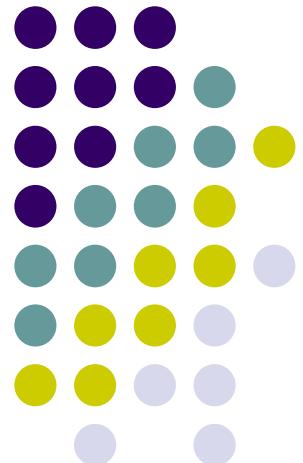


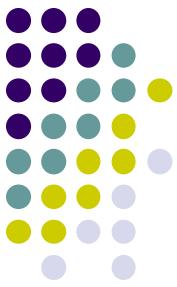
Le langage Java

K. Douzi



Introduction à Java





Qu'est ce que Java?

- Langage portable grâce à l'exécution par une machine virtuelle JVM « *Write once, run everywhere* »
 - » Indépendant des plates-formes
 - Difficile à atteindre: « *Write once, debug everywhere* »
- Simple, orienté objet
- Sûr
- Dynamique et distribué

Historique

Dates clés



- 1991 James Gosling à Sun MicroSystems développe Oak
 - programmer tous les processeurs (ordinateurs ou appareils électroménagers, ...)
 - caractéristiques initiales: robustesse, compatibilité, petite taille du runtime ou des codes générés, facilité de programmation.
 - plateforme pour interpréter les programmes du langage
- 1994 Abandon du projet
- 1995 Après une présentation à Netscape
 - Reprise du projet
 - Intégration dans Netscape sous forme d'applet
- Versions Majeures
 - 1.0
 - 1.1 modèle événementiel, exceptions..
 - 1.2 extensions javax, swing, J2EE
 - 1.3 Hotspot
 - 1.4 Java NIO...
 - ...Java signifie café en Slang (argot américain)...

Historique

Un nouvel environnement pour de nouvelles solutions informatiques



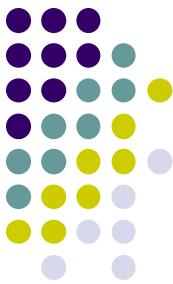
- C++ commence à s'écrouler
 - Complexité des environnements C++
 - Langage C, incrément objet, bibli de classes d'encapsulation des OS.
 - Différences des environnements :
 - Multiplication des Technologies de compilation (dépendance entre le langage et le processeur)
 - Faible durée de vie des composants logiciels
 - Problème de fiabilité et sécurité en utilisation WWW.
 - Utilisation de pointeurs,...



- Un nouveau langage
 - Quelque part entre Smalltalk et C++
- Un nouvel environnement d'exécution adapté au Web
 - interprétation du byte code par une machine virtuelle

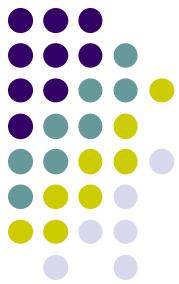
L'environnement Java

Langage et JDK

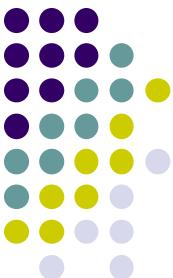


- Langage de programmation orienté objets
 - Situé entre SMALLTALK et C++, permet de développer des applications de taille importante
 - Intégrant les besoins de l'informatique actuelle
 - Répond à des objectifs de portabilité
- Une plateforme de déploiement
 - Une machine virtuelle (JVM)
 - Des bibliothèques de base (API).
- Un ensemble d'outils : Le JDK
 - Compilateur
 - Débugger
 - Documentation
 - ...
- Fournisseur
 - Sun fournit des JDK pour les principales plate-formes :
 - WIN32, MacOS, Solaris et Linux
 - autres éditeurs d'environnement Java :
 - IBM, Microsoft, Novell, Tower J, Blackdown, Appeal Jrockit ...

Caractéristiques



- Simple
 - Syntaxe proche du C / C++
 - Pas de pointeurs
 - Organisation du code (packages)
 - Pas de gestion explicite de la mémoire (ramasse miettes)
 - Tout est objet (sauf les type primitifs)
 - Pas d'héritage multiple(utilisation d'interfaces)
 - Librairies de classes (sockets, BD, graphiques...)
- Orienté Objet
 - paradigme de programmation le plus utilisé
 - tout est classe

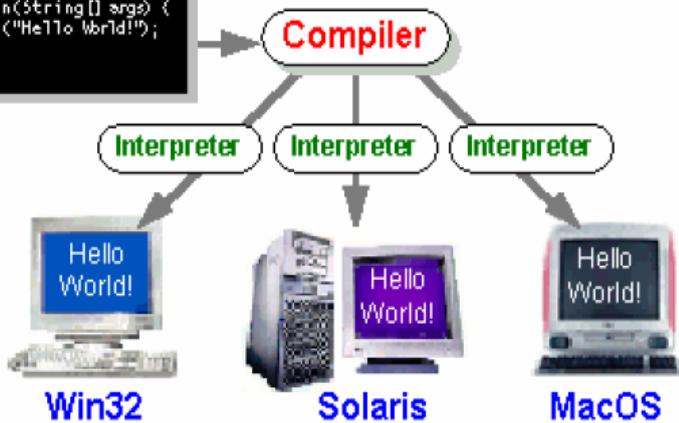


Caractéristiques

- Interpréte, architecturalement neutre et portable
 - Code Source transformé en Bytecode indépendant de l'OS
 - Bytecode interpréte par une machine virtuelle

Java Program

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

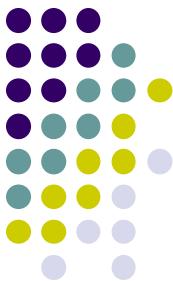


```
// Bytecode consists of opcode and  
operands.
```

```
// Bytecode stream:  
03 3b 84 00 01 1a 05 68 3b a7 ff f9
```

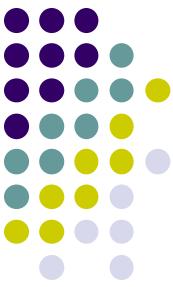
```
// Disassembly:  
iconst_0          // 03  
istore_0          // 3b  
iinc 0, 1         // 84 00 01  
iload_0           // 1a  
iconst_2          // 05  
imul              // 68  
istore_0          // 3b  
goto -7           // a7 ff f9
```

- Taille des types primitifs indépendants de l'OS
- Bibliothèques de classes « standard »



Caractéristiques

- Robuste, Fiable et Sécurisé
 - Langage pour les applications embarquées.
 - Gestion de la mémoire par un ramasse miettes (garbage collector)
 - Impossible de corrompre la mémoire
 - Pas d'accès direct à la mémoire
 - Pas de pointeurs
 - Contrôle du débordement dans les tableaux
 - Mécanisme d'exception.
 - Compilateur strict (erreur si exception non gérée, si utilisation d'une variable non affectée, ...).
 - Bytecode vérifié avant l'exécution par l'interpréteur
 - Accès aux ressources contrôlé



Caractéristiques

- Distribué
 - API réseau (java.net.Socket, java.net.URL, ...).
 - API pour les objets distribués (RMI,CORBA)
 - API pour le Web (servlets)
- Multi-thread (processus légers)
 - Intégrés au langage et aux API :
 - Gestion de la synchronisation
- Dynamique
 - Chargement dynamique des classes
 - Introspection...



Points faibles

- Pas aussi rapide qu'un programme natif
- Gourmand en mémoire
- Absence de surcharge d'opérateurs comme en C++

Plate-forme JAVA et librairies (API)

Les librairies standards

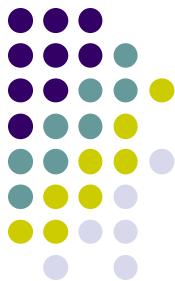


Elles diminuent la charge de travail, standardisent les applications, et fournissent des exemples de source Java de référence

- Les librairies standards :
 - java.lang : Types de bases, Threads, ClassLoader, Exception, Math, ...
 - java.util : Collections (HashMap, ArrayList, TreeMap...), Ressources, Logging, Compression, Préférences
 - java.applet
 - java.awt, javax.swing : Interfaces Graphiques
 - java.io: Accès aux I/O par flux
 - java.net: Socket (UDP, TCP, multicast), URL, ...
 - java.lang.reflect : Introspection
 - java.beans : Composants logiciels
 - java.sql, javax.sql: Accès aux bases de données
 - java.security : signature, cryptographie, authentification
 - java.rmi : Remote Method Invocation
 - java.xml

Plate-forme JAVA et librairies (API)

Les librairies d'extensions Standards



- Les Extensions Standards
 - Java security :
 - cryptography, digital signature, encryption and authentification
 - Java Media API :
 - 2D, Video, Audio, MIDI, Animation, Share, Telephony, 3D
 - Java Enterprise API :
 - JDBC(Java Database Connectivity), IDL, RMI (Remote Methode Invocation)
 - Java Commerce API
 - gestion de services de paiements électroniques, cryptographie évoluée,...
 - Java Server
 - service de gestion de serveurs intranet / internet
 - ...



- Différentes versions
 - Java 1.02: 250 classes, lent
 - Java 1.1: 500 classes: un peu plus rapide
 - Java 2: 2300 classes (différents versions): beaucoup plus rapide
 - Java 5: 3270 classes
 - Java 6
 - ...
- 3 éditions:
 - J2SE: Java 2 standard Edition; JDK = *J2SE Development Kit*, aussi appelé SDK (*Software Development Kit*) pour certaines versions
 - J2EE : Enterprise Edition qui ajoute les API pour écrire des applications installées sur les serveurs dans des applications distribuées : servlet, JSP, EJB,...
 - J2ME : Micro Edition, version allégée de Java pour écrire des programmes embarqués (cartes à puce/Java card, téléphones portables,...)



Premier programme

- Le code source du premier programme:
anatomie d'une classe:

Type de retour void

Signifie pas de valeur
de retour

public class PremiereAppli {

Nom de la classe

public static void main(String[] args){

Nom de la méthode

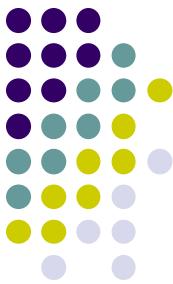
System.out.println("J'apprends java");

*Public pour
que tout
Le monde
puisse y accéder*

}

Afficher sur la sortie standard

La chaîne à afficher



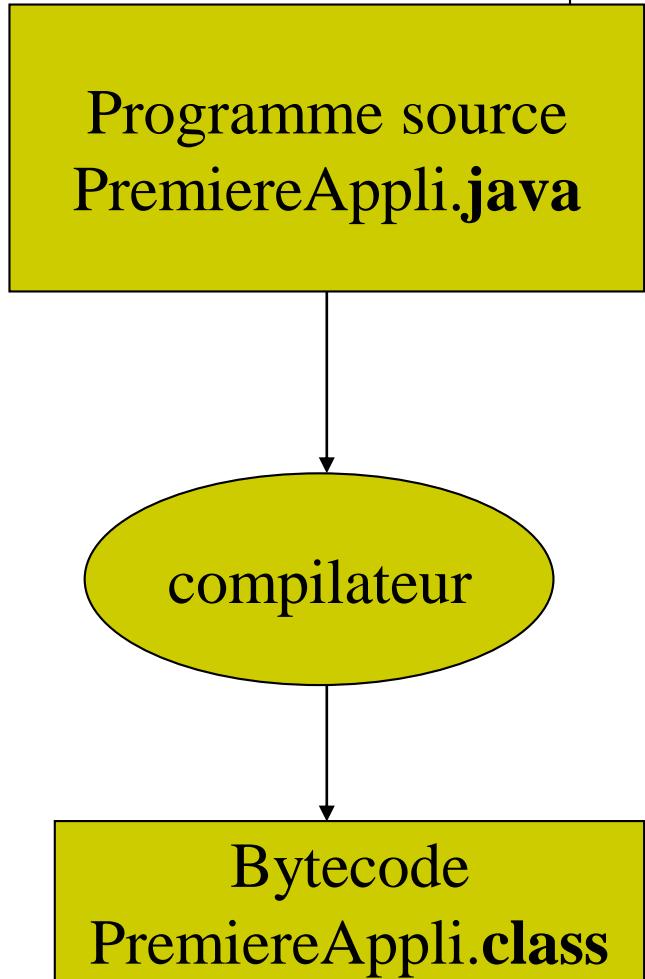
Compilation en Java → *bytecode*

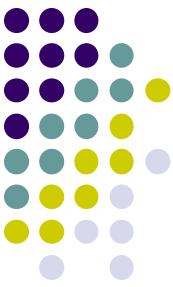
- En Java, le code source n'est pas traduit directement dans le langage de l'ordinateur
- Il est d'abord traduit dans un langage appelé « *bytecode* », langage d'une machine virtuelle (JVM ; *Java Virtual Machine*) définie par *Sun*
- Ce langage est indépendant de l'ordinateur qui va exécuter le programme

La compilation fournit du *bytecode*

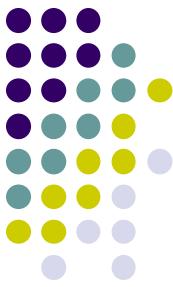


- Programme Java
- Compilateur: **javac**
- Programme en *bytecode*, indépendant de l'ordinateur



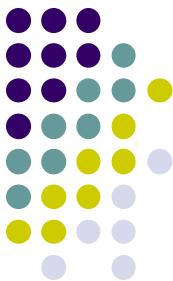


- Compilation: javac PremièreAppli.java
- Exécution: java PremièreAppli
- Dans toute application il faut une classe publique qui contient main():
 - La première méthode à être exécutée

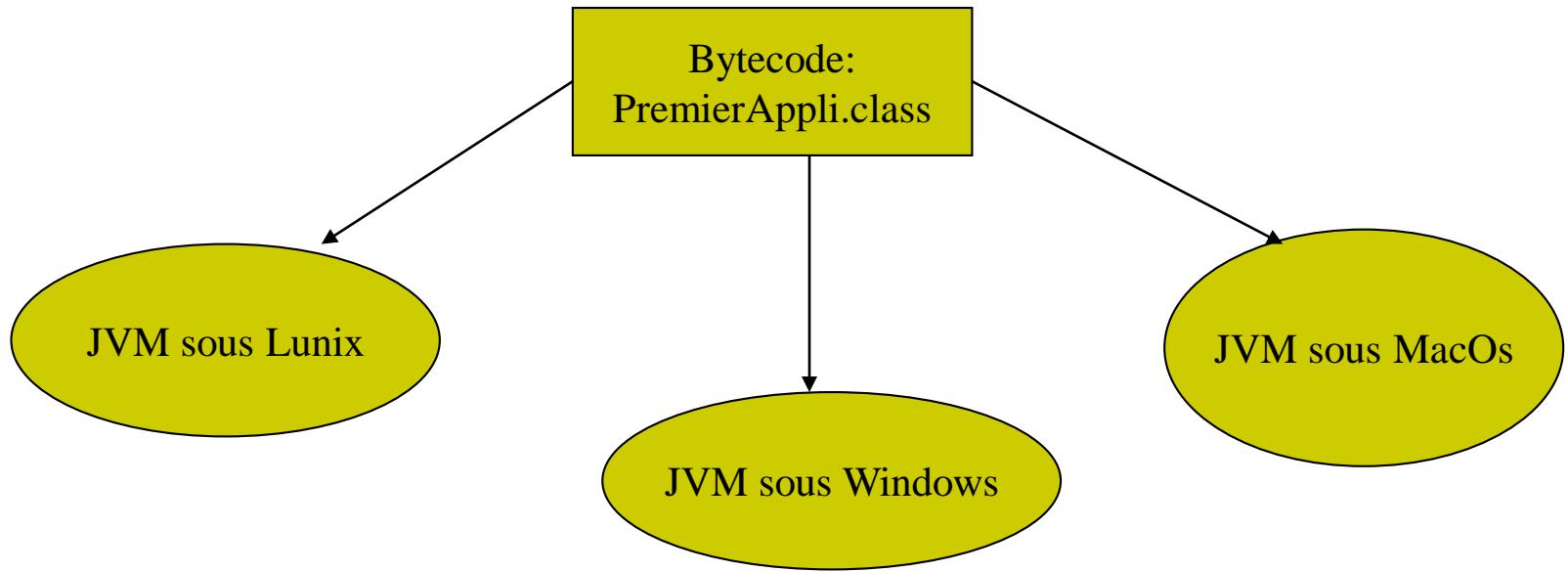


Java Virtual Machine

- Les systèmes qui veulent pouvoir exécuter un programme Java doivent fournir une JVM
- A l'heure actuelle, tous les systèmes ont une JVM (Linux, Windows, MacOs,...)
- Il existe aussi depuis peu quelques JVM « en dur », sous forme de processeurs dont le langage natif est le *bytecode* ; elles sont rarement utilisées (en raison de la portabilité)



Le *bytecode* peut être exécuté par n'importe quelle JVM



- Si un système possède une JVM, il peut exécuter tous les fichiers **.class** compilés sur n'importe quel autre système



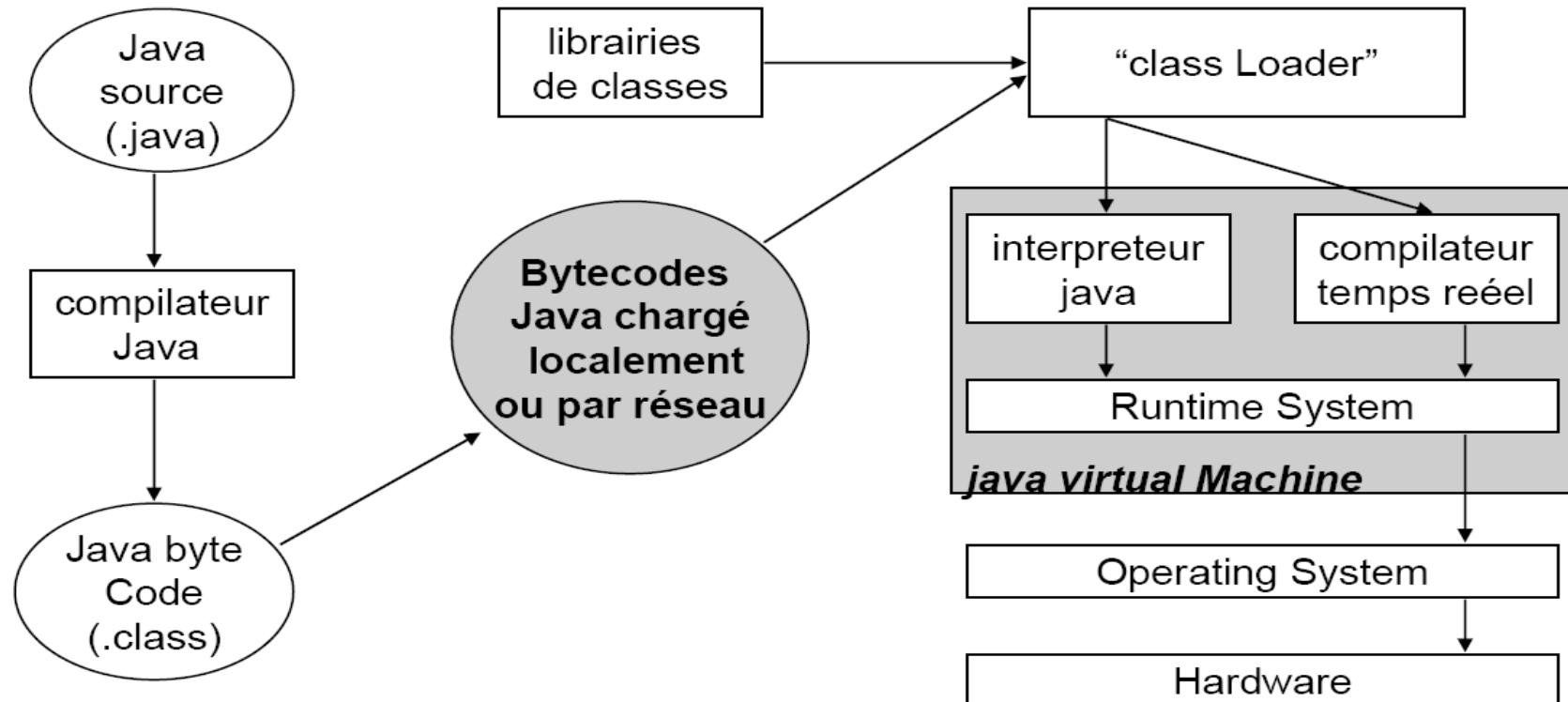
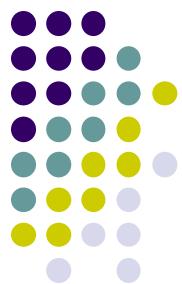
Avantages de la JVM pour Internet

- Grâce à sa portabilité, le *bytecode* d'une classe peut être chargé depuis une machine distante du réseau, et exécutée par une JVM locale
- La JVM fait de nombreuses vérifications sur le *bytecode* avant son exécution pour s'assurer qu'il ne va effectuer aucune action dangereuse
- La JVM apporte donc
 - de la souplesse pour le chargement du code à exécuter
 - mais aussi de la sécurité pour l'exécution de ce code



- Les vérifications effectuées sur le bytecode et l'étape d'interprétation de ce bytecode (dans le langage natif du processeur) ralentissent l'exécution des classes Java
- Les techniques « *Just In Time (JIT)* » ou « *Hotspot* » réduisent ce problème : elles permettent de ne traduire qu'une seule fois en code natif et à la volée les instructions qui sont exécutées

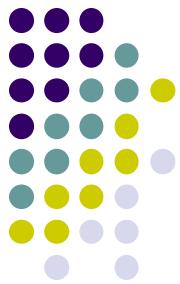
L'environnement de génération et d'exécution Java



Environnement de génération

*Environnement d'exécution
(java Platform)*

Plate-forme JAVA



Programme Java

API

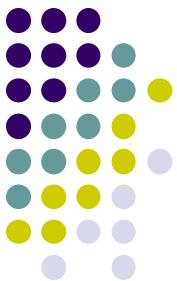
JVM

Machine réelle

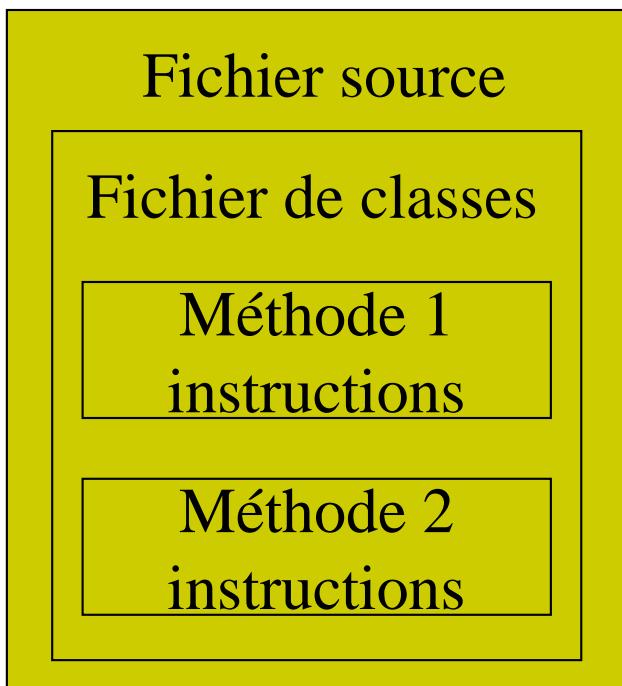


Votre environnement de développement

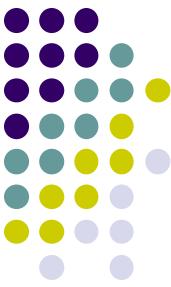
- SE: Linux ou Windows
- Editeur de texte (emacs, word,...)
- Compilateur (javac)
- Interpréteur de bytecode (java)
- Aide en ligne
- Générateur automatique de documentation (javadoc)
- Débogeur (jdb)
- ...
- Un IDE : Eclipse



Structure d'une application

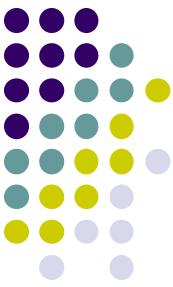


- Placer une classe dans un fichier source
- Placer les méthodes dans une classe
- Placer les instructions dans les méthodes



Exemples: 1 classe point

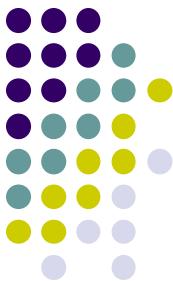
```
/** Modélise un point de coordonnées x, y */
public class Point {
    private int x, y;
    public Point(int x1, int y1) { // un constructeur
        x = x1;
        y = y1;
    }
    public double distance(Point p) { // une méthode
        return Math.sqrt((x-p.x)*(x-p.x) + (y-p.y)*(y-p.y));
    }
    public static void main(String[] args) {
        Point p1 = new Point(1, 2); // on crée deux objets
        Point p2 = new Point(5, 1);
        System.out.println("Distance : " + p1.distance(p2));
    }
}
```



2 classes et 1 fichier

```
/** Modélise un point de coordonnées x, y */
public class Point {
    private int x, y;
    public Point(int x1, int y1) {
        x = x1; y = y1;
    }
    public double distance(Point p) {
        return Math.sqrt((x-p.x)*(x-p.x) + (y-p.y)*(y-p.y));
    }
}
/** Teste la classe Point */
class TestPoint {
    public static void main(String[] args) {
        Point p1 = new Point(1, 2);
        Point p2 = new Point(5, 1);
        System.out.println("Distance : " + p1.distance(p2));
    }
}
```

Fichier Point.java



Compilation et exécution de la classe Point

- La compilation du fichier **Point.java**
javac Point.java fournit 2 fichiers classes :
Point.class et **TestPoint.class**
- On lance l'exécution de la classe **TestPoint** qui a une méthode **main()**:
java TestPoint



2 classes dans 2 fichiers

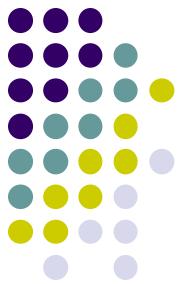
```
/** Modélise un point de coordonnées x, y */
public class Point {
private int x, y;
    public Point(int x1, int y1) {
        x = x1; y = y1;
    }
    public double distance(Point p) {
        return Math.sqrt((x-p.x)*(x-p.x) + (y-p.y)*(y-p.y));
    }
}
```

Fichier **Point.java**

```
/** Pour tester la classe Point */
class TestPoint {
    public static void main(String[] args) {
        Point p1 = new Point(1, 2);
        Point p2 = new Point(5, 1);
        System.out.println("Distance : " + p1.distance(p2));
    }
}
```

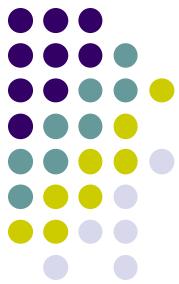
Fichier **TestPoint.java**

Architecture générale d'un programme Java



- Programme source Java = ensemble de fichiers « **.java** »
- Chaque fichier « **.java** » contient une ou *plusieurs* définitions de classes
- Au plus une définition de classe **public** par fichier « **.java** » avec nom du fichier = nom de la classe publique

Chargement dynamique des classes



- Durant l'exécution d'un code Java, les classes (leur *bytecode*) sont chargées dans la JVM au fur et à mesure des besoins
- Une classe peut être chargée:
 - depuis la machine locale (le cas le plus fréquent)
 - depuis une autre machine, par le réseau
 - par tout autre moyen (base de données,...)



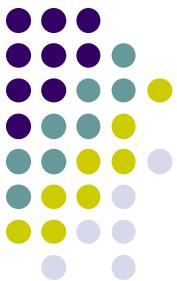
Autre exemple

- L'utilisateur fournit son age en argument de ligne de commande.

```
public class Age {  
    public static void main(String args[]) {  
        int age;  
        age = Integer.parseInt(args[0]);  
        System.out.println("Vous avez " + age + " ans.");  
    }  
}
```

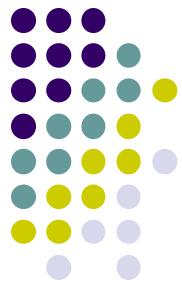
- Exécution: java Age 5

Résultat: Vous avez 5 ans



- String est une classe
- String args[] est un tableau de chaînes de caractères qui stocke les arguments fournis en ligne de commande
- int est un type primitif
- Integer est une classe d'objets de type int
- parseInt() est une méthode de la classe Integer qui convertit String en int

Applications indépendantes et *applets*



- Java permet de développer deux sortes de programmes:
 - Applications indépendantes
 - Applets exécutées dans l'environnement/JVM d'un navigateur Web et chargées par une page HTML

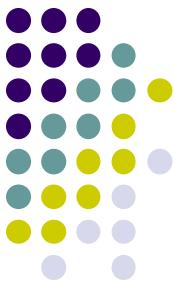


Application indépendante

Exécution de la classe de lancement de l'application (dite classe principale ; *main* en anglais) ; par exemple :

```
java TestPoint
```

java lance l'interprétation du code de la méthode **main()** de la classe **TestPoint**



Applet

- Une applet est une classe compilée héritant de **java.applet.Applet**
- Objet de la classe Java **Applet**, référencé dans une page Web (écrite dans le langage HTML)
- Le lancement d'une (un ?) applet(te ?) se fait quand la partie de la page Web qui référence l'applet est affichée par le client Web

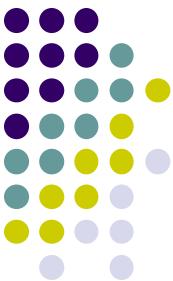
Exemple de page Web qui contient une applet



```
<HTML>
<HEAD>
<TITLE> Une applet </TITLE>
</HEAD>
<BODY>
<H2> Exécution d'une applet </H2>
<APPLET code="HelloApplet.class"
width=500
height=300>
Votre navigateur ne peut exécuter une applet
</APPLET>
</BODY>
</HTML>
```



Dimensions de l'emplacement réservé à l'affichage de l'applet

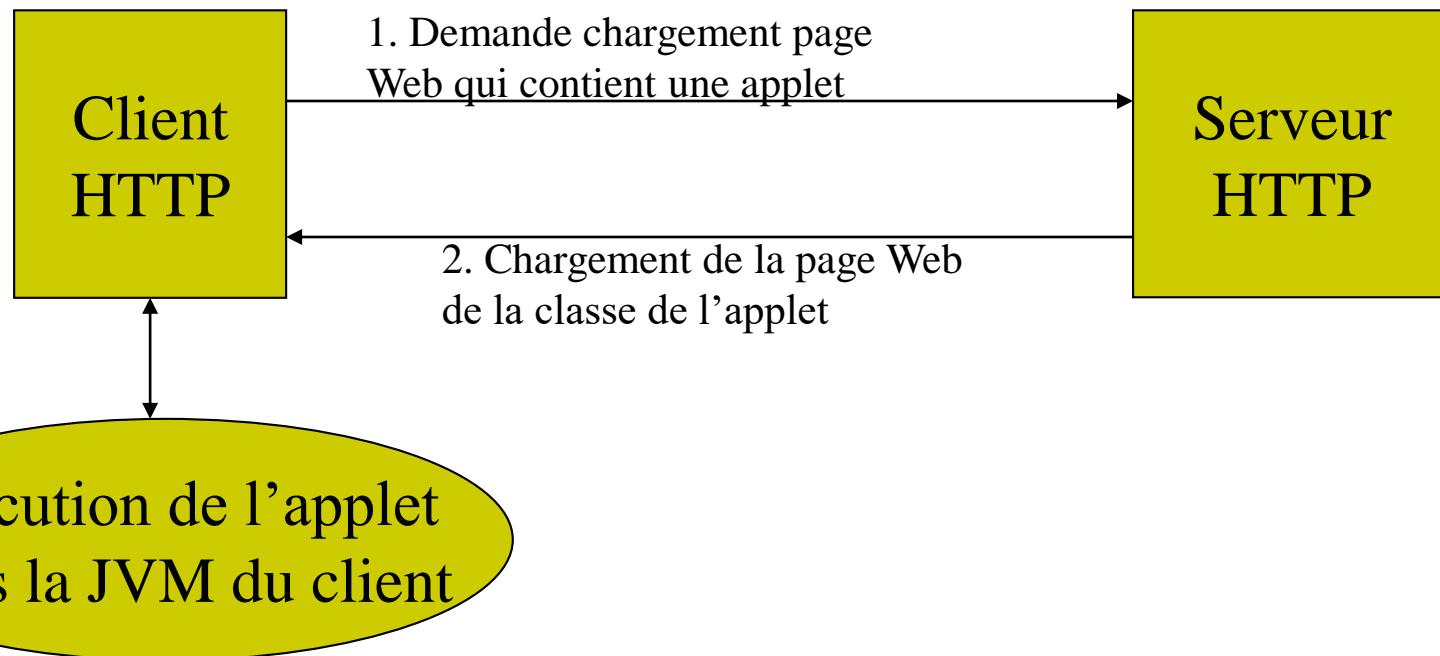
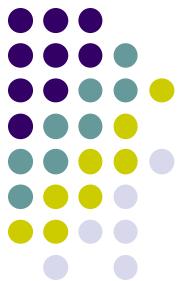


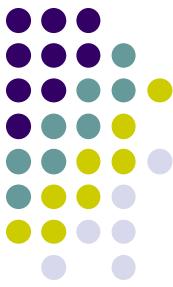
```
import java.awt.Graphics;  
import java.applet.Applet;  
public class HelloApplet extends Applet {  
    public void paint(Graphics g) {  
        g.drawString("Hello world", 50, 25);  
    }  
}
```

Représente l'emplacement
de la page Web où l'applet
s'affichera

Zone où commencera
l'affichage : x = 50 pixels,
y = 25 pixels

Étapes pour l'exécution d'une applet





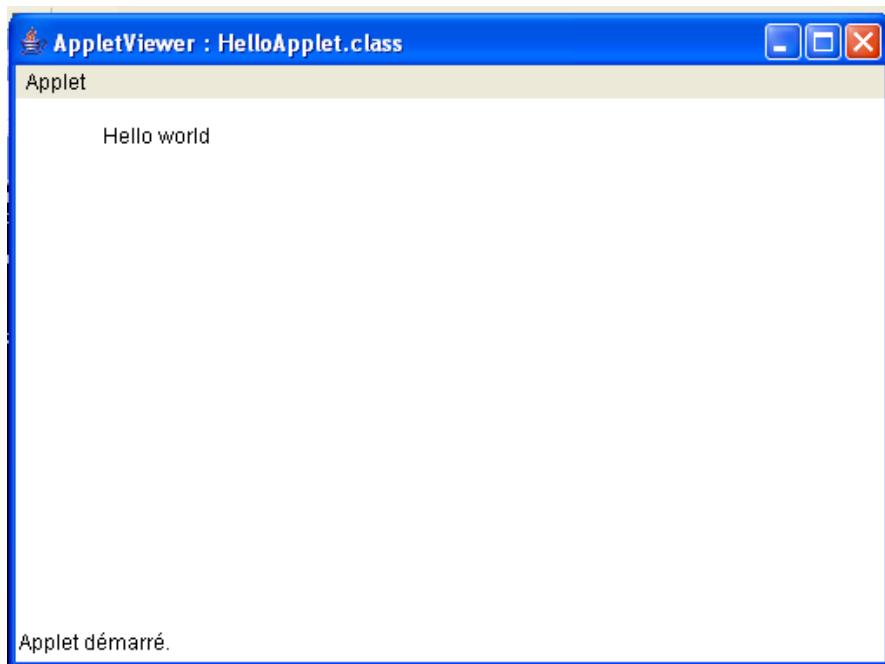
Remarques

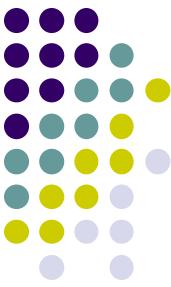
- Le navigateur a sa propre machine virtuelle
- Un programme Java spécial démarré par le navigateur va lancer certaines méthodes de la classe **Applet** : **init()**, **start()**, **stop()**, **destroy()**, **paint()**
- **init()** est exécuté seulement quand l'applet est lancée pour la première fois
- **paint()** dessine l'applet dans la page Web



Exécution de l'applet

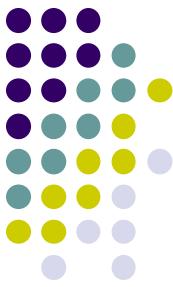
- Soit:
 - A partir du navigateur
 - A partir de l'appletviewer
 - Javac HelloApplet.java -> HelloApplet.class
 - Appletviewer HelloApplet.html





Utilité des applets

- Les applets permettent de faire des pages Web plus riches (grâce aux possibilités offertes par Java)
- La page Web peut contenir
 - des animations ou des mises en forme complexes pour mettre en valeur certaines informations
 - des résultats de calculs complexes
 - des informations « dynamiques » (pas connues au moment où la page Web statique est créée) trouvées en interrogeant une base de données



TD1

- Exo1: Reprendre les exercices étudiés en cours créez les applications, compilez et exécutez.
- Exo2: Créer une application qui calcule la somme de 3 nombres donnés en arguments de ligne de commande

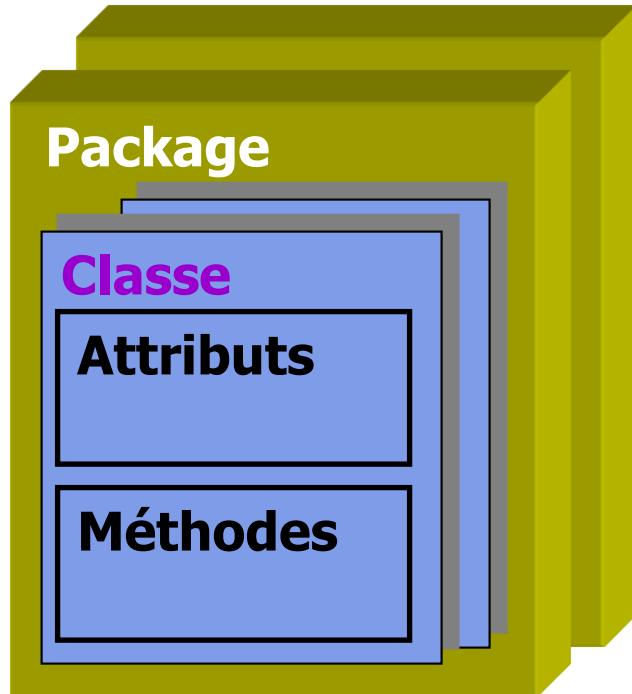
La syntaxe



Constitution d'un programme



- Un programme Java utilise un ensemble de classes
- Les classes sont regroupées par package
- Une classe regroupe un ensemble d'attributs et de méthodes





Déclaration d'une classe

- Le nom de la classe est spécifié derrière le mot clé « class »
- Le corps de la classe est délimité par des accolades
- On définit dans le corps les attributs et les méthodes qui constituent la classe

```
class Test {  
  
    < corps de la classe >  
  
}
```



Définition d'un méthode

- Une méthode est constituée de :
 - D'un nom
 - D'un type de retour
 - De paramètres (éventuellement aucun)
 - D'un bloc d'instructions
- Un paramètre est constitué :
 - D'un type
 - D'un nom
- « void » est le mot-clé signifiant que la méthode ne renvoie pas de valeur

```
class Test {  
    int calculer (int taux, float delta) {  
        < corps de la méthode >  
    }  
}
```

Bloc d'instructions



- Un bloc d'instructions est délimité par des accolades
- Il contient un ensemble d'instructions
- Toute instruction est terminée par un point virgule
- Un bloc d'instructions peut contenir d'autres blocs d'instructions

```
{  
    int i = 0;  
    if (i==0)  
        System.out.println ("Valeur de i : " + i);  
}
```



Instructions possibles

- Déclaration d'une variable
- Appel de méthode
- Affectation
- Instruction de boucle (while, for...)
- Instruction de test (if, switch)



Corps d'une méthode

- Le corps d'une méthode est un bloc d'instructions
- Le mot clé « return » permet de renvoyer une valeur à l'appelant de la méthode
- Il doit renvoyer une valeur du même type que le type de retour de la méthode

```
class Test {  
    int calculer (int taux, float delta) {  
        return taux * delta;  
    }  
}
```



Déclaration d'une variable

- Une variable possède un type et un nom
- Le type peut être un type de base ou une classe
- L'initialisation d'une variable peut se faire au moment de la déclaration

```
{  
    int compteur;  
    int indice = 0;  
  
    Voiture golf;  
    Voiture twingo = new Voiture();  
}
```

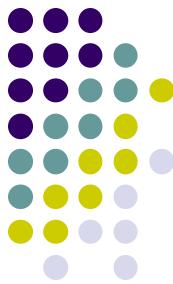


Portée d'une variable

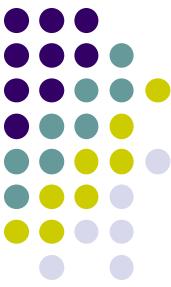
- La portée d'une variable s'étend jusqu'à la fin du bloc dans lequel elle est définie

```
{  
    {  
        int compteur;  
        ...  
        // compteur est accessible  
    }  
  
    // compteur n'est plus accessible  
}
```

Nom de classes, de variables et de méthodes



- Un nom peut être constitué de lettres, de chiffres et du caractère souligné
- Il ne peut pas commencer par un chiffre
- Le caractère \$ est utilisable mais à éviter
- Les mots réservés (if, class, int...) ne peuvent pas être utilisés



Point d'entrée

- Une application possède un point d'entrée fourni par la méthode « main » (sauf les applets)
- Attention à la casse, Java est sensible aux majuscules/minuscules

```
class Test {  
  
    public static void main (String[] args) {  
        ...  
        // corps de la méthode main  
    }  
  
}
```



Les différents types de base (1/2)

- Types arithmétiques

byte	8 bits	signé	(-128 , 127)
short	16 bits	signé	(-32768 , 32767)
int	32 bits	signé	(-2147483648 , 2147483647)
long	64 bits	signé	(-9223372036854775808, 9223372036854775807)
float	32 bits	signé	(1.4E-45 , 3.4028235E38)
double	64 bits	signé	(4.9E-324 , 1.7976931348623157E308)

- Type caractère

char	16 bits	non signé UNICODE2
------	---------	--------------------

- Type booléen

boolean	1 bit	deux valeurs possibles : true ou false
---------	-------	--

Les différents types de base (2/2)



- Les types de base s'écrivent en minuscules (int, float..)
- Le type « int » est codé sur 4 octets → portabilité
- Le type « char » est codé sur 2 octets pour supporter les jeux de caractères Unicode
- Une chaîne de caractères est déclarée avec le mot-clé « String »
 - Ce n'est pas un type de base
 - Il se manipule comme un type de base
 - Ce n'est pas équivalent à un tableaux de caractères

```
String s = "Hello World"
```



L'affectation

- L'opérateur « = » permet d'affecter la valeur de l'expression qui est à droite à la variable qui est à gauche

```
class Test {  
    int calculer () {  
        int i = 0;  
        int j = 6;  
        i = (j + 5) * 3;  
        return i + j;  
    }  
}
```

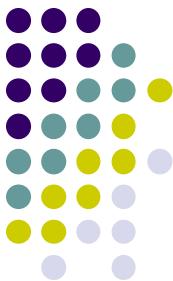


Les opérateurs arithmétiques

- S'appliquent sur les types **entiers** ou **réels**
 - **+, - , * , / , % (modulo) , += , -= , *= , /=**

```
int x, y, z;  
y = 2;  
z = 3;  
  
x = y * z ;      //x vaut 6  
x += 2 ;         //x vaut 8  
y = x / 4 ;      //y vaut 2  
y = x % 2 ;      //y vaut 0
```

- Les opérateur **=** et **+=** peuvent être utilisés sur des variables de type « String »
- En terme de performance, il est recommandé d'utiliser la notion raccourcie



Opérateurs unaires

- S'appliquent à un seul opérande de type **entier** ou **réel**
- **-**, **--**, **+**, **++**

```
int x, y;  
x = 3;  
  
y = -x ;           //y vaut -3  
y = ++x ;          //y vaut 4, x vaut 4  
y = x-- ;          //y vaut 4, x vaut 3
```

- La pré et la post-incrémantation diminuent le nombre de lignes de byte code générées



Opérateurs de comparaison

- S'appliquent sur des **entiers, booléens, réels**
==, !=, <=, >, >=
- Ces opérateurs retournent une valeur du type boolean

```
{  
    boolean droitDeVote;  
    int age;  
  
    droitDeVote = (age >= 18) ;  
}
```



Opérateurs logiques

- S'appliquent au type **boolean**
 - **!** (not) , **&&** (and) , **||** (or)
 - **&**, **|**
- Retournent un type **boolean**
- **&** renvoie « true » si les deux expressions renvoient « true »
- **&&** a le même comportement mais n'évalue pas la seconde expression si la première est « false »
- **|** renvoie « true » si l'une des deux expressions renvoie « true »
- **||** a le même comportement mais n'évalue pas la seconde expression si la première est « true »



Les conversions de type (1/2)

- Il y a 4 contextes possibles de conversion (cast) :
 - Conversion explicite
 - Affectation
 - Appel de méthode
 - Promotion arithmétique
- Certaines conversions provoquent une perte de valeur
 - Float en int, int en short, short en byte
- Le type boolean ne peut pas être converti en entier

Les conversions de type (2/2)



```
double f = 3.14 ;
int i, j ;
short s ;

i = (int)f;           // float → int (conversion explicite)
float ff = (float)3.14;

i = s;                // short → int (affectation)

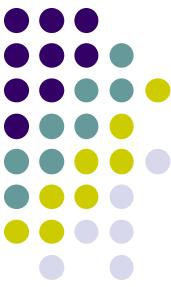
// appel de la méthode obj.m(int i)
obj.m(s);            // short → int (appel de méthode)

// division d'un entier et d'un flottant : l'entier i est
// converti en flottant, puis la division flottante est
// calculée
f = i / (double)j;    // f vaut 0.3333...
```



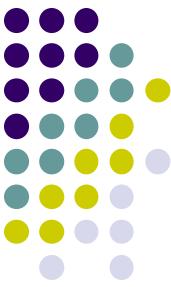
Test conditionnel

```
class Test {  
    public static void main (String args[]) {  
        int compteur = 0;  
        boolean debut;  
  
        if (compteur == 0) {  
            debut = true;  
            System.out.println("Début de la partie");  
        }  
        else if (compteur ==10)  
            System.out.println("Fin de la partie");  
  
        else  
            System.out.println("Partie en cours");  
    }  
}
```



Boucles while

```
class Test {  
    public static void main (String args[]) {  
        int i;  
  
        do {  
            System.out.println("Entrez un nombre < 10");  
            c= lireUnInt();  
        } while (c>10);  
  
        while (c>0) afficher (c--);  
    }  
  
    public static int lireUnInt() {.....}  
  
    public static void afficher (char c) {.....}  
}
```



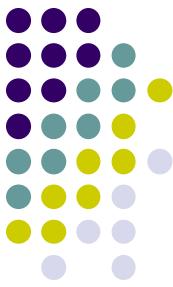
Boucles for

```
class Test {  
    public static void main (String args[]) {  
        int i;  
  
        for (i=0 ; i <=5 ; i++) {  
            System.out.println (i);  
            .....  
        }  
  
        for (j=0 ; j <=5 , j++) {  
            System.out.println (i);  
            .....  
        }  
  
        // i est accessible  
        // j n'est pas accessible  
    }  
}
```



Switch

```
class Test {  
    public static void main (String args[]) {  
  
        char c = (char)System.in.read();  
        switch(c) {  
            case 'o' : case 'O' :  
                System.out.println("Oui");  
                break;  
            case 'n' : case 'N' :  
                System.out.println("Non");  
                break;  
            default :  
                System.out.println("Mauvaise réponse");  
                break;  
        }  
    }  
}
```



Commentaires (1/2)

- L'utilisation de commentaires est fortement recommandé
- */*ceci est un commentaire sur plusieurs lignes */*
- *//ceci est un commentaire sur une ligne*
- Javadoc (fourni dans le JDK) génère la documentation des classes en format HTML
 - */*** : début de commentaire Javadoc
 - *@author* : exemple de tag auteur
 - **/* : fin de commentaire Javadoc



Les tableaux : description

- Nombre fixe d'éléments. Taille fixée à la construction
- Les éléments sont de même type (type de base ou classe)
- Les tableaux sont alloués dynamiquement par « new »
- Un tableau est détruit lorsqu'il n'est plus référencé
- Les tableaux multi-dimensionnels sont des tableaux de tableaux



Les tableaux : syntaxe

- 2 syntaxes pour l'allocation :

```
int[] monTableau = new int[10];  
  
int monTableau[] = new int[10];
```

- Une méthode peut renvoyer un tableau

```
classe Test {  
  
    int[] construireTableau (int dimension) {  
        int tab[] = new int[dimension];  
        return tab;  
    }  
}
```



Les tableaux : initialisation

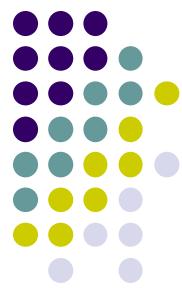
- L'attribut « length » renvoie la dimension du tableau
- L'indice des éléments du tableaux varie de 0 à « tableau.length –1 »
- Initialisation statique :

```
int[] monTableau = {1, 2, 3};
```

- Initialisation dynamique :

```
int[] param = new int[10];  
  
for (int i = 0; i < param.length ; i++)  
    param[i]= i;
```

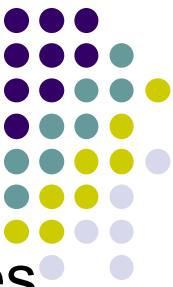
Classes, méthodes, attributs





Déclarer une classe

- Une classe « public » est visible en dehors du package
 - Par défaut, elle n'est pas visible
 - On ne peut pas mettre 2 classes publiques dans un fichier → erreur de compilation
- Une classe « final » ne peut pas être dérivée (pas de sous-classes)
 - Utile pour des raisons de sécurité et de performances
 - De nombreuses classes du JDK sont « final »
- Une classe « abstract » ne peut pas être instanciée (new)



Déclarer un attribut (1/3)

- Lors de la création d'un objet, les attributs sont initialisés par défaut :
 - À zéro pour les valeurs numériques
 - À « null » pour les références
 - À « false » pour les booléens
- Les attributs peuvent être initialisés :
 - Lors de la déclaration
 - Dans le constructeur



Déclarer un attribut (2/3)

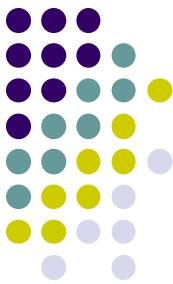
- La valeur d'un attribut déclaré comme « static » est partagée par toutes les instances (objets) de la classe
- La valeur d'un attribut déclaré comme « final » est constante



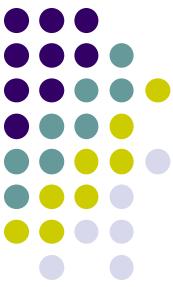
Déclarer un attribut (3/3)

```
class Date {  
    int _jour;                      // initialisé à 0  
    int _mois = 1;                   // initialisation explicite  
    int _an = 2000;  
    final static int max_mois = 12;   // Constante  
  
    void print () {  
        System.out.println(_jour + "/" + _mois + "/" + _an);  
    }  
}  
  
.....  
  
Date d = new Date();           // instantiation de l'objet  
d.print();                     // appel de la méthode print
```

Le passage de paramètres



- Lors de l'appel d'une méthode prenant en paramètre des **types de bases**, les paramètres sont passés par valeur
 - La valeur des variables passées en paramètres est dupliquée
 - Ce sont ces valeurs dupliquées qui sont manipulées dans la méthode
- Lors de l'appel d'une méthode prenant en paramètre des **objets**, les paramètres sont passés par référence
 - Ils peuvent être modifiés dans la méthode



Les arguments variables (varargs)

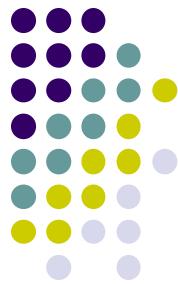
- Cette nouvelle fonctionnalité va permettre de passer un nombre non défini d'arguments d'un même type à une méthode. Ceci va éviter de devoir encapsuler ces données dans une collection.
- Cette nouvelle fonctionnalité implique une nouvelle notation pour préciser la répétition d'un type d'argument. Cette nouvelle notation utilise trois petits points : ...

Exemple (java 1.5) :

```
public class TestVarargs {  
    public static void main(String[] args) {  
        System.out.println("valeur 1 = " +  
            additionner(1,2,3));  
        System.out.println("valeur 2 = "  
            +additionner(2,5,6,8,10)); }  
    public static int additionner(int ... valeurs)  
    {  
        int total = 0;  
        for (int val : valeurs) { total += val; }  
        return total;  
    }  
}
```

Résultat :

```
C:\tiger>java TestVarargs  
valeur 1 = 6  
valeur 2 = 31
```



Constructeurs

- C'est une ou plusieurs méthode(s) permettant d'initialiser les objets
- Le constructeur est appelé lors de la création de l'objet
- Le constructeur a le même nom que la classe
- Il n'a pas de valeur de retour (*void* est un type de retour)
- Le constructeur peut être surchargé
- Java fournit un constructeur par défaut (sans paramètres) si aucun constructeur n'est défini explicitement



Exemple de constructeurs

```
class Date {  
    int _jour = 1;  
    int _mois = 1;  
    int _an = 2000;  
  
    Date() {  
        _an = 1999;  
    }  
  
    Date (int jour, int mois, int an) {  
        _jour = jour;  
        _mois = mois;  
        _an = an;  
    }  
  
.....  
  
Date d = new Date(10,12,2000); // instantiation de l'objet
```



Création d'objets

- Allocation de l'espace mémoire pour stocker les variables d'instances
- Utilisation de l'opérateur « new »
- Appelle du constructeur adéquat
- Retourne une référence sur l'objet créé

```
class Voiture {                                // Définition de la classe
    String _type;
    Voiture (String type) {
        _type = type;
    }
    void demarrer () {.....}
}
.....
Voiture clio = new Voiture("ClioRT"); // Création de l'objet
Voiture renault = clio; // Ajout d'une référence sur l'objet
renault.demarrer();
```

Le Garbage Collector (ramasse-miettes)



- Il prend en charge la gestion de la mémoire
- Il alloue l'espace mémoire lors de la création des objets
- Il libère la mémoire occupé par un objet dès qu'il n'y a plus aucune référence qui pointe vers cet objet
- Il est capable de compacter la mémoire pour éviter la fragmentation
- C'est un « Thread » de la machine virtuel Java

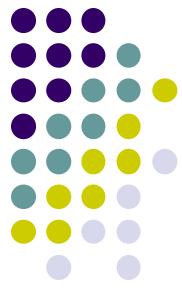


« this »

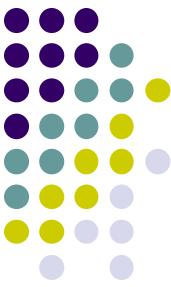
- « this » = référence sur l'objet courant
- Représente l'instance courante en train de s'exécuter

```
class Compte {  
    void crediter(float montant) {.....};  
    void debiter(float montant) {.....};  
}  
  
class Versement {  
    void valider() {.....}  
    void effectuer(Compte s, Compte d, float montant) {  
        s.debiter(montant);  
        d.crediter(montant);  
        this.valider();  
    }  
}
```

Utilisation de « this » dans un constructeur



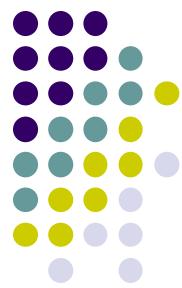
```
class Personne {  
    String _nom ;  
    String _prenom;  
    float _taille;  
  
    Personne(String nom, String prenom) {  
        _nom = nom;  
        _prenom = prenom;  
    }  
  
    Personne(String nom, String prenom, float taille) {  
        this(nom, prenom);  
        _taille = taille;  
    }  
}
```



« null »

- La valeur « **null** » peut être assignée à n'importe quelle référence sur un objet
- Une méthode peut retourner « **null** »
- L'appel d'une méthode sur une référence d'objet valant « **null** » provoque une erreur d'exécution (*NullPointerException*)
- On peut tester une référence pour savoir si elle vaut « **null** »

```
Personne moi = new Personne();  
.....  
if (moi != null) {  
    System.out.println("J'existe toujours ! ");  
}
```



Opérateurs sur les références

- Egalité de deux références : **==**
 - Compare si 2 références pointent vers le même objet
- Différence de deux références : **!=**
 - Compare si 2 références pointent vers des objets différents
- Type d'instance de la référence : **instanceof**
 - Permet de savoir si l'objet référencé est une instance d'une classe donnée ou d'une de ses sous-classes

```
Personne moi = new Personne();  
.....  
if (moi instanceof Personne) {  
    System.out.println("Je suis bien une personne! ");  
}
```



Méthodes « static »

- Le mot-clé « static » permet de définir une méthode comme statique
- Une méthode statique ne peut accéder qu'aux attributs de la classe déclarés comme « static »

- L'appel d'une méthode statique ne se fait pas sur un objet, mais sur une classe
 - Exemple : `Math.cos(3.14);`
- L'utilisation d'une méthode statique ne nécessite pas la création d'un objet



Exemple de méthode « static »

```
class MathUtil {  
    final static double _PI = 3.14 ;  
    static double PI() {  
        return _PI;  
    }  
  
    static double Carre(double x) {  
        return x * x;  
    }  
  
    static double Demi(double x) {  
        return x / 2;  
    }  
}  
.....  
double i = MathUtil.Carre(5);  
double x = MathUtil.PI();
```

X



Static import

- Jusqu'à la version 1.4 de Java, pour utiliser un membre statique d'une classe, il faut obligatoirement préfixer ce membre par le nom de la classe qui le contient.
- Par exemple, pour utiliser la constante Pi définie dans la classe `java.lang.Math`, il est nécessaire d'utiliser `Math.PI`
- Java 1.5 propose une solution pour réduire le code à écrire concernant les membres statiques en proposant une nouvelle fonctionnalité concernant l'importation de package : l'import statique (static import).
- Ce nouveau concept permet d'appliquer les mêmes règles aux membres statiques qu'aux classes et interfaces pour l'importation classique.
- Cette nouvelle fonctionnalité s'utilise comme une importation classique en ajoutant le mot clé static.

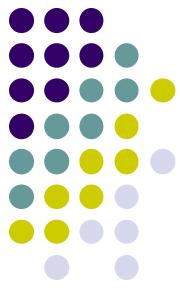
Exemple :

```
public class TestStaticImportOld {  
    public static void main(String[] args)  
    {  
  
        System.out.println(Math.PI);  
  
        System.out.println(Math.sin(0));  
    }  
}
```

Exemple (java 1.5) :

```
import static java.lang.Math.*;  
public class TestStaticImport {  
    public static void main(String[] args)  
    {  
  
        System.out.println(PI);  
  
        System.out.println(sin(0));  
    }  
}
```

Héritage, polymorphisme, encapsulation





Héritage (1/2)



- Toute classe Java est une sous-classe de la classe « Object »
- Java ne permet pas l'héritage multiple
- La classe dérivée peut changer l'implémentation d'une ou plusieurs méthodes héritées : redéfinition
- Il est possible de faire en sorte que l'on ne puisse pas hériter d'une classe en particulier : utilisation du mot-clé « final »
- Il est possible de faire en sorte qu'une méthode ne puisse pas être redéfinie : utilisation du mot-clé « final »



Héritage (2/2)

```
X
class Felin {
    boolean a_faim = true;
    void parler() { }
    void appeler() {
        System.out.println("minou minou,...");
        if (a_faim) parler();
    }
}

final class Chat extends Felin {
    String race;
    void parler() { System.out.println("miaou! "); }
}

final class Lion extends Felin {
    void parler() { System.out.println("roar! "); }
    void chasser() {.....}
}
```



Conversion entre classes

- Si une variable référence un objet d'une classe, elle peut référencer un objet de n'importe laquelle de ses sous-classes

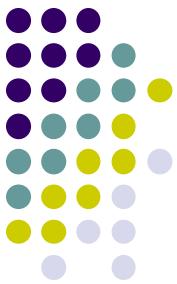
```
class Felin {.....}
class Lion extends Felin {.....}

Lion lion = new Lion();
Felin felin;

felin = lion; // OK conversion implicite : les lions
              // sont des félins

lion = felin // ERREUR : tous les félins ne sont pas
              // des lions
```

Conversion ascendante, conversion descendante



```
class Felin {.....}
class Lion extends Felin {.....}
.....
Felin felin = new Felin();
Lion lion = new Lion();

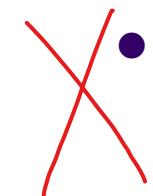
felin = lion;           // Conversion ascendante
felin.parler();         // « roar ! »
felin.chasser();        // Méthode introuvable
lion = felin;          // ERREUR : conversion explicite
                      // nécessaire

lion = (Lion) felin;   // Conversion descendante explicite
lion.parler();          // « roar ! »
lion.chasser();         // OK

Chat chat = new Chat();
felin = chat;           // Conversion ascendante
lion = (Lion) felin   // ERREUR java ClassException
```



Polymorphisme (1/2)

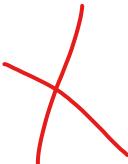


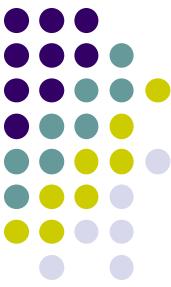
- Une méthode polymorphe est une méthode déclarée dans une super-classe et redéfinie dans une sous-classe
- Une méthode Java est par défaut polymorphe
- Les méthodes « final » ne peuvent pas être redéfinies
 - Permet à la machine virtuel d'optimiser le byte-code
 - Est utilisé pour des raisons de sécurité



Polymorphisme (2/2)

```
class Cirque {  
    Felin f_list[] = new Felin[3];  
  
    Cirque() {  
        f_list[0] = new Lion();  
        f_list[1] = new Chat();  
        f_list[2] = new Tigre();  
    }  
  
    void appeler() {  
        for (int i = 0; i<3 ; i++) {  
            Felin f = f_list[i];  
            f.parler();  
        }  
    }  
}
```





Super

- Le mot-clé « super » permet d'accéder aux méthodes et aux attributs de la super-classe
- « super » est utilisé dans le constructeur de la classe dérivée pour appeler celui de la super-classe
 - Cela permet de factoriser du code
 - « super(...) » doit être la première instruction du constructeur

```
class Felin {  
    int _nbPattes;  
    Felin(int nbpattes) { _nbPattes = nbPattes; } }  
  
class Chat extends Felin {  
    Chat() {  
        super(4);  
        race = "goutière"; } }
```

Méthodes et classes abstraites



- Une méthode abstraite est une méthode dont on donne la signature sans en décrire l'implémentation
 - Le mot-clé « abstract » permet d'indiquer qu'une méthode doit être redéfinie dans une sous-classe
- Une classe abstraite ne peut pas être instanciée
 - Le mot-clé « abstract » permet de bloquer l'instanciation
 - Une classe possédant une méthode abstraite est abstraite

```
abstract class Felin {  
    abstract void parler() {...}  
}  
  
class Chat extends Felin {  
    void parler() { System.out.println("miaou ! ") } }  
  
Felin f = new Felin();           // Ne compile pas
```



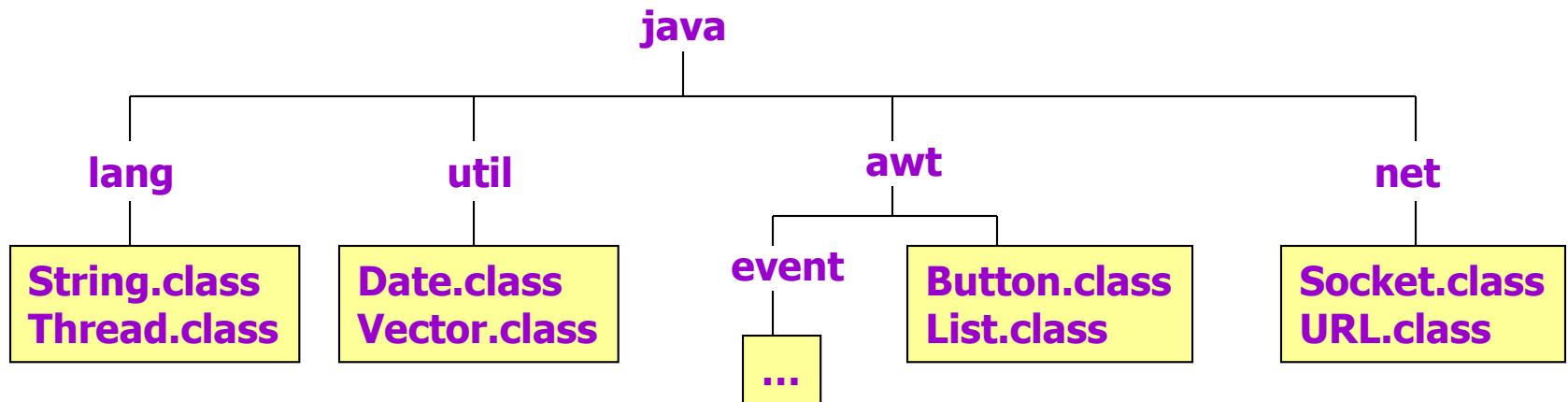
Les packages : définition (1/2)

- Un package est une bibliothèque de classes
- On regroupe les classes d'un même domaine dans un package
- Les packages sont organisés hiérarchiquement
- La notion de package apporte un niveau d'encapsulation supplémentaire

Les packages : définition (2/2)



- Les classes du JDK sont classées dans des packages



- Java importe automatiquement le package « java.lang » qui contient des classes comme « Thread » ou « System »

Les packages : utilisation (1/2)



- Il y a 2 manières d'utiliser une classe stockée dans un package :
 - Utilisation du nom complet

```
java.util.Date dateDuJour = new java.util.Date();  
System.out.println(dateDuJour);
```

- Importer une classe ou toutes les classes du package

```
import java.util.Date;  
  
Date dateDuJour = new Date();  
System.out.println(dateDuJour);
```

```
import java.util.*;  
  
Date dateDuJour = new Date();  
System.out.println(dateDuJour);
```

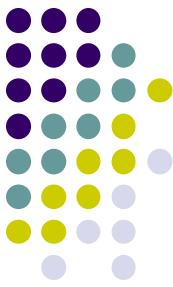


Les packages : utilisation (2/2)

- Le mot-clé « package » permet de définir un nouveau package
- La hiérarchie des packages correspond à la hiérarchie des répertoires

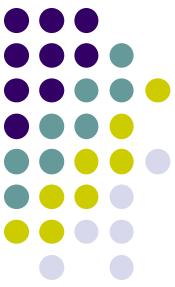
```
//fichier Compte.java dans le répertoire Finance
package finance;
public class Compte {
    .....

//Fichier Client.java dans le répertoire de l'application
import finance.*;
public class Client {
    Compte c = new Compte();
    .....
```



La variable CLASSPATH (1/2)

- Le compilateur utilise la variable d'environnement CLASSPATH pour localiser les classes d'un package sur le disque
- Cette variable doit référencer tous les répertoires ou fichiers dans lesquels sont susceptibles de se trouver des classes Java
- Une classe `Watch` appartenant au package `time.clock` doit se trouver dans le fichier `time/clock/Watch.class`
- On a le droit de placer les classes dans des archives (zip, jar, cab)
 - Dans ce cas, la variable CLASSPATH doit référencer le fichier
- La hiérarchie des classes des packages doit être respectée



La variable CLASSPATH (2/2)

CLASSPATH = \$JAVA_HOME/lib/classes.zip;\$HOME/classes

```
~/classes/graph/2D/Circle.java
package graph.2D;
public class Circle()
{ ... }
```

```
~/classes/graph/3D/Sphere.java
package graph.3D;
public class Sphere()
{ ... }
```

```
~/classes/paintShop/MainClass.java
package paintShop;

import graph.2D.*;
public class MainClass()
{
    public static void main(String[] args) {
        graph.2D.Circle c1 = new graph.2D.Circle(50)
        Circle c2 = new Circle(70);
        graph.3D.Sphere s1 = new graph.3D.Sphere(100);
        Sphere s2 = new Sphere(40); // error: class paintShop.Sphere not found
    }
}
```



Classes publiques

- Le mot-clé « public » permet de définir une classe comme publique
- Seules les classes « public » sont accessibles depuis l'extérieur du package
- Chaque fichier java doit contenir au maximum une classe « public »
 - Cette classe doit porter le nom du fichier dans lequel elle est définie (en respectant les majuscules/minuscules)
- Les classes « non public » ne sont utilisables qu'à l'intérieur du fichier dans lequel elles sont définies

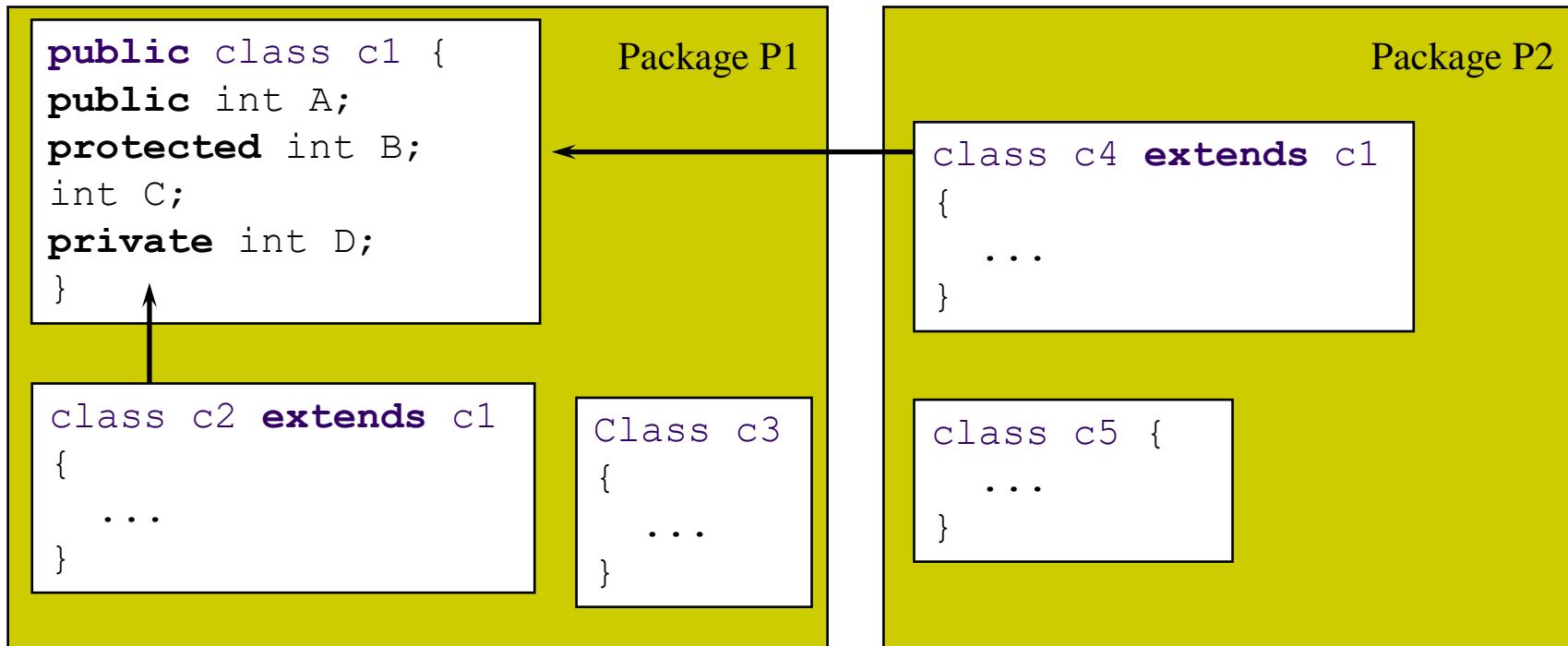
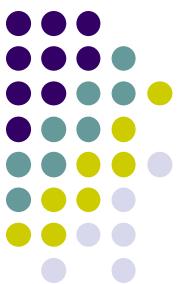


Visibilité - Encapsulation



- Permet de distinguer les services offerts (interface) de l'implémentation
- L'encapsulation des classes Java est définie au niveau du package
- L'encapsulation agit au niveau des classes et non des objets
 - Un attribut « private » dans un objet sera accessible depuis un autre objet de la même classe
- Il y a 4 niveaux de visibilité : public, private, protected, friendly (valeur par défaut)

Règles de visibilité



	A	B	C	D
Accessible par c2	o	o	o	-
Accessible par c3	o	o	o	-
Accessible par c4	o	o	-	-
Accessible par c5	o	-	-	-



Règles de visibilité

- Pour une bonne encapsulation, il est préférable de définir les attributs comme « private »
- On définit alors des méthodes « publiques » (accesseurs) permettant de lire et/ou de modifier les attributs
 - Si un accesseur retourne une référence sur un objet, rien n'empêche la modification de cet objet à travers cette référence → cela brise l'encapsulation

```
class Personne {  
    private Vector children = new Vector();  
    public Vector getChildren() { return children; } }  
.....  
Personne moi = new Personne();  
Vector v = moi.getChildren();  
v.addElement(new Personne("Paul"));
```

Rupture de
l'encapsulation



Encapsulation des constantes

```
class EtatPorte {          // Classe non publique
    public final static EtatPorte OUVERTE = new EtatPorte();
    public final static EtatPorte FERME = new EtatPorte();

    // Empeche la creation d'un nouvel objet
    private EtatPorte() { }
}

public class Porte {          // Classe publique
    private EtatPorte etat = EtatPorte.FERMEE;

    public void ouvrir() {
        etat = EtatPorte.OUVERTE;
    }

    public estOuverte() {
        return (etat == EtatPorte.OUVERTE);
    }
}
```

Les interfaces





Définition

- Une interface définit un ou plusieurs services offerts
- Elle est composée d'un ensemble de méthodes abstraites et de constantes (« static » et « final »)
- Une classe peut implémenter une ou plusieurs interfaces
 - Elle doit fournir une implémentation pour chaque méthode

```
interface Printable {
    void print();
}

class Point extends Object implements Printable {
    private double x, y;
    void print() {
        System.out.println(x);
        System.out.println(y);
    }
}
```



Héritage multiple d'interface

- Une interface peut hériter d'une ou plusieurs autres interfaces
 - Java supporte l'héritage multiple d'interface
- Une classe peut implémenter plusieurs interfaces

```
interface Printable {
    void print();
}
interface Persistent {
    void save();
}
interface SGBD extends Persistent {
    void connect();
}

class Point implements Printable , SGBD {
    private double x, y;
    void print() {.....};
    void save() {.....};
    void connect() {.....};
}
```



Interfaces et types

- Une interface définit un nouveau type
- Des objets différents peuvent répondre au même message à condition qu'ils implémentent la même interface
- L'opérateur « instanceof » peut être utilisé pour savoir si un objet implémente une interface donnée

```
Point point = new Point();

if (point instanceof Printable) {
    point.print();
    .....
}
```



Variables d'interfaces

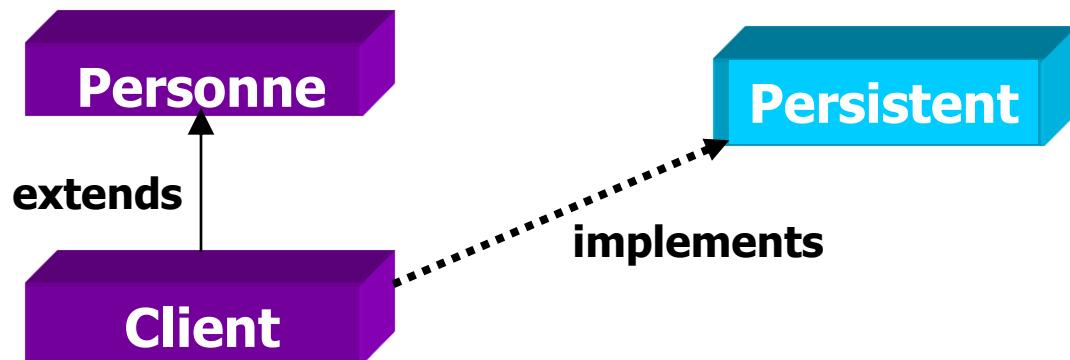
- Les variables déclarées dans une interface sont des constantes
 - Les mots clés « static » et « final » ne sont pas nécessaires
- Les variables des interfaces doivent obligatoirement être initialisées

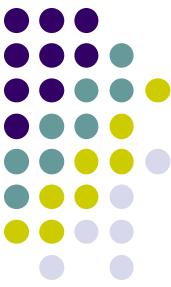
```
interface MyInterface {  
    int MIN = 0;  
    int MAX = 100;  
    ..... }  
  
int i = MyInterface.MAX;
```



Quand utiliser les interfaces ?

- Pour définir des services techniques
- Pour contourner le fait que Java ne supporte pas l'héritage multiple
- Interface vs Héritage :
 - On utilise l'héritage quand un objet est un sous-type d'un autre
 - On utilise une interface pour décrire le fait qu'une classe implémente un service particulier





Les *inner classes* (1/6)

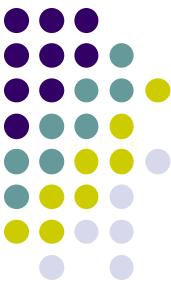
- Introduites avec java 1.1
- Elles permettent de
 - Déclarer une classe dans une bloc (*inner class*)
 - Instancier une classes anonymes (*anonymous class*)
- Elles affinent la localisation des classes
- Elles simplifient le développement
- Elles offrent une (autre) réponse pour les *pointeurs de fonctions*
- Elles sont une caractéristique du compilateur et non de la JVM
- **Attention : elles peuvent réduire la lisibilité des sources.**



Les *inner classes* (2/6)

- Exemple :

```
public class FixedStack {  
    Object array[];  
    int top = 0;  
  
    public void push(Object item) { ... }  
    public Object pop() { ... }  
    public isEmpty() { ... }  
    public java.util.Enumeration element() { return new Enumerator(); }  
  
class Enumerator implements java.util.Enumeration {  
    int count = top;  
    public boolean hasMoreElements() { return count > 0; }  
    public Object nextElement() {  
        if (count == 0) throw  
NoSuchElementException("FixedStack");  
        return array[--count];  
    }  
}
```



Les *inner classes* (3/6)

- Ce qui est produit pas le compilateur :

```
public class FixedStack {  
    ...  
    public java.util.Enumeration element() {return new  
        FixedStack$Enumerator(this);}  
}  
  
class FixedStack$Enumerator implements java.util.Enumeration {  
    private FixedStack this$0;  
  
    FixedStack$Enumerator(FixedStack this$0) {  
        this.this$0 = this$0;  
        this.count = this$0.top;  
    }  
    int count;  
    public boolean hasMoreElements() { return count > 0; }  
    public Object nextElement() {  
        if (count == 0) throw NoSuchElementException("FixedStack");  
        return this$0.array[--count];  
    }  
}
```

Les *inner classes* (4/6)



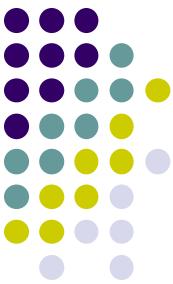
- Exemple de classe locale :

...

```
Enumeration myEnumeration(final Object array[])
{
    class E implements java.util.Enumeration
    {
        int count = top;
        public boolean hasMoreElements() { return count > 0; }
        public Object nextElement() {
            if (count == 0) throw
NoSuchElementException("FixedStack");
            return array[--count];
        }
    }

    return new E();
}
```

Les *inner classes* (5/6)



- Exemple de classe anonyme :

...

```
Enumeration myEnumeration(final Object array[]) {  
  
    return new java.util.Enumeration() {  
        int count = 0;  
        public boolean hasMoreElements() { return count <  
array.length; }  
        public Object nextElement() {  
            return array[count++];  
        }  
    }  
}
```



Les *inner classes* (6/6)

- Ce qui est produit pas le compilateur :

```
Enumeration myEnumeration(final Object array[]) {  
  
    return new MyOuterClass$19(array);  
}  
  
...  
  
class MyOuterClass$19 implements java.util.Enumeration {  
    private Object val$array;  
    int count;  
    MyOuterClass$19(Object val$array) {  
        this.val$array = val$array;  
        count = 0;  
    }  
    public boolean hasMoreElements() { return count <  
        val$array.length; }  
    public Object nextElement() {  
        return val$array[count++];  
    }  
}
```

Les classes de base



La classe « Object » : définition



- Classe mère de toutes les classes
 - Permet d'utiliser les classes conteneurs et le polymorphisme
- La méthode « *toString()* » retourne une description de l'objet
- La méthode « *equals(Object)* » teste l'égalité sémantique de deux objets
- Le méthode « *getClass()* » retourne le descripteur de classe. Il permet de :
 - connaître la classe d'un objet
 - connaître le nom de la classe, ses ancêtres
 - découvrir la structure des objets (JDK 1.1)



La classe « Object » : exemple

```
class NamedObject extends Object {  
    protected String _nom;  
    public String toString() {  
        return "Objet : " + _nom + " de la classe " +  
               getClass() .getName();  
    }  
  
    public boolean equals (NamedObject obj) {  
        return obj._nom.equals(this._nom);  
    }  
}  
class Personne extends NamedObject {.....}  
  
Personne moi = new Personne("Marcel Dupond");  
Personne lui = new Personne("Marcel Dupond");  
  
System.out.println(moi);  
  
if (moi == lui) {.....}  
if (moi.equals(lui)) {.....}
```

Objet Marcel Dupond
de la classe Personne

Test des références = false

Test des valeurs = true



Destruction d'un objet

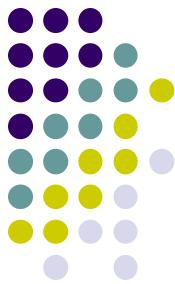
- Tout objet qui n'est plus référencé est détruit par le ramasse-miettes
 - Il n'y a pas de “delete” en java
- Si l'objet possède la méthode ***finalize()***, celle-ci est appelé avant que le ramasse-miettes ne libère la mémoire occupée par l'objet
 - Permet de libérer des ressources allouées par l'objet
- On peut forcer l'appel au ramasse-miettes en invoquant
 - *System.gc()* : déclencher le ramasse-miettes
 - *System.runFinalisation()* : déclencher la finalisation des objets
- Vous pouvez lancer la machine virtuelle en bloquant le déclenchement automatique du ramasse-miettes
 - Option : *java -nosyncgc*
 - Permet de contrôler le déclenchement du ramasse-miettes

Les classes « Wrapper » : description

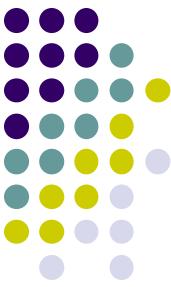


- Les types de base Java ne sont pas des objets
 - Il est parfois indispensable de les manipuler comme des objets
- Les classes Wrapper représentent un type de base
 - Récupérer les valeurs min et max
 - Créer une instance à partir d'un type de base ou d'une chaîne
 - Conversions entre types de base et chaînes
 - Conversions entre chaînes et types de base
 - Utiliser des types de base dans des conteneurs
- Boolean, Integer, Float, Double, Long, Character
- Attention : ne supporte pas d'opérateurs (+, -, ...)
- Elles sont dans le package *java.lang*

« Wrapper » vs « types de base »



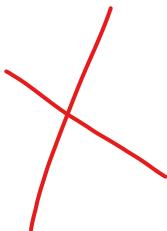
- Avantages des type de bases :
 - Plus léger en mémoire
 - Supporte un certain nombre d'opérateurs
 - Autorise certaines conversions automatiques de types
- Avantages des Wrappers :
 - Est passé par référence
 - Dispose de méthodes de conversions statiques vers les autres types
 - Peut être géré par les objets container



Exemples

- Convertir une chaîne en entier :

```
static int convertir(String s) {  
    try {  
        return Integer.parseInt(s);  
    } catch(Exception e) { return 0; }  
}
```



- Convertir un entier en chaîne :

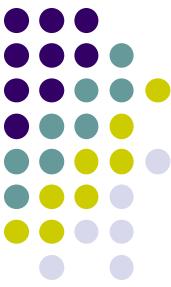
```
static String convertir(int i) {  
    Integer j = new Integer(i);  
    return j.toString();  
}
```



La classe `java.lang.String`

- Contient une chaîne de caractères
- Classe connue par le compilateur Java
 - Une chaîne constante sera convertie en objet de la classe String
 - `String msg = "Hello Java World !"`
- Un objet de la classe String ne peut pas être modifié
 - Ne brise pas l'encapsulation

```
class Personne {  
    String nom;  
  
    public String getNom() {  
        return nom;  
    }  
}
```



String : concaténation

```
int area = 33;
int prefixe = 1;
int suffixe = 02030405;

// Chaque concaténation crée un nouvel objet
// l'ancien est détruit par le Garbage Collector

String numTel = "(" + area + ")";      // (33)
numTel += prefixe                      // (33)1
numTel += "-"                           // (33)1-
numTel += suffixe                      // (33)1-02030405

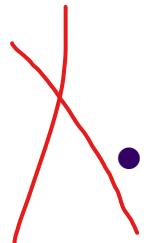
// Un seul objet est créé
numtel = "(" + area + ")" + prefixe + "-" + suffixe

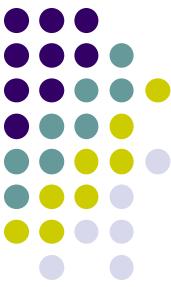
// Conversion d'un entier en chaîne
String un = 1 + "";
```



String : manipulation

- **length()** : cette méthode permet de connaître le nombre de caractères d'une chaîne
 - Ex : `for (int i = 0; i < monTexte.length(); i++) {.....}`
- **indexOf(char c, int i)** : retourne la position du caractère “c” à partir de la position “i”
 - `if (monTexte.indexOf('$', 5) <> -1) {.....}`
- **substring(int i, int j)** : extrait de la chaîne les caractères de la position “i” à la position “j”
 - `String nouveauMot = ancienMot.substring(2,5);`
- **equals()** : retourne un booléan signifiant si les deux chaînes sont identiques
 - Ex : `if (mot1.equals(mot2)) {.....}`





La classe java.lang.StringBuffer

- C'est une chaîne de caractères modifiable et de taille variable
 - La classe StringBuffer gère des chaînes de caractères (char) modifiable (setCharAt(), append(), insert())
 - Son utilisation est moins simple que « String »
 - pas d'utilisation possible de l'opérateur +
 - **append(p)** : ajoute "p" en fin de la chaîne courante
 - "p" peut être de n'importe quel type de base
 - **length()** : retourne la longueur de la chaîne
 - **setLength()** : change la longueur de la chaîne
 - Si elle est étendue, les nouveaux caractères sont initialisés à 0
 - La méthode **toString()** convertie une StringBuffer en String (pas de recopie, le même tableau est partagé, jusqu'à modification)
- X

```
StringBuffer sb = "abc";    // Error: can't convert String to StringBuffer
StringBuffer sb = new StringBuffer("abc");

sb.setCharAt(1, 'B');           // sb= "aBc"
sb.insert(1, "1234");          // sb = "a1234Bc"
sb.append("defg");             // sb = "a1234Bcdefg"

String s = sb.toString();       // s = "a1234Bcdefg"
sb.append("hij");              // sb = "a1234Bcdefghij" s = "a1234Bcdefg"133
```

La classe java.util.Scanner



- Avant le JDK 5.0 (ou 1.5), il n'existait aucune méthode commode pour lire des entrées depuis la fenêtre de la console.
- Heureusement, cette situation vient d'être rectifiée.
- La lecture d'une entrée au clavier s'effectue en construisant un Scanner attaché sur l'unité "entrée standard - System.in" :

```
Scanner clavier = new Scanner(System.in);
System.out.print("Quel est ton nom ? ");
String nom = clavier.nextLine();
System.out.print("Quel est ton prénom ? ");
String prénom = clavier.next();
System.out.print("Quel est ton âge ? ");
int âge = clavier.nextInt();
```

- Penser à importer le paquetage `java.util.*` ; ou `java.util.Scanner` ;

La classe java.util.Scanner



Méthodes

`Scanner(InputStream in)`

Caractéristiques

Construit un objet Scanner à partir du flux de saisie.
Peut-être utilisé pour d'autres type de flux et peut donc remplaceravantageusement BufferedReader.

`String nextLine()`

Lit la prochaine ligne saisie.

`String next()`

Lit le prochain mot saisi (délimité par un espace).

`int nextInt()`

Lit et transforme la prochaine ligne de caractères qui représente un entier.

`double nextDouble()`

Lit et transforme la prochaine ligne de caractères qui représente un nombre à virgule flottante.

`boolean hasNext()`

Teste s'il y a un autre mot dans la saisie.

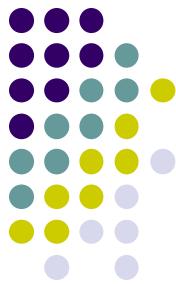
`boolean hasNextInt()`

Teste si la prochaine suite de catactères représente un entier.

`boolean hasNextDouble()`

Teste si la prochaine suite de catactères représente un nombre à virgule flottante.

La classe java.util.Scanner

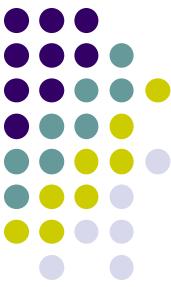


```
package clavier;
import java.util.*;
import static java.lang.System.*;
public class Clavier {
    public static void main(String[] args)
{
    Scanner saisie = new Scanner(in);
    out.print("Entier : ");
    int entier = saisie.nextInt();
    out.println("Réel : ");
    double réel = saisie.nextDouble();
    out.println("Entier = "+entier);
    out.println("Réel = "+réel);
}
}
```



La classe `java.util.Vector`

- Tableau de références à taille variable
- On ne peut y stocker que des références sur les objets
 - Impossible de stocker directement un type de base dans une collection de type Vector : utiliser pour cela les classes wrapper
 - Souvent utilisé pour coder une relation 1-n entre deux classes
- Possibilité de savoir si un objet est présent dans le tableau et quelle est sa position
- Vous pouvez :
 - Insérer ou supprimer des références
 - Parcourir le contenu



Vector : insérer, supprimer

```
Vector vInt = new Vector();

for (int i = 0; i<10 ; i++) {
    Integer elt = new Integer(i);
    vInt.addElement(elt);
}

// 0123456789

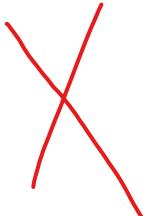
Integer i = new Integer(0);
vInt.addElementAt(i, 5);

// 01234056789

vInt.removeElementAt(0);

// 1234056789

vInt.removeAllElements();
```

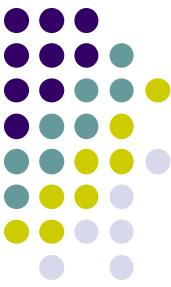




Vector : modifier, rechercher ...



- **Object elementAt(int index)**
 - Retourne l'élément pointé par index
- **Void setElementAt(Object obj, int index)**
 - Place obj à la position index
- **Boolean contains(Object elem)**
 - Retourne true si elem est dans le tableau
- **Int indexOf(Object elem)**
- **Int indexOf(Object elem, int index)**
 - Retourne la position de l'objet ou -1 si l'objet n'est pas dans le tableau
 - L'objet est recherché par référence
- **Int size()**
 - Retourne la taille courante



L'interface java.util.Enumeration

- Objet permettant de parcourir les éléments d'un conteneur
 - Une énumération ne stocke aucune information
 - Ne contient qu'une position courante dans le conteneur
 - Interface unique pour le parcours de tous les types de conteneurs
 - Ne fonctionne que pour un seul parcours
 - C'est une interface, pas une classe
- ~~Elements()~~ appliquée sur un Vector retourne une Enumeration
- Deux méthodes :
 - Boolean **hasMoreElements()** : teste s'il reste des éléments
 - Object **nextElement()** : récupère l'élément courant et passe au suivant



Enumeration : exemple

```
Vector vInt = new Vector();
for (int i = 0; i<10 ; i++) {
    Integer elt = new Integer(i);
    vInt.addElement(elt);
}
// 0123456789

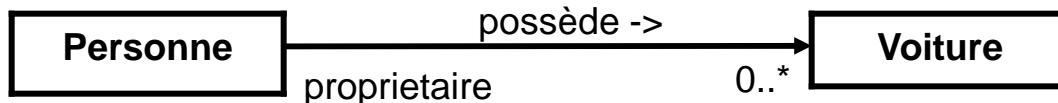
// Première façon de parcourir le vector
for (int i = 0; i<vInt.size() ; i++)
    System.out.println(vInt.elementAt(i));

// Seconde façon de parcourir le vector
// la plus élégante
for (Enumeration e = vInt.elements(); e.hasMoreElements();)
    System.out.println(e.nextElement());
```

Quand utiliser une Enumeration ?

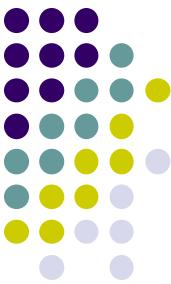


- Utilisez une Enumeration pour parcourir une association 1..*



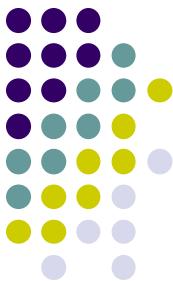
- Permet de ne pas briser l'encapsulation
- Dans l'exemple suivant :
 - la classe « Personne » possède une méthode publique pour parcourir la collection de voitures
 - L'ajout ou la suppression d'une voiture dans cette collection doit se faire en manipulant une instance de la classe « Voiture »

Quand utiliser une Enumeration ?



```
public class Personne {  
    private Vector _voitures = new Vector();  
  
    public Enumeration voitures() { return _voitures.elements(); }  
    void addVoiture(Voiture v) { _voitures.addElement(v); }  
    void remVoiture(Voiture v) { _voitures.removeElement(v); }  
}  
.....  
package simulateur;  
class Voiture {  
    private Personne _proprio;  
  
    public void setProprietaire (Personne p) {  
        if (_proprio != null) _proprio.remVoiture(this);  
        _proprio = p; .  
        if (_proprio != null) _proprio.addVoiture(this);  
    }  
  
    public Personne getProprietaire() { return _proprio; }  
}
```

La boucle for améliorée



- Le JDK 5.0 a introduit une construction de boucle performante qui vous permet de parcourir chaque élément d'un tableau (ainsi que d'autres collections d'éléments) sans avoir à vous préoccuper des valeurs d'indice.

for (variable : collection) instruction

```
X
package testclavier;
import static java.lang.System.*;
import java.util.Scanner;
public class Main {
    public static void main(String[] args) {
        Scanner clavier = new Scanner(in);
        out.print("Nombre de cases de votre tableau : ");
        int nombre = clavier.nextInt();
        int[] tableau = new int[nombre];
        for (int indice=0; indice<nombre; indice++) {
            out.print("tableau["+indice+"] = ");
            tableau[indice] = clavier.nextInt();
        }
        out.println("Récapitulatif de votre saisie :");
        for (int élément : tableau) out.println(élément);
    }
}
```



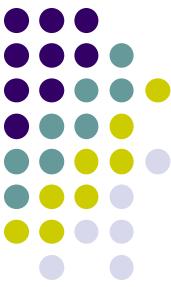
La classe `java.util.Hashtable`

- Tableau associatif permettant :
 - D'indexer un objet par une clé
 - De retrouver rapidement un objet à partir d'une clé
- Les clés des références sont des objets
 - Chaînes de caractères, numéros d'identification ...
 - Elles sont recherchées par valeurs (méthodes `equals`)
 - Les clés doivent implémenter la méthode `Object.hashCode()`
- Une clé ne peut identifier qu'une seule valeur
 - N'accepte pas les doublons
 - Pour stocker des relations n-n, il suffit de stocker des instances de `Vector` dans une `Hashtable`
- Très performant pour l'accès aux valeurs



La classe `java.util.Hashtable`

- `Object put(Object key, Object value)` : insère un élément
- `Object remove(Object key)` : supprime la clé
- `Object get(Object key)` : retourne la valeur associée à la clé
- `Boolean containsKey(Object key)` : teste si la clé existe
- `Keys()` : retourne une Enumeration sur les clés
- `Elements()` : retourne une Enumeration sur les valeurs



Hashtable : un exemple

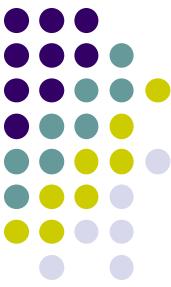
```
Hashtable deptable = new Hashtable();

// Stockage d'objets de type Departement en utilisant
// une clé de type String
deptable.put("Ain", new Departement(1));
deptable.put("Aisne", new Departement(2));
deptable.put("Allier", new Departement(3));

Departement dep = (Departement)deptable.get("Aisne");

// Parcours de toutes les clés de la hashtable
for (Enumeration noms = deptable.keys();
     noms.hasMoreElements();) {
    String nom = (String)noms.nextElement();
    Departement dep2 = (Departement)deptable.get(nom);
    System.out.println(nom + " = " + dep2);
}
```





Autoboxing / unboxing

- jusqu'à la version 1.4 de Java, pour ajouter des entiers dans un conteneur (, il est nécessaire d'encapsuler chaque valeur dans un objet de type Integer).

Exemple :

```
import java.util.*;
public class TestAutoboxingOld {
    public static void main(String[] args) {
        Vector vecteur = new Vector();
        Integer valeur = null;
        for(int i = 0; i < 10; i++) {
            valeur = new Integer(i);
            vecteur.addElement(valeur);
        }
    }
}
```



Autoboxing / unboxing



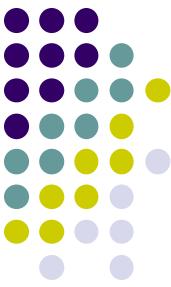
- L'autoboxing permet de transformer automatiquement une variable de type primitif en un objet du type du wrapper correspondant.
L'unboxing est l'opération inverse.

Exemple (java 1.5) :

```
import java.util.*;
public class TestAutoboxing {
    public static void main(String[] args) {
        Vector vecteur = new Vector();
        for(int i = 0; i < 10; i++) {
            vecteur.addElement(i);
        }
    }
}
```

Les exceptions





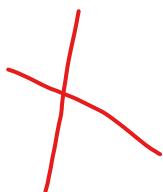
A quoi servent les exceptions ?

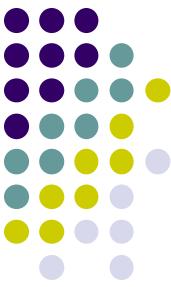
- Le mécanisme des exceptions en Java permet de traiter les erreurs d'exécution
- Il permet aussi de gérer toutes les situations "hors contrat" définies par le concepteur
- Le travail du concepteur :
 - Définir les ensembles d'instructions à risque
 - Implémenter les gestionnaires d'exceptions pour chaque cas
- Le travail de la machine virtuelle Java :
 - Distinguer un résultat valide d'un code erreur
 - Propager jusqu'au gestionnaire de ce cas exceptionnel les informations qui lui sont nécessaires (code erreur, données de contexte)



Les exceptions

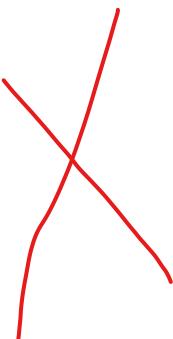
- Ce sont des instances de classes dérivant de `java.lang.Exception`
- La levée d'une exception provoque une remontée dans l'appel des méthodes jusqu'à ce qu'un bloc `catch` acceptant cette exception soit trouvé. Si aucun bloc `catch` n'est trouvé, l'exception est capturée par l'interpréteur et le programme s'arrête.
- L'appel à une méthode pouvant lever une exception doit :
 - soit être contenu dans un bloc `try/catch`
 - soit être situé dans une méthode propageant (`throws`) cette classe d'exception
- Un bloc (optionnel) `finally` peut-être posé à la suite des `catch`. Son contenu est exécuté après un `catch` ou après un `break`, un `continue` ou un `return` dans le bloc `try`

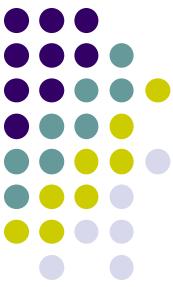




Exemple (1/2)

```
class Equation {  
    double _a;  
    double _b;  
    double c;  
  
    // Constructeur  
    Equation(double a, double b, double c) {  
        _a = a;  
        _b = b;  
        _c = c; }  
  
    // Calcul du delta  
    public double delta() { return _b*_b - 4*_a*_c; }  
  
    // Solution  
    public double solution() throws PasDeSolution {  
        double discr = delta();  
        if (discr<0) throw new PasDeSolution();  
        return (_b + Math.sqrt(discr))/(2*_a); }  
}
```

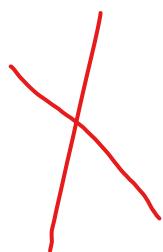




Exemple (2/2)

```
class PasDeSolution extends Exception {  
  
    // Retourne la description du problème  
    String toString() {  
        return "L'équation n'a pas de solution";  
    }  
}  
  
... ...  
// Méthode appellante  
.....  
try {  
    Equation eq = new Equation(1,0,1);      // x2 + 1 = 0  
    double resultat = eq.solution();  
    .....  
}  
catch(PasdeSolution p) {  
    System.out.println(p.toString());  
}
```

*Branchement
vers le catch*





X Throw et throws

- Le mot-clé “**Throw**” permet de soulever une exception

```
if (discr<0) throw new PasDeSolution();  
.....
```

- Le mot-clé “**Throws**” s’utilise au niveau de la signature d’une méthode pour préciser que celle-ci est susceptible de soulever une exception

```
public double solution() throws PasDeSolution {  
.....
```



Try - Catch et Throws

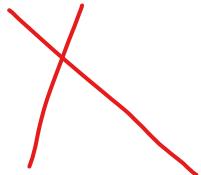
- Lorsque que l'on utilise une méthode susceptible de soulever une exception, il faut spécifier comment on la gère.
- 2 Solutions :
 - Traiter le cas dans la méthode : Try - Catch

```
try {  
    double resultat = eq.solution();  
} catch(PasdeSolution p) {  
    System.out.println(p.toString()); }
```

- Faire “remonter” l'exception au niveau supérieur : Throws

```
void calculer() throws PasDeSolution {  
    .....  
    double resultat = eq.solution();  
    .....  
}
```

Finally



- Le mot-clé “**Finally**” permet de déclarer un bloc d'instruction qui sera exécuté dans tous les cas, qu'une exception survienne ou non.

```
try {  
    // Ouvrir le fichier  
    // Lire les données  
    // .....  
} catch (IOException i) {  
    System.out.println("Erreur I/O sur le fichier");  
    return;  
} finally {  
    // Fermer le fichier  
    // .....  
}
```

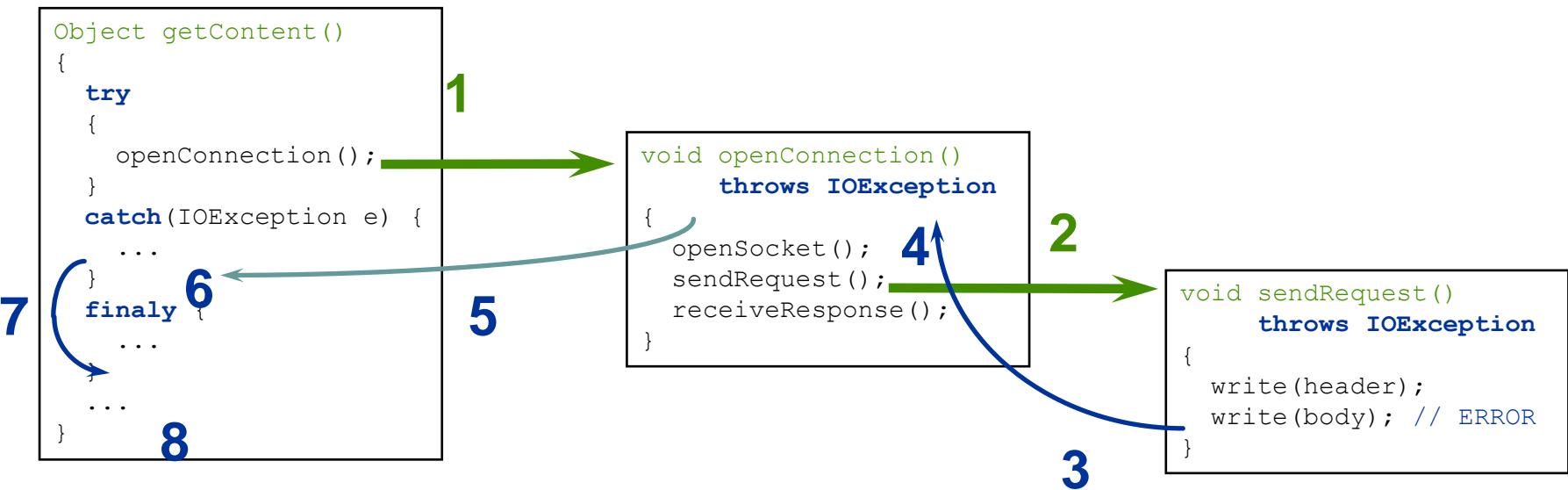
1 Erreur ici !

2 Retour aux instructions suivant la lecture

3 Bloc d'instructions toujours traité



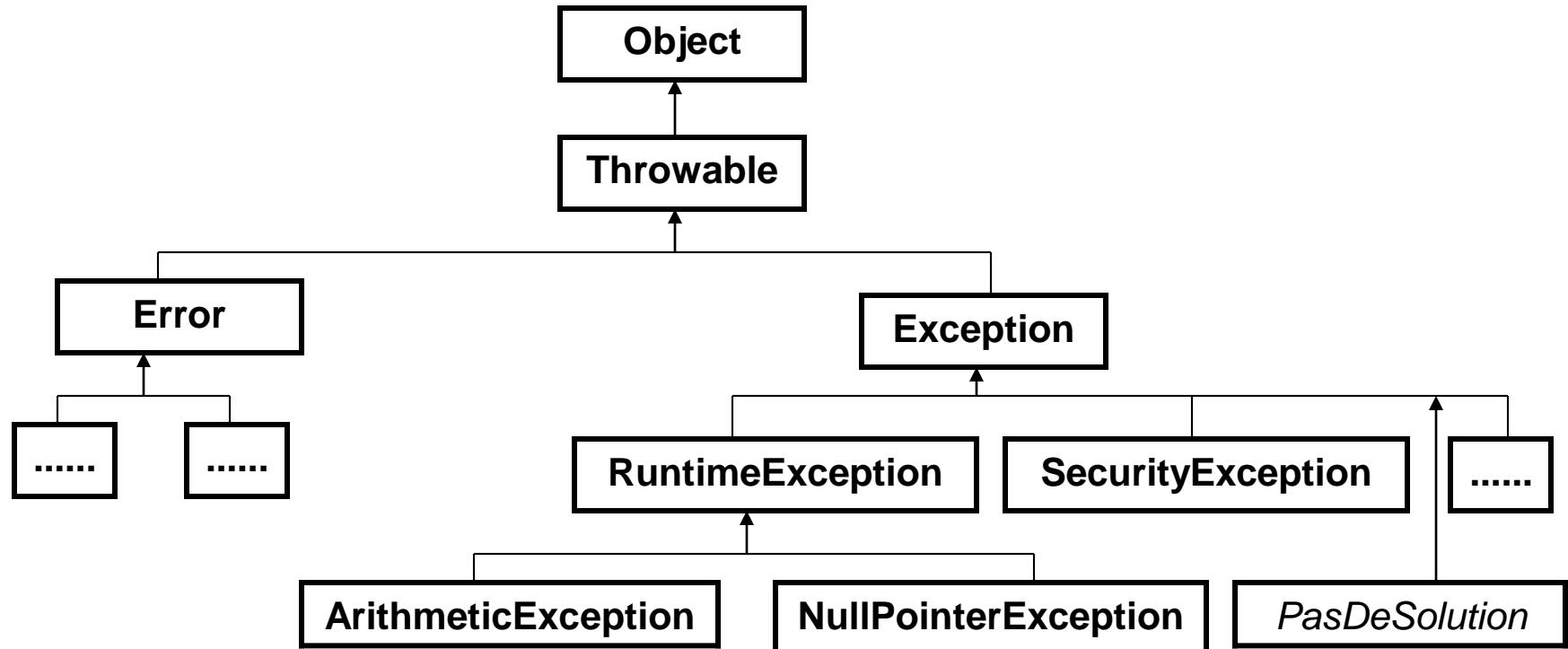
Les exceptions récapitulatif



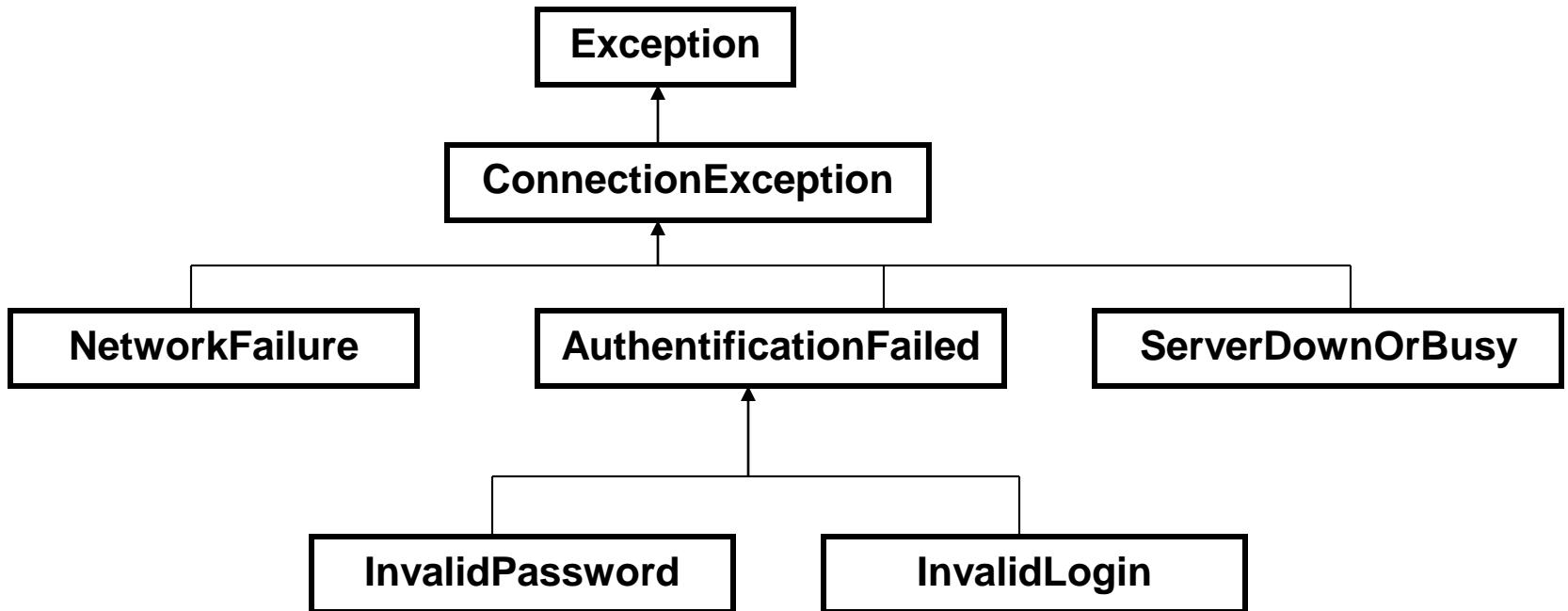
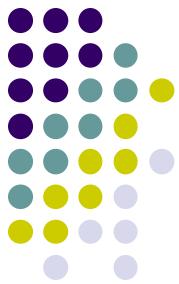


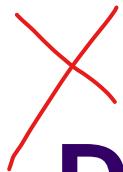
Classification des exceptions

- Le JDK contient des classes d'exceptions dont voici un extrait :



Bâtir une hiérarchie d'exceptions métier





Debugging : les traces

- Une exception dérive de la classe ***Object***. Utilisez la méthode ***toString()*** pour garder des traces d'exécution
- La méthode ***PrintStackTrace()*** permet d'afficher la pile des appels depuis la méthode qui a déclenché l'exception
- La classe ***Throwable*** prend une String à sa construction. Cela permet d'enrichir les traces avec des messages spécifiques, récupérés avec ***getMessage()***

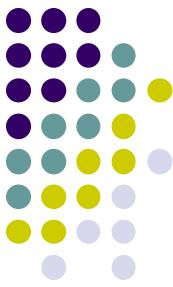
Les entrées-sorties





Les entrées / sorties

- Dans la plupart des langages de programmation les notions d'entrées / sorties sont considérées comme une technique de base, car les manipulations de fichiers, notamment, sont très fréquentes.
- En Java, et pour des raisons de sécurité, on distingue deux cas :
 - le cas des applications Java autonomes, où, comme dans n'importe quel autre langage, il est généralement fait un usage important de fichiers,
 - le cas des applets Java qui, ne peuvent pas, en principe, accéder, tant en écriture qu'en lecture, aux fichiers de la machine sur laquelle s'exécute le navigateur (machine cliente).



La gestion des fichiers

- La gestion de fichiers proprement dite se fait par l'intermédiaire de la classe File.
- Cette classe possède des méthodes qui permettent d'interroger ou d'agir sur le système de gestion de fichiers du système d'exploitation.
- Un objet de la classe File peut représenter un fichier ou un répertoire.

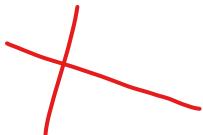


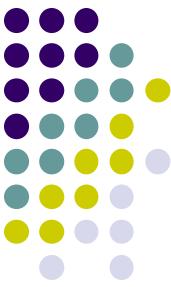
java.io.File

```
File f = new File("/etc/passwd");
System.out.println(f.exists());      // --> true
System.out.println(f.canRead());     // --> true
System.out.println(f.canWrite());    // --> false
System.out.println(f.getLength());   // --> 11345

File d = new File("/etc/");
System.out.println(d.isDirectory()); // --> true

String[] files = d.list();
for(int i=0; i < files.length; i++)
    System.out.println(files[i]);
```





Notion de flux (1)

- Les E / S sont gérées de façon portable (selon les OS) grâce à la notion de flux (*stream* en anglais).
- Un flux est en quelque sorte un canal dans lequel de l'information transite. L'ordre dans lequel l'information y est transmise est respecté.
- Un flux peut être :
 - Soit une source d'octets à partir de laquelle il est possible de lire de l'information. On parle de flux d'entrée.
 - Soit une destination d'octets dans laquelle il est possible d'écrire de l'information. On parle de flux de sortie.



Notion de flux (2)

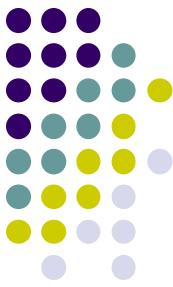
- Certains flux de données peuvent être associés à des ressources qui fournissent ou reçoivent des données comme :
 - les fichiers,
 - les tableaux de données en mémoire,
 - les lignes de communication (connexion réseau)



Notion de flux (3)



- L'intérêt de la notion de flux est qu'elle permet une gestion homogène : **ouverture du flux**, **écriture ou lecture**, puis **fermeture du flux**.
 - quelle que soit la ressource associée au flux de données,
 - quel que soit le flux (entrée ou sortie).
- Certains flux peuvent être associés à des filtres
 - Combinés à des flux d'entrée ou de sortie, ils permettent de traduire les données.



Notion de flux (4)

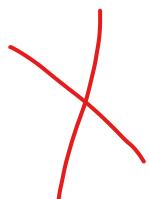
- Les flux sont regroupés dans le paquetage java.io
- Il existe de nombreuses classes représentant les flux
 - il n'est pas toujours aisément de se repérer.
- Certains types de flux agissent sur la façon dont sont traitées les données qui transitent par leur intermédiaire :
 - E / S bufferisées, traduction de données, ...
- Il va donc s'agir de combiner ces différents types de flux pour réaliser la gestion souhaitée pour les E / S.

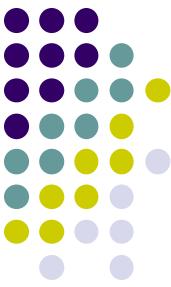


Flux d'octets et flux de caractères



- Il existe des flux de bas niveau et des flux de plus haut niveau (travaillant sur des données plus évoluées que les simples octets). Citons :
 - Les flux d'octets
 - classes abstraites **InputStream** et **OutputStream** et leurs sous-classes concrètes respectives,
 - Les flux de caractères
 - classes abstraites **Reader** et **Writer** et leurs sous-classes concrètes respectives.



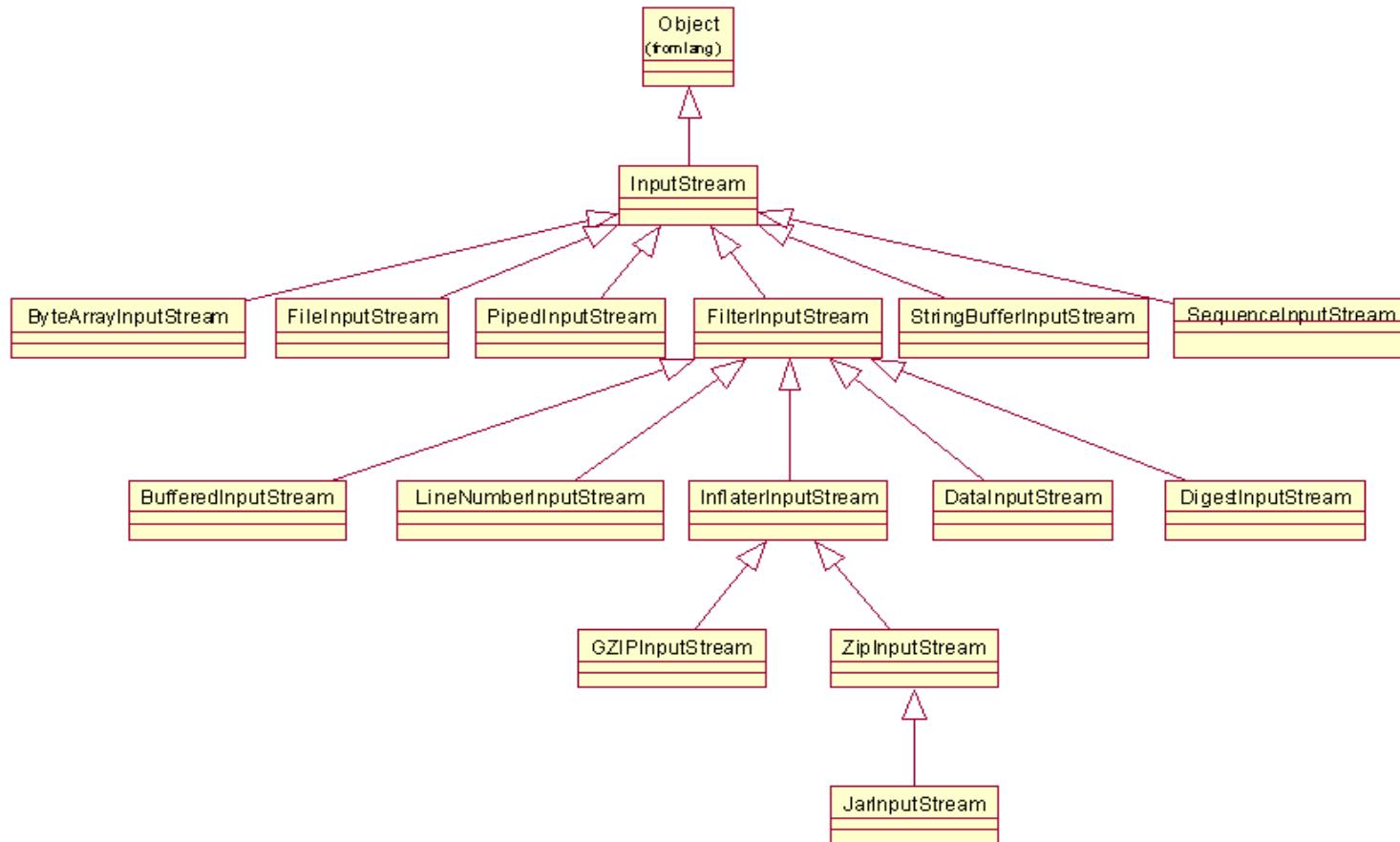


Différents types de flux

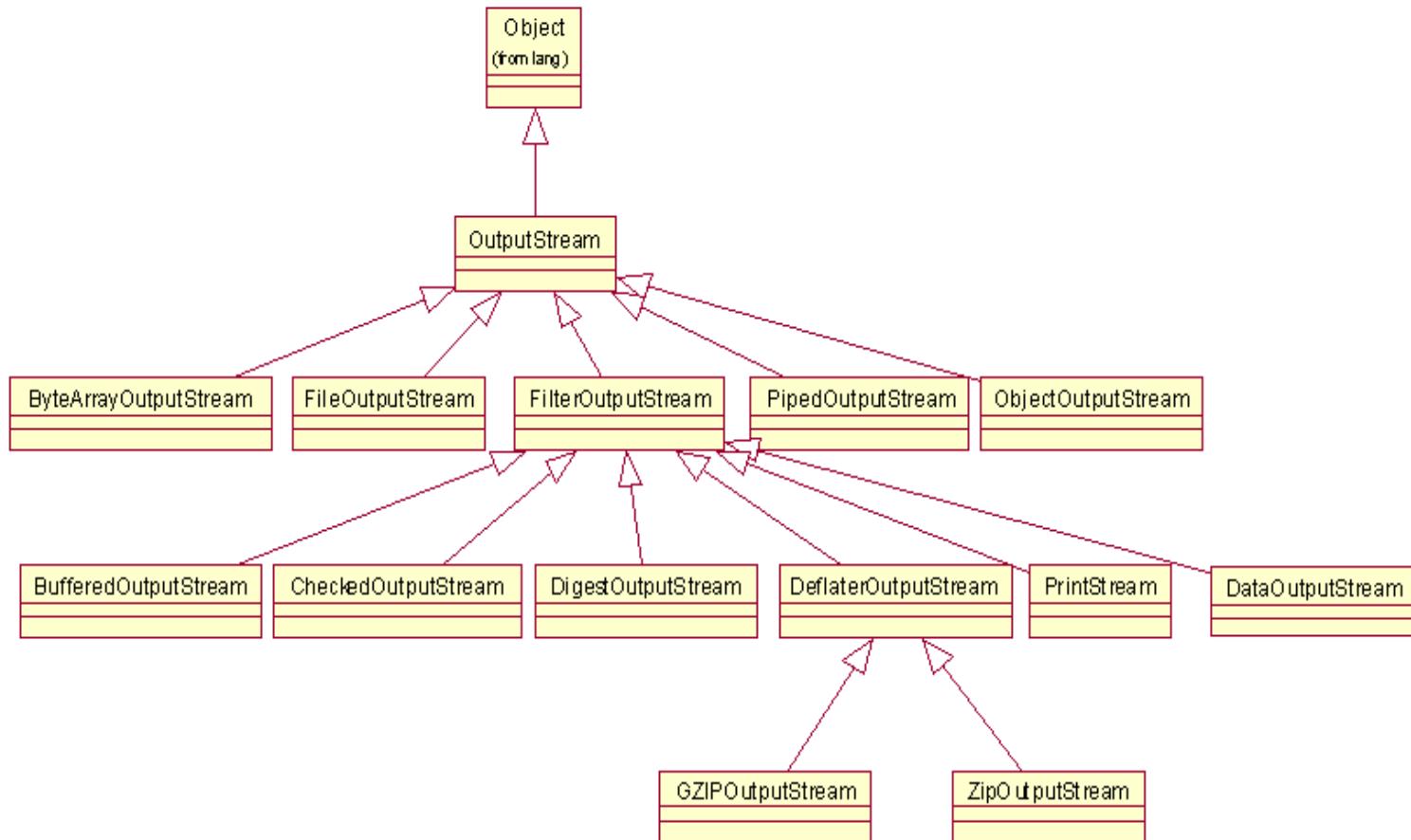
On distingue pour chaque type de flux, des **flux de communication**, qui servent à créer une liaison entre le programme et une autre entité, ainsi que **flux de traitement**, qui servent à traiter les données échangées.



La hiérarchie des flux d'octets en entrée



La hiérarchie des flux d'octets en sortie





Les flux de données prédéfinis

Il existe 3 flux prédéfinis :

- l'entrée standard System.in (instance de InputStream)
- la sortie standard System.out (instance de PrintStream)
- la sortie standard d'erreurs System.err (instance de PrintStream)

```
try {  
    int c;  
    while((c = System.in.read()) != -1) {  
        System.out.print(c);  
    }  
} catch(IOException e) {  
    System.out.print(e);  
}
```



Flux de communication d'octets

- **FileInputStream, FileOutputStream** : permettent de créer un flux avec un fichier en argument (de type **File**) ou un String représentant le chemin vers le fichier.
 - les flux d'écritures de fichier créent automatiquement le fichier si il n'existe pas. Il est possible aussi d'ajouter du texte à la suite du fichier (au lieu de tout écraser) à l'aide d'un boolean dans le constructeur :
`FileOutputStream(String nomFichier,boolean append)`
- **ByteArrayInputStream, ByteArrayOutputStream** : le flux est créé à partir d'un tableau d'octets.
- **PipedInputStream, PipedOutputStream** : Permet de créer un tube de données. Cela signifie que ce qu'on écrit dans un PipedOutputStream peut être directement lu dans un PipedInputStream.

Exemples (1)



```
// Copie d'un fichier source vers un fichier dest

try {
    // On cree un flux binaire d'entree de fichier
    FileInputStream fis = new FileInputStream("source");

    try {
        // On cree un flux binaire de sortie
        FileOutputStream fos = new FileOutputStream("dest");

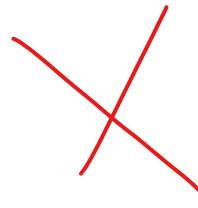
        // La methode read() renvoie un int representant l'octet lu
        // La valeur -1 presente la fin du fichier
        // La methode write permet d'ecrire un octet dans le flux de sortie

        int byteLu;

        try {
            while((byteLu = fis.read()) != -1) {
                fos.write(byteLu);
            }
        } finally{
            // On ferme toujours le flux pour liberer les ressources
            fos.close();
        }
    } finally {
        // On ferme toujours le flux pour liberer les ressources
        fis.close();
    }
}

catch(FileNotFoundException e) { }
catch(IOException e) { }
```

Exemples (2)



```
// Lecture
// On cree un buffer d'octets
byte[] buffer = new byte[10];
int byteLu;
// On remplit le buffer
for(byte i = 0; i < 10; i++) {
    buffer[i] = i;
}
// On cree un flux d'entree qui a comme source un tableau d'octets
ByteArrayInputStream bis = new ByteArrayInputStream(buffer);
// Puis on lit le flux, comme pour le FileStream
// A noter que read() renvoie un int et non un byte
while((byteLu = bis.read()) != -1) {
    System.out.println(byteLu);
}

// On peut maintenant faire l'inverse, en écriture
ByteArrayOutputStream bos = new ByteArrayOutputStream();
int byteLu;
byte[] tab = new byte[10];

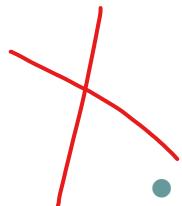
// On écrit dans le flux
for(byte i = 0; i < 10; i++) {
    bos.write(i);
}
// On récupère le flux sous forme de tableau d'octets
tab = bos.toByteArray();

// Puis on affiche les données
for(int j=0; j<10; j++) {
    System.out.println(tab[j]);
}
```

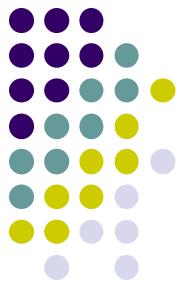
Flux de traitement d'octets



- **BufferedInputStream, BufferedOutputStream** : permet la lecture et l'écriture de données grâce à un **tampon**. Ce tampon est un tableau d'octets, il est très utile car il permet d'écrire ou de lire une suite de données d'un seul coup, plutôt que de lire ou écrire octet par octet. Cela optimise grandement les performances.
- **DataInputStream, DataOutputStream** : le flux permet de manipuler des données représentants des types primitifs de Java (int, boolean, double, byte). Il est optimisé pour ce type de données car elles sont facilement manipulées à l'aide de méthodes comme **writeInt()**, **writeDouble()** ou encore **readBoolean()**.
- **ObjectInputStream, ObjectOutputStream** : permet la sérialisation et deserialization, c'est à dire l'écriture et la restauration d'un objet, **implémentant** `java.io.Serializable`.

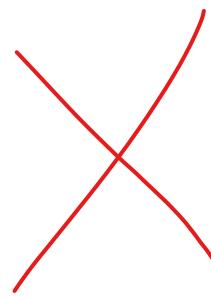


Exemple (1)



```
// Copie d'un fichier source vers un fichier dest
try {
    // On cree un buffer sur une lecture de fichier "source"
    // Cela permet juste d'améliorer les performances
    BufferedInputStream bis = new BufferedInputStream(new FileInputStream("source"));

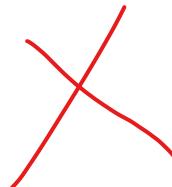
    // La méthode available permet de connaître le nombre d'octets qui pourront être lus
    // de manière non bloquante.
    System.out.println(bis.available());
    try {
        BufferedOutputStream bos = new BufferedOutputStream(new FileOutputStream("dest"));
        int byteLu;
        try {
            while((byteLu = bis.read()) != -1) {
                bos.write(byteLu);
            }
            // Vide le contenu du buffer
            bos.flush();
        } finally {
            bos.close();
        }
    } finally {
        bis.close();
    }
}
catch(FileNotFoundException e) {}
catch(IOException e) {}
```



Exemple (2)



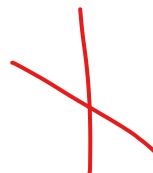
```
// On écrit d'abord dans un fichier "sortie"
try {
    // On crée un flux
    DataOutputStream dos = new DataOutputStream(new FileOutputStream("sortie"));
    // On écrit quelques données de base
    boolean test = true;
    int i = 100;
    try {
        // On écrit les données
        dos.writeBoolean(test);
        dos.writeInt(i);
        // Puis on vide le buffer
        dos.flush();
    } finally {
        dos.close();
    }
} catch(IOException e) {}
// Puis on lit les données de ce fichier
try {
    // On cree un flux
    DataInputStream dis = new DataInputStream(new FileInputStream("sortie"));
    try {
        // On lit les données
        boolean test = dis.readBoolean();
        int i = dis.readInt();
    } finally {
        dis.close();
    }
    // On affiche les données
    // Si tout se passe bien, vous devriez avoir un true et un 100 :)
    System.out.println(test);
    System.out.println(i);
} catch(IOException e) {}
```

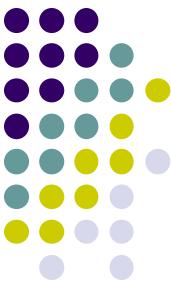




La sérialisation (1)

- La sérialisation consiste à prendre un objet en mémoire et à en sauvegarder l'état sur un flux de données (vers un fichier, par exemple).
- Ce concept permet aussi de reconstruire, ultérieurement, l'objet en mémoire à l'identique de ce qu'il pouvait être initialement.
- La sérialisation peut donc être considérée comme une forme de persistance des données.





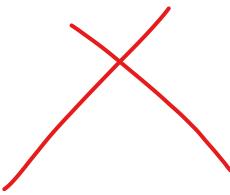
La sérialisation (2)

- 2 classes `ObjectInputStream` et `ObjectOutputStream` proposent, respectivement, les méthodes `readObject` et `writeObject`
- Par défaut, les classes ne permettent pas de sauvegarder l'état d'un objet sur un flux de données. Il faut implémenter l'interface `java.io.Serializable`.

Exemple (3)



```
// Imaginons qu'on ait une classe Article avec les attributs String titre et auteur
try {
    Article p = new Article("La serialization", "Killian");
    Article nouvelArticle;
    // On crée le flux d'écriture d'objets à partir d'un flux d'écriture de fichiers dans
    // un fichier "fichierObjet"
    ObjectOutputStream oos = new ObjectOutputStream(new FileOutputStream("fichierObjet"));
    try {
        // On écrit objet, puis on vide le buffer
        oos.writeObject(p);
        oos.flush();
    } finally {
        oos.close();
    }
    // On fait l'opération inverse (lecture)
    ObjectInputStream ois = new ObjectInputStream(new FileInputStream("fichierObjet"));
    try {
        // On cast l'entrée
        nouvelArticle = (Article)ois.readObject();
        // Puis on affiche
        System.out.println(nouvelArticle.titre);
    } finally {
        ois.close();
    }
} catch(Exception e) { }
```





Les flux de caractères (1)

- Ce sont des sous-classes de Reader et Writer.
- Ces flux utilisent le codage de caractères Unicode.
- Exemples
 - conversion des caractères saisis au clavier en caractères dans le codage par défaut

```
InputStreamReader in = new InputStreamReader(System.in);
```

Conversion des caractères d'un fichier
avec un codage explicitement indiqué

```
InputStreamReader in = new InputStreamReader (   
    new FileInputStream ("chinois.txt"), "ISO2022CN");
```

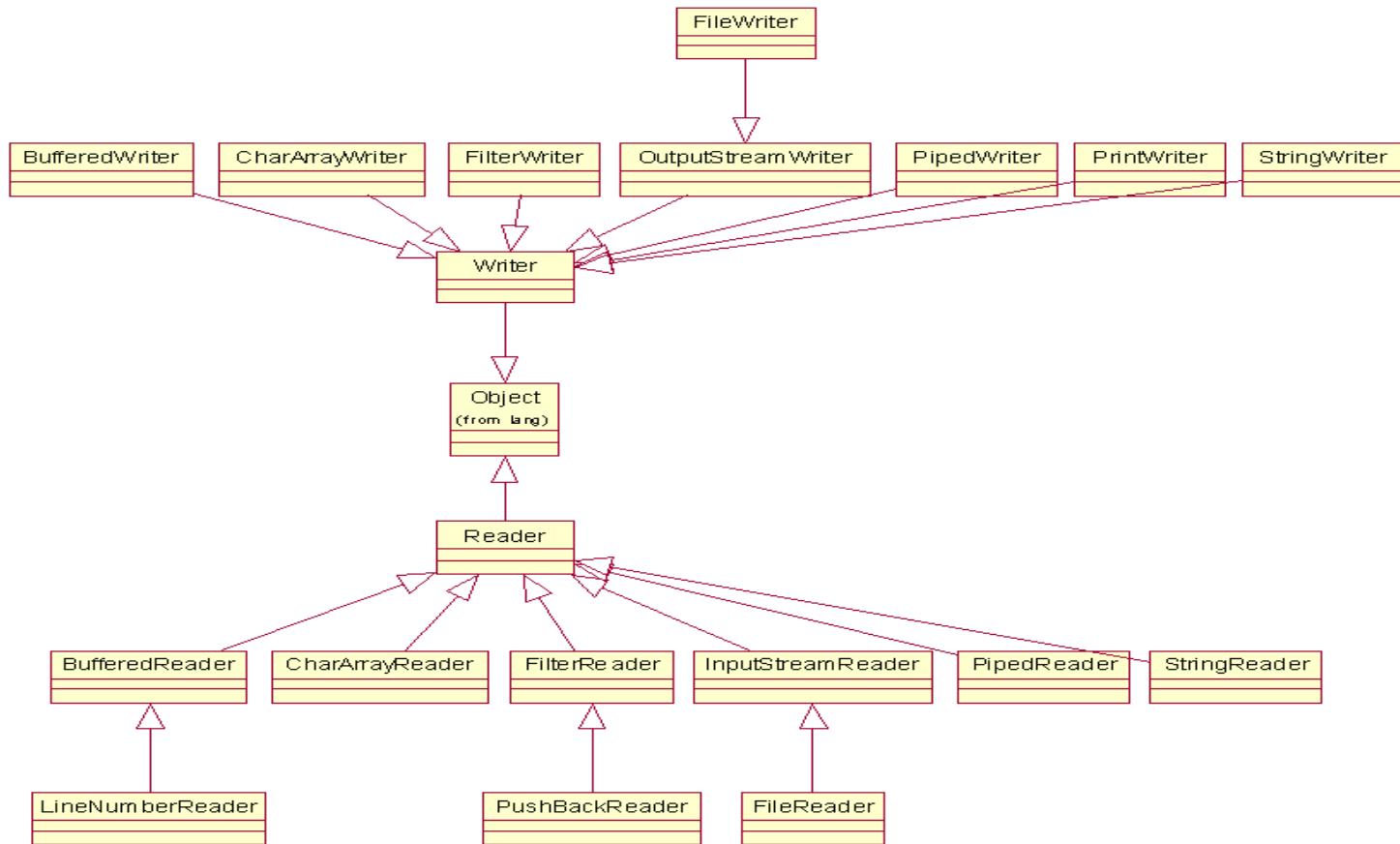
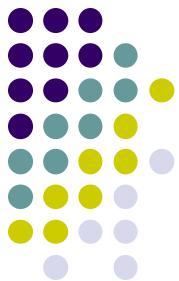


Les flux de caractères (2)

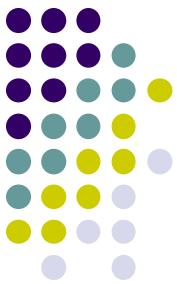
- Pour écrire des chaînes de caractères et des nombres sous forme de texte
 - on utilise la classe **PrintWriter** qui possède un certain nombre de méthodes **print (...)** et **println (...)**.
- Pour lire des chaînes de caractères sous forme texte, il faut utiliser, par exemple,
 - **BufferedReader** qui possède une méthode **readLine()**.
 - Pour la lecture de nombres sous forme de texte, il n'existe pas de solution toute faite : il faut par exemple passer par des chaînes de caractères et les convertir en nombres.



La hiérarchie des flux de caractères



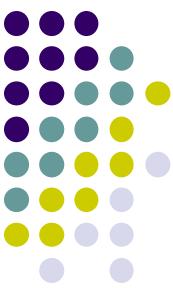
Flux de communication de caractères



- **CharArrayReader, CharArrayWriter** : l'équivalent du flux ByteArray, stocke les caractères dans un tableau indexé.
- **StringReader, StringWriter** : permet de lire / écrire des caractères dans un StringBuffer.
- **PipedReader, PipedWriter** : comme pour les PipedStream, permet de créer un tube qui échange des caractères.
- **FileReader, FileWriter** : ces classes étendent InputStreamReader et OutputStreamWriter. Elles permettent de lire / écrire en utilisant le tampon et l'encodage par défaut.



Exemple (1)

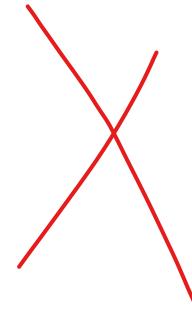


```
// Lecture
try {
    char[] tab = {'a','b','c','d','e'};
    // On cree le flux de lecture
    CharArrayReader car = new CharArrayReader(tab);
    // On lit les données
    for(int i = 0; i < tab.length; i++){
        System.out.println((char)car.read());
    }
}

} catch(IOException e) {}

// Ecriture
try {
    // On cree le flux
    // La taille du tableau est dynamique
    CharArrayWriter car = new CharArrayWriter();
    // On écrit les données
    for(char c = 'a'; c < 'e'; c++){
        car.write(c);
    }
    // On convertit en tableau de caractères
    char[] tab = car.toCharArray();

    // Et on affiche
    for(int i = 0; i < tab.length; i++){
        System.out.println(car[i]);
    }
    // A noter que close() n'agit pas sur ce flux
} catch(IOException e) {}
```





Exemple (2)

```
// Lecture d'une chaine
try {
    int i;
    String chaine = "Killian écrit de bons articles";

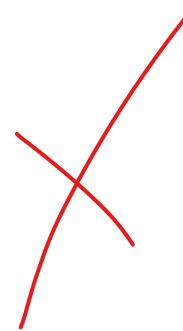
    // On cree le flux
    StringReader sr = new StringReader(chaine);

    // On lit le flux, caractère par caractère
    while((i=sr.read()) != -1){
        System.out.println((char)i);
    }
} catch(IOException e) { }

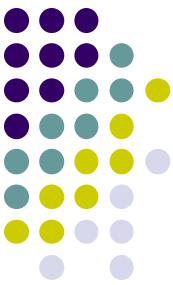
// Ecriture d'une chaine
try {
    // On cree le flux
    StringWriter sw = new StringWriter();

    // On écrit dans le flux
    sw.write("Killian écrit");
    sw.write(" de bien bons articles");

    // On affiche
    System.out.println(sw.toString());
    sw.flush();
    // La méthode close() est inefficace
} catch(IOException e) { }
```



Exemple (3)



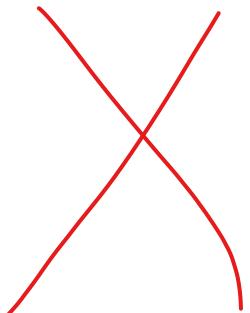
```
// Lecture d'un fichier "source"
// L'écriture = exactement la même opération dans le sens inverse

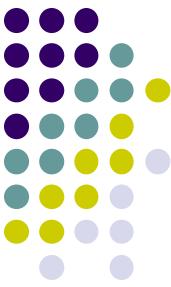
try {
    int i;

    // On cree le flux
    FileReader fr = new FileReader("source");

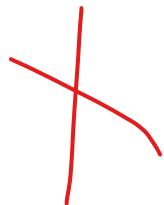
    try {
        // On lit les données
        while((i = fr.read()) != -1) {
            System.out.println((char)i);
        }
    } finally {
        fr.close();
    }

} catch(IOException e) { }
```



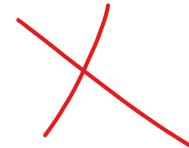


Flux de traitement de caractères



- **InputStreamReader, OutputStreamWriter** : permettent de transformer un flux de données binaires en un flux de caractères. Permet de lire / écrire quand l'encodage par défaut (de FileReader / FileWriter) ne convient pas.
- **BufferedReader, BufferedWriter** : permet d'employer un tampon pendant la lecture / l'écriture d'un flux de caractères. Améliore grandement les performances.
- **LineNumberReader** : sous classe de BufferedReader, profite de l'utilisation du tampon tout en permettant de connaître le numéro de la ligne lue.
- **PrintWriter** : permet d'écrire des caractères formatés.

Exemple



```
// Copie d'un fichier "source" vers un fichier "dest"

try {
    String ligne;

    // Création du flux de lecture
    BufferedReader br = new BufferedReader(new FileReader("source"));

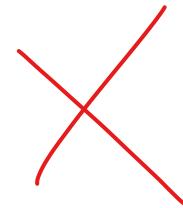
    try {
        // Création du flux d'écriture
        BufferedWriter bw = new BufferedWriter(new FileWriter("dist"));

        try {
            while((ligne = br.readLine()) != null){
                bw.write(ligne);
                // On insère un saut de ligne
                bw.newLine();
            }

            bw.flush();
        } finally {
            bw.close();
        }
    } finally {
        br.close();
    }
} catch(IOException e) {}
```

Flux de fichiers à accès direct

(1)



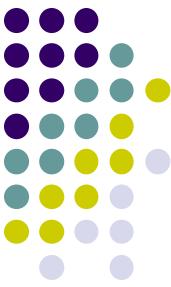
- La classe RandomAccessFile
 - permet de lire ou d'écrire dans un fichier à n'importe quel emplacement (par opposition aux fichiers à accès séquentiels).
- Elle implémente les interfaces DataInput et DataOutput
 - permettent de lire ou d'écrire tous les types Java de base, les lignes, les chaînes de caractères ascii ou unicode, etc ...

Flux de fichiers à accès direct (2)



- Un fichier à accès direct peut être
 - ouvert en lecture seule (option "r") ou
 - en lecture / écriture (option "rw").
- Ces fichiers possèdent un pointeur de fichier qui indique constamment la donnée suivante.
 - La position de ce pointeur est donnée par **long getFilePointer ()** et celui-ci peut être déplacé à une position donnée grâce à **seek (long off)**.





Exemple

```
try {
    // On ouvre un fichier
    File f = new File("nomDuFichier");

    // On ouvre un flux à accès aléatoire
    // "rw" signifie lecture écriture
    RandomAccessFile raf = new RandomAccessFile(f, "rw");

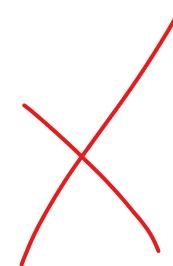
    // On lit un caractère
    char ch = raf.readChar();

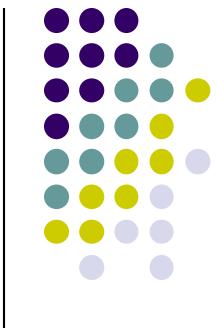
    // On peut se déplacer au sein du fichier
    // Ici on se positionne à la fin
    raf.seek(f.length());

    // Et on ajoute les caractères
    raf.writeChars("uneChaine");

    // Puis on ferme le flux
    raf.close();
}

} catch(FileNotFoundException ef) {}
} catch (IOException e) {}
```

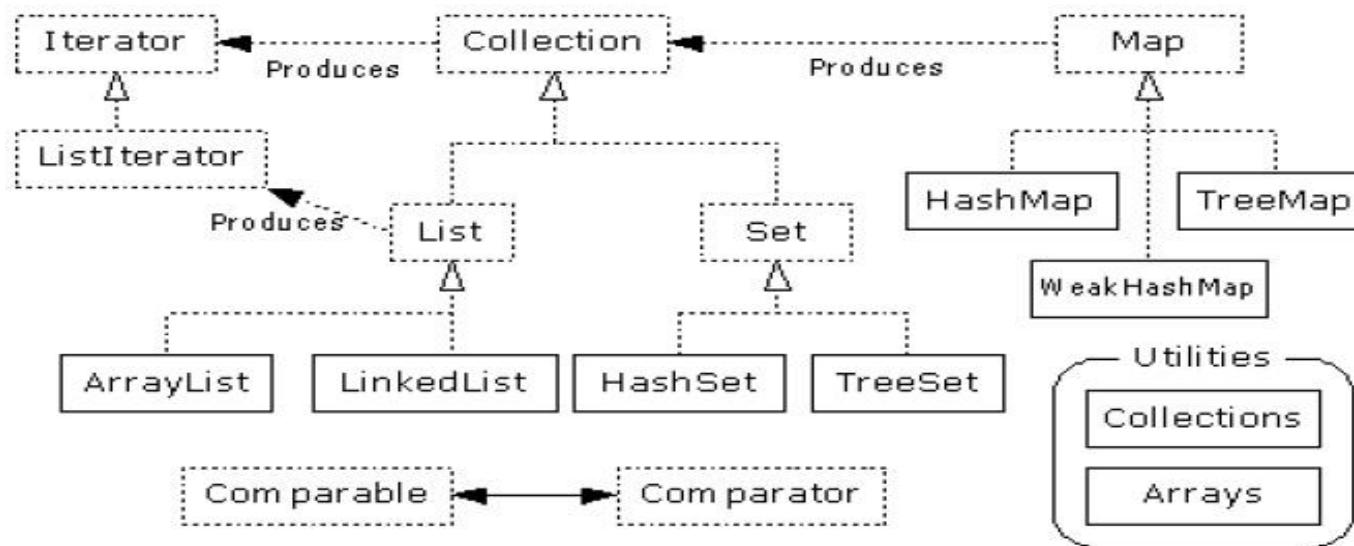


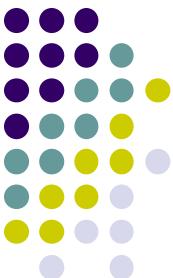




Les collections (Les containers)

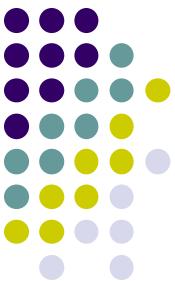
- Les collections sont des objets qui permettent de gérer des ensembles d'objets. Ces ensembles de données peuvent être définis avec plusieurs caractéristiques : la possibilité de gérer des doublons, de gérer un ordre de tri, etc. ...
- Chaque objet contenu dans une collection est appelé un élément.





Les collections (Les containers)

- Les structures de données classiques
 - Listes
 - Accès séquentiel : premier, suivant ==> recherche lente
 - Dynamique : nombre d'éléments variable dans le temps
 - Insertion, suppression faciles et efficace
 - Tableaux
 - Accès direct
 - Nombre d'éléments fixés à l'avance
 - Insertion et suppression lentes
- Containers : les **structures classiques + les principaux algorithmes** disponibles
 - Structures linéaires
 - Listes
 - Ensembles, ensembles triés: éléments ne peuvent être dupliqués
 - Tableaux, Tableaux dynamiques
 - Structures associatives ou dictionnaire =Map
 - Eléments : couple clé-valeur
 - Exemples : (nom, n°telephone)
 - Structures plus complexes : tables hachage, arbres binaire, arbres balancés
 - Méthodes et algorithmes
 - Ajout, suppression, appartenance, recherche
 - Tri,



Présentation du framework collection en Java (1)

Dans la version 1 du J.D.K., il n'existe qu'un nombre restreint de classes pour gérer des ensembles de données :

- Vector
- Stack
- Hashtable
- Bitset

- L'interface Enumeration permet de parcourir le contenu de ces objets.
- Pour combler le manque d'objets adaptés, la version 2 du J.D.K. apporte un framework complet pour gérer les collections. Les interfaces à utiliser par des objets qui gèrent des collections sont :
 - Collection : interface qui est implementée par la plupart des objets qui gèrent des collections
 - Map : interface qui définit des méthodes pour des objets qui gèrent des collections sous la forme clé/valeur
 - Set : interface pour des objets qui n'autorisent pas la gestion des doublons dans l'ensemble
 - List : interface pour des objets qui autorisent la gestion des doublons et un accès direct à un élément
 - SortedSet : interface qui étend l'interface Set et permet d'ordonner l'ensemble
 - SortedMap : interface qui étend l'interface Map et permet d'ordonner l'ensemble



Présentation du framework collection en Java (2)

- Le framework propose plusieurs objets qui implémentent les interfaces citées auparavant et qui peuvent être directement utilisés :
 - HashSet : HashTable qui implémente l'interface Set
 - TreeSet : arbre qui implémente l'interface SortedSet
 - ArrayList : tableau dynamique qui implémente l'interface List
 - LinkedList : liste doublement chaînée (parcours de la liste dans les deux sens) qui implémente l'interface List
 - HashMap : HashTable qui implémente l'interface Map
 - TreeMap : arbre qui implémente l'interface SortedMap
- Le framework définit aussi des interfaces pour faciliter le parcours des collections et leur tri :
 - Iterator : interface pour le parcours des collections
 - ListIterator : interface pour le parcours des listes dans les deux sens et modifier les éléments lors de ce parcours
 - Comparable : interface pour définir un ordre de tri naturel pour un objet
 - Comparator : interface pour définir un ordre de tri quelconque



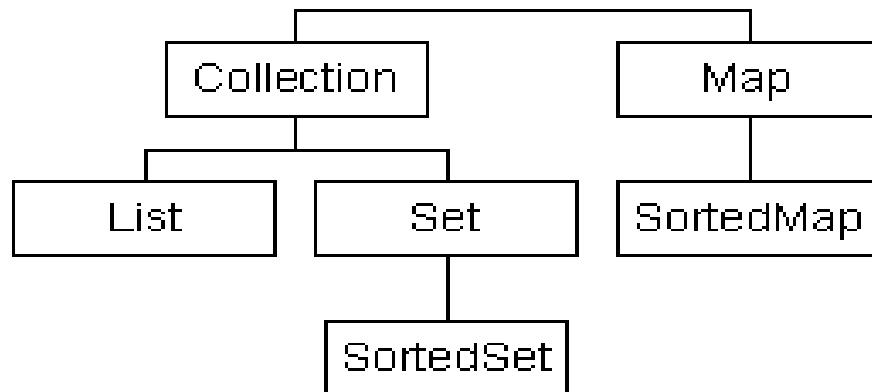
Présentation du framework collection en Java (3)

- Deux classes existantes dans les précédentes versions du JDK ont été modifiées pour implémenter certaines interfaces du framework :
 - Vector : tableau à taille variable qui implémente maintenant l'interface List
 - HashTable : table de hashage qui implémente maintenant l'interface Map
- Les objets du framework stockent toujours des références sur les objets contenus dans la collection et non les objets eux mêmes. Ce sont obligatoirement des objets qui doivent être ajoutés dans une collection. Avant le JDK 5.0 (ou Java1.5), il n'est pas possible de stocker directement des types primitifs : il faut obligatoirement encapsuler ces données dans des wrappers .
- Toutes les classes de gestion de collection du framework ne sont pas synchronisées par défaut : elles ne prennent pas en charge les traitements multi-threads. Le framework propose des méthodes qui prennent en charge cette fonctionnalité. Les classes Vector et Hashtable étaient synchronisées.
- Lors de l'utilisation de ces classes, il est préférable de stocker la référence de ces objets sous la forme d'une interface qu'ils implémentent plutôt que sous leur forme objet. Ceci rend le code plus facile à modifier si le type de l'objet qui gèrent la collection doit être changé.

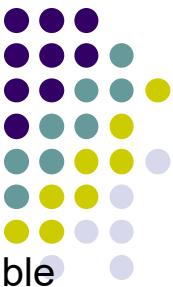


Les interfaces des collections (1)

- Le framework de java 2 définit 6 interfaces en relation directe avec les collections qui sont regroupées dans deux arborescences :



- Le JDK ne fourni pas de classes qui implémentent directement l'interface `Collection`.



Quels containers choisir ?

- En général, on sait si on a besoin d'une liste, d'un tableau, d'un ensemble, d'un ensemble trié, d'un dictionnaire, d'un dictionnaire trié, d'un arbre binaire de recherche.
- On définit un objet du type de l'interface la plus générique : Collection
- On crée ensuite l'objet réel du type voulu :
 - LinkedList : liste séquentielle classique avec pointeur
 - ArrayList : liste implantée par un tableau avec accès par index rapide. On peut aussi considérer que c'est un tableau dont la taille peut varier dans le temps.
 - Vector, historique, identique à ArrayList,
 - Queue (FIFO) : file !!!
 - HashSet : ensemble implémenté par une table de hashage
 - Ensemble : une seule occurrence de chaque élément autorisé
 - Accès aux éléments rapide
 - TreeSet, : ensemble implémenté avec un arbre binaire de recherche, trié en permanence
 - HashMap : ensemble associatif (clé valeur)
 - TreeMap : Ensemble associatif trié en permanence sur la clé
 - Accès aux éléments rapide en $\log(n)$



Les interfaces des collections (2)

- Le tableau ci-dessous présente les différentes classes qui implémentent les interfaces de bases Set, List et Map :

	Set collection d'éléments uniques	List collection avec doublons	Map collection sous la forme clé/valeur
Tableau redimensionnable		ArrayList, Vector (JDK 1.1)	
Arbre	TreeSet		TreeMap
Liste chaînée		LinkedList	
Collection utilisant une table de hashage	HashSet		HashMap, HashTable (JDK 1.1)
Classes du JDK 1.1		Stack	

L'interface Collection



Méthode	Rôle
boolean add(Object)	ajoute l'élément fourni en paramètre à la collection. La valeur de retour indique si la collection a été mise à jour
boolean addAll(Collection)	ajoute à la collection tous les éléments de la collection fournie en paramètre
void clear()	supprime tous les éléments de la collection
boolean contains(Object)	indique si la collection contient au moins un élément identique à celui fourni en paramètre
boolean containsAll(Collection)	indique si tous les éléments de la collection fournie en paramètre sont contenus dans la collection
boolean isEmpty()	indique si la collection est vide
Iterator iterator()	renvoie un objet qui permet de parcourir l'ensemble des éléments de la collection
boolean remove(Object)	supprime l'élément fourni en paramètre de la collection. La valeur de retour indique si la collection a été mise à jour
boolean removeAll(Collection)	supprime tous les éléments de la collection qui sont contenus dans la collection fournie en paramètre
int size()	renvoie le nombre d'éléments contenu dans la collection
Object[] toArray()	renvoie d'un tableau d'objets qui contient tous les éléments de la collection

L'interface Iterator



Cette interface définit des méthodes pour des objets capables de parcourir les données d'une collection.

Méthode	Rôle
boolean hasNext()	indique si il reste au moins à parcourir dans la collection
Object next()	renvoie le prochain élément dans la collection
void remove()	supprime le dernier élément parcouru

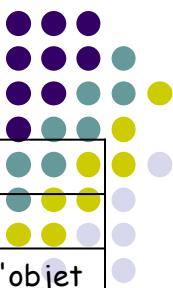
Exemple (code Java 1.2) :

```
Iterator iterator = collection.iterator();
while (iterator.hasNext()) {
    System.out.println("objet = "+iterator.next());
}
```

Exemple (code Java 1.2) : suppression du premier élément

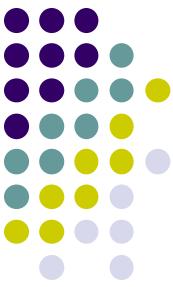
```
Iterator iterator = collection.iterator();
if (iterator.hasNext()) {
    iterator.next();
    iterator.remove();
}
```

L'interface List



Méthode	Rôle	
Iterator iterator()	renvoie un objet capable de parcourir la liste	
Object set (int, Object)	remplace l'élément contenu à la position précisée par l'objet fourni en paramètre	
void add(int, Object)	ajouter l'élément fourni en paramètre à la position précisée	
Object get(int)	renvoie l'élément à la position précisée	
int indexOf(Object)	renvoie l'index du premier élément fourni en paramètre dans la liste ou -1 si l'élément n'est pas dans la liste	
ListIterator listIterator()	renvoie un objet pour parcourir la liste et la mettre à jour	
List subList(int,int)	renvoie un extrait de la liste contenant les éléments entre les deux index fournis (le premier index est inclus et le second est exclus). Les éléments contenus dans la liste de retour sont des références sur la liste originale. Des mises à jour de ces éléments impactent la liste originale.	
int lastIndexOf(Object)	renvoie l'index du dernier élément fourni en paramètre dans la liste ou -1 si l'élément n'est pas dans la liste	
Object set(int, Object)	remplace l'élément à la position indiquée avec l'objet fourni	

Le framework propose des classes qui implémentent l'interface List : LinkedList et ArrayList.



Les listes chaînées : la classe LinkedList

- Cette classe hérite de la classe AbstractSequentialList et implémente donc l'interface List.
- Elle représente une liste doublement chaînée.
- Cette classe possède un constructeur sans paramètre et un qui demande une collection. Dans ce dernier cas, la liste sera initialisée avec les éléments de la collection fournie en paramètre.

Exemple (code Java 1.2) :

```
LinkedList listeChaine = new LinkedList();
Iterator iterator = listeChaine.iterator();
listeChaine.add("element 1");
listeChaine.add("element 2");
listeChaine.add("element 3");
while (iterator.hasNext()) {
    System.out.println("objet =
"+iterator.next());
}
```

Méthode	Rôle
void addFirst(Object)	insère l'objet en début de la liste
void addLast(Object)	insère l'objet en fin de la liste
Object getFirst()	renvoie le premier élément de la liste
Object getLast()	renvoie le dernier élément de la liste
Object removeFirst()	supprime le premier élément de la liste et renvoie le premier élément
Object removeLast()	supprime le dernier élément de la liste et renvoie le premier élément

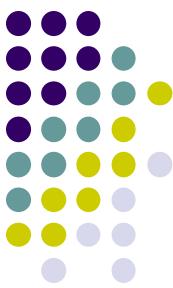


L'interface ListIterator

- Cette interface définit des méthodes pour parcourir la liste dans les deux sens et effectuer des mises à jour qui agissent par rapport à l'élément courant dans le parcours
- En plus des méthodes définies dans l'interface Iterator dont elle hérite, elle définit les méthodes suivantes :

Méthode	Rôles
<code>void add(Object)</code>	ajoute un élément dans la liste en tenant de la position dans le parcours
<code>boolean hasPrevious()</code>	indique s'il reste au moins un élément à parcourir dans la liste dans son sens inverse
<code>Object previous()</code>	renvoi l'élément précédent dans la liste
<code>void set(Object)</code>	remplace l'élément courante par celui fourni en paramètre

Les tableaux redimensionnables : la classe ArrayList



Méthode	Rôle
boolean add(Object)	ajoute un élément à la fin du tableau
boolean addAll(Collection)	ajoute tous les éléments de la collection fournie en paramètre à la fin du tableau
boolean addAll(int, Collection)	ajoute tous les éléments de la collection fournie en paramètre dans la collection à partir de la position précisée
void clear()	supprime tous les éléments du tableau
void ensureCapacity(int)	permet d'augmenter la capacité du tableau pour s'assurer qu'il puisse contenir le nombre d'éléments passé en paramètre
Object get(index)	renvoie l'élément du tableau dont la position est précisée
int indexOf(Object)	renvoie la position de la première occurrence de l'élément fourni en paramètre
boolean isEmpty()	indique si le tableau est vide
int lastIndexOf(Object)	renvoie la position de la dernière occurrence de l'élément fourni en paramètre
Object remove(int)	supprime dans le tableau l'élément fourni en paramètre
void removeRange(int,int)	supprime tous les éléments du tableau de la première position fourni incluse jusqu'à la dernière position fournie exclue
Object set(int, Object)	remplace l'élément à la position indiquée par celui fourni en paramètre
int size()	renvoie le nombre d'élément du tableau
void trimToSize()	ajuste la capacité du tableau sur sa taille actuelle

- Cette classe représente un tableau d'objets dont la taille est dynamique.
- Elle hérite de la classe `AbstractList` donc elle implémente l'interface `List`.
- Le fonctionnement de cette classe est identique à celui de la classe `Vector`.
- La différence avec la classe `Vector` est que cette dernière est multi thread (toutes ces méthodes sont synchronisées). Pour une utilisation dans un thread unique, la synchronisation des méthodes est inutile et coûteuse. Il est alors préférable d'utiliser un objet de la classe `ArrayList`.
- Elle définit plusieurs méthodes dont les principales sont :

L'interface Set



- Cette classe définit les méthodes d'une collection qui n'accepte pas de doublons dans ses éléments. Elle hérite de l'interface Collection mais elle ne définit pas de nouvelle méthode.
- Pour déterminer si un élément est déjà inséré dans la collection, la méthode equals() est utilisée.
- Le framework propose deux classes qui implémentent l'interface Set : TreeSet et HashSet
- Le choix entre ces deux objets est lié à la nécessité de trier les éléments :
 - les éléments d'un objet HashSet ne sont pas triés : l'insertion d'un nouvel élément est rapide
 - les éléments d'un objet TreeSet sont triés : l'insertion d'un nouvel élément est plus long



L'interface SortedSet

- Cette interface définit une collection de type ensemble triée. Elle hérite de l'interface Set.
- Le tri de l'ensemble peut être assuré par deux façons :
- les éléments contenus dans l'ensemble implémentent l'interface Comparable pour définir leur ordre naturel
- il faut fournir au constructeur de l'ensemble un objet Comparator qui définit l'ordre de tri à utiliser
- Elle définit plusieurs méthodes pour tirer parti de cette ordre :

Méthode	Rôle
<code>Comparator comparator()</code>	renvoie l'objet qui permet de trier l'ensemble
<code>Object first()</code>	renvoie le premier élément de l'ensemble
<code>SortedSet headSet(Object)</code>	renvoie un sous ensemble contenant tous les éléments inférieurs à celui fourni en paramètre
<code>Object last()</code>	renvoie le dernier élément de l'ensemble
<code>SortedSet subSet(Object, Object)</code>	renvoie un sous ensemble contenant les éléments compris entre le premier paramètre inclus et le second exclus
<code>SortedSet tailSet(Object)</code>	renvoie un sous ensemble contenant tous les éléments supérieurs ou égaux à celui fourni en paramètre



La classe HashSet

- Cette classe est un ensemble sans ordre de tri particulier.
- Les éléments sont stockés dans une table de hashage : cette table possède une capacité.

Exemple (code Java 1.2) :

```
import java.util.*;
public class TestHashSet {
    public static void main(String args[]) {
        Set set = new HashSet();
        set.add("CCCCC");
        set.add("BBBBB");
        set.add("DDDDD");
        set.add("BBBBB");
        set.add("AAAAA");
        Iterator iterator = set.iterator();
        while (iterator.hasNext())
            {System.out.println(iterator.next());}
    }
}
```

Résultat :

```
AAAAA
DDDDD
BBBBB
CCCCC
```



La classe TreeSet

- Cette classe est un arbre qui représente un ensemble trié d'éléments.
- Cette classe permet d'insérer des éléments dans n'importe quel ordre et de restituer ces éléments dans un ordre précis lors de son parcours.
- L'implémentation de cette classe insère un nouvel élément dans l'arbre à la position correspondant à celle déterminée par l'ordre de tri. L'insertion d'un nouvel élément dans un objet de la classe TreeSet est donc plus lent mais le tri est directement effectué.
- L'ordre utilisé est celui indiqué par les objets insérés s'ils implémentent l'interface Comparable pour un ordre de tri naturel ou fournir un objet de type Comparator au constructeur de l'objet TreeSet pour définir l'ordre de tri.

Exemple (code Java 1.2) :

```
import java.util.*;
public class TestTreeSet {
public static void main(String args[]) {
    Set set = new TreeSet();
    set.add("CCCCC");
    set.add("BBBBB");
    set.add("DDDDD");
    set.add("BBBBB");
    set.add("AAAAA");
    Iterator iterator = set.iterator();
    while (iterator.hasNext()) {System.out.println(iterator.next());}
}
```

Résultat :

```
AAAAA
BBBBB
CCCCC
DDDDD
```

L'interface Map



- Cette interface est une des deux racines de l'arborescence des collections. Les collections qui implémentent cette interface ne peuvent contenir des doublons. Les collections qui implémentent cette interface utilisent une association entre une clé et une valeur.
- La méthode entrySet() permet d'obtenir un ensemble contenant toutes les clés.
- La méthode values() permet d'obtenir une collection contenant toutes les valeurs. La valeur de retour est une Collection et non un ensemble car il peut y avoir des doublons (plusieurs clés peuvent être associées à la même valeur).
- Le J.D.K. 1.2 propose deux nouvelles classes qui implémentent cette interface :
- HashMap qui stocke les éléments dans une table de hashage
- TreeMap qui stocke les éléments dans un arbre
- La classe HashTable a été mise à jour pour implémenter aussi cette interface.

Méthode	Rôle
void clear()	supprime tous les éléments de la collection
boolean containsKey(Object)	indique si la clé est contenue dans la collection
boolean containsValue(Object)	indique si la valeur est contenue dans la collection
Set entrySet()	renvoie un ensemble contenant les valeurs de la collection
Object get(Object)	renvoie la valeur associée à la clé fournie en paramètre
boolean isEmpty()	indique si la collection est vide
Set keySet()	renvoie un ensemble contenant les clés de la collection
Object put(Object, Object)	insère la clé et sa valeur associée fournies en paramètres
void putAll(Map)	insère toutes les clés/valeurs de l'objet fourni en paramètre
Collection values()	renvoie une collection qui contient toutes les éléments des éléments
Object remove(Object)	supprime l'élément dont la clé est fournie en paramètre
int size()	renvoie le nombre d'éléments de la collection



L'interface SortedMap

- Cette interface définit une collection de type Map triée sur la clé. Elle hérite de l'interface Map.
- Le tri peut être assuré par deux façons :
 - les clés contenues dans la collection implémentent l'interface Comparable pour définir leur ordre naturel
 - il faut fournir au constructeur de la collection un objet Comparator qui définit l'ordre de tri à utiliser
- Elle définit plusieurs méthodes pour tirer parti de cette ordre :

Méthode	Rôle
Comparator comparator()	renvoie l'objet qui permet de trier la collection
Object first()	renvoie le premier élément de la collection
SortedSet headMap(Object)	renvoie une sous collection contenant tous les éléments inférieurs à celui fourni en paramètre
Object last()	renvoie le dernier élément de la collection
SortedMap subMap(Object, Object)	renvoie une sous collection contenant les éléments compris entre le premier paramètre inclus et le second exclus
SortedMap tailMap(Object)	renvoie une sous collection contenant tous les éléments supérieurs ou égaux à celui fourni en paramètre



La classe Hashtable

- Cette classe qui existe depuis le premier jdk implémente une table de hachage. La clé et la valeur de chaque élément de la collection peut être n'importe quel objet non nul.
- A partir de Java 1.2 cette classe implémente l'interface Map.
- Une des particularités de classe HashTable est qu'elle est synchronisée.

Exemple (code Java 1.2) :

```
import java.util.*;public class TestHashtable {  
    public static void main(String[] args) {  
        Hashtable htable = new Hashtable();  
        htable.put(new Integer(3), "données 3");  
        htable.put(new Integer(1), "données 1");  
        htable.put(new Integer(2), "données 2");  
        System.out.println(htable.get(new Integer(2)));  
    }  
}
```

Résultat :

données 2



La classe TreeMap

- Cette classe gère une collection d'objets sous la forme clé/valeur stockés dans un arbre de type rouge-noir (Red-black tree). Elle implémente l'interface SortedMap. L'ordre des éléments de la collection est maintenu grâce à un objet de type Comparable.
- Elle possède plusieurs constructeurs dont un qui permet de préciser l'objet Comparable pour définir l'ordre dans la collection.

Exemple (code Java 1.2) :

```
import java.util.*;
public class TestTreeMap {
    public static void main(String[] args) {
        TreeMap arbre = new TreeMap();
        arbre.put(new Integer(3), "données 3");
        arbre.put(new Integer(1), "données 1");
        arbre.put(new Integer(2), "données 2");
        Set cles = arbre.keySet();
        Iterator iterator = cles.iterator();
        while (iterator.hasNext()) {
            System.out.println(arbre.get(iterator.next()));
        }
    }
}
```

Résultat :

```
données 1
données 2
données 3
```



La classe HashMap

- La classe HashMap est similaire à la classe Hashtable. Les trois grandes différences sont :
- elle est apparue dans le JDK 1.2
- elle n'est pas synchronisée
- elle autorise les objets null comme clé ou valeur
- Cette classe n'étant pas synchronisée, pour assurer la gestion des accès concurrents sur cet objet, il faut l'envelopper dans un objet Map en utilisant la méthode synchronizedMap de la classe Collection.



L'interface Comparable

- Tous les objets qui doivent définir un ordre naturel utilisé par le tri d'une collection avec cet ordre doivent implémenter cette interface.
- Cette interface ne définit qu'une seule méthode : int compareTo(Object).
- Cette méthode doit renvoyer :
 - une valeur entière négative si l'objet courant est inférieur à l'objet fourni
 - une valeur entière positive si l'objet courant est supérieur à l'objet fourni
 - une valeur nulle si l'objet courant est égal à l'objet fourni
- Les classes wrappers, String et Date implémentent cette interface.



L'interface Comparator

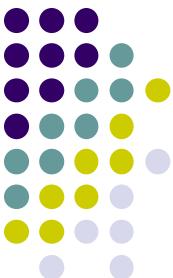
- Cette interface représente un ordre de tri quelconque. Elle est utile pour permettre le tri d'objet qui n'implémente pas l'interface Comparable ou pour définir un ordre de tri différent de celui défini avec Comparable (l'interface Comparable représente un ordre naturel : il ne peut y en avoir qu'un)
- Cette interface ne définit qu'une seule méthode : int compare(Object, Object).
- Cette méthode compare les deux objets fournis en paramètre et renvoie :
 - une valeur entière négative si le premier objet est inférieur au second
 - une valeur entière positive si le premier objet est supérieur au second
 - une valeur nulle si les deux objets sont égaux



Les algorithmes (1)

- La classe Collections propose plusieurs méthodes statiques qui effectuer des opérations sur des collections. Ces traitements sont polymorphiques car ils demandent en paramètre un objet qui implémente une interface et retourne une collection.
- Si la méthode sort(List) est utilisée, il faut obligatoirement que les éléments inclus dans la liste implémentent tous l'interface Comparable sinon une exception de type ClassCastException est levée.
- Cette classe propose aussi plusieurs méthodes pour obtenir une version multi-thread ou non modifiable des principales interfaces des collections : Collection, List, Map, Set, SortedMap, SortedSet
 - XXX synchronizedXXX(XXX) pour obtenir une version multi-thread des objets implémentant l'interface XXX
 - XXX unmodifiableXXX(XXX) pour obtenir une version non modifiable des objets implémentant l'interface XXX

Méthode	Rôle
void copy(List, List)	copie tous les éléments de la seconde liste dans la première
Enumeration enumeration(Collection)	renvoie un objet Enumeration pour parcourir la collection
Object max(Collection)	renvoie le plus grand élément de la collection selon l'ordre naturel des éléments
Object max(Collection, Comparator)	renvoie le plus grand élément de la collection selon l'ordre naturel précisé par l'objet Comparator
Object min(Collection)	renvoie le plus petit élément de la collection selon l'ordre naturel des éléments
Object min(Collection, Comparator)	renvoie le plus petit élément de la collection selon l'ordre précisé par l'objet Comparator
void reverse(List)	inverse l'ordre de la liste fournie en paramètre
void shuffle(List)	réordonne tous les éléments de la liste de façon aléatoire
void sort(List)	trie la liste dans un ordre ascendant selon l'ordre naturel des éléments
void sort(List, Comparator)	trie la liste dans un ordre ascendant selon l'ordre précisé par l'objet Comparator



Les algorithmes (2)

Exemple (code Java 1.2) :

```
import java.util.*;
public class TestUnmodifiable{
    public static void main(String args[])
    {
        List list = new LinkedList();
        list.add("1");
        list.add("2");
        list = Collections.unmodifiableList(list);
        list.add("3");
    }
}
```

Résultat :

```
C:\>java TestUnmodifiable
Exception in thread "main"
java.lang.UnsupportedOperationException
at
java.util.Collections$UnmodifiableCollection.add(Unkn
wn Source)
at TestUnmodifiable.main(TestUnmodifiable.java:13)
```

Exemple (code Java 1.2) :

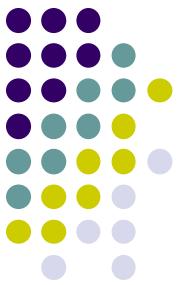
```
import java.util.*;
public class TestSynchronized{
    public static void main(String args[])
    {
        List maList = new LinkedList();    maList.add("1");
        maList.add("2");
        maList.add("3");
        maList=Collections.synchronizedList(maList);
        synchronized(maList) {
            Iterator i = maList.iterator();
            while (i.hasNext())
                System.out.println(i.next());
        }
    }
}
```

Méthode compareTo de l'interface comparable

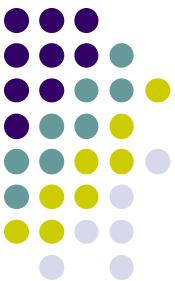


- Certaines classes comme string, classes enveloppes (Integer, Float,...) implémentent l'interface comparable et dispose donc d'une méthode compareTo qui conduit à un ordre naturel
 - Lexicographique pour les chaînes de caractères
 - Numériques pour les classes enveloppes numériques
 - Pour des objets propres à l'utilisateur: nécessaire de redéfinir la méthode compareTo (object o)

Utilisation d'un objet comparateur



- Dans quels cas
 - Les éléments sont des objets d'une classe existante n'implémentant pas l'interface comparable
 - Besoin de définir plusieurs ordre différents sur une même collection

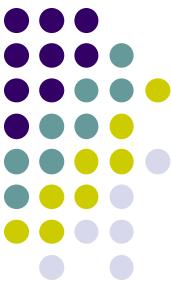


Exemple: Méthode compare

```
import java.util.*;  
  
public class Chien {  
    int nombre;  
    Chien (int i) {  
        nombre = i;  
    }  
    void imprimer() {  
        System.out.println("Chien "+nombre);  
    }  
    public String toString() {  
        return "Chien : "+nombre;  
    }  
}
```

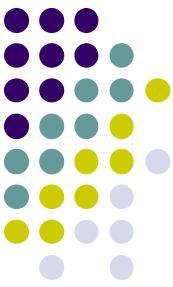


```
class ChienComparator implements Comparator{  
    public int compare (Object o1, Object o2){  
        if (((Chien) (o1)).nombre)<((Chien) (o2)).nombre))  
            return -1;  
        else if (((Chien) (o1)).nombre)>((Chien) (o2)).nombre))  
            return 1;  
        else return 0;  
    }  
  
}  
  
class Main_chien {  
    public static void main(String[] args) {  
        ArrayList c= new ArrayList();  
        Random r = new Random();  
        for (int i=0;i<10;i++) c.add(new Chien(r.nextInt(100)));  
        Random r = new Random();  
        for (int i=0;i<10;i++) c.add(new Chien(r.nextInt(100)));  
        for(int i=0; i<c.size();i++) System.out.println(i + " " +c.get(i));  
        Comparator comp = new ChienComparator();  
        System.out.println("max " + Collections.max(c,comp));  
        Collections.sort(c,comp);  
        for(int i=0; i<c.size();i++)  
            System.out.println(i + " " + c.get(i));  
    }  
}
```

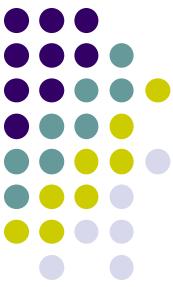


Méthode compareTo

```
import java.util.*;  
  
public class Chienc implements Comparable{  
    int nombre;  
    Chienc (int i){  
        nombre = i;  
    }  
    public int compareTo(Object o) {  
        if ((this.nombre) < (((Chienc) (o)).nombre))  
            return -1;  
        else if ((this.nombre) > (((Chienc) (o)).nombre))  
            return 1;  
        else  
            return 0;  
    }  
    void imprimer(){  
        System.out.println("Chienc "+nombre);  
    }  
    public String toString(){  
        return "Chienc :" + nombre;  
    }  
}
```

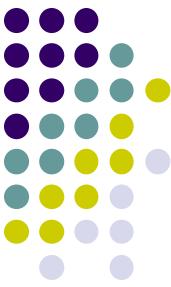


```
class Main_Chienc1 {  
public static void main(String[] args) {  
ArrayList c= new ArrayList();  
Random r = new Random();  
for (int i=0;i<10;i++)  
c.add(new Chienc (r.nextInt(100)));  
for(int i=0; i<c.size();i++)  
System.out.println(i + " " +c.get(i));  
System.out.println("max " + Collections.max(c));  
Collections.sort(c);  
for(int i=0; i<c.size();i++)  
System.out.println(i + " " + c.get(i));  
}  
}
```



Combinaison des deux Tri sur différents critères

```
import java.util.*;
class Chien implements Comparable{
int nombre;
int longueur;
Chien (int i, int j){
nombre = i;
longueur = j;
}
public int compareTo(Object o) {
if ((this.nombre) < (((Chien) (o)).nombre))
return -1;
else if ((this.nombre) > (((Chien) (o)).nombre))
return 1;
else
return 0;
}
void imprimer(){
System.out.println("Chien "+nombre);
}
public String toString(){
return "Chien :" +nombre+ ":" +longueur;
}
}
```



```
class ChienComparator implements Comparator{
public int compare (Object o1, Object o2){
if (((Chien) (o1)).longueur) < (((Chien) (o2)).longueur))
return -1;
else if (((Chien) (o1)).longueur) > (((Chien) (o2)).longueur))
return 1;
else return 0;
}
}

public class Main_chien_2 {
public static void main(String[] args) {
ArrayList c= new ArrayList();
Random r = new Random();
for (int i=0;i<10;i++)
c.add(new Chien(r.nextInt(100),r.nextInt(10)));
for(int i=0; i<c.size();i++)
System.out.println(i + " " +((Chien) c.get(i)).nombre+":"+((Chien)
c.get(i)).longueur);
Comparator comp = new ChienComparator();
System.out.println("max comparateur" + Collections.max(c,comp));
Collections.sort(c,comp);
for(int i=0; i<c.size();i++)
System.out.println(i + " " +((Chien) c.get(i)).nombre+":"+((Chien)
c.get(i)).longueur);

System.out.println("max comparable" + Collections.max(c));
Collections.sort(c);
for(int i=0; i<c.size();i++)
System.out.println(i + " " +((Chien) c.get(i)).nombre+":"+((Chien)
c.get(i)).longueur);
}

}
```

Généricité



- La généricité permet de définir des classes, et des méthodes paramétrées par une ou plusieurs classes.
- Pourquoi la généricité ?
 - Les erreurs types sont immédiatement détectées à la compilation
 - Simplification au niveau des casts
 - Factorisation du code

Exemple : Soit la classe suivante :

```
public class Paire {  
    Object premier ;  
    Object second;  
    public Paire (Object a, Object b){  
        premier= a; second = b;  
    }  
    public object getPremier(){  
        return premier;  
    }  
    public object getSecond(){  
        return second;  
    }  
}
```

Généricité



- Les deux inconvénients sont :

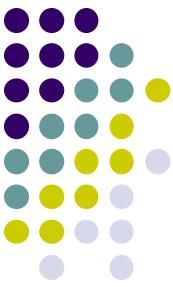
```
Paire p = new Paire ("abc", "xyz");
String x = (String)p.getPremier(); // le casting est obligatoire
Double y = (Double)p.getSecond(); // Il faut attendre l'exécution pour avoir
                                // une levée d'exception(ClassCastException)
```

- On définit alors une classe paramétrée :

```
public class Paire<T> {
    T premier ;
    T second;
    public Paire (T a, T b) {
        premier = a;
        second = b;
    }
    public T getPremier(){
        return premier;
    }
    public T getSecond(){
        return second;
    }
}
```

- Le programme est alors plus simple et plus sûr :

```
Paire<String> p = new Paire<String> ("abc", "xyz");
String x = p.getPremier(); // pas de cast
Double y = p.getSecond(); // erreur de compilation (type mismatch)
```



Méthode générique

- On peut définir une méthode générique dans une classe de la façon suivante :

```
public class X {  
    public <T> void affiche(Paire<T> p) {  
        System.out.println(p);  
    }  
    public <T> T choix(T a, T b) {  
        return (int) (Math.random() * 2) == 1 ? a : b;  
    }  
    public static void main(String []a) {  
        Paire<String> ps = new Paire<String>("un", "deux");  
        Paire<Integer> pi = new Paire<Integer>(1, 2);  
        X x = new X();  
        x.affiche(ps);  
        x.affiche(pi);  
        Number n = x.choix(new Integer(2), new Double(3.14159));  
    }  
}
```

Dans la dernière ligne, le compilateur fait une **inférence de type**, et calcule la première super classe commune aux deux classes *Integer* et *Double*, soit *Number*.



Limite pour les types paramètre

- Supposons que nous voulions ajouter à la classe *Paire<T>* la méthode suivante :

```
public T min(){  
    if(premier.compareTo(second)<=0) return premier;  
    else return second;  
}
```

- Le compilateur signale alors que la méthode *compareTo* n'est pas définie pour le type *T*.
- Il faut restreindre *T* à une classe qui a cette méthode, et définir la classe *Paire* de la façon suivante :

```
public class Paire<T extends Comparable> {  
    T premier ;  
    T second;  
    public Paire (T a, T b){  
        premier = a;  
        second = b;  
    }  
    public T getPremier(){  
        return premier;  
    }  
    public T getSecond(){  
        return second;  
    }  
    public T min(){  
        if(premier.compareTo(second)<=0) return premier;  
        else return second;  
    }  
}
```

Le type limitant peut être une classe ou une interface. On définira :

public class Paire<T extends Number> { ... } pour définir une paire de nombres... et

public class Paire<T extends A & Comparable> { ... } pour définir une paire d'élément de la classe *A* ou de 235 classes dérivées de *A* et qui sont *Comparable*.

Effacement



- Les classes paramétrées sont compilées vers une classe représentant le type *brut* : le type équivalent débarrassé des paramètres de la classe. On dit que les paramètres de type sont effacés. Ceci a des conséquences :
- On ne peut pas avoir une classe *Paire* et une classe *Paire<T>*

```
Paire p1 = new Paire("abc");
if(p1 instanceof Paire) ; // OK
if(p1 instanceof Paire<String>) ; // NON OK
```

- La dernière ligne provoque l'erreur de compilation suivante : Impossible d'effectuer une vérification instanceof sur le type paramétré Paire. A la place, utilisez sa forme brute Paire car les informations du type générique seront effacées lors de l'exécution.
- Lors de l'effacement la classe générique *Paire<T>* est générée en la classe brute *Paire* définie comme suit :

```
public class Paire {
    Object premier ;
    Object second;
    public Paire (Object a, Object b) {
        premier= a; second = b;
    }

    public object getPremier() {
        return premier;
    }

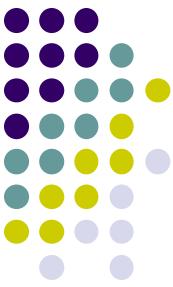
    public object getSecond() {
        return second;
    }
}
```



Effacement

- Lors de l'effacement la classe générique *Paire<T extends A>* est générée en la classe brute *Paire* définie comme suit :

```
public class Paire {  
    A premier ;  
    A second;  
    public Paire (A, A b){  
        premier= a; second = b;  
    }  
  
    public A getPremier(){  
        return premier;  
    }  
  
    public A getSecond(){  
        return second;  
    }  
  
    Paire<String> p1 = new Paire<String>("abc", "1");  
    Paire p2 = new Paire("abc", "def");  
  
    p2 = p1; // OK  
    p1 = p2; // avertissement, sécurité de type :  
    // l'expression du type Paire requiert une conversion non contrôlée en Paire<String>
```



Généricité et héritage

- Soient les deux classes suivantes :

```
public class Super {  
    ...  
}
```

```
public class Sous extends Super {  
    ...  
}
```

- Il n'existe pas de relation d'héritage entre Paire<Super> et Paire<Sous>.

```
Paire<Super> pSup = new Paire<Super>(new Super(), new Super());  
Paire<Sous> pSous = new Paire<Sous>(new Sous(), new Sous());  
pSup = pSous; // INTERDIT
```



Joker (Wildcards)

- Supposons que nous définissions dans la classe X une autre méthode pour afficher des paires de Super, de la façon suivante :

```
public void afficheS(Paire<Super>p) {  
    System.out.println(p);  
}  
  
Paire <Super> ps = new Paire<Super>();  
Paire <Sous> pso = new Paire< Sous>();  
  
x.afficheS(ps);      // OK, pas de problème.  
x.afficheS(pso);    // NON, pso n'est pas une Paire de Super
```

- Pour que la dernière instruction soit possible il faudrait définir la méthode `afficheS` de la façon suivante :

```
public void afficheS(Paire<?> p) {  
    System.out.println(p);  
}
```

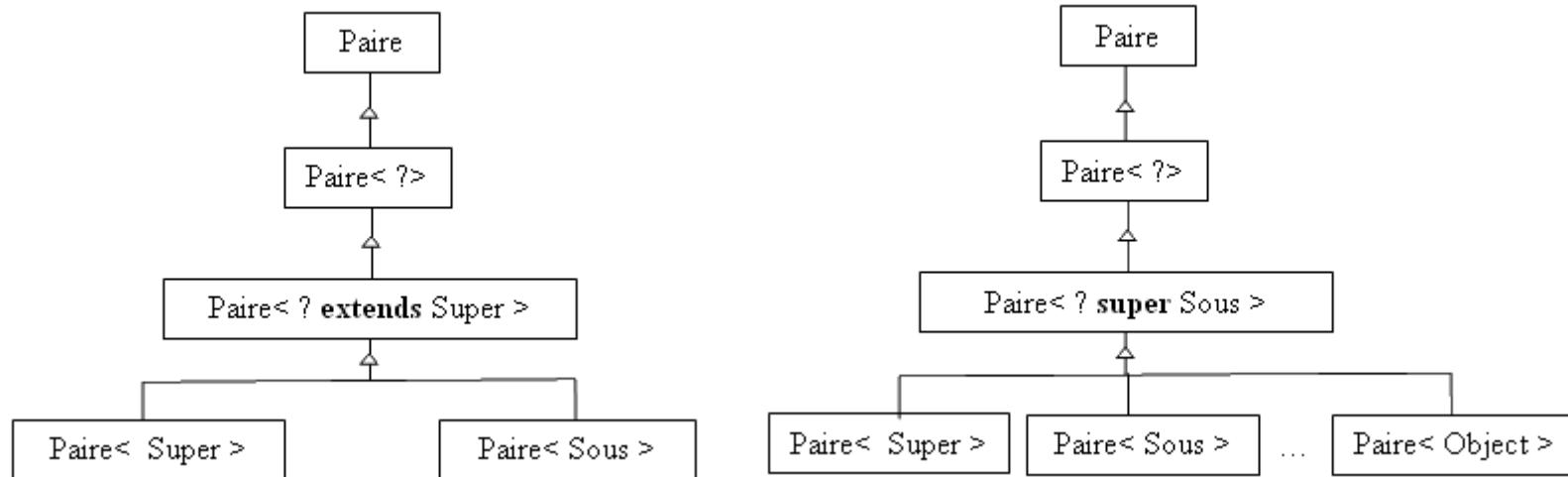
- Mais dans ce cas tout est possible, et si on veut se limiter aux classes qui dérivent de Super :

```
public void afficheS(Paire<? extends Super>p) {  
    System.out.println(p);  
}
```

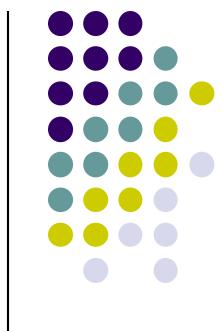


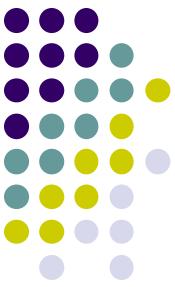
Joker (Wildcards)

- Les jokers peuvent être limités vers le haut (*extends*) ou vers le bas (*super*).
- On a les relations d'héritage suivantes :

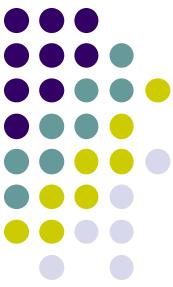


- La relation d'héritage entre *Paire* et *Paire<?>* est là pour assurer la compatibilité avec les codes existants avant la version 5 de Java.
- Toutes les collections en Java 5 sont génériques.
 - Interfaces et classes ont été remplacées par des versions génériques





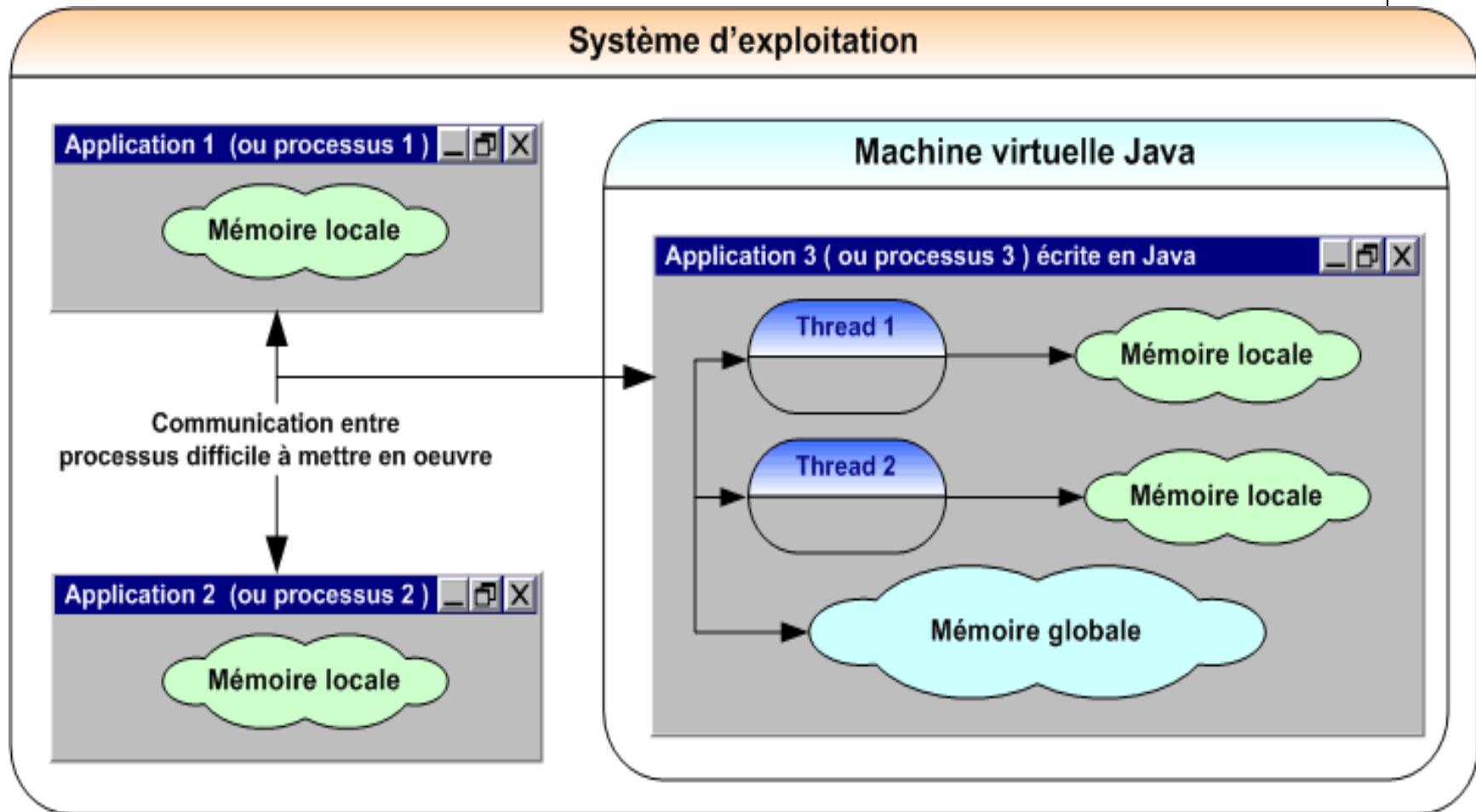
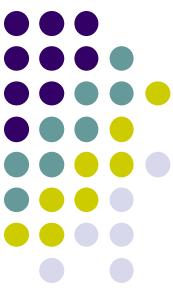
Les Threads



Qu'est-ce qu'un Thread ?

- les threads sont différents des processus :
 - ils partagent code, données et ressources : « processus légers »
 - mais peuvent disposer de leurs propres données.
 - ils peuvent s'exécuter en "parallèle"

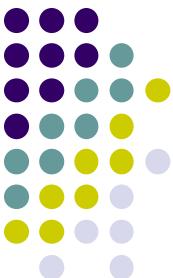
Qu'est-ce qu'un Thread ?





Création

- La classe `java.lang.Thread` permet de créer de nouveaux threads
- Un thread doit implémenter obligatoirement l'interface `Runnable`
 - le code exécuté se situe dans sa méthode `run()`
- 2 méthodes pour créer un Thread :
 - 1) une classe qui dérive de `java.lang.Thread`
 - `java.lang.Thread` implémente `Runnable`
 - il faut redéfinir la méthode `run()`
 - 2) une classe qui implémente l'interface `Runnable`
 - il faut implémenter la méthode `run()`



Methode 1 : Sous-classer Thread

```
class Procl extends Thread {  
Procl() {...} // Le constructeur  
...  
public void run() {  
... // Ici ce que fait le processus  
}  
}  
...  
Procl p1 = new Procl(); // Création du processus p1  
p1.start(); // Demarre le processus et execute p1.run()
```



La sortie de la méthode « run » met fin à la vie du Thread.



Méthode 2 : une classe qui implémente Runnable

```
class Proc2 implements Runnable {  
    public Thread monThread ;  
    Proc2() { ...  
        Thread monThread = new Thread(this);  
        ...  
    } // Constructeur  
    ...  
    public void run() {  
        ... // Ici ce que fait le processus  
    }  
    ...  
    Proc2 p = new Proc2(); p.monThread.start() ;  
    Thread p2 = new Thread(p);  
    ...  
    p2.start(); // Démarre un processus qui execute p.run()
```

*Notez le this pour
associer le Thread à cette
classe*

Quelle solution choisir ?

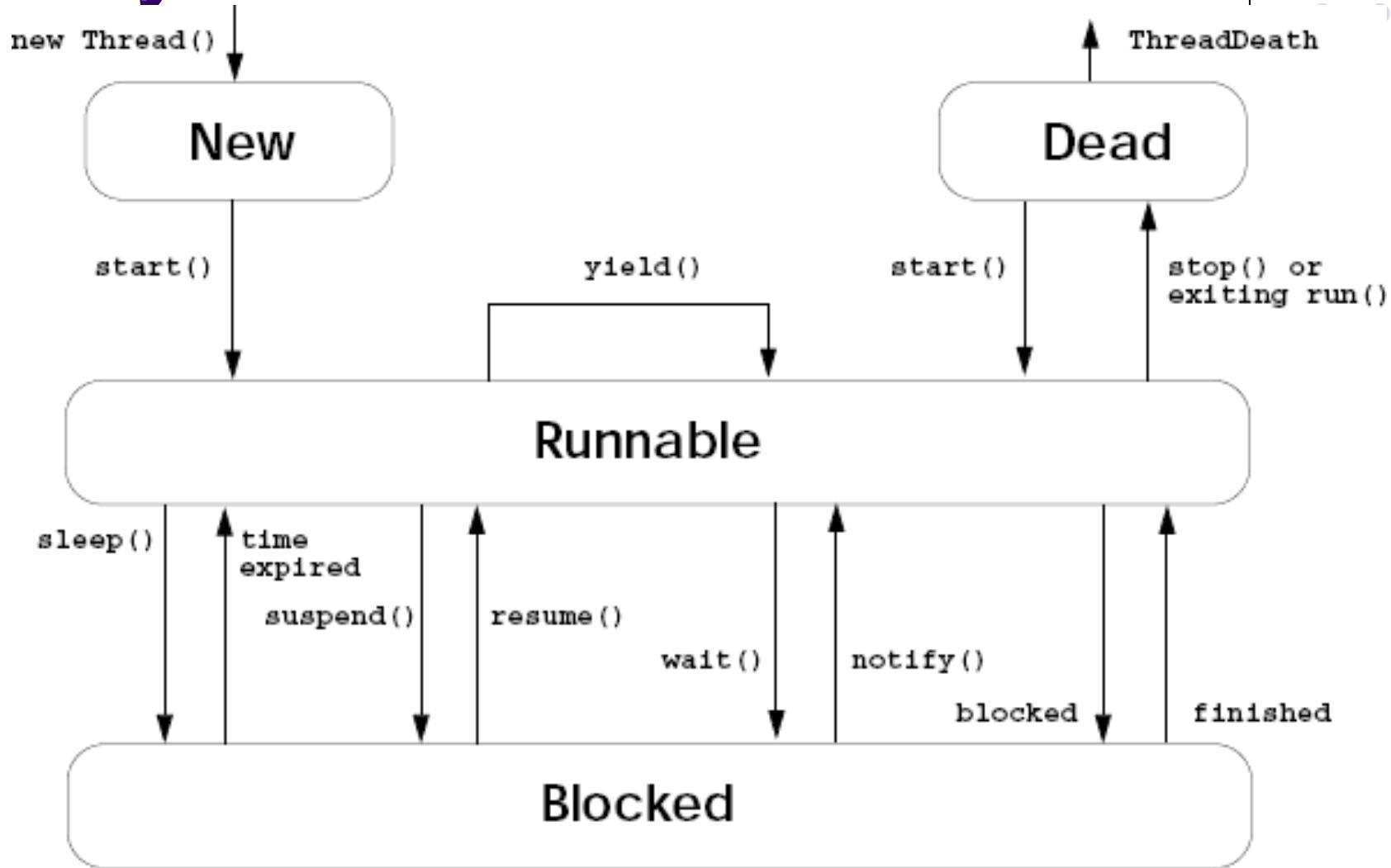


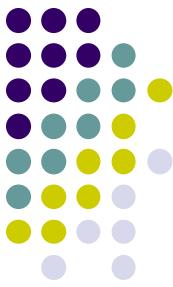
- Méthode 1 : sous-classer Thread
 - lorsqu'on désire paralléliser une classe qui n'hérite pas déjà d'une autre classe (attention : héritage simple)
 - cas des applications autonomes
- Méthode 2 : implémenter Runnable
 - lorsqu'une super-classe est imposée
 - cas des applets

```
public class MyThreadApplet
extends Applet implements Runnable { }
```
- *Distinguer la méthode run (qui est le code exécuté par l'activité) et la méthode start (méthode de la classe Thread qui rend l'activité exécutable) ;*
- *Dans la première méthode de création, attention à définir la méthode run avec strictement le prototype indiqué (il faut redéfinir Thread.run et non pas la surcharger).*



Le cycle de vie





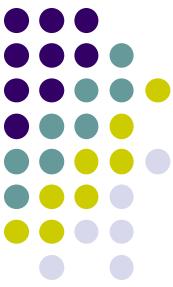
Les états d'un thread

- Crée :
 - comme n'importe quel objet Java
 - ... mais n'est pas encore actif
- Actif :
 - après la création, il est activé par start() qui lance run().
 - il est alors ajouté dans la liste des threads actifs pour être exécuté par l'OS en temps partagé
 - peut revenir dans cet état après un resume() ou un notify()

Exemple



```
public class ExempleThread extends java.lang.Thread
{
    public static int threadCompteur = 0;
    public int numThread = 0;
    public int count = 5;
    public ExempleThread()
    {
        numThread = ThreadCompteur++;
        System.out.println("Création du thread n°" + numThread );
    }
    public void run()
    {
        while ( count != 0 )
        {
            System.out.println("Thread n°" + numThread + " , compteur = " +
            count-- );
        }
    }
    public static void main( String [] args )
    {
        for ( int i=0; i<3; i++ )
            new ExempleThread().start();
        System.out.println("Tous les threads sont lancés");
    }
}
```



A l'exécution...

Création du thread n°1

Création du thread n°2

Création du thread n°3

Thread n°1, compteur = 5

Thread n°2, compteur = 5

Thread n°2, compteur = 4

Thread n°2, compteur = 3

Thread n°3, compteur = 5

Thread n°1, compteur = 4

Tous les threads sont lancés

Thread n°3, compteur = 4



L'ordonnancement est imprévisible.

...

Les états d'un Thread (suite)



- Endormi ou bloqué :
 - après sleep() : endormi pendant un intervalle de temps (ms)
 - suspend() endort le Thread mais resume() le réactive
 - une entrée/sortie bloquante (ouverture de fichier, entrée clavier) endort et réveille un Thread
- Mort :
 - si stop() est appelé explicitement
 - quand run() a terminé son exécution



Interruption et reprise d'un Thread

- On peut interrompre un thread par l'intermédiaire de l'opération **suspend()**.
- Pour relancer l'exécution d'un thread, on fait appel à la méthode **resume()**.
- On peut également marquer une pause dans l'exécution d'un thread en employant l'opération **sleep()**.
- La méthode **yield()** applique une pause à l'exécution du thread courant, afin de libérer le processeur pour d'autres threads en attente.
- Enfin, un thread peut attendre la fin d'un autre thread en appliquant l'opération **join()** sur le thread en question.
- Pour arrêter un thread on utilise l'opération **stop()** :
`public final void stop();`

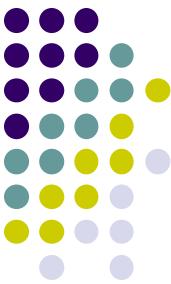
La méthode `sleep()` et la méthode `yield()` étant déclarée statique, ne peut être invoquée sur une instance de la classe `Thread`.



Exemple d'utilisation de sleep

```
class ThreadCompteur extends Thread {  
    int no_fin; int attente;  
    ThreadCompteur (int fin,int att) {  
        no_fin = fin; attente=att;}  
    // On redéfinit la méthode run()  
    public void run () {  
        for (int i=1; i<=no_fin ; i++) {  
            System.out.println(this.getName() + ":" + i);  
            try {sleep(attente);}  
            catch(InterruptedException e) {};  
        }  
    }  
    public static void main (String args[]) {  
        // On instancie les threads  
        ThreadCompteur cp1 = new ThreadCompteur (100,100);  
        ThreadCompteur cp2 = new ThreadCompteur (50,200);  
        cp1.start();  
        cp2.start();  
    } }
```

Les priorités



- Principes :
 - Java permet de modifier les priorités des Threads par la méthode setPriority()
 - Par défaut, chaque nouveau Thread a la même priorité que le Thread qui l'a créée
 - Rappel : seuls les Threads actifs peuvent être exécutés et donc accéder au CPU
 - La JVM choisit d'exécuter le Thread actif qui a la plus haute priorité : priority-based scheduling
 - si plusieurs Threads ont la même priorité, la JVM répartit équitablement le temps CPU (time slicing) entre tous : round-robin scheduling

Les priorités (suite)

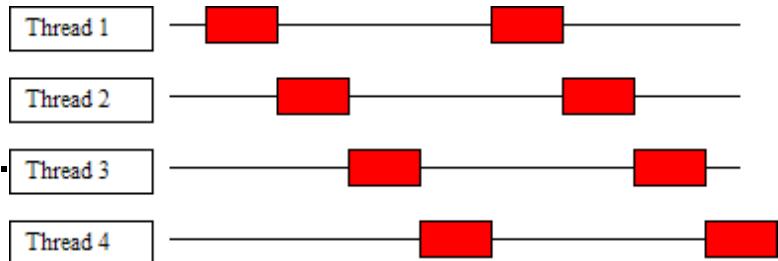


- Les méthodes :
 - `setPriority(int)` : fixe la priorité du receveur.
 - le paramètre doit appartenir à :
`[MIN_PRIORITY, MAX_PRIORITY]`
 - sinon `IllegalArgumentException` est levée
 - `int getPriority()` : pour connaître la priorité d'un Thread
 - `NORM_PRIORITY` : donne le niveau de priorité "normal"

La gestion du CPU



- Time-slicing (ou round-robin scheduling) :
 - La JVM répartit de manière équitable le CPU entre tous les threads de même priorité. Ils s'exécutent en "parallèle".
- Préemption (ou priority-based scheduling) :
 - Le premier thread du groupe des threads à priorité égale monopolise le CPU. Il peut le céder :
 - involontairement : sur entrée/sortie
 - volontairement : appel à la méthode statique `yield()`
Attention : ne permet pas à un thread de priorité inférieure de s'exécuter (seulement de priorité égale)
 - implicitement en passant à l'état endormi (`wait()`, `sleep()`, ou `suspend()`)





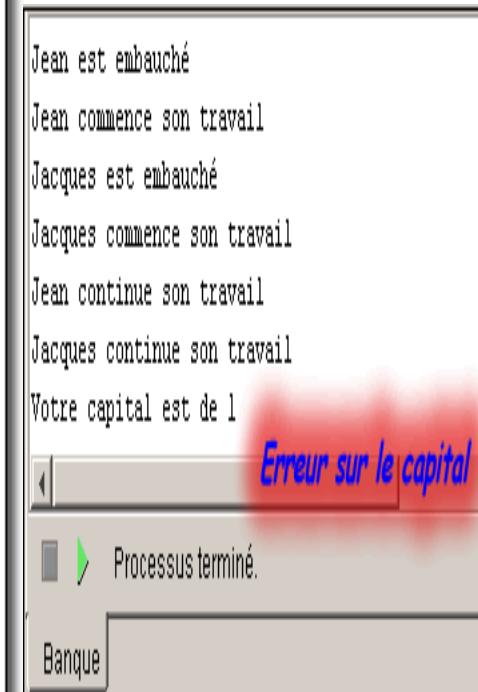
La concurrence d'accès

- Le problème : espace de travail commun, pas de "mémoire privée" pour chaque thread :
 - inconvénient : accès simultané à une même ressource.
Il faut garantir l'accès exclusif à un objet pendant l'exécution d'une ou plusieurs instructions
- Pour se faire : le mot-clé synchronized permet de gérer les concurrence d'accès :
 - d'une méthode
 - d'un objet
 - ou d'une instruction (ou d'un bloc)

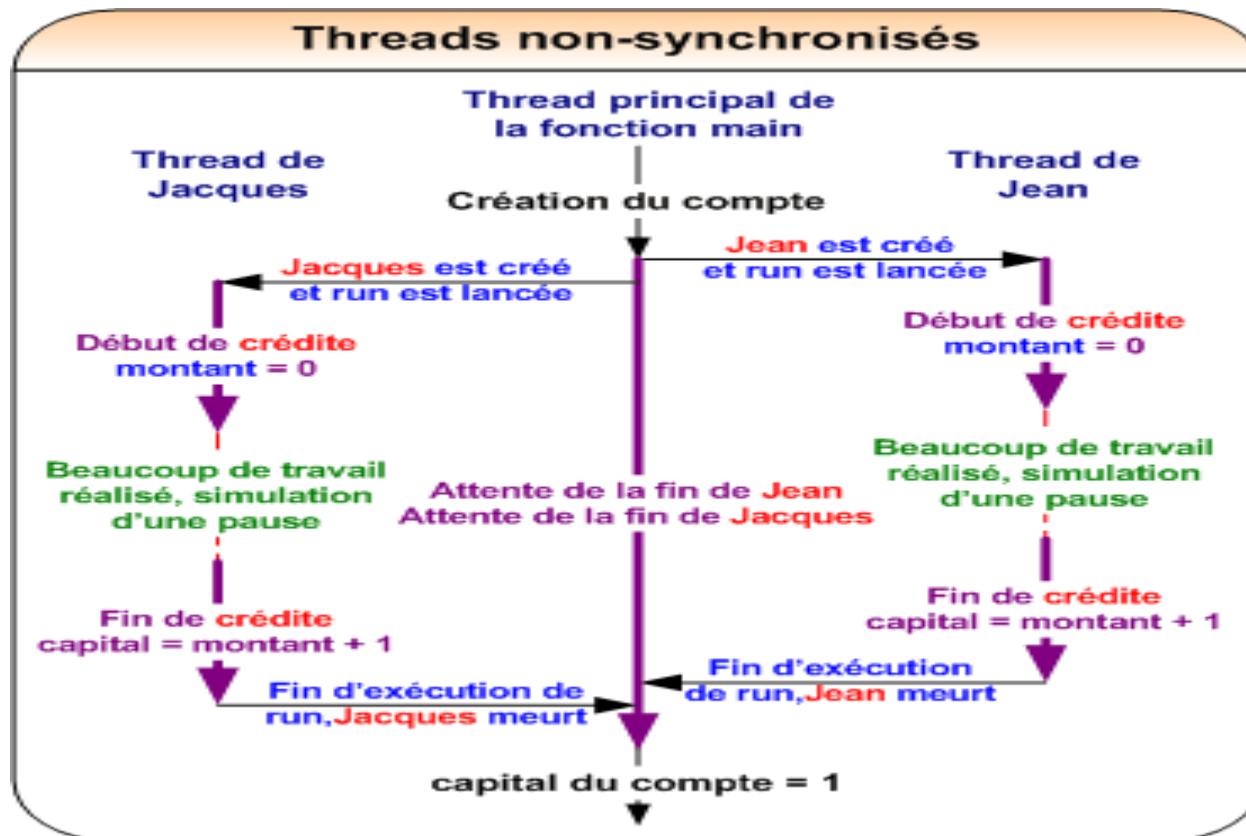
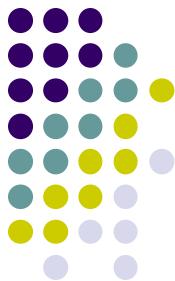


Exemple : Problématique

```
class Compte {  
    int capital = 0;  
    void crédite(Banquier banquier, int deCombien) {  
        int montant;  
        System.out.println(banquier.nom + " commence son travail");  
        montant = capital;  
        for (int i=0; i<100000000; i++); // beaucoup d'opérations à réaliser (simulation)  
        System.out.println(banquier.nom + " continue son travail");  
        capital = montant + deCombien;  
    }  
}  
  
class Banquier extends Thread {  
    Compte compte;  
    String nom;  
    Banquier(Compte compte, String nom) { this.compte = compte; this.nom = nom; }  
    public void run() {  
        System.out.println(nom + " est embauché");  
        compte.crédite(this, 1);  
    }  
}  
  
public class Banque {  
    public static void main(String[] args) {  
        Compte compte = new Compte();  
        Banquier Jean = new Banquier(compte, "Jean"); Jean.start();  
        Banquier Jacques = new Banquier(compte, "Jacques"); Jacques.start();  
        try {  
            Jean.join(); // attend que le thread Jean ait cessé d'être vivant  
            Jacques.join(); // attend que le thread Jacques ait cessé d'être vivant  
        }  
        catch (InterruptedException e) {}  
        System.out.println("Votre capital est de " + compte.capital);  
    }  
}
```



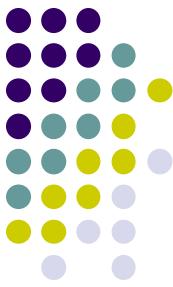
Exemple : Problématique (suite)





La synchronisation

- Basée sur la technique de l'exclusion mutuelle :
 - à chaque objet Java est associé un « verrou » géré par le thread quand une méthode (ou un objet) synchronized est accédée.
 - garantit l'accès exclusif à une ressource (la section critique) pendant l'exécution d'une portion de code.
- Une section critique :
 - une méthode : déclaration précédée de synchronized
 - une instruction (ou un bloc) : précédée de synchronized
 - un objet : le déclarer synchronized
- Attention à l'inter-blocage !!



Utiliser synchronized

- Pour gérer la concurrence d'accès à une méthode :
 - si un thread exécute cette méthode sur un objet, un autre thread ne peut l'exécuter pour le même objet
 - en revanche, il peut exécuter cette méthode pour un autre objet

```
public synchronized void maMethode() { . . . }
```

- Pour Contrôler l'accès à un objet :

```
public void maMethode() { . . .
    synchronized(objet) {
        objet.saMethode(); } }
```

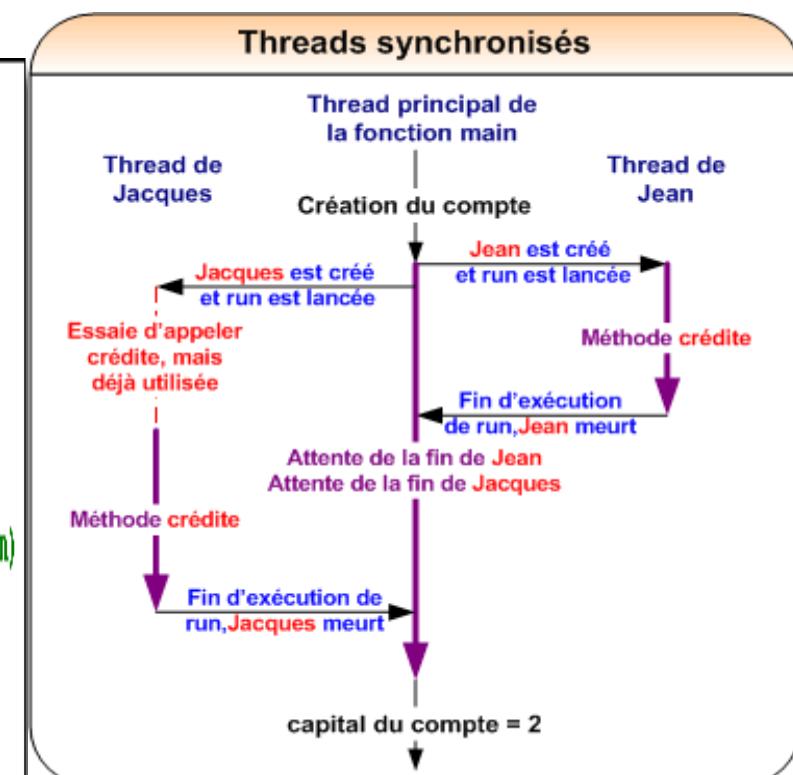
- l'accès à l'objet passé en paramètre de synchronized(Object) est réservé à un unique thread sur le bloc synchronisé.



Threads synchronisés

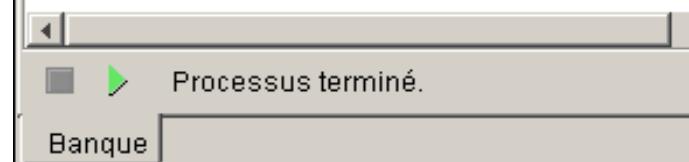
```

class Compte {
    int capital = 0;
    synchronized void crédite(Banquier banquier, int deCombien) {
        int montant;
        System.out.println(banquier.nom + " commence son travail");
        montant = capital;
        for (int i=0; i<100000000; i++); // beaucoup d'opérations à réaliser (simulation)
        System.out.println(banquier.nom + " continue son travail");
        capital = montant + deCombien;
    }
}
  
```



```

Jean est embauché
Jean commence son travail
Jacques est embauché
Jean continue son travail
Jacques commence son travail
Jacques continue son travail
Votre capital est de 2
  
```





Le JDK 1.2 et les opérations sur les threads

- Plusieurs opérations sont notés « **deprecated** » dans le JDK 1.2.
- En particulier les opérations « **suspend** » et « **resume** » ne doivent plus être utilisées.
 - Utiliser à la place une synchronisation à partir d'un verrou.
- De plus, l'opération « **stop** » est également déconseillée au profit de l'utilisation d'une variable :

```
public void run()
{
    while ( stop != true )
        { // ... }
}
public void stop()
{ stop = true; }
```



Les signaux

- ◆ Toutes les classes Java héritent de Object et disposent donc d 'un moniteur d 'accès à travers wait() et notify().
- ◆ Lorsqu'un processus accède à un objet il peut appeler la méthode wait() de cet objet. (monObjet.wait())
 - Relâche l 'accès à l 'objet et s 'endort.
 - attend une notification des autres processus.
 - redémarre dès qu'il obtient à nouveau l'accès exclusif
- ◆ Lorsqu 'un processus a «fini » avec un objet, il appelle sa méthode notify (). (monObjet.notify()).
 - Signale aux processus qui ont fait un wait, qu'ils peuvent se réveiller.
 - Ne relâche pas l 'accès (=>à faire en fin de bloc synchronisé).
 - Ne peut se faire que si on a eu l 'accès (dans un bloc synchronisé)
- ◆ Permet une synchronisation de type Producteur-Consommateur.

Exemple 1:

un producteur de nombres

- ◆ Dépose dans un tableau de taille limitée des nombres qui sont pris par des consommateurs.

```
class Producteur extends Thread {  
    public void run () {  
        while (true){  
            try {sleep (1000);} catch (InterruptedException e) {};  
            synchronized (table) {  
                if (table.estPleine()) try {table.wait();} catch (InterruptedException e) {};  
                table.ajoute((int)( Math.random()*1000));  
                table.notify();  
            }  
        }  
    }  
}
```

Section critique

Une pause d 'une seconde

Attente si table est pleine

Signale aux consommateurs et aux autres producteurs (!) de se réveiller...



Exemple 1:

un consommateur de nombres

- ◆ Prends dans un tableau de taille limitée des nombres qui y sont placés par des producteurs.

```
class Consommateur extends Thread {  
    public void run () {  
        while (true){  
            try {sleep (1000);} catch (InterruptedException e) {};  
            synchronized (table) {  
                if (table.estVide()) try {table.wait();} catch (InterruptedException e) {};  
                System.out.println (table.retire());  
                table.notify();  
            }  
        }  
    }  
}
```

Section critique

Une pause d 'une seconde

Attente si table est vide

Signale aux producteurs et aux
autres consommateurs (!) de se réveiller...



Daemons

- Un thread peut être déclarer comme daemon :
 - comme le "garbage collector", l'"afficheur d'images", ...
 - en général de faible priorité, il "tourne" dans une boucle infinie
 - arrêt implicite dès que le programme se termine
- Les méthodes :
 - `setDaemon()` : déclare un thread daemon
 - `isDaemon()` : ce thread est-il un daemon ?



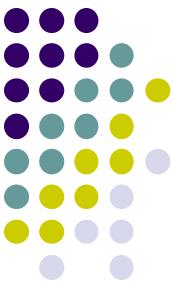
Les « ThreadGroup »

Pour contrôler plusieurs threads

- Plusieurs processus (Thread) peuvent s'exécuter en même temps, il serait utile de pouvoir les manipuler comme une seule entité
 - pour les suspendre
 - pour les arrêter, ...

Java offre cette possibilité via l'utilisation des groupes de threads : `java.lang.ThreadGroup`

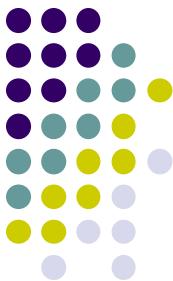
- on groupe un ensemble nommé de threads
- ils sont contrôlés comme une seule unité



Les groupes de threads

- Une arborescence :
 - la classe `ThreadGroup` permet de constituer une arborescence de Threads et de ThreadGroups
 - elle donne des méthodes classiques de manipulation récursives d'un ensemble de threads : `suspend()`, `stop()`, `resume()`, ...
 - et des méthodes spécifiques : `setMaxPriority()`, ...
- Fonctionnement :
 - la JVM crée au minimum un groupe de threads nommé `main`
 - par défaut, un thread appartient au même groupe que celui qui l'a créé (son père)
 - `getThreadGroup()` : pour connaître son groupe

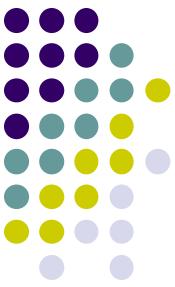
Création d'un groupe de threads



- Pour créer un groupe de processus :

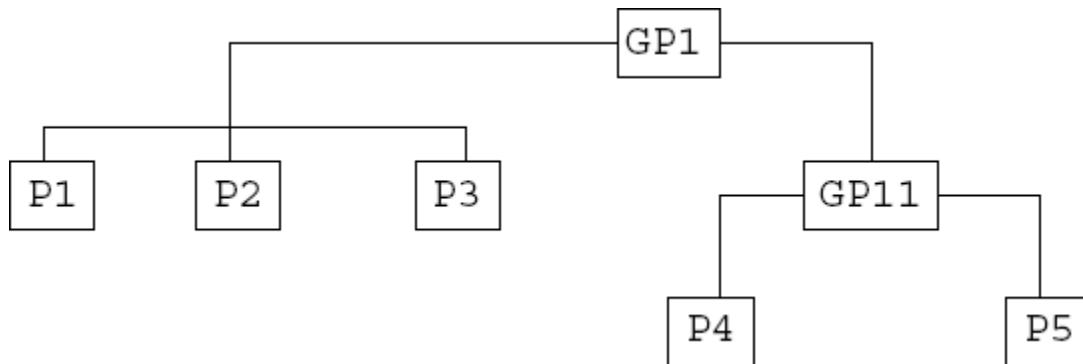
```
ThreadGroup groupe = new  
    ThreadGroup ("Mon groupe");  
  
Thread p1 = new Thread(groupe, "P1");  
Thread p2 = new Thread(groupe, "P2");  
Thread p3 = new Thread(groupe, "P3");
```

- On peut créer des sous-groupes de threads pour la création d'arbres sophistiqués de processus
 - des ThreadGroup contiennent des ThreadGroup
 - des threads peuvent être au même niveau que des ThreadGroup

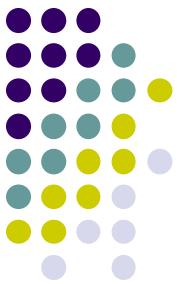


Création de groupe de threads (suite)

```
ThreadGroup groupe1 = new ThreadGroup("GP1");
Thread p1 = new Thread(groupe1, "P1");
Thread p2 = new Thread(groupe1, "P2");
Thread p3 = new Thread(groupe1, "P3");
ThreadGroup groupe11 = new ThreadGroup(groupe1,
    "GP11");
Thread p4 = new Thread(groupe11, "P4");
Thread p5 = new Thread(groupe11, "P5");
```



Contrôler les ThreadGroup



- Le contrôle des ThreadGroup passe par l'utilisation des méthodes standards qui sont partagées avec Thread :

`resume()`, `suspend()`, `stop()`, ...

- Par exemple : appliquer la méthode `stop()` à un ThreadGroup revient à invoquer pour chaque Thread du groupe cette même méthode

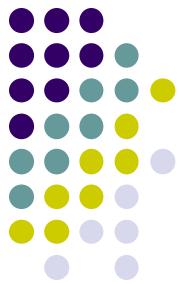
Avantages / Inconvénients des threads



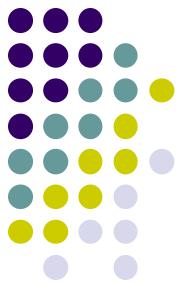
- Programmer facilement des applications où des traitements se résolvent de façon concurrente (applications réseaux, par exemple)
- Améliorer les performances en optimisant l'utilisation des ressources
- Code plus difficile à comprendre, peu réutilisable et difficile à débuguer

AWT

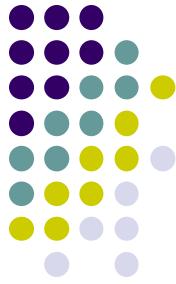
The Abstract Windowing Toolkit



Développement des interfaces graphiques



- Les interfaces graphiques assurent le dialogue entre les utilisateurs et une application. Cette partie contient les chapitres suivants :
- Le graphisme
 - Objets et méthodes de base pour le graphisme
- Les éléments d'interfaces graphiques de l'AWT
 - Différents composants fournis dans la bibliothèque AWT
 - Création d'interfaces graphiques avec AWT
- L'interception des actions de l'utilisateur



Le graphisme (1)

- **Le tracé de formes géométriques**
- À l'exception des lignes, toutes les formes peuvent être dessinées vides (méthode `drawXXX`) ou pleines (méthode `fillXXX`).
- La classe `Graphics` possède de nombreuses méthodes qui permettent de réaliser des dessins.

Méthode	Role	
drawRect(x, y, largeur, hauteur), fillRect(x, y, largeur, hauteur)	dessiner un carré ou un rectangle	
drawRoundRect(x, y, largeur, hauteur, hor_arr, ver_arr), fillRoundRect(x, y, largeur, hauteur, hor_arr, ver_arr)	dessiner un carré ou un rectangle arrondi	
drawLine(x1, y1, x2, y2)	Dessiner une ligne	
drawOval(x, y, largeur, hauteur), fillOval(x, y, largeur, hauteur)	dessiner un cercle ou une elipse en spécifiant le rectangle dans lequel ils s'inscrivent	
drawPolygon(int[], int[], int) fillPolygon(int[], int[], int)	<p>Dessiner un polygone ouvert ou fermé. Les deux premiers paramètres sont les coordonnées en abscisses et en ordonnées. Le dernier paramètre est le nombre de points du polygone. Pour dessiner un polygone fermé il faut joindre le dernier point au premier.</p> <div style="background-color: #d3d3d3; padding: 5px;"> Exemple (code Java 1.1) : <pre>int[] x={10,60,100,80,150,10}; int[] y={15,15,25,35,45,15}; g.drawPolygon(x,y,x.length); g.fillPolygon(x,y,x.length);</pre> </div> <p>Il est possible de définir un objet Polygon.</p> <div style="background-color: #d3d3d3; padding: 5px;"> Exemple (code Java 1.1) : <pre>int[] x={10,60,100,80,150,10}; int[] y={15,15,25,35,45,15}; Polygon p = new Polygon(x, y,x.length); g.drawPolygon(p);</pre> </div>	
drawArc(x, y, largeur, hauteur, angle_deb, angle_bal), fillArc(x, y, largeur, hauteur, angle_deb, angle_bal);	dessiner un arc d'ellipse inscrit dans un rectangle ou un carré. L'angle 0 se situe à 3 heures. Il faut indiquer l'angle de début et l'angle balayé	279



Le graphisme (2)

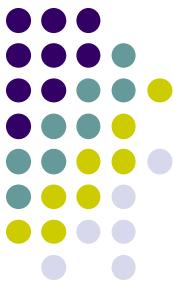
Le tracé de texte

- La méthode `drawString()` permet d'afficher un texte aux coordonnées précisées

Exemple (code Java 1.1) :

```
g.drawString(texte, x, y);
```

- Pour afficher des nombres int ou float, il suffit de les concatener à une chaîne éventuellement vide avec l'opérateur+.



Le graphisme (3)

- On dessine dans un composant awt en implémentant sa méthode **paint(g)**.
- l'argument g des méthodes de dessin est un objet de type **Graphics**.
- **repaint()** : redessine tout le composant en commençant par invoquer la méthode **update (Graphics g)** puis la méthode **paint (Graphics g)**.
- **update** est une méthode qui peint le composant graphique avec la couleur du fond afin de réaliser son effacement.



Le graphisme (4)

- Les méthodes **paint** et **repaint** sont appelées automatiquement quand le composant est affiché ou quand il doit être réaffiché (changement de taille).
- La manière dont l'affichage se déroule doit être décrit par la méthode **paint()**. Cette méthode est implémentée mais n'est pas invoquée par le programme. Elle l'est implicitement par le système.
- **repaint()** est appelée explicitement dans le programme mais n'est pas implémentée.
- On pourrait appeler **paint**, en récupérant d'abord le contexte avec **getGraphics()**. Mais si on veut redessiner (= ou dessiner dans) un composant à un autre moment, on utilisera en fait la méthode **repaint**.



```
import java.applet.*;
import java.awt.*;
public class anim1 extends Applet implements Runnable
{ Thread runner = null;      //variables de classe
  double t;
  public void init()
  { setBackground (Color.lightGray);}
  public void paint(Graphics g)
  { int X=140+(int)(120*Math.sin(t)); //calcul de la position
    t+=0.05;           //déplacement lors de l'appel suivant
    g.fillOval(X,20,50,100);}
  public void start() //surcharge de la méthode
  { if (runner == null) //test d'existence
    { runner = new Thread(this); //création , this désigne l'applet
      runner.start();}} // lancement de la méthode run( )
  public void stop()
  { if (runner != null)
    { runner.stop();
      runner = null;}} //destruction
  public void run()
  { while (true) // boucle sans fin
    { try           //action à réaliser
      { repaint(); //redessiner
        Thread.sleep(20);} //pause de 20 ms
      catch (InterruptedException e)
      { stop();}}}} //traitement (facultatif) de l'exception
}
```

Le graphisme (5)

L'utilisation des fontes



- La classe Font permet d'utiliser une police de caractères particulière pour afficher un texte.

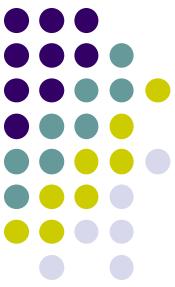
Exemple (code Java 1.1) :

```
Font fonte = new Font("TimesRoman",Font.BOLD,30);
```

- Le constructeur de la classe Font est Font(String, int, int). Les paramètres sont : le nom de la police (Dialog, Helvetica, TimesRoman, Courier, ZapfDingBats), le style (BOLD, ITALIC, PLAIN ou 0,1,2) et la taille des caractères en points.
- Pour associer plusieurs styles, il suffit de les additionner : Font.BOLD + Font.ITALIC
- La méthode getName() de la classe Font retourne le nom de la fonte.
- La méthode setFont() de la classe Graphics permet de changer la police d'affichage des textes

Exemple (code Java 1.1) :

```
Font fonte = new Font("TimesRoman",Font.BOLD,30);
g.setFont(fonte);
g.drawString("bonjour",50,50);
```



Le graphisme (6)

La gestion de la couleur

- La méthode `setColor()` permet de fixer la couleur des éléments graphiques des objets de type `Graphics` créés après à son appel.

Exemple (code Java 1.1) :

```
g.setColor(Color.black); // (green, blue, red, white, black, ...)
```

L'effacement d'une aire

- La méthode `clearRect(x1, y1, x2, y2)` dessine un rectangle dans la couleur de fond courante.

La copie d'une aire rectangulaire

- La méthode `copyArea(x1, y1, x2, y2, dx, dy)` permet de copier une aire rectangulaire. Les paramètres `dx` et `dy` permettent de spécifier un décalage en pixels de la copie par rapport à l'originale.

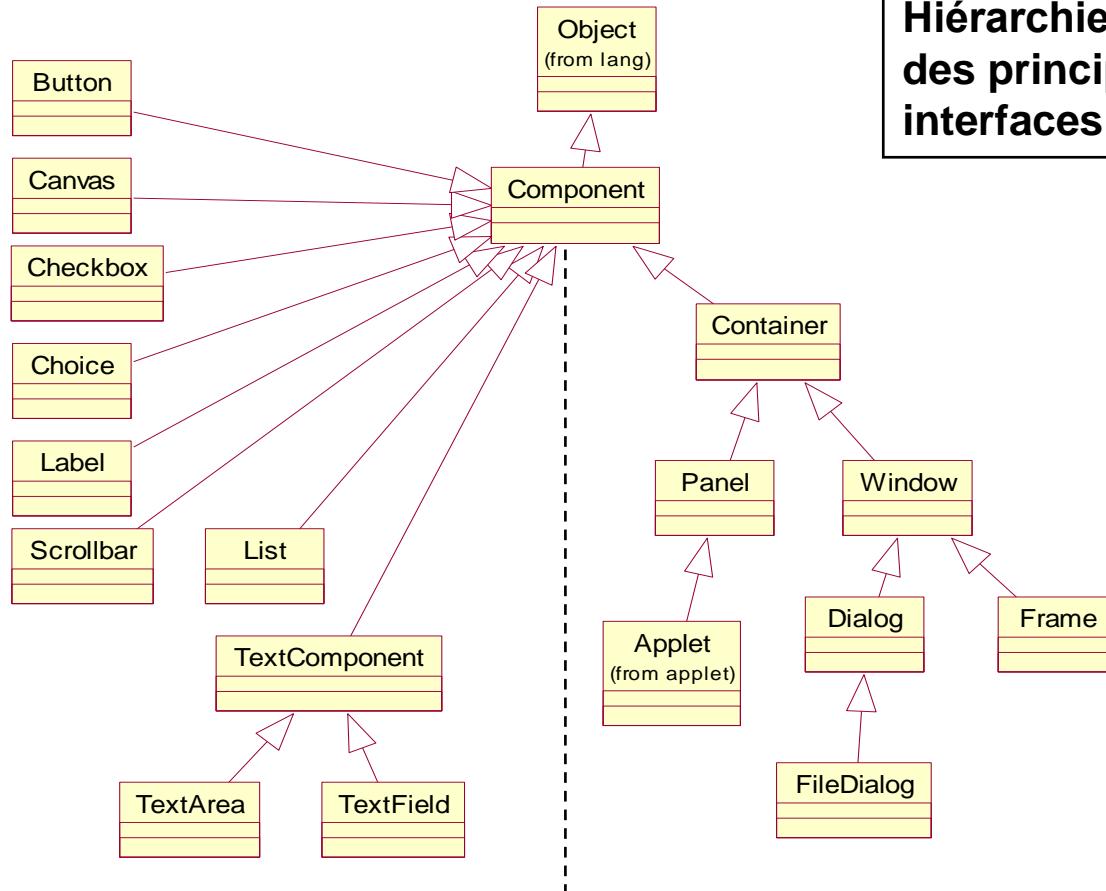


Conteneurs et composants (1)

- Une interface graphique en Java est un assemblage conteneurs (*Container*) et de composants (*Component*).
- **Un composant** est une partie "visible" de l'interface utilisateur Java.
 - C'est une sous-classes de la classe abstraite **java.awt.Component**.
 - Exemple : les boutons, les zones de textes ou de dessin, etc.
- **Un conteneur** est un espace dans lequel on peut positionner plusieurs composants.
 - Sous-classe de la classe **java.awt.Container**
 - La classe Container est elle-même une sous-classe de la classe Component
 - Par exemple les fenêtres, les applets, etc.



Conteneurs et composants (2)



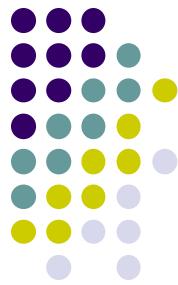
Hiérarchie d'héritage
des principaux éléments des
interfaces graphiques en Java



Conteneurs et composants (3)

- Les deux conteneurs les plus courants sont le **Frame** et le **Panel**.
- Un **Frame** représente une fenêtre de haut niveau avec un titre, une bordure et des angles de redimensionnement.
 - La plupart des applications utilisent au moins un **Frame** comme point de départ de leur interface graphique.
- Un **Panel** n'a pas une apparence propre et ne peut pas être utilisé comme fenêtre autonome.
 - Les **Panel** sont créés et ajoutés aux autres conteneurs de la même façon que les composants tels que les boutons
 - Les **Panel** peuvent ensuite redéfinir une présentation qui leur soit propre pour contenir eux-mêmes d'autres composants.

Conteneurs et composants (5)



- On ajoute un composant dans un conteneur, avec la méthode add() :

```
Panel p = new Panel();
Button b = new Button();
p.add(b);
```

- De manière similaire, un composant est retiré de son conteneur par la méthode remove() :

```
p.remove(b);
```

- Un composant a (notamment) :
 - une taille préférée que l'on obtient avec **getPreferredSize()**
 - une taille minimum que l'on obtient avec **getMinimumSize()**
 - une taille maximum que l'on obtient avec **getMaximumSize()**



Conteneurs et composants (6)

```
import java.awt.*;  
  
public class EssaiFenetre1  
{  
    public static void main(String[] args)  
    {  
        Frame f = new Frame("Ma première fenêtre");  
        Button b = new Button("coucou");  
        f.add(b); ←  
        f.pack(); ←  
        f.show(); ←  
    }  
}
```



Création d'une fenêtre (un objet de la classe Frame) avec un titre

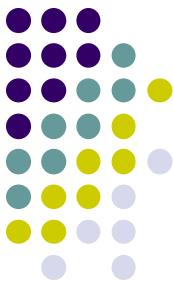
Création du bouton ayant pour label « coucou »

Ajout du bouton dans la fenêtre

On demande à la fenêtre de choisir la taille minimum avec pack() et de se rendre visible avec show()

Gestionnaire de présentation

(1)

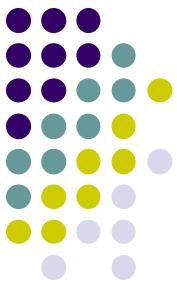


- A chaque conteneur est associé un gestionnaire de présentation (*layout manager*)
- Le gestionnaire de présentation gère le positionnement et le (re)dimensionnement des composants d'un conteneur.
- Le ré-agencement des composants dans un conteneur a lieu lors de :
 - la modification de sa taille,
 - le changement de la taille ou le déplacement d'un des composants.
 - l'ajout, l'affichage, la suppression ou le masquage d'un composant.
- Les principaux gestionnaires de présentation de l'AWT sont :
FlowLayout, BorderLayout, GridLayout, CardLayout, GridBagLayout

Gestionnaire de présentation (2)

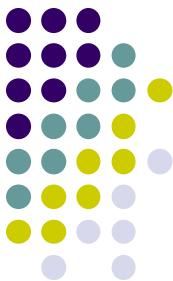


- Tout conteneur possède un gestionnaire de présentation par défaut.
 - Tout instance de *Container* référence une instance de *LayoutManager*.
 - Il est possible d'en changer grâce à la méthode **setLayout()**.
- Les gestionnaires de présentation par défaut sont :
 - Le **BorderLayout** pour **Window** et ses descendants (**Frame**, **Dialog**, ...)
 - Le **FlowLayout** pour **Panel** et ses descendants (**Applet**, etc.)
- Une fois installé, un gestionnaire fonctionne "tout seul" en interagissant avec le conteneur.



FlowLayout (1)

- Le FlowLayout est le plus simple des managers de l'AWT
- Gestionnaire de présentation utilisé par défaut dans les Panel si aucun LayoutManager n'est spécifié.
- Un FlowLayout peut spécifier :
 - une justification à gauche, à droite ou centrée,
 - un espacement horizontal ou vertical entre deux composants.
 - Par défaut, les composants sont centrés à l'intérieur de la zone qui leur est allouée.

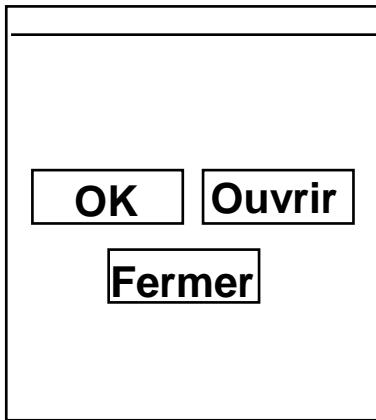


FlowLayout (2)

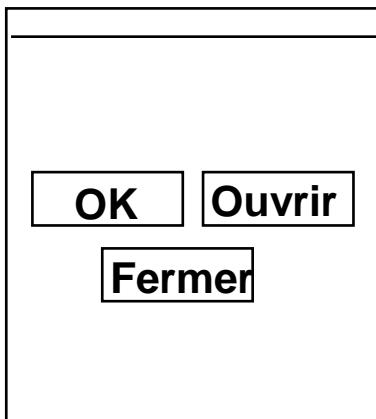
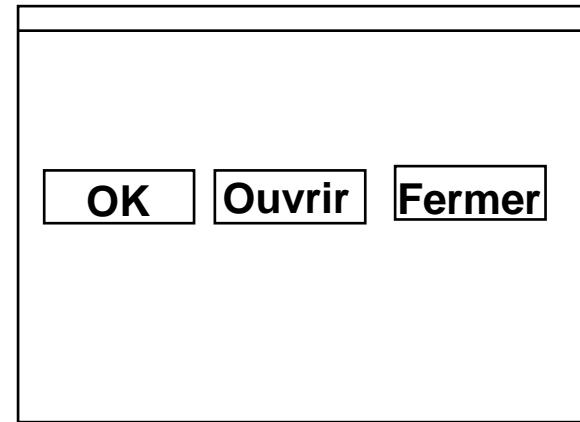
- La stratégie de disposition du FlowLayout est la suivante :
 - Respecter la **taille préférée** de tous les composants contenus.
 - Disposer autant de composants que l'on peut en faire tenir horizontalement à l'intérieur de l'objet **Container**.
 - Commencer une nouvelle rangée de composants si on ne peut pas les faire tenir sur une seule rangée.
 - Si tous les composants ne peuvent pas tenir dans l'objet **Container**, ce n'est pas géré (c'est-à-dire que les composants peuvent ne pas apparaître).



FlowLayout (3)



Redimensionnement



Redimensionnement



plus visible



FlowLayout (4)

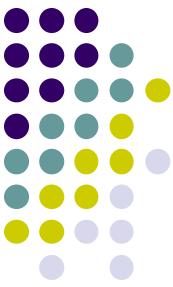


Redimensionnement

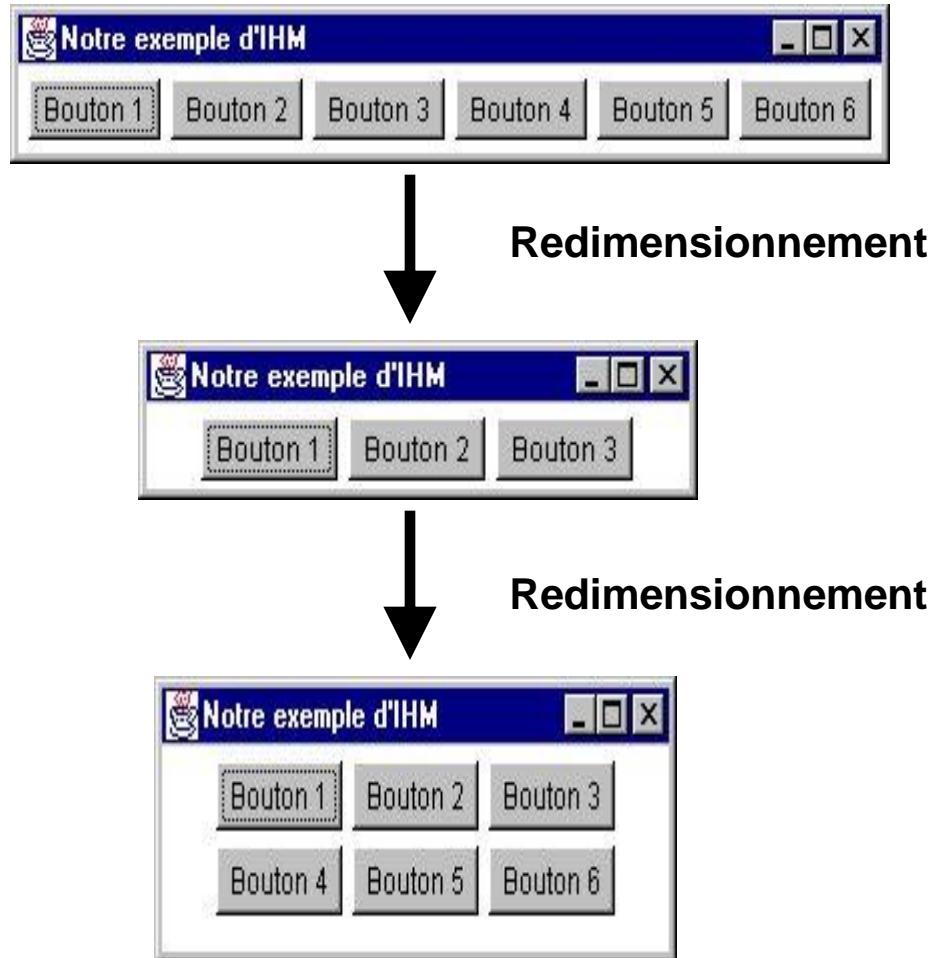


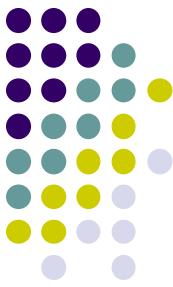
Redimensionnement





FlowLayout (5)





FlowLayout (6)

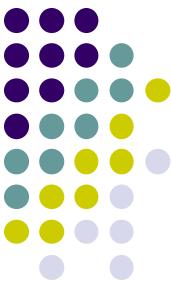
- Le FlowLayout cache réellement et effectivement les composants qui ne rentrent pas dans le cadre.
- Le FlowLayout n'a d'intérêt que quand il y a peu de composants.
- L'équivalent vertical du FlowLayout n'existe pas
- La présentation FlowLayout positionne les composants ligne par ligne.
 - Chaque fois qu'une ligne est remplie, une nouvelle ligne est commencée.
- Le gestionnaire FlowLayout n'impose pas la taille des composants mais leur permet d'avoir la taille qu'ils préfèrent.



BorderLayout (1)

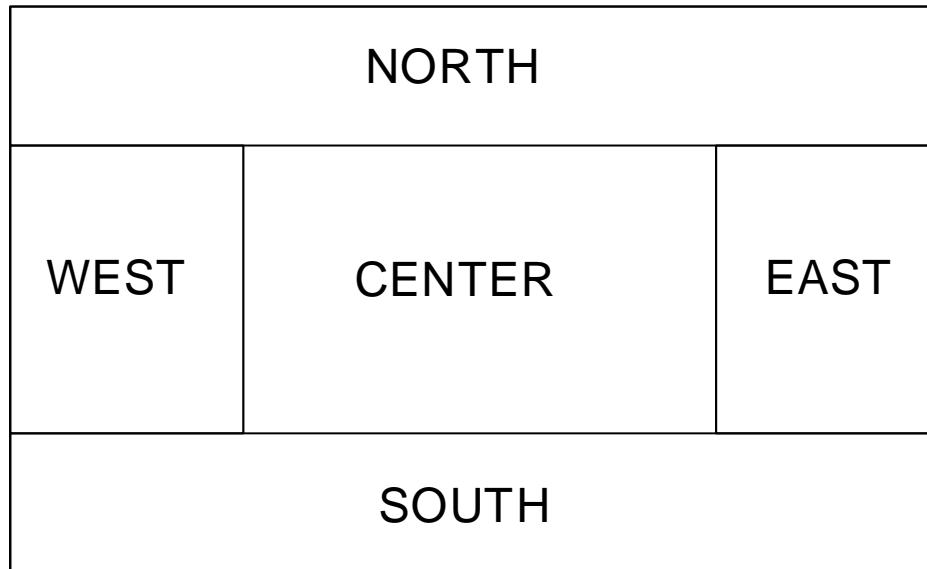
- BorderLayout divise son espace de travail en cinq zones géographiques : North, South, East, West et Center.
- Les composants sont ajoutés par nom à ces zones (un seul composant par zone).
 - Exemple

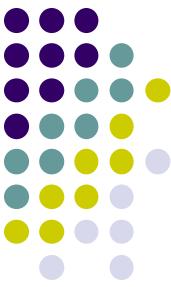
```
add("North", new Button("Le bouton nord !"));
```
 - Si une des zones de bordure ne contient rien, sa taille est 0.



BorderLayout (2)

- Division de l'espace avec le BorderLayout

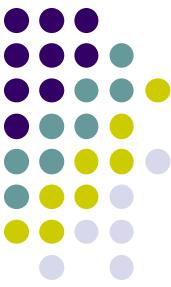




BorderLayout (3)

```
import java.awt.*;  
  
public class EssaiBorderLayout extends Frame  
{  
    private Button b1,b2,b3,b4, b5;  
    public EssaiBorderLayout() {  
        setLayout(new BorderLayout());  
        b1 = new Button ("Nord"); b2 = new Button ("Sud");  
        b3 = new Button ("Est"); b4 = new Button ("Ouest");  
        b5 = new Button ("Centre");  
        this.add(b1, BorderLayout.NORTH);  
        this.add(b2 , BorderLayout.SOUTH);  
        this.add(b3, BorderLayout.EAST);  
        this.add(b4, BorderLayout.WEST);  
        this.add(b5, BorderLayout.CENTER);  
    }  
    public static void main (String args []) {  
        EssaiBorderLayout essai = new EssaiBorderLayout();  
        essai.pack (); essai.setVisible(true) ;  
    }  
}
```





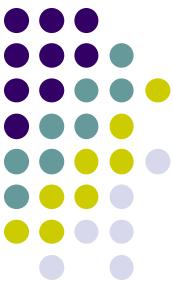
BorderLayout (4)

- Stratégie de disposition du BorderLayout
 - S'il y a un composant dans la partie placée dans la partie **NORTH**, il récupère sa ***taille préférée***, respecte sa ***hauteur préférée*** si possible et fixe sa largeur à la totalité de la largeur disponible de l'objet Container.
 - S'il y a un composant dans la partie placée dans la partie **SOUTH**, il fait pareil que dans le cas de la partie **NORTH**.
 - S'il y a un composant dans la partie placée dans la partie **EAST**, il récupère sa ***taille préférée***, respecte sa ***largeur préférée*** si possible et fixe sa hauteur à la totalité de la hauteur encore disponible.
 - S'il y a un composant dans la partie placée dans la partie **WEST**, il fait pareil que dans le cas de la partie **EAST**.
 - S'il y a un composant dans la partie **CENTER**, il lui donne la place qui reste, s'il en reste encore.

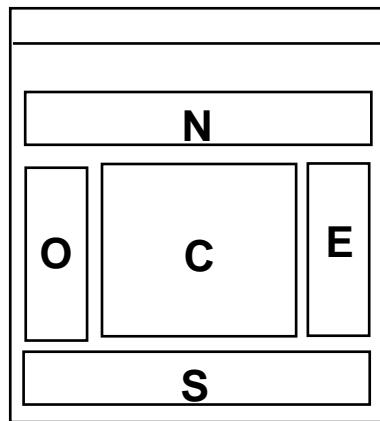


BorderLayout (5)

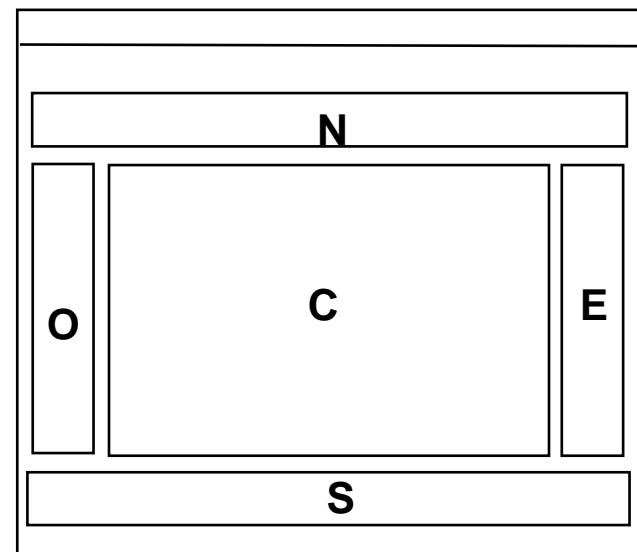
- Lors du redimensionnement, le composant est lui-même redimensionné en fonction de la taille de la zone, c-à-d :
 - les zones nord et sud sont éventuellement élargies mais pas allongées.
 - les zones est et ouest sont éventuellement allongées mais pas élargies,
 - la zone centrale est étirée dans les deux sens.



BorderLayout (6)

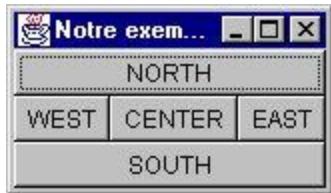


Redimensionnement
→

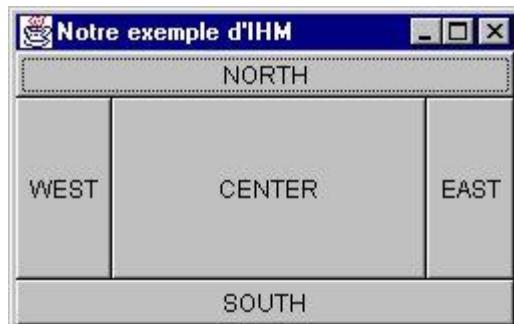




BorderLayout (7)

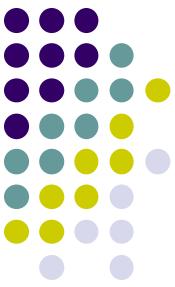


Redimensionnement



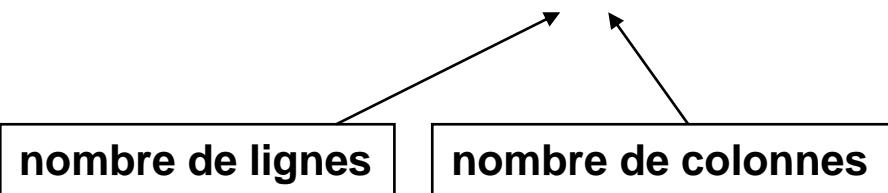
Redimensionnement





GridLayout (1)

- Le GridLayout dispose les composants dans une grille.
 - Découpage de la zone d'affichage en lignes et en colonnes qui définissent des cellules de dimensions égales.
 - Chaque composant à la même taille
 - quand ils sont ajoutés dans les cellules le remplissage s'effectue de gauche à droite et de haut en bas.
 - Les 2 paramètres sont les rangées et les colonnes.
 - Construction d'un GridLayout : **new GridLayout(3,2);**



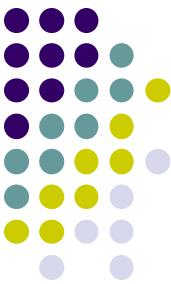


GridLayout (2)

```
import java.awt.*;
public class ApplGridLayout extends Frame
{
    public ApplGridLayout()
    {
        super("ApplGridLayout");
        this.setLayout(new GridLayout(3,2));
        for (int i = 1; i < 7; i++)
            add(new Button(Integer.toString(i)));
        this.pack();
        this.show();
    }

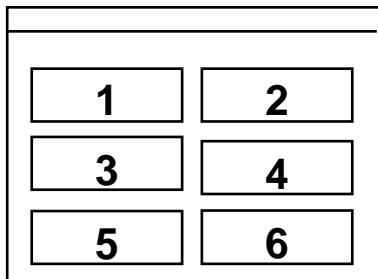
    public static void main(String args[])
    {
        ApplGridLayout appl = new ApplGridLayout();
    }
}
```



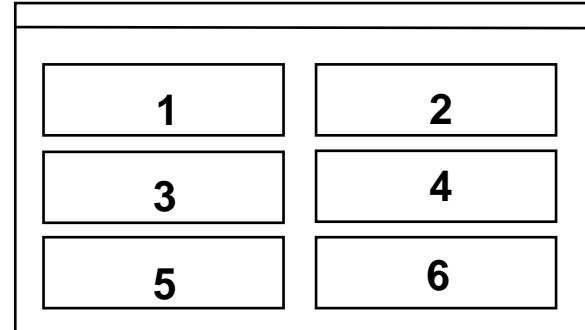


GridLayout (3)

- Lors d'un redimensionnement les composants changent tous de taille mais leurs positions relatives ne changent pas.



Redimensionnement



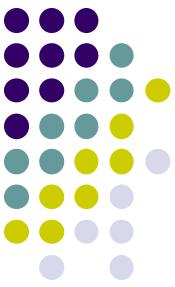


GridLayout (4)



Redimensionnement





CardLayout

- Le CardLayout n'affiche qu'un composant à la fois :
 - les composants sont considérées comme empilées, à la façon d'un tas de cartes.
- La présentation CardLayout permet à plusieurs composants de partager le même espace d'affichage de telle sorte que seul l'un d'entre eux soit visible à la fois.
- Pour ajouter un composant à un conteneur utilisant un CardLayout il faut utiliser `add(Component monComposant, String cle)`
- Permet de passer de l'affichage d'un composant à un autre en appelant les méthodes **first**, **last**, **next**, **previous** ou **show**



```
import java.awt.*;
import java.awt.event.*;
public class CardTest implements ActionListener
{
Panel p1, p2, p3, p4, p5;
Label l1, l2, l3, l4, l5;
Button monBouton;
CardLayout myCard;
Frame f;
Panel panneauCentral;
public static void main (String arg):
CardTest ct = new CardTest();
ct.init();
}
public void init() {
f = new Frame("Card Test");
panneauCentral = new Panel();
myCard = new CardLayout();
panneauCentral.setLayout(myCard);
p1 = new Panel(); p2 = new Panel();
p3 = new Panel();
p4 = new Panel(); p5 = new Panel();
```

```
l1 = new Label("Premiere carte");
p1.setBackground(Color.yellow); p1.add(l1);
l2 = new Label("Seconde carte");
p2.setBackground(Color.green); p2.add(l2);
l3 = new Label("Troisieme carte");
p3.add(l3);
rte");
:); p4.add(l4);
te");
); p5.add(l5);
");
ind");
d");
th");
i");
. "First");
al);
r passer au suivant:");
(this);
```

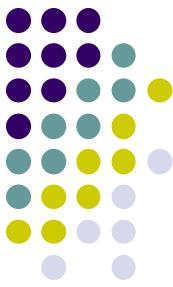


```
i.setVisiblity(true, s
public void actionPerformed(ActionEvent e){
myCard.next (panneauCentral); }
```



GridBagLayout (1)

- Le gestionnaire GridBagLayout fournit des fonctions de présentation complexes
 - basées sur une grille dont les lignes et les colonnes sont de taille variables.
 - permet à des composants simples de prendre leur taille préférée au sein d'une cellule, au lieu de remplir toute la cellule.
 - permet aussi l'extension d'un même composant sur plusieurs cellules.
- Le GridBagLayout est compliqué à gérer.
 - Dans la plupart des cas, il est possible d'éviter de l'utiliser en associant des objets Container utilisant des gestionnaires différents.



GridLayout (2)

- Le gestionnaire GridLayout est associé à un objet GridBagConstraints
 - l'objet **GridBagConstraints** définit des contraintes de positionnement, d'alignements, de taille, etc. d'un composant dans un conteneur géré par un **GridLayout**.
 - On associe chaque composant que l'on place dans le **GridLayout** avec un objet **GridBagConstraints**
 - Un même objet **GridBagConstraints** peut-être associé à plusieurs composants.
 - Définir les objets **GridBagConstraints** en spécifiant les différents paramètres est assez fastidieux...

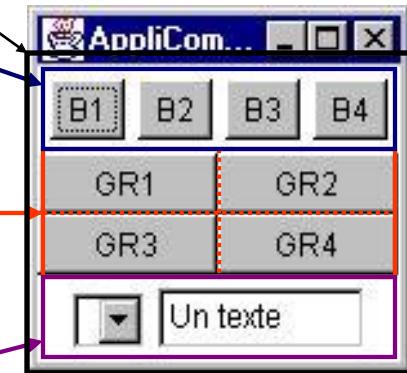


Mise en forme complexe (1)

```
super("AppliComplexeLayout");
setLayout(new BorderLayout());
Panel pnorth = new Panel();
pnorth.add(b1); pnorth.add(b2);
pnorth.add(b3); pnorth.add(b4);
this.add(pnorth,BorderLayout.NORTH);

Panel pcenter = new Panel();
pcenter.setLayout(new GridLayout(2,2));
pcenter.add(gr1); pcenter.add(gr2);
pcenter.add(gr3); pcenter.add(gr4);
this.add(pcenter,BorderLayout.CENTER);

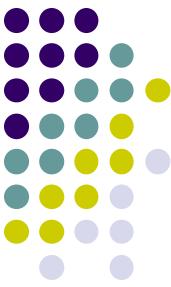
Panel psouth = new Panel();
psouth.setLayout(new FlowLayout());
psouth.add(ch); psouth.add(tf);
this.add(psouth, BorderLayout.SOUTH);
```





D'autres gestionnaires?

- On peut imposer à un objet « container » de n'avoir pas de gestionnaire en fixant son LayoutManager à la valeur `null`
 - **Frame f = new Frame(); f.setLayout(null);**
 - A la charge alors du programmeur de positionner chacun des composants « manuellement » en indiquant leur position absolue dans le repère de la fenêtre.
 - C'est à éviter, sauf dans des cas particuliers.,
- Il est possible d'écrire ses propres LayoutManager...



Récapitulatif

- FlowLayout
 - Flux : composants placés les uns derrière les autres
- BorderLayout
 - Ecran découpé en 5 zones (« North », « West », « South », « East », « Center »)
- GridLayout
 - Grille : une case par composant, chaque case de la même taille
- CardLayout
 - « Onglets » : on affiche un élément à la fois
- GridBagLayout
 - Grille complexe : plusieurs cases par composant

Les événements graphiques

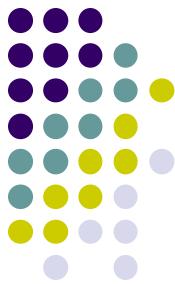
(1)



- L'utilisateur effectue
 - une action au niveau de l'interface utilisateur (clic souris, sélection d'un item, etc)
 - alors un **événement graphique** est émis.
- Lorsqu'un événement se produit
 - il est reçu par le composant avec lequel l'utilisateur interagit (par exemple un bouton, un curseur, un champ de texte, etc.).
 - Ce composant transmet cet événement à un autre objet, un **écouteur** qui possède une méthode pour traiter l'événement (on parle de traitement d'événement)
 - cette méthode reçoit l'objet événement généré de façon à traiter l'interaction de l'utilisateur.

Les événements graphiques

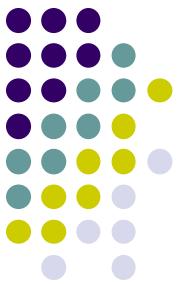
(2)



- La gestion des événements passe par l'utilisation d'objets "écouteur d'événements" (les *Listener*) et d'objets sources d'événements.
 - Un objet écouteur est l'instance d'une classe implémentant l'interface **XXXXListener**.
 - Une source d'événements est un objet pouvant recenser des objets écouteurs et leur envoyer des objets événements.
- Lorsqu'un événement se produit,
 - la source d'événements envoie un objet événement correspondant à tous ses écouteurs.
 - Les objets écouteurs utilisent alors l'information contenue dans l'objet événement pour déterminer leur réponse.

Les événements graphiques

(3)



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

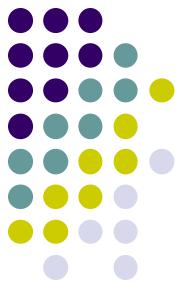
class MonAction implements ActionListener {
    public void actionPerformed (ActionEvent e) {
        System.out.println ("Une action a eu lieu") ;
}

public class TestBouton {
    public TestBouton(){
        Frame f = new Frame ("TestBouton");
        Button b = new Button ("Cliquer ici");
        f.add (b) ;
        f.pack (); f.setVisible (true) ;
        b.addActionListener (new MonAction ());
    }

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

Les événements graphiques

(4)



- Les écouteurs sont des interfaces
- Donc une même classe peut implémenter plusieurs interfaces écouteur.
 - Par exemple une classe héritant de Frame implémentera les interfaces MouseMotionListener (pour les déplacements souris) et MouseListener (pour les clics souris).
- Chaque composant de l'AWT est conçu pour être la source d'un ou plusieurs types d'événements particuliers.
 - Cela se voit notamment grâce à la présence dans la classe de composant d'une méthode nommée addXXXListener().

Les événements graphiques

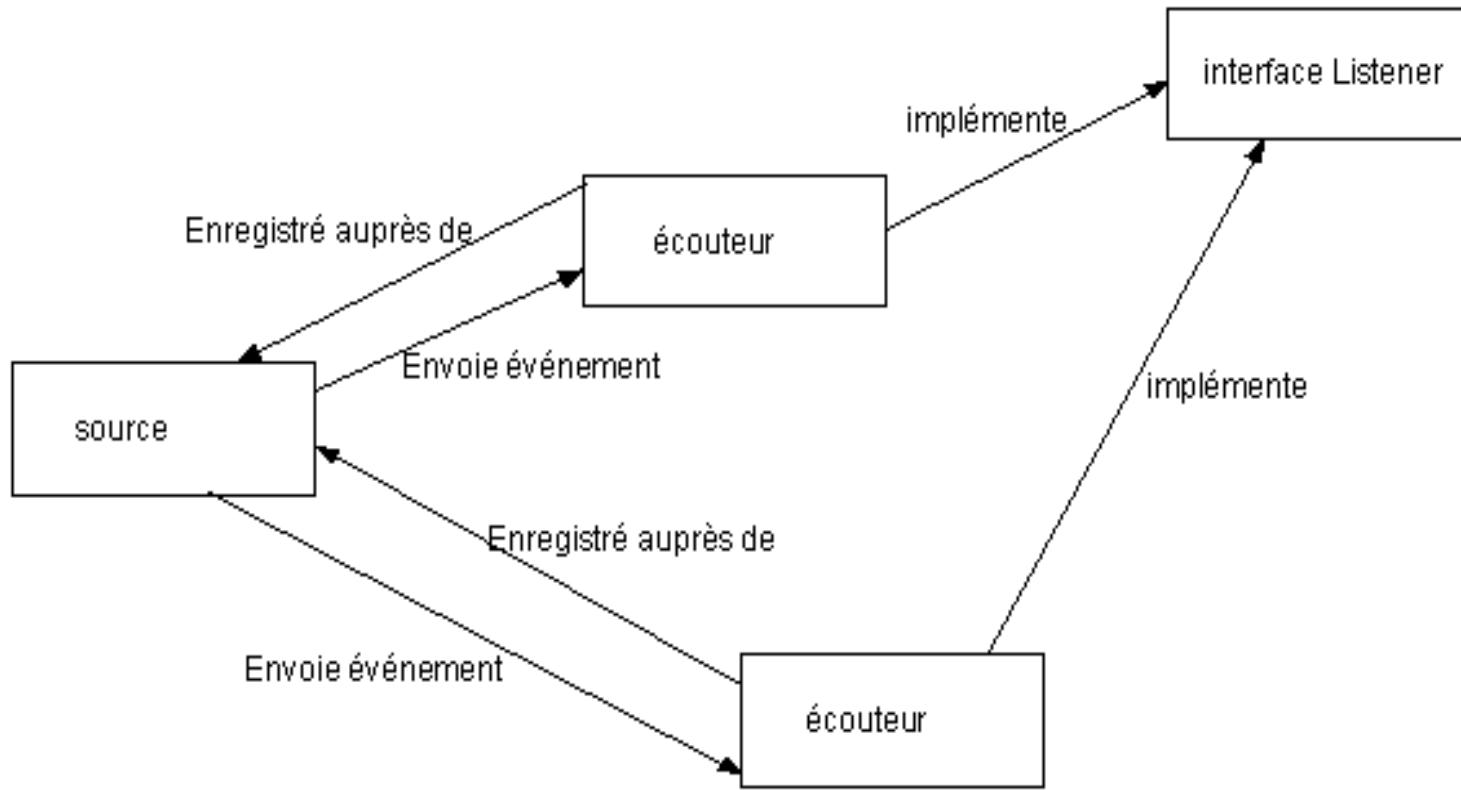
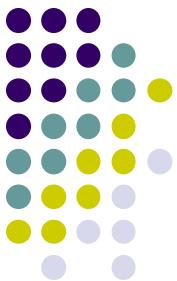
(5)



- L'objet événement envoyé aux écouteurs et passé en paramètres des fonctions correspondantes peut contenir des paramètres intéressants pour l'application.
 - Par exemple, `getX()` et `getY()` sur un `MouseEvent` retournent les coordonnées de la position du pointeur de la souris.
 - Une information généralement utile quelque soit le type d'événement est la source de cet événement que l'on obtient avec la méthode `getSource()`.

Les événements graphiques

(6)



Catégories d'événements graphiques (1)



- Plusieurs types d'événements sont définis dans le package `java.awt.event`.
- Pour chaque catégorie d'événements, il existe une interface qui doit être définie par toute classe souhaitant recevoir cette catégorie événements.
 - Cette interface exige aussi qu'une ou plusieurs méthodes soient définies.
 - Ces méthodes sont appelées lorsque des événements particuliers surviennent.

Catégories d'événements graphiques (2)



Catégorie	Nom de l'interface	Méthodes
Action	ActionListener	actionPerformed (ActionEvent)
Item	ItemListener	itemStateChanged (ItemEvent)
Mouse	MouseMotionListener	mouseDragged (MouseEvent) mouseMoved (MouseEvent)
Mouse	MouseListener	mousePressed (MouseEvent) mouseReleased (MouseEvent) mouseEntered (MouseEvent) (MouseEvent) mouseExited mouseClicked
Key	KeyListener	keyPressed (KeyEvent) keyReleased (KeyEvent) keyTyped (KeyEvent)
Focus	FocusListener	focusGained (FocusEvent) focusLost (FocusEvent)

Catégories d'événements graphiques (3)



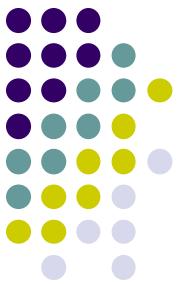
Adjustment	AdjustmentListener	adjustmentValueChanged (AdjustmentEvent)
Component	ComponentListener	componentMoved (ComponentEvent) componentHidden (ComponentEvent) componentResize (ComponentEvent) componentShown (ComponentEvent)
Window	WindowListener	windowClosing (WindowEvent) windowOpened (WindowEvent) windowIconified (WindowEvent) windowDeiconified (WindowEvent) windowClosed (WindowEvent) windowActivated (WindowEvent) windowDeactivated (WindowEvent)
Container	ContainerListener	componentAdded (ContainerEvent) componentRemoved(ContainerEvent)
Text	TextListener	textValueChanged (TextEvent)



Catégories d'événements graphiques (4)

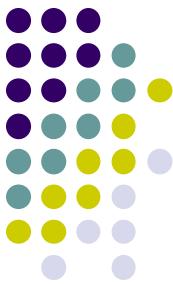
- ActionListener
 - Action (clic) sur un bouton, retour chariot dans une zone de texte, « tic d'horloge » (Objet Timer)
- WindowListener
 - Fermeture, iconisation, etc. des fenêtres
- TextListener
 - Changement de valeur dans une zone de texte
- ItemListener
 - Sélection d'un item dans une liste

Catégories d'événements graphiques (5)



- **MouseListener**
 - Clic, enfoncement/relâchement des boutons de la souris, etc.
- **MouseMotionListener**
 - Déplacement de la souris, drag&drop avec la souris, etc.
- **AdjustmentListener**
 - Déplacement d'une échelle
- **ComponentListener**
 - Savoir si un composant a été caché, affiché ...
- **ContainerListener**
 - Ajout d'un composant dans un Container

Catégories d'événements graphiques (6)



- FocusListener
 - Pour savoir si un élément a le "focus"
- KeyListener
 - Pour la gestion des événements clavier

Catégories d'événements graphiques (7)



```
import java.awt.*;
import java.awt.event.*;
public class EssaiActionEvent1 extends Frame
    implements ActionListener
{
    public static void main(String args[])
    {EssaiActionEvent1 f= new EssaiActionEvent1();}
    public EssaiActionEvent1()
    {
        super("Utilisation d'un ActionEvent");
        Button b = new Button("action");
        b.addActionListener(this);
        add(BorderLayout.CENTER,b);pack();show();
    }
    public void actionPerformed( ActionEvent e )
    {
        setTitle("bouton cliqué !");
    }
}
```



Implémentation de l'interface ActionListener

On enregistre l'écouteur d'evt action auprès de l'objet source "b"

Lorsque l'on clique sur le bouton dans l'interface, le titre de la fenêtre change



Catégories d'événements graphiques (8)



```
public class EssaiActionEvent2 extends Frame  
    implements ActionListener  
{ private Button b1,b2;  
    public static void main(String args[])  
    {EssaiActionEvent2 f= new EssaiActionEvent2();}  
    public EssaiActionEvent2(){  
        super("Utilisation d'un ActionEvent");  
        b1 = new Button("action1");  
        b2 = new Button("action2");  
        b1.addActionListener(this);  
        b2.addActionListener(this);  
        add(BorderLayout.CENTER,b1);  
        add(BorderLayout.SOUTH,b2);  
        pack();show(); }  
  
    public void actionPerformed( ActionEvent e ) {  
        if (e.getSource() == b1) setTitle("action1 cliqué");  
        if (e.getSource() == b2) setTitle("action2 cliqué");  
    } }
```



Les 2 boutons ont le même écouteur (la fenêtre)

e.getSource()" renvoie l'objet source de l'événement. On effectue un test sur les boutons (on compare les références)



Catégories d'événements graphiques (9)



```
import java.awt.*; import java.awt.event.*;
public class WinEvt extends Frame
    implements WindowListener {
    public static void main(String[] args) {
        WinEvt f= new WinEvt();
        public WinEvt() {
            super("Cette fenêtre se ferme");
            addWindowListener(this);
            pack();show();}
        public void windowOpened(WindowEvent e){}
        public void windowClosing(WindowEvent e) {
            System.exit(0);}
        public void windowClosed(WindowEvent e){}
        public void windowIconified(WindowEvent e){}
        public void windowDeiconified(WindowEvent e){}
        public void windowActivated(WindowEvent e){}
        public void windowDeactivated(WindowEvent e){} }
```



Implémenter cette interface impose l'implémentation de bcp de méthodes

La fenêtre est son propre écouteur

WindowClosing() est appelé lorsque l'on clique sur la croix de la fenêtre

"System.exit(0)" permet de quitter une application java



Les adaptateurs (1)

- Les classes « adapteur » permettent une mise en œuvre simple de l'écoute d'événements graphiques.
 - Ce sont des classes qui implémentent les écouteurs d'événements possédant le plus de méthodes, en définissant un corps vide pour chacune d'entre elles.
 - Plutôt que d'implémenter l'intégralité d'une interface dont une seule méthode est pertinente pour résoudre un problème donné, une alternative est de sous-classer l'adaptateur approprié et de redéfinir juste les méthodes qui nous intéressent.
 - Par exemple pour la gestion des événements fenêtres...



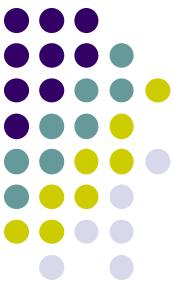
Les adaptateurs (2)

Solution en implémentant l'interface

```
class Terminator implements WindowListener
{
    public void windowClosing (WindowEvent e) {System.exit(0);}
    public void windowClosed (WindowEvent e) {}
    public void windowIconified (WindowEvent e) {}
    public void windowOpened (WindowEvent e) {}
    public void windowDeiconified (WindowEvent e) {}
    public void windowActivated (WindowEvent e) {}
    public void windowDeactivated (WindowEvent e)
}
```

Solution en utilisant un WindowAdapter

```
class Terminator extends WindowAdapter
{
    public void windowClosing (WindowEvent e) {System.exit(0);}
}
```



Les adapteurs (3)

- Il existe 7 classes d'adapteurs (autant que d'interfaces d'écouteurs possédant plus d'une méthode) :
 - **ComponentAdapter**
 - **ContainerAdapter**
 - **FocusAdapter**
 - **KeyAdapter**
 - **MouseAdapter**
 - **MouseMotionAdapter**
 - **WindowAdapter**



Les adapteurs (4)

- En pratique, et notamment avec la classe WindowAdapter, on utilise très souvent une classe anonyme

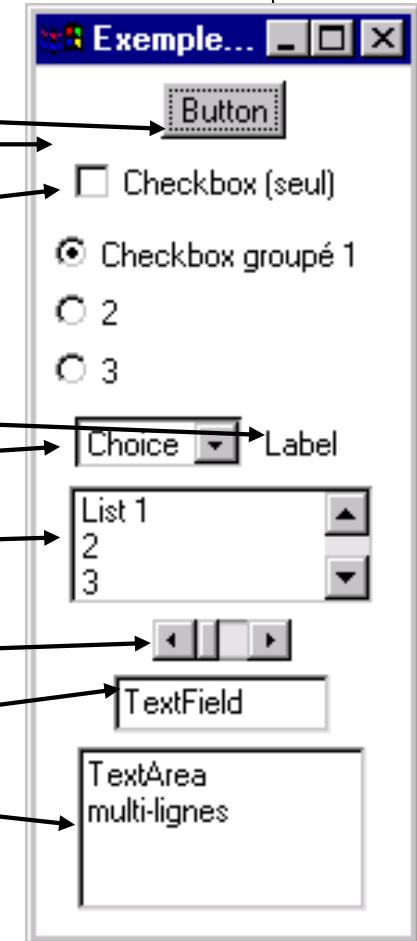
```
Frame f = new Frame("Machin")
f.addWindowListener(new WindowAdapter() {
    public void windowClosing(WindowEvent e)
    {
        System.exit(0);
    } });
}
```

Les composants graphiques

(1)



- Button
- Canvas (zone de dessin)
- Checkbox (case à cocher)
- CheckboxGroup
- Label
- Choice (Sélecteur)
- List
- Scrollbar (barre de défilement)
- TextField (zone de saisie d'1 ligne)
- TextArea (zone de saisie multilignes)



Les composants graphiques

(2)



- Button
 - C'est un composant d'interface utilisateur de base de type "appuyer pour activer".
 - Il peut être construit avec une étiquette de texte (un label) précisant son rôle à l'utilisateur.
 - Un objet de la classe Button est une source d>ActionEvent
 - Les écouteurs associés à des objets de la classe Button doivent implémenter interface ActionListener
 - Il n'y a qu'une méthode dans l'interface ActionListener, c'est la méthode public void actionPerformed(ActionEvent e).

```
Button b = new Button ("Sample") ;  
add (b) ;  
b.addActionListener (...) ;
```

Les composants graphiques

(3)



- CheckBox
 - La case à cocher fournit un dispositif d'entrée "actif / inactif" accompagné d'une étiquette de texte.
 - La sélection ou la désélection est notifiée par un ItemEvent à un écouteur implémentant l'interface ItemListener.
 - la méthode getStateChange() de ItemEvent retourne une constante : ItemEvent.DESELECTED ou ItemEvent.SELECTED.
 - le méthode getItem() de ItemEvent renvoie la chaîne contenant l'étiquette de la case à cocher considérée.

```
Checkbox one = new Checkbox("One", false);
add(one);
one.addItemListener(...);
```

Les composants graphiques

(4)



- CheckboxGroup
 - On peut regrouper des cases à cocher dans un CheckboxGroup pour obtenir un comportement de type boutons radio
 - On ne peut sélectionner qu'une seule case du groupe de cases à cocher en même temps.
 - Sélectionner une case fera que toute autre case précédemment cochée sera désélectionnée

```
CheckboxGroup cbg = new CheckboxGroup();
Checkbox one = new Checkbox("One", cbg, false);
...
add(one);
...
```

Les composants graphiques

(5)

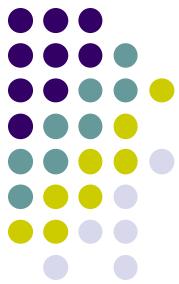


- Choice
 - Ce composant propose une liste de choix.
 - On ajoute des éléments dans l'objet Choice avec la méthode addItem(String nomItem).
 - La chaîne passée en paramètre sera la chaîne visible dans la liste
 - On récupère la chaîne de caractère correspondant à l'item actuellement sélectionné avec la méthode String getSelectedItem()
 - Cet objet est source de ItemEvent, l'écouteur correspondant étant un ItemListener

```
Choice c = new Choice();
c.addItem("First");
c.addItem("Second");
...
c.addItemListener (...);
```

Les composants graphiques

(6)

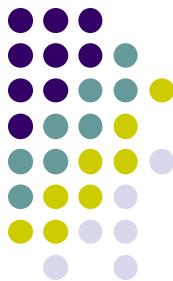


- Label
 - Un Label affiche une seule ligne de texte (étiquette) non modifiable.
 - En général, les étiquettes ne traitent pas d'événements.

```
Label l = new Label ("Bonjour !");  
add(l);
```

Les composants graphiques

(7)



- List
 - Un objet de la classe List permet de présenter à l'utilisateur plusieurs options de texte parmi lesquelles il peut sélectionner un ou plusieurs éléments.
 - Source d'ActionEvent et d'ItemEvent
 - méthode String getSelectedItem() et String[] getSelectedItems() pour récupérer des items.

```
List l =new List (4, false);  
l.add("item1");
```

nombre d'items visibles
(ici 4 éléments seront
visible en même temps)

sélections multiples possibles ou non.
Ici, avec la valeur false, non possible

Les composants graphiques

(8)



- **TextField**
 - Le champ de texte est un dispositif d'entrée de texte sur une seule ligne.
 - Il est source d>ActionEvent
 - On peut définir un champ de texte comme étant éditable ou non.
 - Méthodes void setText(String text) et String getText() pour mettre ou retirer du texte dans le TextField

```
TextField f = new TextField ("Une ligne seulement ...", 30);  
add(f);
```

Texte par défaut mis
dans le TextField

Nombre de caractères visibles
dans le TextField

Les composants graphiques

(9)



- **TextArea**

- La zone de texte est un dispositif d'entrée de texte multi-lignes, multi-colonnes avec éventuellement la présence ou non de « scrollbars » (barres de défilement) horizontal et/ou vertical.
- Il peut être ou non éditable.
- Méthode setText(), getText() et append() (pour ajouter du texte à la fin d'un texte existant déjà dans le TextArea)

```
TextArea t = new TextArea ("Hello !", 4, 30,TextArea.SCROLLBARS_BOTH);  
add(t);
```

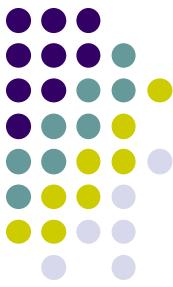
Texte par défaut mis
dans le TextArea

Nombre de lignes

Nombre de colonnes
(en nbre de caractères)

Valeur constante
précisant la
présence ou
l'absence de
« scrollbar »

Les composants graphiques (10)

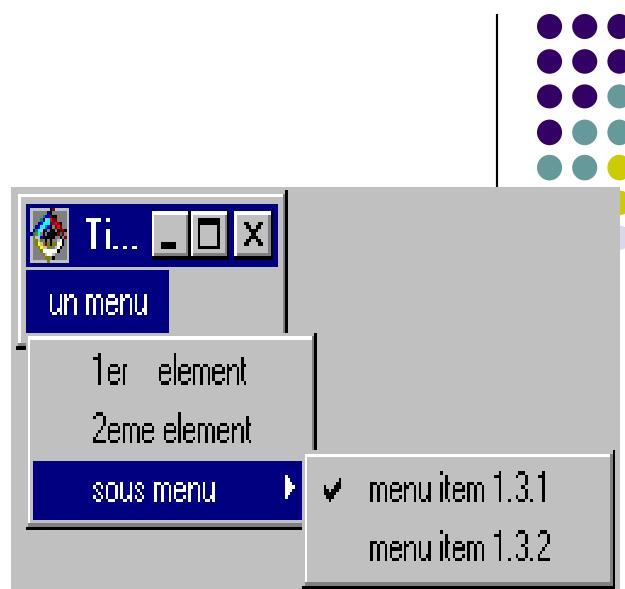


- Menu :
 - menu déroulant de base, qui peut être ajoutée à une barre de menus (MenuBar) ou à un autre menu.
 - Les éléments de ces menus sont
 - des MenuItem
 - ils sont rajoutés à un menu
 - En règle générale, ils sont associés à un ActionListener.
 - des CheckBoxMenuItem, ie. des éléments de menus à cocher
 - ils permettent de proposer des sélections de type "activé / désactivé " dans un menu.

Exemple (code Java 1.1) :

```
import java.awt.*;
public class MaFrame extends Frame {
    public MaFrame() {
        super();
        setTitle(" Titre de la Fenetre ");
        setSize(300, 150);
        MenuBar mb = new MenuBar();
        setMenuBar(mb);
        Menu m = new Menu(" un menu ");
        mb.add(m);
        m.add(new MenuItem(" 1er element "));
        m.add(new MenuItem(" 2eme element "));
        Menu m2 = new Menu(" sous menu ");
        CheckboxMenuItem cbm1 = new CheckboxMenuItem(" menu item 1.3.1 ");
        m2.add(cbm1);
        cbm1.setState(true);
        CheckboxMenuItem cbm2 = new CheckboxMenuItem(" menu item 1.3.2 ");
        m2.add(cbm2);
        m.add(m2);
        pack();
        show(); // affiche la fenetre
    }

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



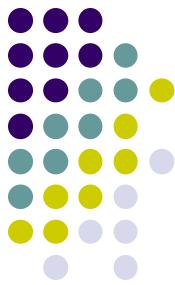
Les composants graphiques

(11)



- PopupMenu
 - des menus autonomes pouvant s'afficher instantanément sur un autre composant.
 - Ces menus doivent être ajoutés à un composant parent (par exemple un Frame), grâce à la méthode add(...).
 - Pour afficher un PopupMenu, il faut utiliser la méthode show(...).

Les composants graphiques (12)



- Canvas
 - Il définit un espace vide
 - Sa taille par défaut est zéro par zéro (Ne pas oublier de la modifier avec un setSize(...)) et il n'a pas de couleur.
 - pour forcer un canvas (ou tout autre composant) à avoir une certaine taille il faut redéfinir les méthodes getMinimumSize() et getPreferredSize().
 - On peut capturer tous les événements dans un Canvas.
 - Il peut donc être associé à de nombreux écouteurs : KeyListener, MouseMotionListener, MouseListener.
 - On l'utilise en général pour définir une zone de dessin

Les composants graphiques

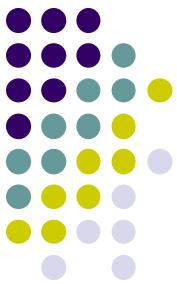
(13)



```
class Ctrait extends Canvas implements MouseListener
{
    Point pt;
    public Ctrait() { addMouseListener(this); }
    public void paint(Graphics g)
    {g.drawLine(0,0,pt.x,pt.y);
     g.setColor(Color.red);
     g.drawString((""+pt.x+" ; "+pt.y+""),pt.x,pt.y+5);}
    public Dimension getMinimumSize()
    {return new Dimension(200,100);}
    public Dimension getPreferredSize()
    {return getMinimumSize();}
    public void mouseClicked(MouseEvent e){}
    public void mousePressed(MouseEvent e){}
    public void mouseReleased(MouseEvent e)
    {pt=e.getPoint();repaint();}
    public void mouseEntered(MouseEvent e){}
    public void mouseExited(MouseEvent e){}}
```

Les composants graphiques

(14)



```
import java.awt.*;
import java.awt.event.*;
public class Dessin extends Frame
{
    public static void main(String[] args)
    {
        Dessin f= new Dessin();
    }
    public Dessin()
    {
        super("Fenêtre de dessin");
        Ctrait c= new Ctrait();
        add(BorderLayout.CENTER,c);
        pack();
        show();
    }
}
```

Utilisation de la classe précédente
(Ctrait) par une autre classe.



Les composants graphiques (15)



- Contrôle des couleurs d'un composant
 - Deux méthodes permettent de définir les couleurs d'un composant
 - `setForeground (Color c)` : la couleur de l'*encre* avec laquelle on écrira sur le composant
 - `setBackground (Color c)` : la couleur du fond
 - Ces deux méthodes utilisent un argument instance de la classe `java.awt.Color`.
 - La gamme complète de couleurs prédéfinies est listée dans la page de documentation relative à la classe `Color`.
 - Il est aussi possible de créer une couleur spécifique (RGB)

```
int r = 255, g = 255, b = 0 ;  
Color c = new Color (r, g, b) ;
```

Les composants graphiques (16)



- Contrôle des polices de caractères
 - La police utilisée pour afficher du texte dans un composant peut être définie avec `setFont(...)` avec comme argument une instance de `java.awt.Font`.
 - **Font f = new Font ("TimesRoman", Font.PLAIN, 14) ;**
 - Les constantes de style de police sont en réalité des valeurs entières, parmi celles citées ci-après :
 - **Font.BOLD**
 - **Font.ITALIC**
 - **Font.PLAIN**
 - **Font.BOLD + Font.ITALIC**
 - Les tailles en points doivent être définies avec une valeur entière.



Conteneurs particuliers (1)

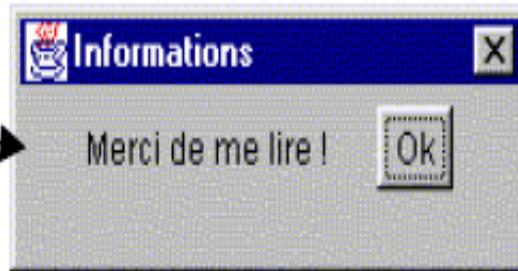


- Dialog
 - Un objet Dialog ressemble à un objet Frame mais ne sert qu'à afficher des messages devant être lus par l'utilisateur.
 - Il n'a pas de boutons permettant de le fermer ou de l'iconiser.
 - On y associe habituellement un bouton de validation.
 - Il est réutilisable pour afficher tous les messages au cours de l'exécution d'un programme.
 - Un objet Dialog dépend d'un objet Frame (ou héritant de Frame)
 - ce Frame est passé comme premier argument au constructeur).
 - Un Dialog n'est pas visible lors de sa création. Utiliser la méthode show() pour la rendre visible (il existe une méthode hide() pour la cacher).



```
import java.awt.*;
class AppliUnDialog2 extends Dialog
{
    public AppliUnDialog2(Frame mere)
    {
        super(mere,"Informations");
        Label etiq = new Label("Merci de me lire !");
        Button bout1 = new Button("Ok");
        setSize(200,100);
        setLayout(new FlowLayout());
        add(etiq);
        add(bout1);
        setVisible ( true );
    }
    public static void main(String[ ] args) {
        Frame fen = new Frame("Bonjour");
        AppliUnDialog2 dlg = new AppliUnDialog2(fen);
    }
}
```

voici l'effet visuel obtenu :





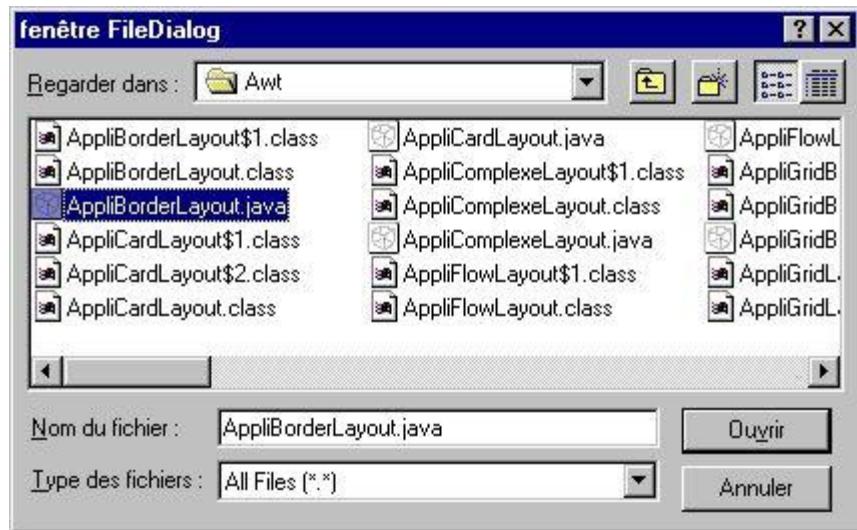
Conteneurs particuliers (2)

- `FileDialog`
 - C'est une sous-classe de `Dialog` ; par défaut elle n'est pas visible.
 - C'est un dispositif de sélection de fichier : on peut préciser si c'est en vue d'une sauvegarde ou du chargement d'un fichier
 - Un `FileDialog` ne gère généralement pas d'événements.
 - Comme pour un objet `Dialog`, un `FileDialog` dépend d'un objet `Frame`
 - Un `FileDialog` est une fenêtre modale : à partir du moment où la fenêtre a été rendue visible par la méthode `show(...)`, la main n'est rendu à l'utilisateur que quand un fichier a été sélectionné.



Conteneurs particuliers (3)

- **FileDialog**



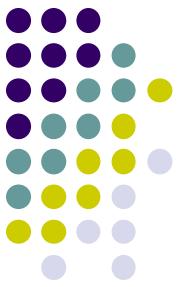


Conteneurs particuliers (4)

- ScrollPane
 - C'est un conteneur général de type Panel
 - tout comme un Panel il ne peut pas être utilisé de façon indépendante
 - il fournit des barres de défilement (scrollbars) verticales et/ou horizontales
 - Un ScrollPane ne peut contenir qu'un seul composant
 - Ce conteneur n'est pas très intéressant mais est cité pour information.
- Avec Swing que nous allons voir dans la suite, de nouveaux très intéressants composants font leur apparition...



Applet



Qu'est-ce qu'une applet ?

Une applet est une application Java :

- C'est une classe Java compilée sous forme de bytecode
- Elle dérive d'une classe mère : `java.applet.Applet`
- Elle est exécutée par une machine virtuelle Java(JVM)



Qu'est-ce qu'une applet ?

Elle possède néanmoins certaines particularités :

- Elle réside sur un serveur web
- Elle est véhiculée par une page HTML qui contient son URL
- Le navigateur Java-compatible qui charge cette page télécharge alors le code de l'applet (.class ou .jar) et l'exécute grâce à sa propre JVM ou une JVM extérieure dans le cas d'un plug-in



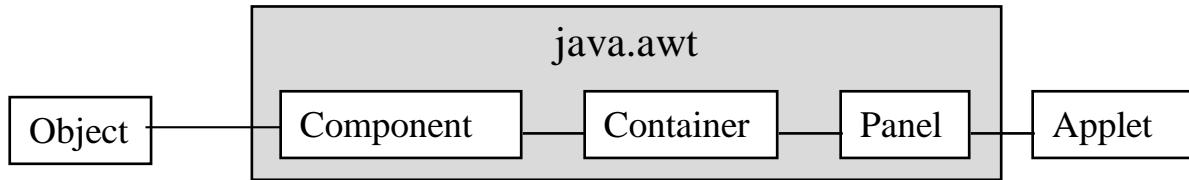
Applets et sécurité...

- Les conditions de sécurité sont plus strictes que pour une application ordinaire, et sont gérées par le navigateur lui-même :
 1. Pas d'accès au système de fichier de l'hôte
 2. Pas d'exécution de code natif sur l'hôte
 3. Communication restreinte avec le serveur d'origine



Applets généralités

- Une **applet** est une **Panel** spécialisée.



- Une applet n'est donc pas une application autonome : elle n'a pas la méthode `main()`.

Méthodes de gestion

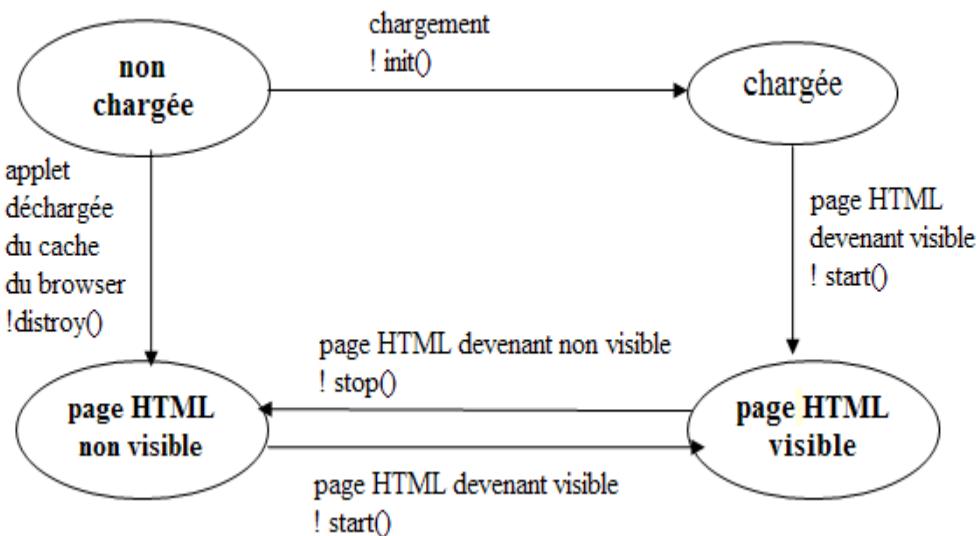
- `init()`
- `start()`
- `stop()`
- `destroy()`
- et `paint(Graphics g)`

héritées de la superclasse : `java.applet.Applet`



Méthodes d'Applets

- **public void init()**
 - appelée une seule fois par le navigateur à l'initialisation de l'applet, lorsqu'elle est chargée
- **public void start()**
 - appelée à chaque fois que l'applet devient visible
- **public void stop()**
 - appelée à chaque fois que l'applet est masquée
- **public void destroy()**
 - appelée une seule fois par le navigateur à la destruction de l'applet, lorsque la page HTML change





Méthodes d'Applets

- **public void paint (Graphics g) { ... }**
 - décrit la manière dont une applet affiche quelque chose à l'écran : texte, ligne, fond en couleur ou image.
- **Cycle de vie d'une applet :**

Init() --> start() --> paint() --> stop() --> destroy()

```
graph TD; Init[Init] --> Start[start]; Start --> Paint[paint]; Paint --> Stop[stop]; Stop --> Destroy[destroy];
```
- **Méthodes pour la régénération de l'applet :**
 - à Démarrage de génération : **repaint()**
 - à Effacement du composant graphique, peinture du composant avec la couleur du fond et appel de la fonction pain : **update()**

Repaint() --> update() --> paint()

Balise d'Applet

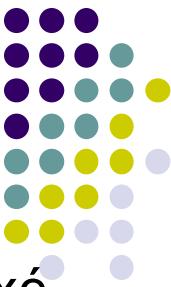


- Dans une page HTML, balises **<applet>** et **<param>**.
- Attributs de la balise **<applet>**:
name : donne un nom à une applette; permet de distinguer deux occurrences de la même applette;
codebase : adresse URL où se trouve le binaire, par défaut ".".
code : la classe de l'applette; le nom local, sans répertoire.
- Attributs de la balise **<param>**:
name : nom d'un paramètre
value : la valeur associée au paramètre.

- **Balisage minimal** : code, largeur, hauteur

```
<applet code=Bonjour  
        width=200  
        height=200>  
</applet>
```

```
<APPLET  
    [CODEBASE = url du répertoire]  
    CODE = fichier de l'applette  
    WIDTH = largeur du conteneur  
    HEIGHT = sa hauteur  
    [ALT = texte de remplacement]  
    [ARCHIVE = fichiers archives]  
    [NAME = nom de l'instance de l'applette ]  
    [ALIGN = top, middle, left (dans la page)]  
    [VSPACE = marges haute et basse]  
    [HSPACE = marges gauche et droite]>  
    [<PARAM NAME = nom VALUE = sa valeur>]  
    [< ... >]  
    ["Remplaçant-html" si balise APPLET inconnue]  
</APPLET>
```



Méthodes d'Applets

- **URL getCodeBase()** : le répertoire contenant le fichier suffixé **class** de l'*applet*.
- **URL getDocumentBase()** : le répertoire contenant le *fichier html* contenant la balise <applet>.
- **String getParameter(String)** : retourne la valeur associée au paramètre:

```
Image i = getImage(getDocumentBase(),getParameter("image"));
```



Obtenir une image

- La classe abstraite **java.awt.Image** étend **Object**.
- Les images sont obtenus, dans les applets, par la méthode
java.applet.Applet.getImage()
- Un producteur d'images (**ImageProducer**) les fournit,
- Un observateur d'images (**ImageObserver**) les récupère.
- L'interface **ImageObserver** est implémentée par **Component**, et tout composant graphique peut donc "observer" des images.





Afficher une image

- C'est un **Graphics** qui affiche l'image, par:
g.drawImage(image, . . . , observateur)
- L'observateur est le composant où s'affiche l'image.
- Il y a 6 variantes de **drawImage**, la plus simple indique l'origine de la zone d'affichage, la plus sophistiquée permet d'afficher une partie de l'image dans un rectangle spécifiée.

```
public void paint(Graphics g) {  
    g.drawImage(im,0,0,this);  
}
```

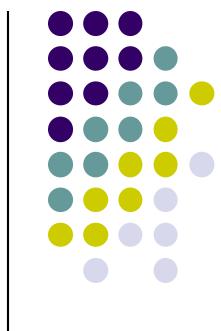


Un exemple (en applet)

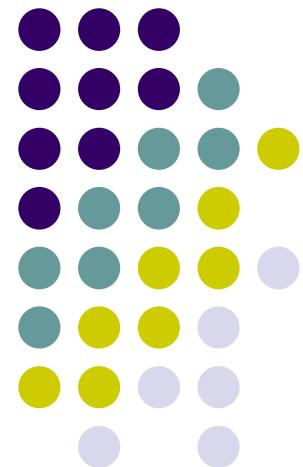
- Package `net` pour les URL.
- Les dimensions de l'applette sont indiquées dans le fichier `html`
- L'image est prise où elle se trouve.

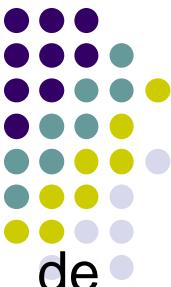
```
import java.net.URL;
import java.applet.Applet;
import java.awt.Graphics;
import java.awt.Image;

public class ImageTestAppletSimple extends Applet {
    private Image im;
    public void init() {
        URL cb = getCodeBase();
        im = getImage(cb, "saint.gif");
    }
    public void paint(Graphics g) {
        g.drawImage(im,0,0,this);
    }
}
```



Swing





Introduction à Swing (1)

- La bibliothèque Swing est une nouvelle bibliothèque de composants graphiques pour Java.
 - Swing est intégré à Java 1.2.
 - Swing peut être téléchargé séparément pour une utilisation avec des versions de Java antérieures (1.1.5+)
- Cette bibliothèque s'ajoute à celle qui était utilisée jusqu'alors (AWT) pour des raisons de compatibilité.
 - Swing fait cependant double emploi dans beaucoup de cas avec AWT.
 - L'ambition de Sun est que, progressivement, les développeurs réalisent toutes leurs interfaces avec Swing et laissent tomber les anciennes API graphiques.

Composants graphiques

lourds (1)



- Un composant graphique lourd (*heavyweight GUI component*) s'appuie sur le gestionnaire de fenêtres local, celui de la machine sur laquelle le programme s'exécute.
 - awt ne comporte que des composants lourds.
 - Ce choix technique a été initialement fait pour assurer la portabilité.

Composants graphiques

lourds (2)



- Exemple :
 - Un bouton de type `java.awt.Button` intégré dans une application Java sur la plate-forme Unix est représenté grâce à un vrai bouton Motif (appelé son pair - *peer* en anglais).
 - Java communique avec ce bouton Motif en utilisant la Java Native Interface. Cette communication induit un coût.
 - C'est pourquoi ce bouton est appelé composant lourd.



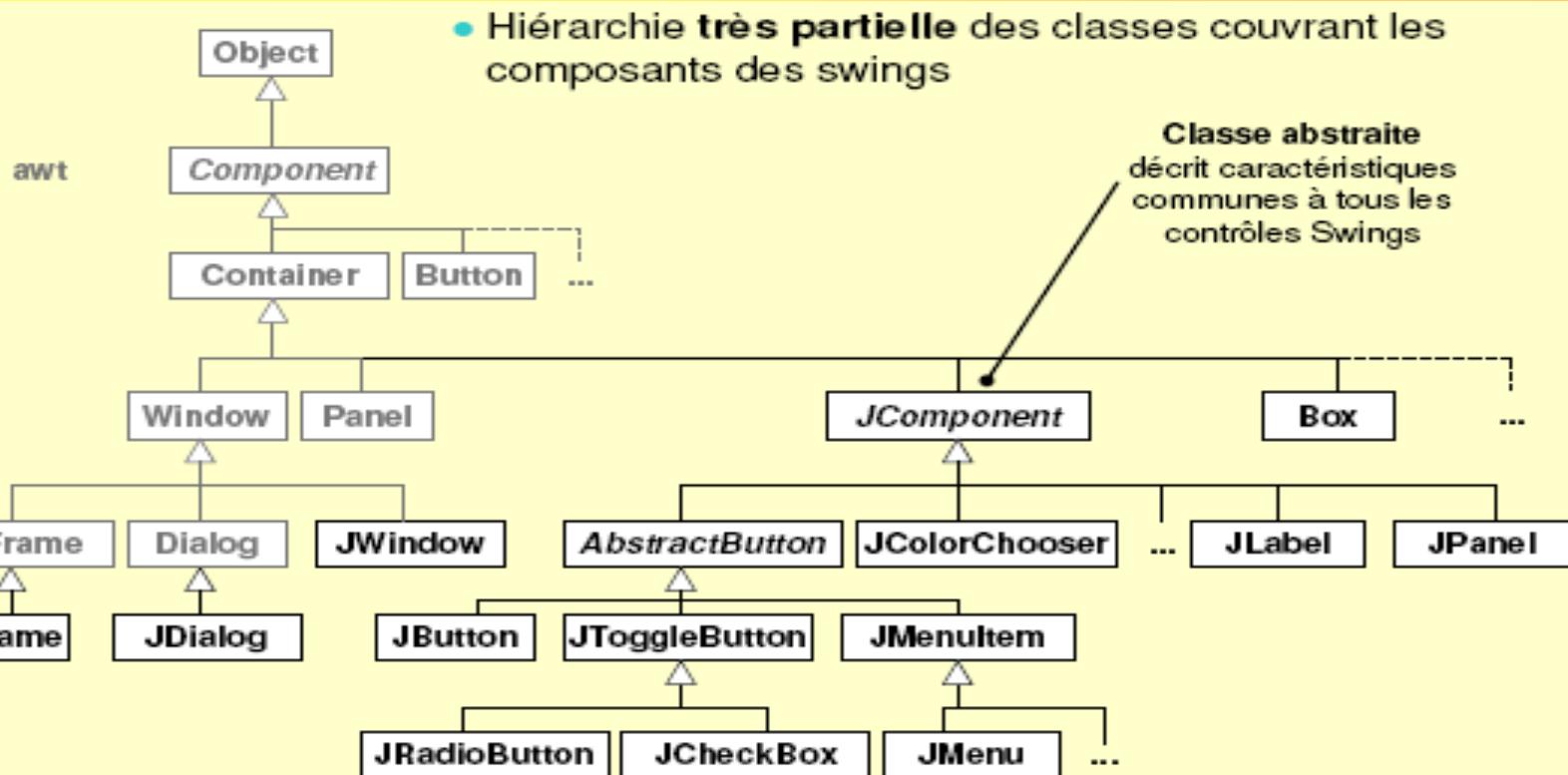
Composants légers de Swing

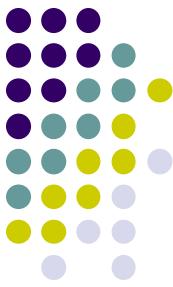
- Un composant graphique léger (en anglais, *lightweight GUI component*) est un composant graphique indépendant du gestionnaire de fenêtre local.
 - Un composant léger ressemble à un composant du gestionnaire de fenêtre local mais n'en est pas un : un composant léger émule les composants de gestionnaire de fenêtre local.
 - Un bouton léger est un rectangle dessiné sur une zone de dessin qui contient une étiquette et réagit aux événements souris.
 - Tous les composants de Swing, exceptés **JApplet**, **JDialog**, **JFrame** et **JWindow** sont des composants légers.



Les classes Swing

Composants graphiques Swing





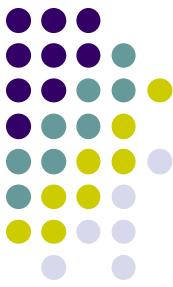
Atouts de Swing

- Plus de composants, offrant plus de possibilités.
- Les composants Swing dépendent moins de la plate-forme :
 - Il est plus facile d'écrire une application qui satisfasse au slogan "*Write once, run everywhere*"
 - Swing peut pallier aux faiblesses (bogues ?) de chaque gestionnaire de fenêtre.



Look & Feel (1)

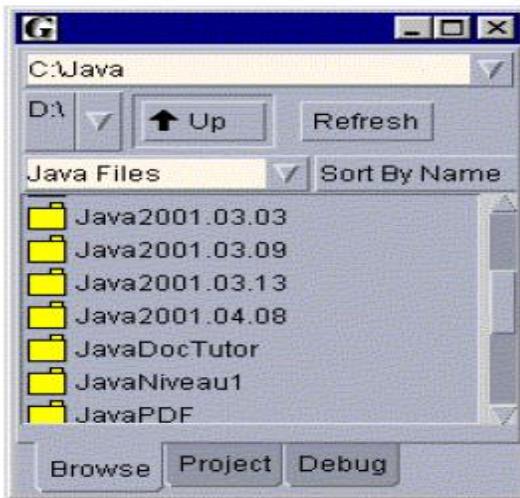
- Les programmeurs peuvent choisir l'aspect qu'ils désirent pour leur application
 - en effet ceux-ci sont émulés.
 - On parle de **pluggable look and feel** ou **plaf**.
 - Exemple : Une application exécutée sur un système Windows ayant l'aspect d'une application Motif.
 - Ce choix peut même intervenir en cours d'exécution.



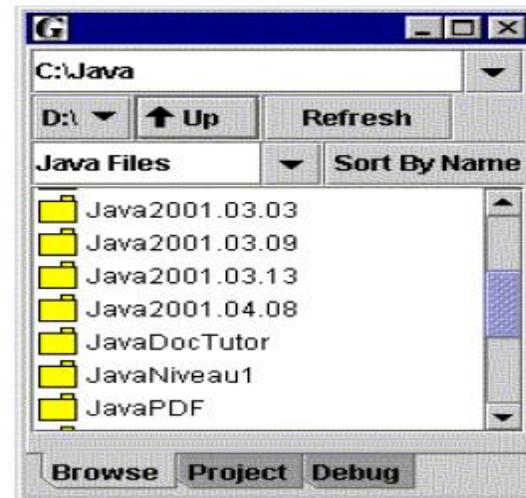
Look & Feel (2)

- Sun a choisi de créer son propre look-and-feel,
 - il permet de donner une "identité graphique" aux applications Java.
 - C'est le look-and-feel **Metal**.
- Un inconvénient est que
 - pour des raisons de copyright tous les **look-and-feel** ne sont pas disponibles sur toutes les plate-formes.

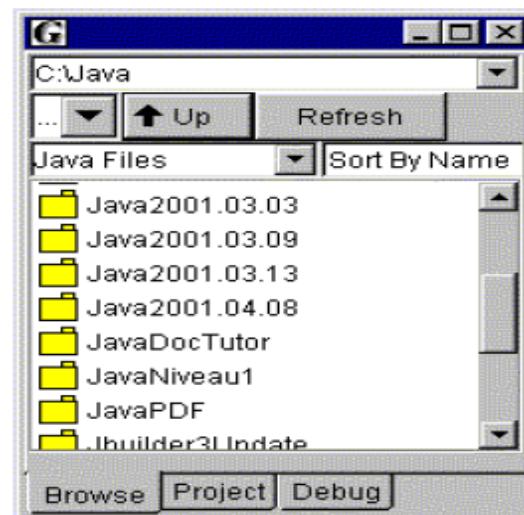
Aspect motif de l'IHM



Aspect métal de l'IHM



Aspect Windows de l'IHM :





Conventions de nommage

- Les composants Swing sont situés dans le paquetage `javax.swing` et ses sous paquetages.
- Ils portent des noms similaires à leurs correspondants de AWT précédés d'un J.
 - `JFrame`, `JPanel`, `JTextField`, `JButton`, `JCheckBox`, `JLabel`, etc.



Architecture Swing

Une application est composée de plusieurs Swing :

- Un composant **top-level**
- Plusieurs composants **conteneur intermédiaire**, ils contiennent d'autre composants
- Des **composants atomiques**



Le composant JComponent

- Tous les composants Swing héritent de JComponent
- Les composants ont des Tool Tips
- Les composants ont des bordures
- Entité graphique la plus abstraite



Top-Level

- Swing propose 3 composants top-level: JFrame, JDialog et JApplet.
- JWindow est aussi top-level mais il n'est pas utilisé.
- JInternalFrame ressemble à un top-level mais il n'en est pas un
- Une application graphique doit avoir un composant top-level comme composant racine (composant qui inclus tous les autres composants)

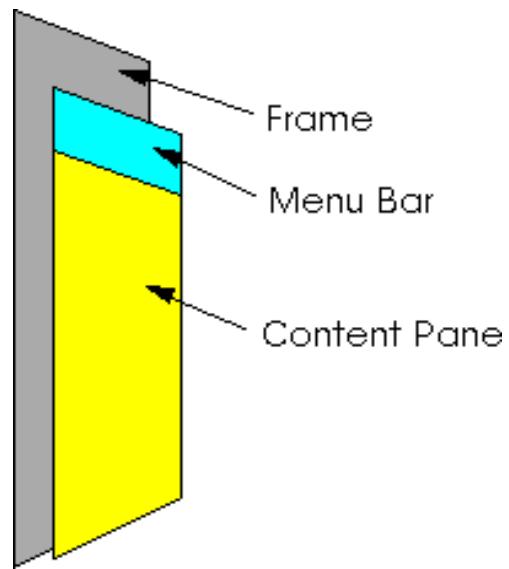
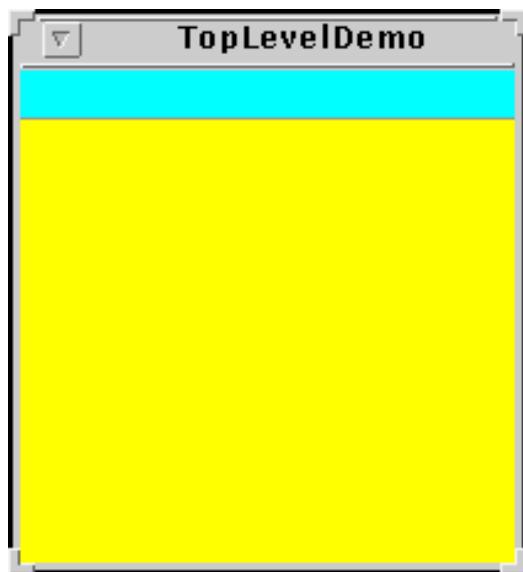
Top-Level Containers





Top-Level

- Les composants top-level possèdent un content pane qui contient tous les composants visibles d'un top-level
- Un composant top-level peut contenir une barre de menu





JDialog

Un JDialog est une fenêtre qui a pour but un échange d'information



Un JDialog dépend d'une fenêtre, si celle-ci est détruite, le JDialog l'est aussi

Un JDialog peut aussi être modal, il bloque tout les inputs sur lui

Conteneur Intermédiaire

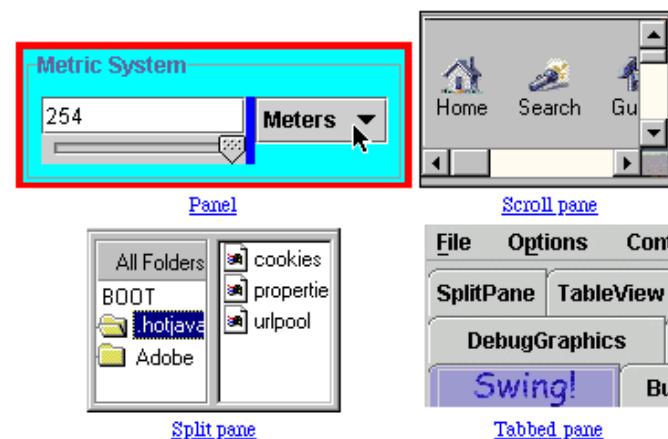


- Les conteneur intermédiaire sont utilisés pour structurer l'application graphique
- Le composant top-level contient des composants conteneur intermédiaire
- Un conteneur intermédiaire peut contenir d'autre conteneurs intermédiaire

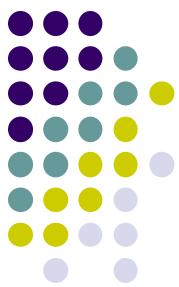
Swing propose plusieurs conteneurs intermédiaire:

- JPanel
- JScrollPane
- JSplitPane
- JTabbedPane
- JToolBar
- ...

General-Purpose Containers

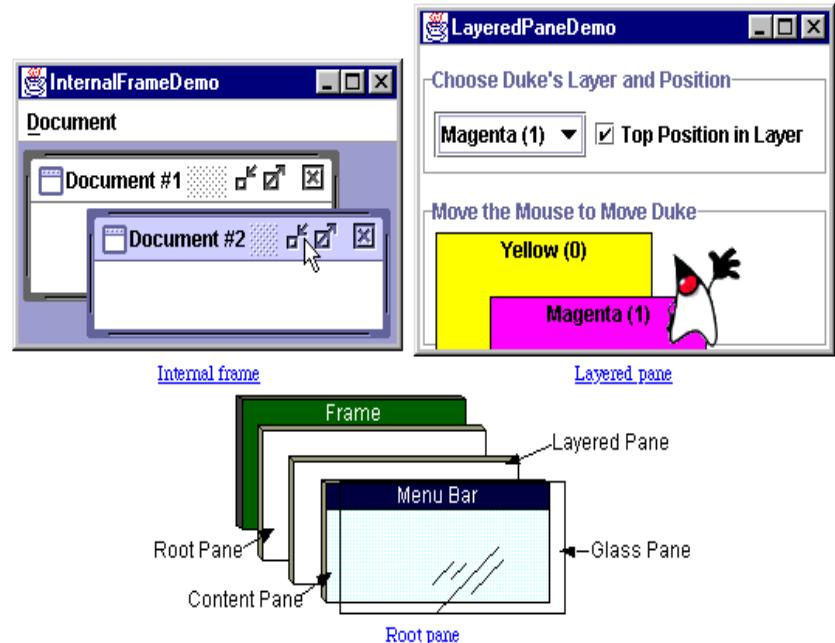


Conteneur Intermédiaire Spécialisé

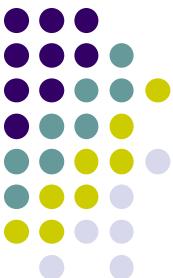


- Les conteneurs Intermédiaire spécialisé sont des conteneurs qui offrent des propriétés particulières aux composants qu'ils accueillent
 - JRootPane
 - JLayeredPane
 - JInternalFrame

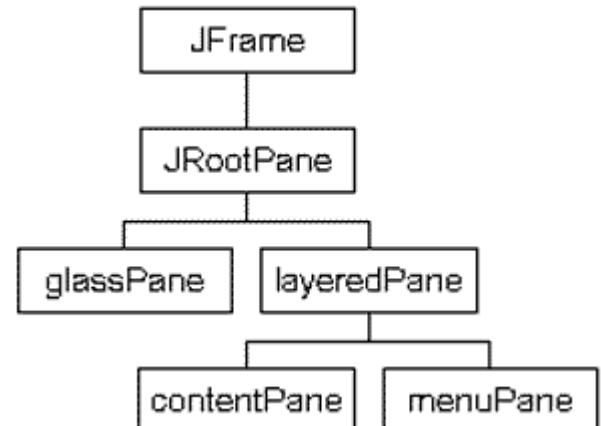
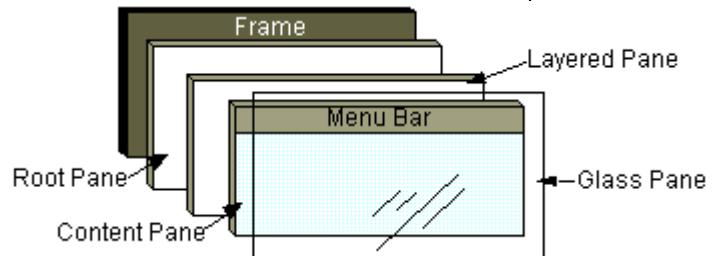
Special-Purpose Containers



JRootPane



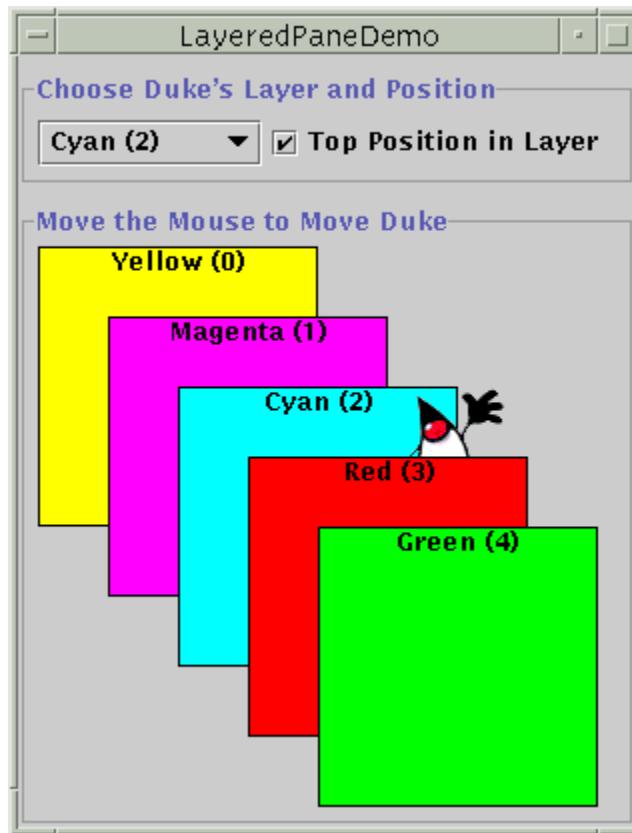
- En principe, un JRootPane est obtenu à partir d'un top-level ou d'une JInternalFrame
- Le dispositif d'affichage d'un JFrame est un JRootPane, composé de deux objets : un glass pane et un layered pane.
- Le glass pane est invisible, mais toujours devant le layered pane, et permet l'affichage des tooltips et des popups
- Le layered pane est constitué d'un menubar optionnel, et d'un content pane, utilisé habituellement
- Le contentPane est par défaut un JPanel opaque dont le gestionnaire de présentation est un BorderLayout.
- Tous les composants ajoutés au JFrame doivent être ajoutés à un des Panes du JRootPane et non au JFrame directement.
- C'est aussi à un de ces Panes qu'il faut associer un layout manager si nécessaire

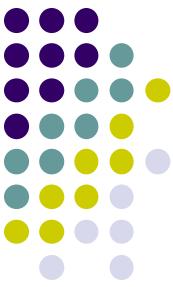




JLayeredPane

- Le JLayeredPane permet d'afficher des composants dans des couches différentes, ce qui permet des superpositions:
- `layeredPane.add (component, new Integer(5));`
- Le défaut est `JLayeredPane.DEFAULT_LAYER`.
- On peut placer des objets relativement à cette couche, devant ou derrière
- Le LayoutManager détermine l'ordre d'affichage, et empêche les superpositions au sein d'une même couche.

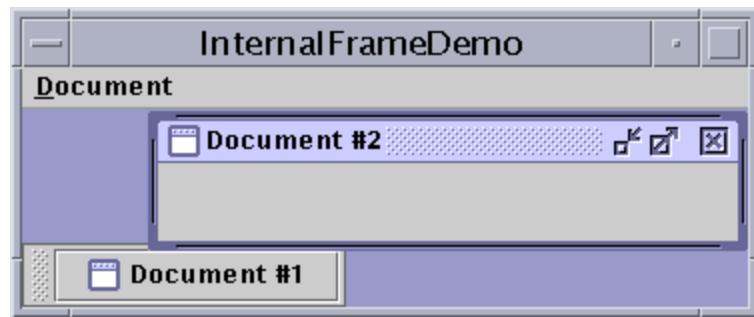




JInternalFrame

Un JInternalFrame permet d'avoir des petites fenêtres dans une fenêtre.

Une JInternalFrame ressemble très fortement à une JFrame mais ce n'est pas un container Top-Level





Les composants atomiques(1)

- Un composant atomique est considéré comme étant une entité unique.
- Java propose beaucoup de composants atomiques:
 - boutons, CheckBox, Radio
 - Combo box
 - List, menu
 - TextField, TextArea, Label
 - FileChooser, ColorChooser,
 - ...



Les composants atomiques(2)

Basic Controls



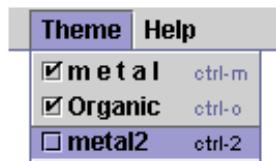
[Buttons](#)



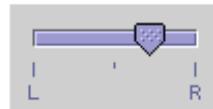
[Combo box](#)



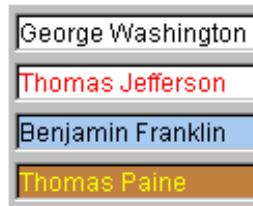
[List](#)



[Menu](#)



[Slider](#)



[Text fields](#)

Uneditable Information Displays



[Label](#)



[Progress bar](#)



[Tool tip](#)

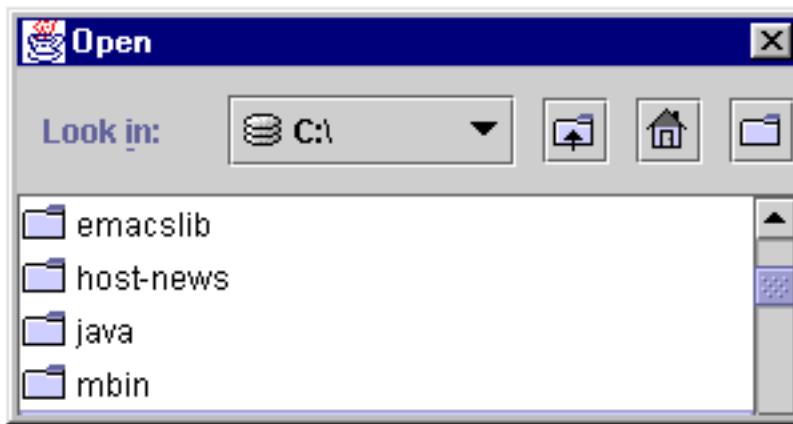
Aperçu des composants Swing (3)



Editable Displays of Formatted Information



[Color chooser](#)



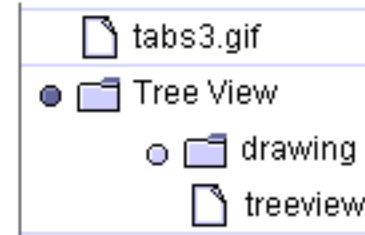
[File chooser](#)

First Name	Last Name
Mark	Andrews
Tom	Ball
Alan	Chung
Jeff	Dinkins

[Table](#)

Verify that the RJ45 cable is connected to the WAN plug on the back of the Pipeline unit.

[Text](#)



[Tree](#)

Principaux paquetages Swing

(1)



- javax.swing
 - le paquetage général
- javax.swing.border
 - pour dessiner des bordures autour des composants
- javax.swing.colorchooser
 - classes et interfaces utilisées par le composant JColorChooser
- javax.swing.event
 - les événements générés par les composants Swing
- javax.swing.filechooser
 - classes et interfaces utilisées par le composant JFileChooser

Principaux paquetages Swing (2)



- javax.swing.plaf
 - les classes et interfaces autour du **plaf**
- javax.swing.table
 - classes et interfaces pour gérer les JTable
- javax.swing.text
 - classes et interfaces pour la gestion des composants « texte »
- javax.swing.tree
 - classes et interfaces pour gérer les JTree
- javax.swing.undo
 - pour la gestion de undo/redo dans une application

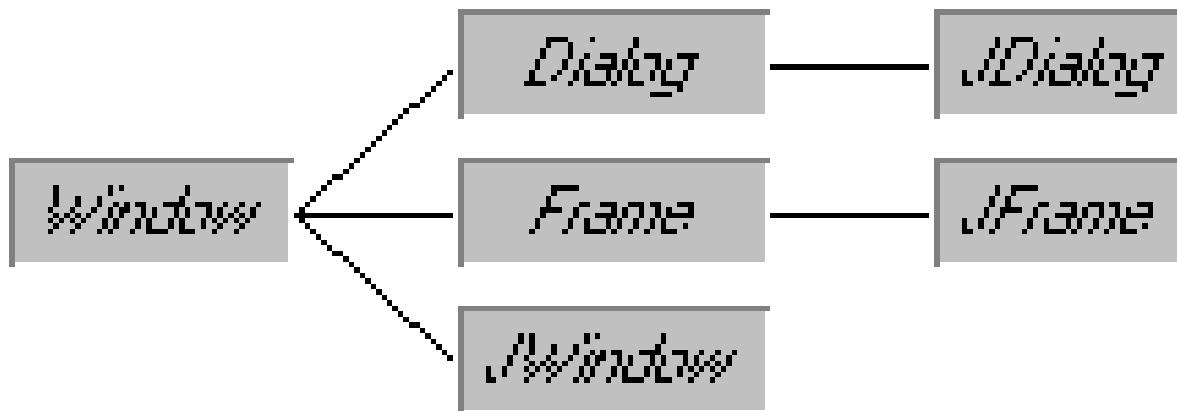
Présentation de la Bibliothèque Java Swing



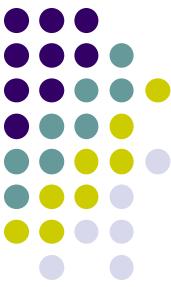


Hiérarchie des Fenêtres

- La hiérarchie des classes fenêtre Swing s'intègre sous la classe Windows de AWT

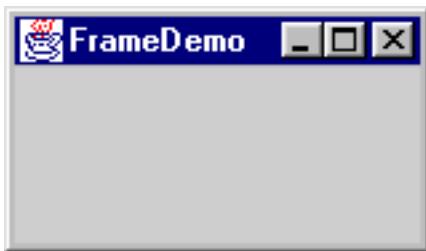


- Elles ne sont donc pas "lightweight", sont associées à une fenêtre graphique, et ne peuvent pas être transparentes.
- On peut utiliser setJMenuBar().
- De même que pour JWindow et JDialog, on doit ajouter les éléments à un container obtenu par getContentPane()



JFrame

Une JFrame est une fenêtre avec un titre et une bordure



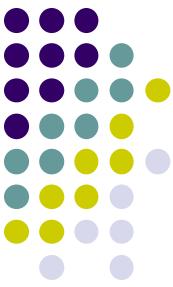
Quelques méthodes :

```
public JFrame();
public JFrame(String name);
public Container getContentPane();
public void setMenuBar(JMenuBar menu);
public void setTitle(String title);
public void setIconImage(Image image);
```



JFrame et Close

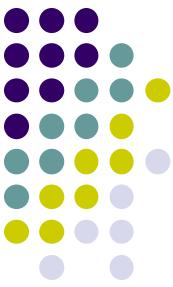
- Il est possible de préciser comment un objet JFrame (JInternalFrame, ou JDialog) réagit à sa fermeture grâce à la méthode setDefaultCloseOperation().
- Cette méthode attend en paramètre une valeur qui peut être :
 - DO NOTHING ON CLOSE: comme AWT
 - HIDE ON CLOSE: le défaut (setVisible(true) remappe la fenêtre)
 - DISPOSE ON CLOSE: récupère les ressources
- HIDE ON CLOSE et DISPOSE ON CLOSE laissent s'exécuter les event listeners



Création d'une classe "frame"

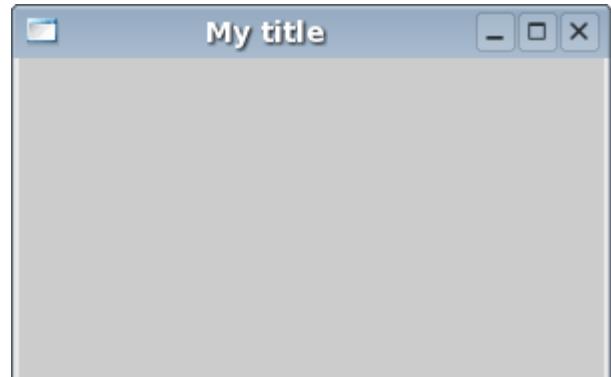
```
import javax.swing.JFrame;

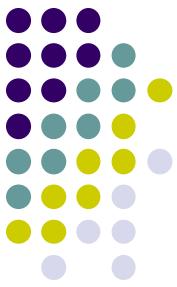
public class MainFrame extends JFrame
{
    public MainFrame()
    {
        super("My title");
        setSize(300, 300);
    }
}
```



Création d'une application

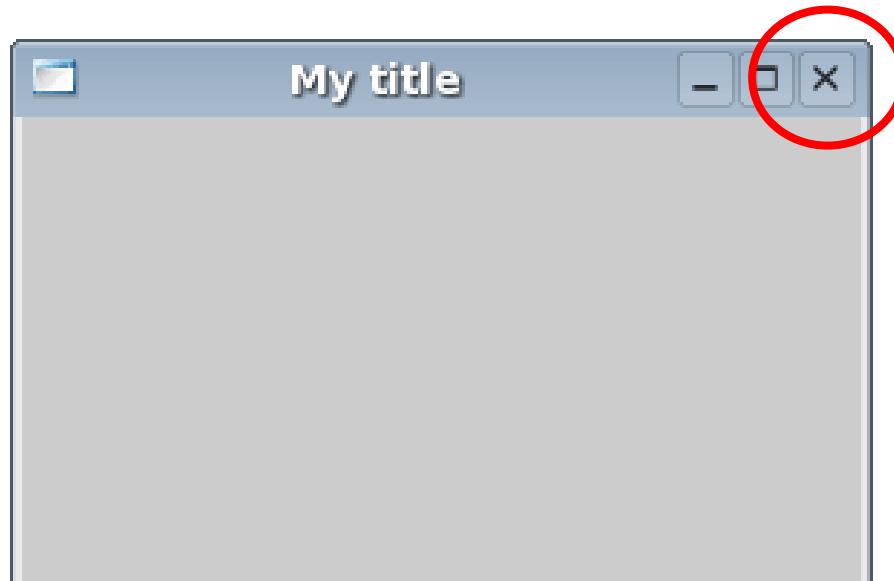
```
public class Application
{
    public static void main(String[] args)
    {
        // perform any initialization
        MainFrame mf = new MainFrame();
        mf.setVisible(true);
    }
}
```





Fermer l'application

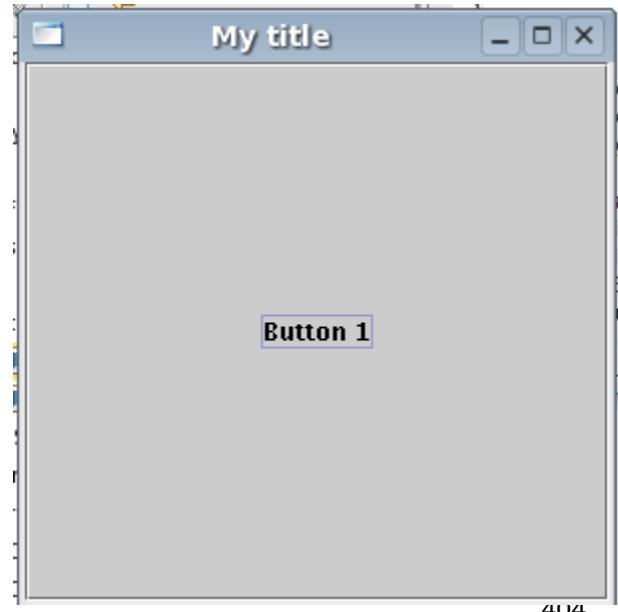
```
setDefaultCloseOperation(  
    JFrame.EXIT_ON_CLOSE);
```





Ajout de composants

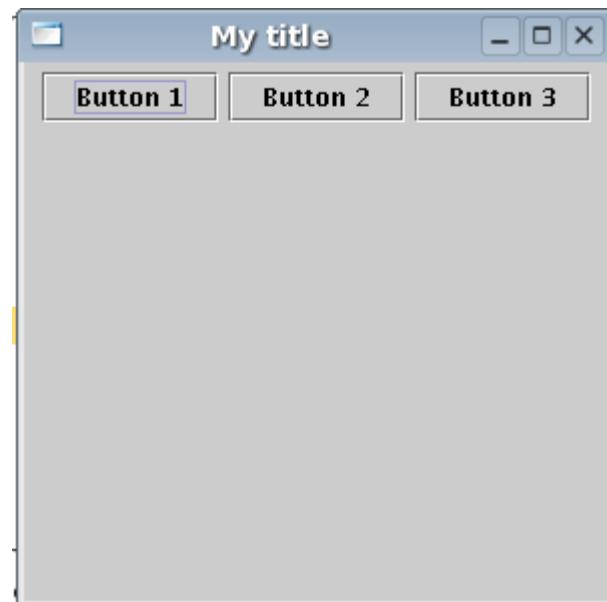
```
Container content = getContentPane();  
content.add(new JButton("Button 1"));
```





Grouper des composants

```
JPanel panel=new JPanel();
panel.add(new JButton("Button 1"));
panel.add(new JButton("Button 2"));
panel.add(new JButton("Button 3"));
content.add(panel);
```



Gérer le Positionnement Variable



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

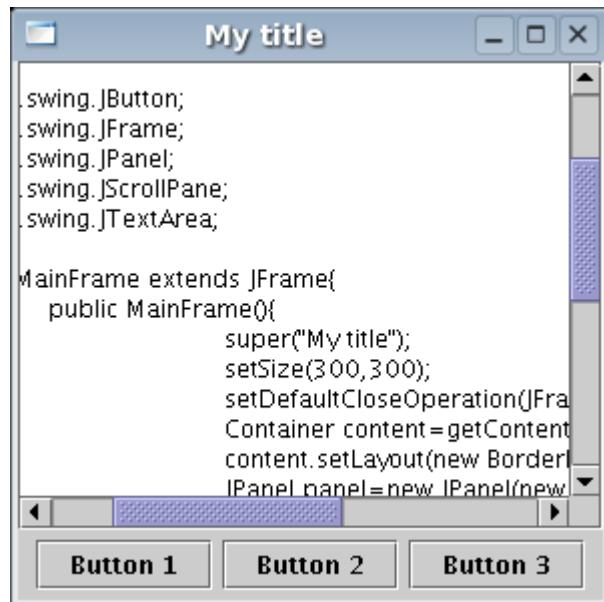
public class MainFrame extends JFrame{
    public MainFrame() {
        super("My title");
        setSize(300,300);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        Container content = getContentPane();
        content.setLayout(new BorderLayout());
        JPanel panel = new JPanel(new FlowLayout());
        panel.add(new JButton("Button 1"));
        panel.add(new JButton("Button 2"));
        panel.add(new JButton("Button 3"));
        content.add(panel, BorderLayout.SOUTH);
        content.add(new JTextArea(), BorderLayout.CENTER);
    }
}
```

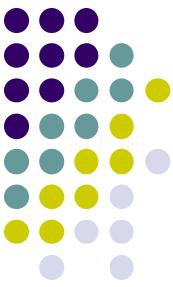




Ascenseurs

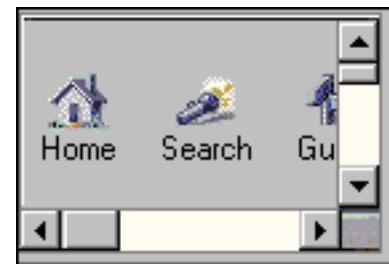
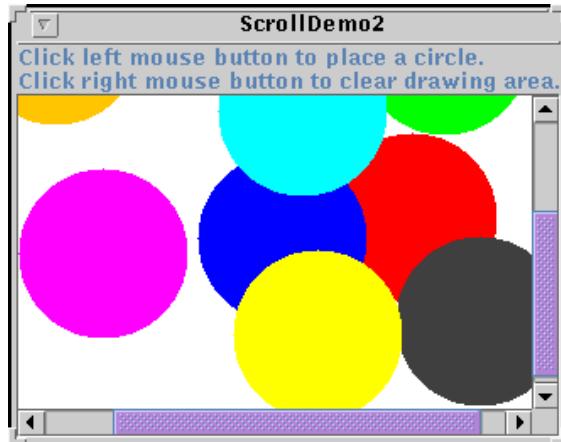
```
content.add(new JScrollPane(new JTextArea()), BorderLayout.CENTER);
```





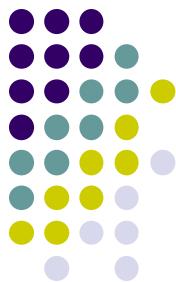
JScrollPane (Ascenseurs)

Un JScrollPane est un conteneur qui offre des ascenseurs, il permet de visionner un composant plus grand que lui



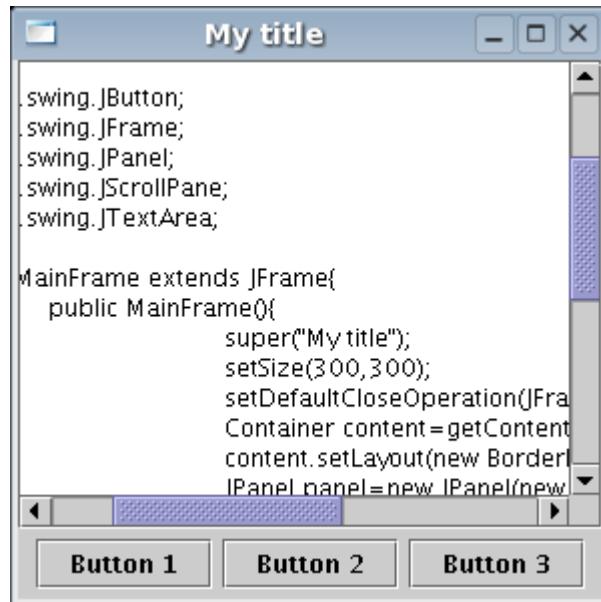
Quelques méthodes:

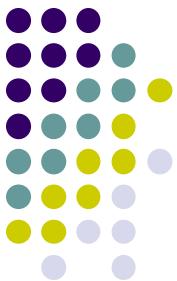
```
public JScrollPane(Component comp);  
public void setCorner(String key,Component comp);
```



JScrollPane (Ascenseurs)

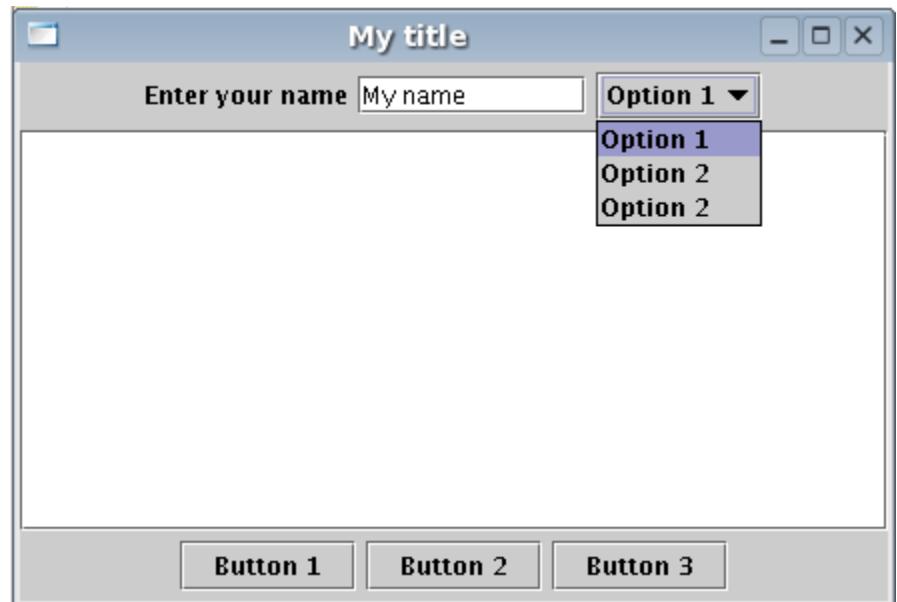
```
content.add(new JScrollPane(new JTextArea()), BorderLayout.CENTER);
```





Mixer les layouts

```
panel=new JPanel(new FlowLayout());
panel.add(new JLabel("Enter your name"));
panel.add(new JTextField(10));
String options[] = new String[]{"Option 1","Option 2","Option 2"};
panel.add(new JComboBox(options));
content.add(panel, BorderLayout.NORTH);
```





Événements

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

public class MainFrame extends JFrame {
    public MainFrame(){
        super("My title");
        ...
        JButton button1 = new JButton("Button 1");
        panel.add(button1);
        button1.addActionListener( new MyButtonListener(this));
        ...
    }

    private class MyButtonListener implements ActionListener
    {
        private JFrame parentComponent;
        MyButtonListener(JFrame parentComponent){
            this.parentComponent=parentComponent; }

        public void actionPerformed(ActionEvent e) {
            JOptionPane.showMessageDialog(parentComponent, "BUTTON PRESSED!");
        }
    }
}
```



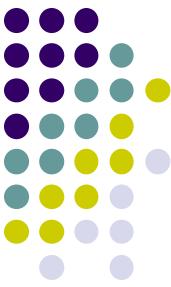


Evénements

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

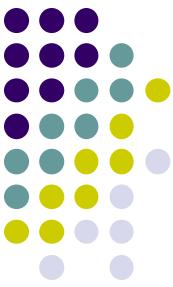
public class MainFrame extends JFrame {
    public MainFrame(){
        super("My title");
        ...
        JButton button1 = new JButton("Button 1");
        panel.add(button1);
        button1.addActionListener( new ActionListener(this) {
            public void actionPerformed(ActionEvent e) {
                JOptionPane.showMessageDialog(this, "BUTTON PRESSED!");
            }
        });
        ...
    }
}
```





Squelette d'application Swing

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
public class JWinApp extends JFrame{
    public JWinApp(String title, JPanel panel){
        super(title);
        getContentPane().add(panel, BorderLayout.CENTER);
        setSize(200,200);
        addWindowListener(new WindowAdapter(){
            public void windowClosing(WindowEvent we){ exitApp();
        });
    }
    protected void exitApp(){
        setVisible(false);
        dispose();
        System.exit(0);
    }
    public static void main(String args[]) { new JWinApp(...).setVisible(true); }
}
```

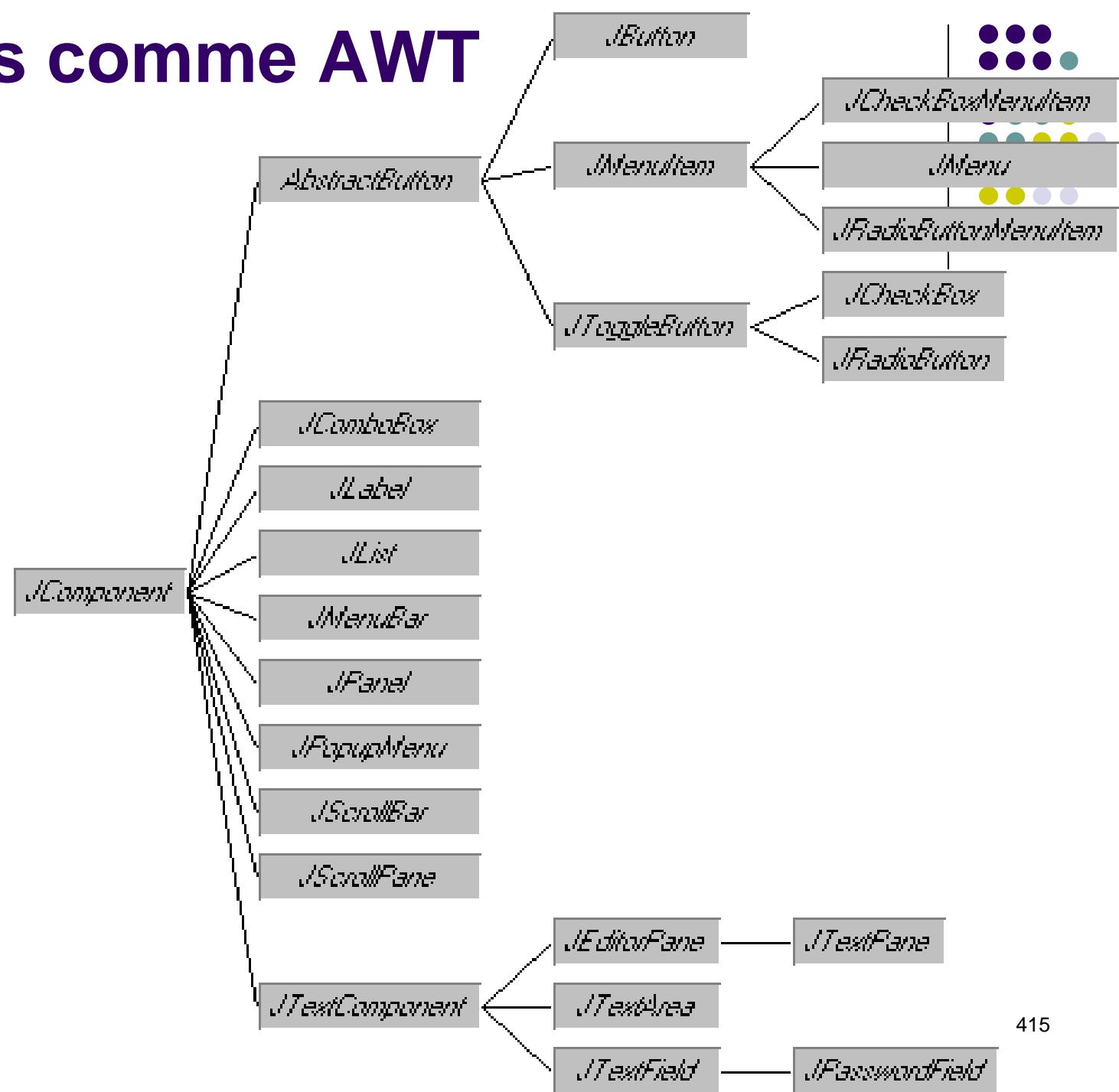


Hello World

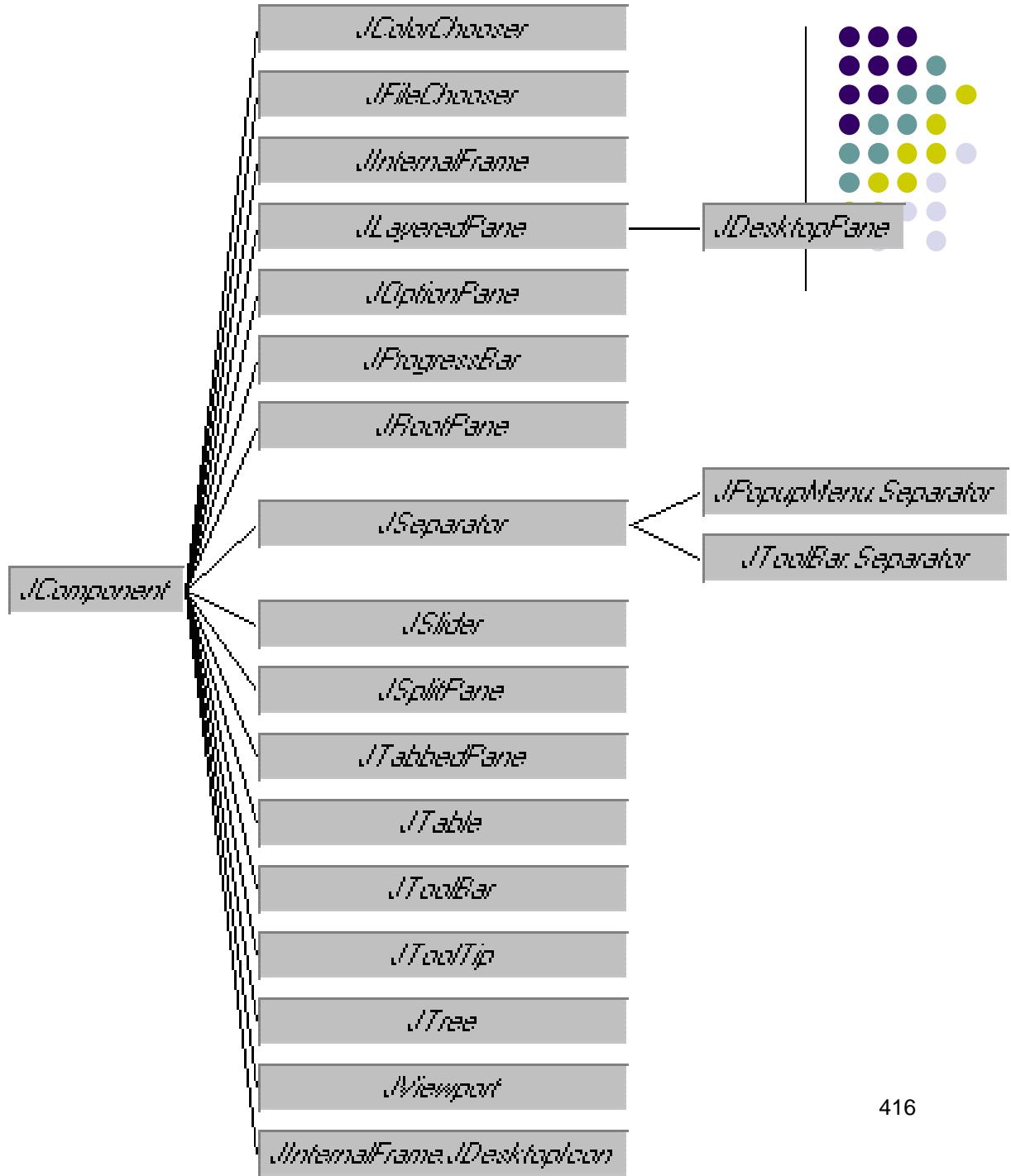
```
class WinHelloPanel extends JPanel implements ActionListener{  
    JLabel label = new JLabel("Hello World "); // un label  
    JButton button = new JButton("Click!"); // un bouton  
    public WinHelloPanel(){  
        add(label);  
        add(button);  
        button.addActionListener(this);  
    }  
    public void actionPerformed(ActionEvent ae){  
        JOptionPane.showMessageDialog(this, "Thanks for Clicking.");  
    }  
}
```



Widgets comme AWT



Widgets Swing

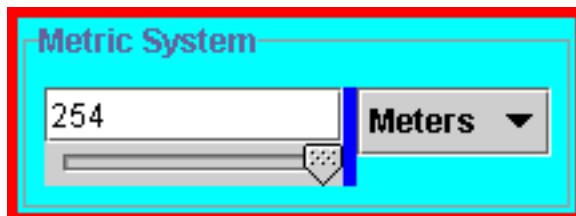




JPanel

Le JPanel est le conteneur intermédiaire le plus neutre

On ne peut que choisir la couleur de fond



Quelques méthodes de JPanel:

```
public JPanel();  
public Component add(Component comp);  
public void setLayout(LayoutManager lm);
```



JPanel

- C'est un Panel léger offrant un support pour le double buffering (technique d'affichage en deux temps permettant d'éviter les scintillements et défaut d'aspects)
- Quand le buffering est activé (constructeur) tous les composants se dessinent d'abord dans un buffer non affiché

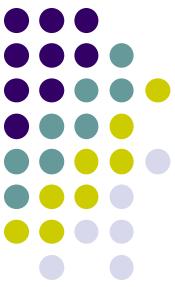


Icones

- Les icônes sont utilisées avec tous les boutons ou autres composants.

```
public interface Icon {  
    void paintIcon(Component c, Graphics g, int x, int y);  
    int getIconWidth();  
    int getIconHeight();  
}
```

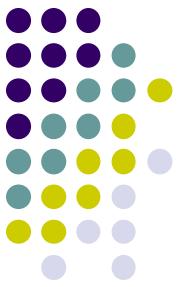
- l'argument “c” sert à fournir une information complémentaire au moment du dessin (police, couleur)
- x et y spécifient l'origine du dessin



ImageIcon

```
Icon i = new ImageIcon("Image.gif");
```

- Avantages :
 - url ou fichier,
 - chargement asynchrone : pas de blocage de l'interface
 - l'image n'est pas sérializable



Créer sa propre icône

```
public class RedOval implements Icon {  
    public void paintIcon (Component c, Graphics g, int x, int y) {  
        g.setColor(Color.red);  
        g.drawOval (x, y, getIconWidth(), getIconHeight());  
    }  
    public int getIconWidth() {    return 10;  }  
    public int getIconHeight() {    return 10;  }  
}
```

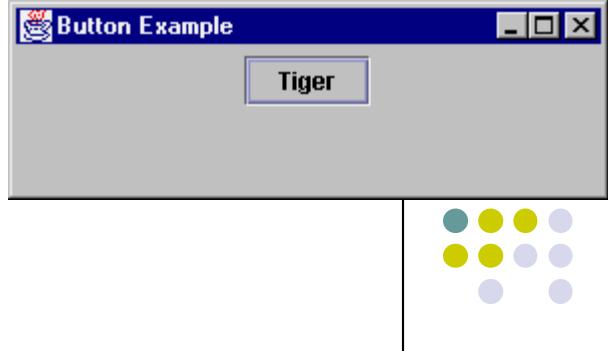
JLabel



```
public class LabelPanel extends JPanel {  
    public LabelPanel() {  
        JLabel plainLabel = new JLabel("Plain Small Label");  
        add(plainLabel);  
  
        JLabel fancyLabel = new JLabel("Fancy Big Label");  
        Font fancyFont = new Font("Serif", Font.BOLD | Font.ITALIC, 32);  
        fancyLabel.setFont(fancyFont);  
  
        Icon tigerIcon = new ImageIcon("SmallTiger.gif");  
        fancyLabel.setIcon(tigerIcon);  
        fancyLabel.setHorizontalAlignment(JLabel.RIGHT);  
        add(fancyLabel);  
    }  
}
```



JButton



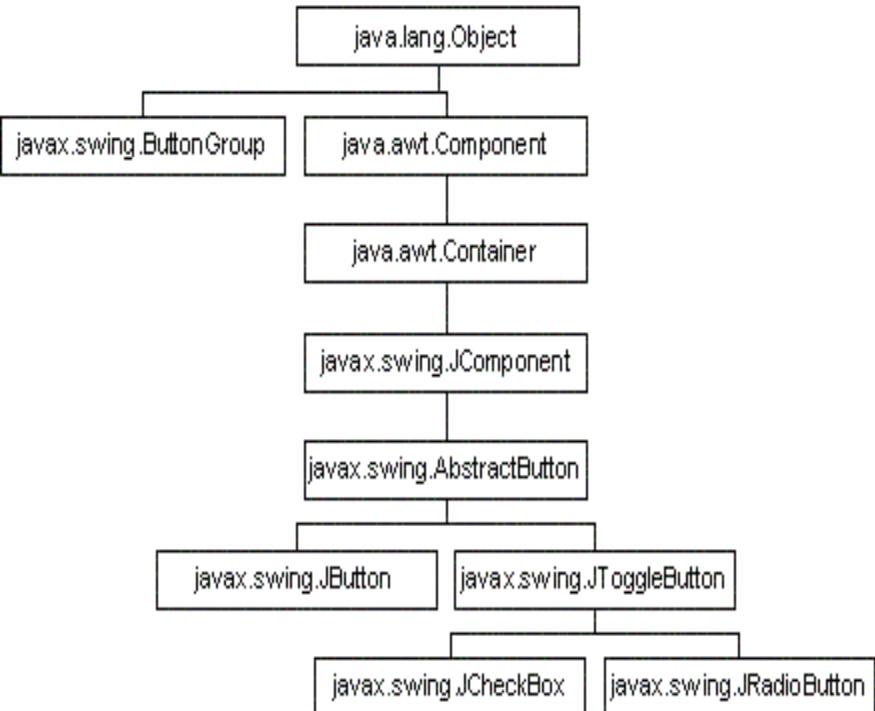
```
public class ButtonPanel extends JPanel {  
    public ButtonPanel() {  
        Icon tigerIcon = new ImageIcon("SmallTiger.gif");  
        JButton myButton = new JButton("Tiger", tigerIcon);  
        add(myButton);  
    }  
}
```





AbstractButton

- Plusieurs classes Swing implémentent abstractButton
- setMnemonic() – accélérateur clavier : les constantes VK_* de KeyEvent
- doClick() – déclencher un clic par programmation
- setDisabledIcon(),
setDisabledSelectedIcon(),
setPressedIcon(), setRolloverIcon(),
setRolloverSelectedIcon(),
setSelectedIcon() – modifications dynamique de l'icone
- setVerticalAlignment(),
setHorizontalAlignment()
- setVerticalTextPosition(),
setHorizontalTextPosition() – place le texte par rapport à l'icone





JCheckBox

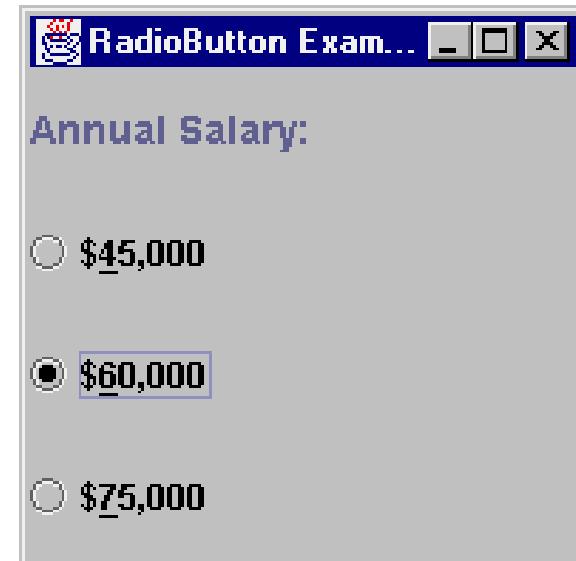
```
class CheckboxPanel extends JPanel {  
  
Icon no = new ToggleIcon (false);  
Icon yes = new ToggleIcon (true);  
  
public CheckboxPanel() {  
  
setLayout(new GridLayout(2, 1));  
  
JCheckBox cb1 = new  
    JCheckBox("Choose Me", true);  
cb1.setIcon(no);  
cb1.setSelectedIcon(yes);  
  
JCheckBox cb2 = new  
    JCheckBox("No Choose Me", false);  
cb2.setIcon(no);  
cb2.setSelectedIcon(yes);  
  
add(cb1); add(cb2);  
}
```

```
class ToggleIcon implements Icon {  
    boolean state;  
    public ToggleIcon (boolean s) {  
        state = s;  
    }  
  
    public void paintIcon (Component c, Graphics  
        g, int x, int y) {  
  
        int width = getIconWidth();  
        int height = getIconHeight();  
        g.setColor (Color.black);  
        if (state) g.fillRect (x, y, width, height);  
        else g.drawRect (x, y, width, height);  
    }  
  
    public int getIconWidth() { return 10; }  
    public int getIconHeight() { return 10; }  
}
```

JRadioButton



```
class RadioButtonPanel extends JPanel {  
    public RadioButtonPanel() {  
        setLayout(new GridLayout(4,1));  
        JRadioButton radioButton;  
        ButtonGroup rbg = new ButtonGroup();  
        JLabel label = new JLabel("Annual Salary: ");  
        label.setFont(new Font("SansSerif", Font.BOLD, 14));  
        add(label);  
  
        radioButton = new JRadioButton("$45,000");  
        radioButton.setMnemonic(KeyEvent.VK_4);  
        add (radioButton);    rbg.add (radioButton);  
        radioButton.setSelected(true);  
  
        radioButton = new JRadioButton("$60,000");  
        radioButton.setMnemonic(KeyEvent.VK_6);  
        add (radioButton);    rbg.add (radioButton);  
        ...  
    }  
}
```

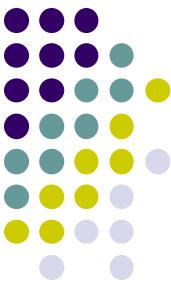


JToggleButton



```
class ToggleButtonPanel extends JPanel {  
  
    public ToggleButtonPanel() {  
  
        // Set the layout to a GridLayout  
  
        setLayout(new GridLayout(4,1, 10, 10));  
  
        add (new JToggleButton ("Fe"));  
        add (new JToggleButton ("Fi"));  
        add (new JToggleButton ("Fo"));  
        add (new JToggleButton ("Fum"));  
    }  
}
```





Méthodes de JTextComponent

- copy()
- cut()
- paste()
- getSelectedText()
- setSelectionStart()
- setSelectionEnd()
- selectAll()
- replaceSelection()
- getText()
- setText()
- setEditable()
- setCaretPosition()



JTextField & JTextArea

```
JTextField tf = new JTextField();
```

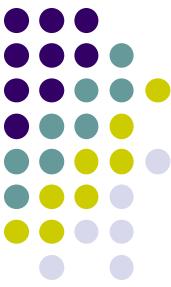
```
JTextArea ta = new JTextArea();
```

```
tf.setText("TextField");
```

```
ta.setText("JTextArea\n Multi Lignes");
```

```
add(tf);
```

```
add(new JScrollPane(ta)); // scroll au cas où
```



JTextPane

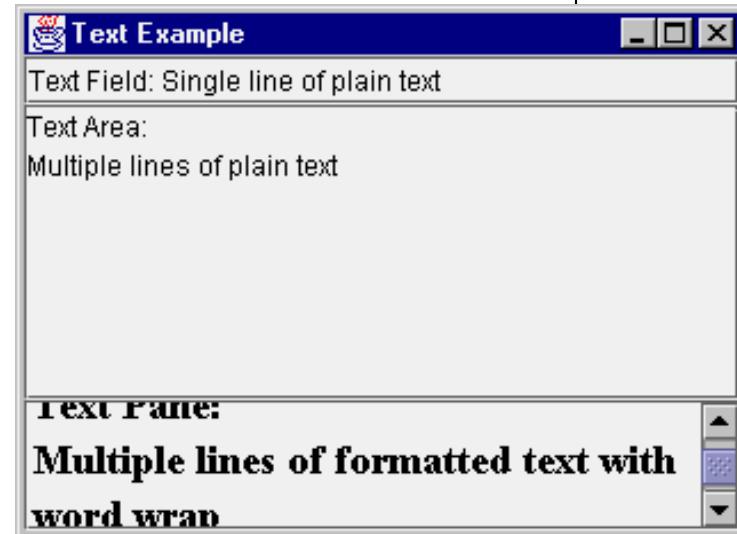
JTextPane est un éditeur de texte complet (avec insertions d'images). Il s'appuie sur une liste de blocs dotés de styles

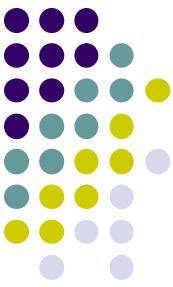
- `JTextPane tp = new JTextPane();`
- `MutableAttributeSet attr = new SimpleAttributeSet();`
- `StyleConstants.setFontFamily(attr, "Serif");`
- `StyleConstants.setFontSize(attr, 18);`
- `StyleConstants.setBold(attr, true);`
- `tp.setCharacterAttributes(attr, false);`
- `add(new JScrollPane(tp));`

JTextPane exemple



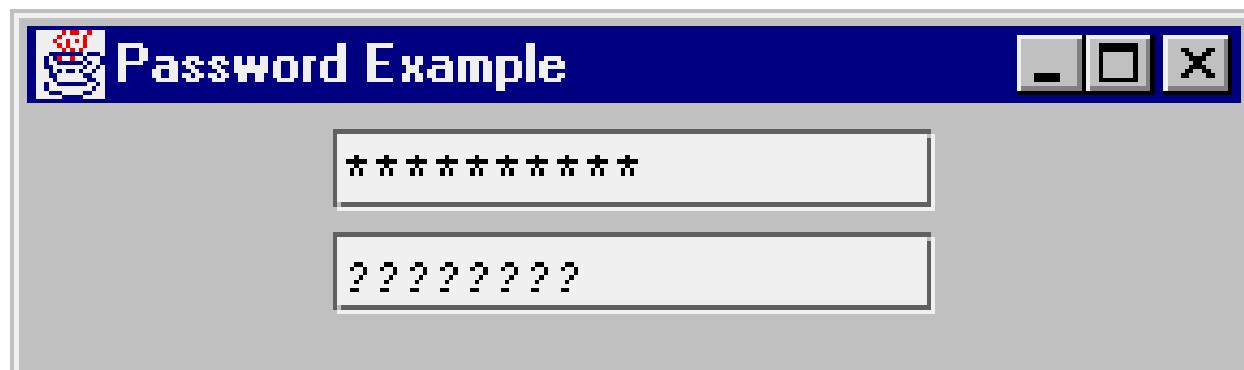
```
class TextPanel extends JPanel {  
  
    public TextPanel() {  
        setLayout(new BorderLayout());  
  
        JTextField textField = new JTextField();  
        JTextArea textArea = new JTextArea();  
        JTextPane textPane = new JTextPane();  
  
        MutableAttributeSet attr = new SimpleAttributeSet();  
        StyleConstants.setFontFamily(attr, "Serif");  
        StyleConstants.setFontSize(attr, 18);  
        StyleConstants.setBold(attr, true);  
        textPane.setCharacterAttributes(attr, false);  
  
        add(textField, BorderLayout.NORTH);  
        add(new JScrollPane(textArea), BorderLayout.CENTER);  
        add(new JScrollPane(textPane), BorderLayout.SOUTH);}}}
```

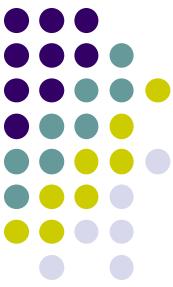




JPasswordField

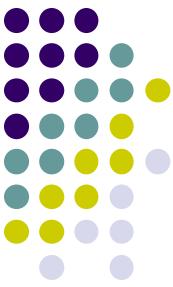
```
class PasswordPanel extends JPanel {  
    PasswordPanel() {  
        JPasswordField p1 = new JPasswordField(20);  
        JPasswordField p2 = new JPasswordField(20);  
        p2.setEchoChar ('?');  
        add(p1);  
        add(p2);  
    }  
}
```





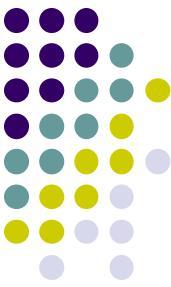
JEditorPane

- JEditorPane est un éditeur de textes
- permettant l'affichage de contenu html, ou rtf, identifié par une URL,
- et permettant de suivre les liens



Source

```
class Browser extends JPanel {  
    Browser() {  
        setLayout (new BorderLayout (5, 5));  
        final JEditorPane jt = new JEditorPane();  
        final JTextField input = new JTextField("http://java.sun.com");  
        jt.setEditable(false);  
  
        // suivre les liens :  
        jt.addHyperlinkListener(new HyperlinkListener () {  
            public void hyperlinkUpdate(final HyperlinkEvent e) {  
                if (e.EventType() == HyperlinkEvent.EventType.ACTIVATED) {  
                    Document doc = jt.getDocument();  
                    try { URL url = e.getURL(); jt.setPage(url);  
                        input.setText (url.toString());  
                    } catch (IOException io) {  
                        JOptionPane.showMessageDialog (this, "Can't follow link", "Invalid Input",  
                            JOptionPane.ERROR_MESSAGE);  
                    jt.setDocument (doc);}}};  
    }});
```



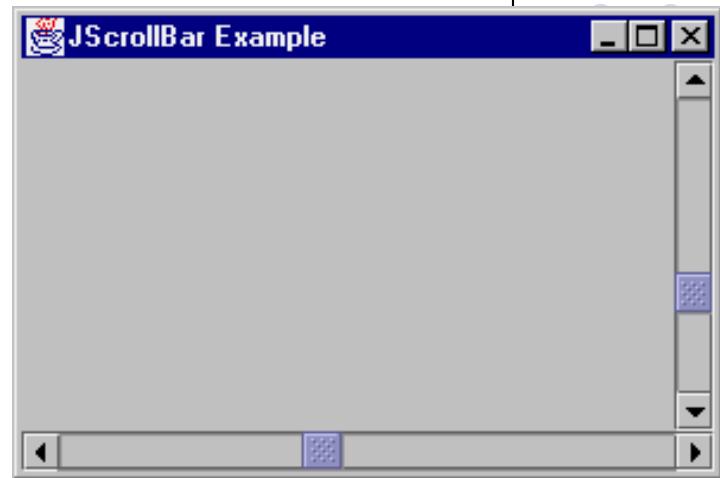
suite

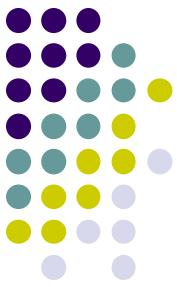
```
JScrollPane pane = new JScrollPane();
pane.setBorder (
    BorderFactory.createLoweredBevelBorder());
pane.setViewport().add(jt);
add(pane, BorderLayout.CENTER);
input.addActionListener (new ActionListener() {
    public void actionPerformed (ActionEvent e) {
        try { jt.setPage (input.getText());      }
        catch (IOException ex) {
            JOptionPane.showMessageDialog (
                Browser.this, "Invalid URL", "Invalid Input",
                JOptionPane.ERROR_MESSAGE);  }
    } });
add (input, BorderLayout.SOUTH);}}
```



JScrollBar

```
class ScrollbarPanel extends JPanel {  
    public ScrollbarPanel() {  
        setLayout(new BorderLayout());  
  
        JScrollPane sb1 =  
            new JScrollPane (JScrollPane.VERTICAL, 0, 5, 0, 100);  
        add(sb1, BorderLayout.EAST);  
  
        JScrollPane sb2 =  
            new JScrollPane (JScrollPane.HORIZONTAL, 0, 5, 0, 100);  
        add(sb2, BorderLayout.SOUTH);  
    }  
}
```





JSlider

Les JSlider permettent la saisie graphique d'un nombre
Un JSlider doit contenir les bornes max et min



Quelques méthodes:

```
public JSlider(int min ,int max, int value);  
public void setLabelTable(Dictionary d);
```

JSlider

```
public class SliderPanel extends JPanel {
```

```
    public SliderPanel() {
```

```
        setLayout(new BorderLayout());
```

```
        JSlider s1 = new JSlider (JSlider.VERTICAL, 0, 100, 50);
```

```
        s1.setPaintTicks(true);
```

```
        s1.setMajorTickSpacing(10);    s1.setMinorTickSpacing(2);
```

```
        add(s1, BorderLayout.EAST);
```

```
        JSlider s2 = new JSlider (JSlider.VERTICAL, 0, 100, 50);
```

```
        s2.setPaintTicks(true);    s2.setMinorTickSpacing(5);
```

```
        add(s2, BorderLayout.WEST);
```

```
        JSlider s3 = new JSlider (JSlider.HORIZONTAL, 0, 100, 50);
```

```
        s3.setPaintTicks(true);    s3.setMajorTickSpacing(10);
```

```
        add(s3, BorderLayout.SOUTH);
```

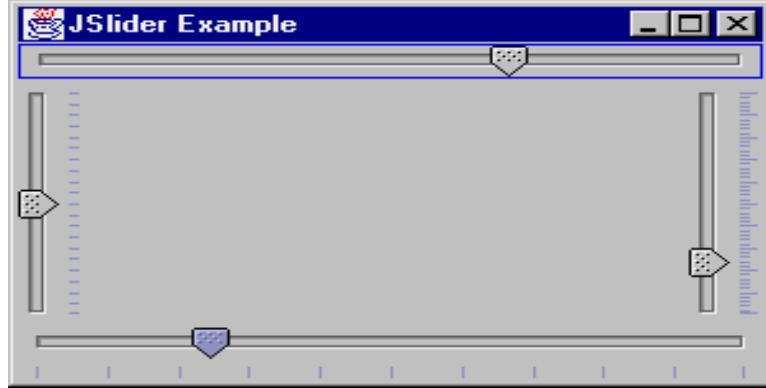
```
        JSlider s4 =
```

```
            new JSlider (JSlider.HORIZONTAL, 0, 100, 50);
```

```
            s4.setBorder(BorderFactory.createLineBorder(Color.blue));
```

```
            add(s4, BorderLayout.NORTH);
```

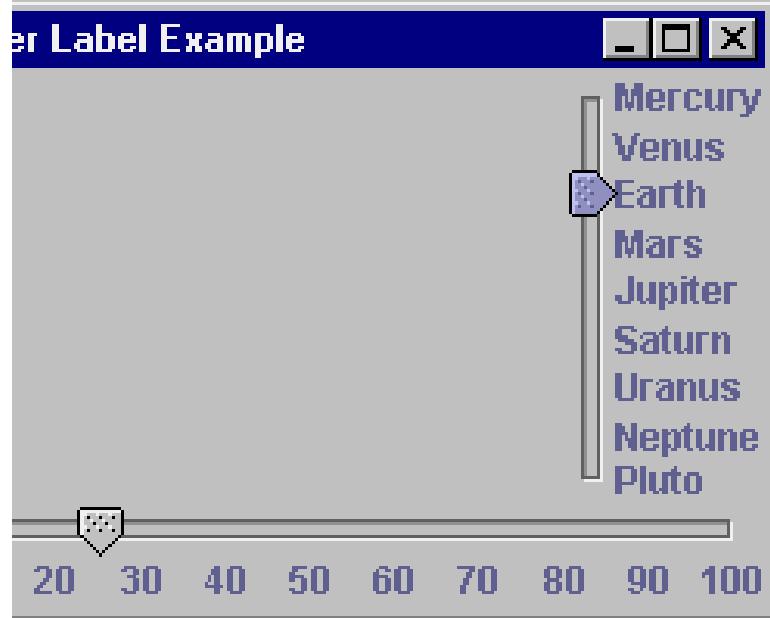
```
}
```



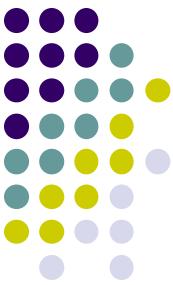


JSlider et Labels

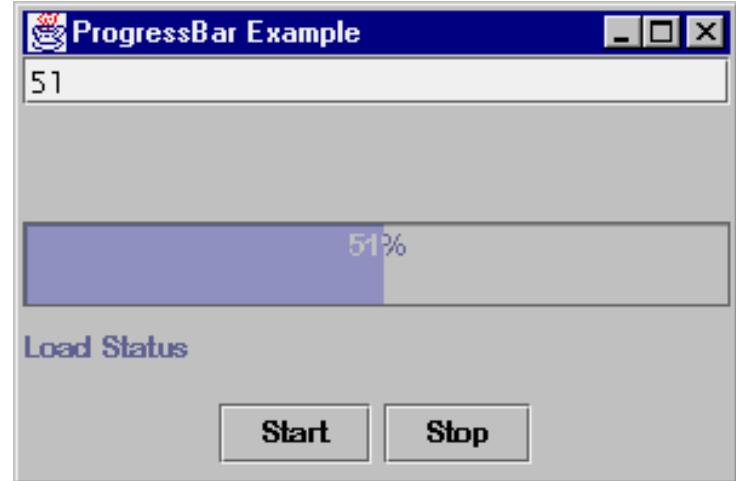
```
public class SliderPanel2 extends JPanel {  
    public SliderPanel2() {  
        setLayout(new BorderLayout());  
  
        JSlider right, bottom;  
        right = new JSlider(JSlider.VERTICAL, 1, 9, 3);  
        Hashtable h = new Hashtable();  
        h.put (new Integer (1), new JLabel("Mercure"));  
        ...  
  
        right.setLabelTable (h);  
        right.setPaintLabels (true);  
        right.setInverted (true);  
  
        bottom = new JSlider(JSlider.HORIZONTAL, 0, 100, 25);  
        bottom.setMajorTickSpacing (10);  
        bottom.setPaintLabels (true);  
        add(right, BorderLayout.EAST);  
        add(bottom, BorderLayout.SOUTH);  
    }  
}
```



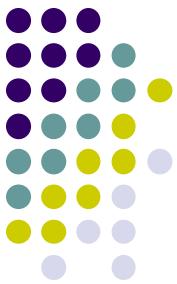
JProgressBar



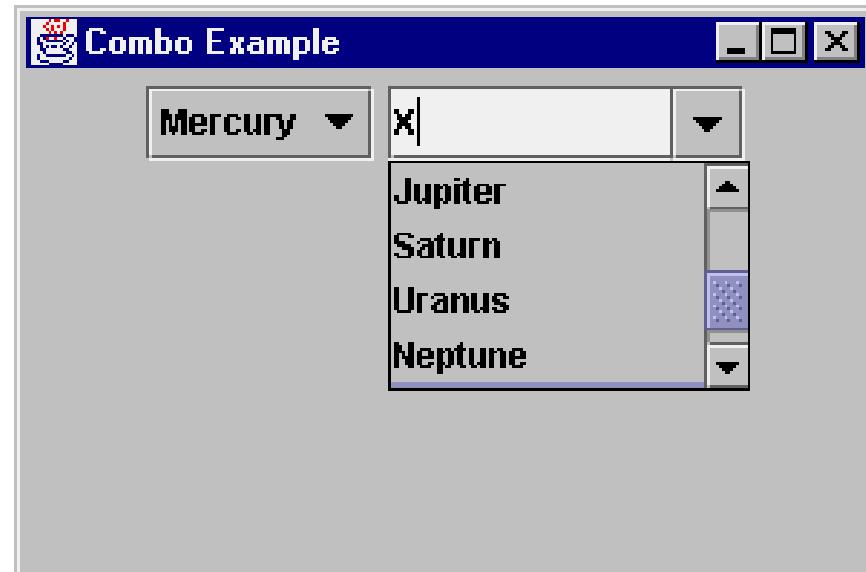
- Créer une Progress Bar
 - `progressBar = new JProgressBar(0, task.getLengthOfTask());`
 - `progressBar.setValue(0);`
 - `progressBar.setStringPainted(true);`
- Changer la valeur courante :
 - `progressBar.setValue(task.getCurrent());`
- Utilisation du mode « indeterminate »
 - `progressBar = new JProgressBar();`
 - `progressBar.setIndeterminate(true);`
 - *... // la taille devient connue*
 - `progressBar.setMaximum(newLength);`
 - `progressBar.setValue(newValue);`
 - `progressBar.setIndeterminate(false);`



JComboBox



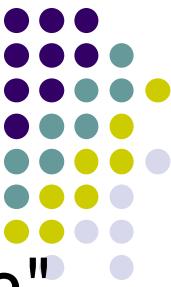
```
public class ComboPanel extends JPanel {  
    String choices[] = {"Mercure", "Venus", "Terre", "Mars",  
    "Jupiter", "Saturne", "Uranus", "Neptune", "Pluton"};  
    public ComboPanel() {  
        JComboBox combo1 = new JComboBox();  
        JComboBox combo2 = new JComboBox();  
        for (int i=0;i<choices.length;i++) {  
            combo1.addItem (choices[i]); combo2.addItem (choices[i]);  
        }  
        combo2.setEditable(true);  
        combo2.setSelectedItem("X");  
        combo2.setMaximumRowCount(4);  
        add(combo1);    add(combo2);  
    }  
    public static void main (String args[]) {  
    ...  } }
```





Callbacks ComboBox

```
public class ComboBoxDemo implements ActionListener {  
    ...  
    combo.addActionListener(this);  
    ...  
    public void actionPerformed(ActionEvent e) {  
        JComboBox cb = (JComboBox)e.getSource();  
        String item = (String)cb.getSelectedItem();  
        ...  
    }  
    ...  
}
```

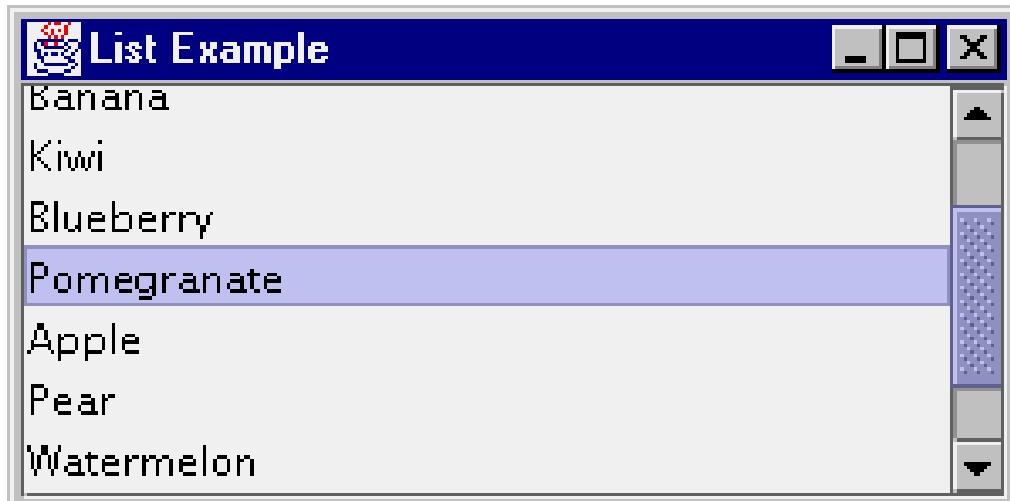


JList

```
String label [] = {"a", "b", "c", "d", "e", "f", "g", "h",
    "i", "j", "k" };
```

```
JList list = new JList(label);
```

```
JScrollPane pane = new JScrollPane(list);
```





JList : Sélection

```
static Vector v;  
l = new JList(v);  
l.setSelectionMode(  
    ListSelectionModel.SINGLE_SELECTION);  
// SINGLE_INTERVAL_SELECTION  
// MULTIPLE_INTERVAL_SELECTION
```





Modification dynamique de JList

```
listModel = new DefaultListModel();
listModel.addElement("A");
listModel.addElement("B");
listModel.addElement("C");
```

```
JList list = new JList(listModel);
```

```
...
```

```
listModel.remove(index);
```



listSelectionListener

```
public void valueChanged(ListSelectionEvent e) {  
    if (e.getValueIsAdjusting()) return;  
  
    JList theList = (JList)e.getSource();  
    if (theList.isSelectionEmpty()) {  
  
        ...  
    } else {  
        int index = theList.getSelectedIndex();  
  
        ...  
    }  
}
```

Borders

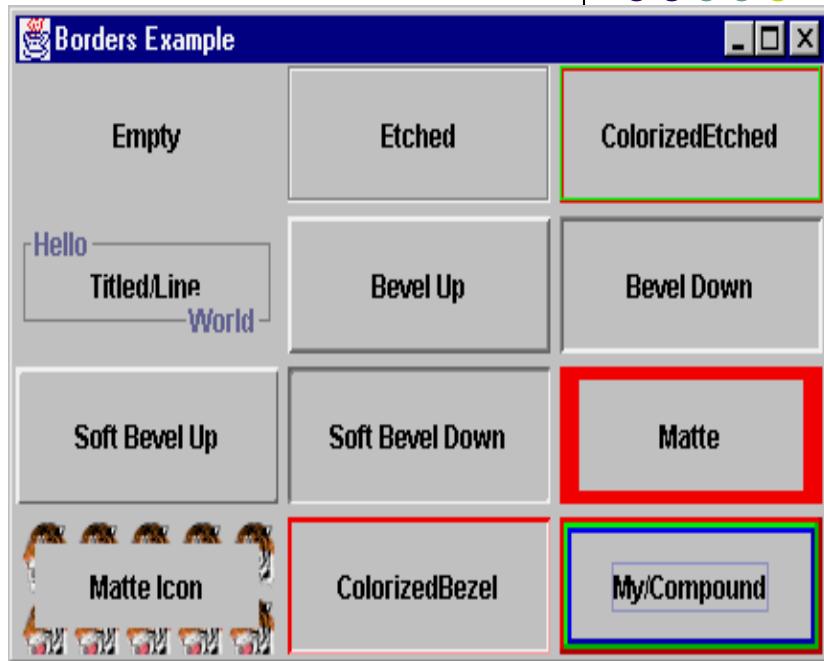


```
JButton b = new JButton("Empty");
b.setBorder (new EmptyBorder (1,1,1,1));
b = new JButton ("Etched");
b.setBorder (new EtchedBorder ());
b = new JButton ("ColorizedEtched");

b.setBorder (new EtchedBorder (Color.red, Color.green));
b = new JButton ("Titled/Line");
b.setBorder(new TitledBorder (
    new
    TitledBorder(LineBorder.createGrayLineBorder(),"Hello
    "),"World",
    TitledBorder.RIGHT, TitledBorder.BOTTOM));

b = new JButton ("Bevel Up"); b.setBorder(new
    BevelBorder(BevelBorder.RAISED));

b = new JButton ("Bevel Down"); b.setBorder(new
    BevelBorder(BevelBorder.LOWERED));
```



Borders

```
SoftBevelBorder(SoftBevelBorder.RAISED);
```

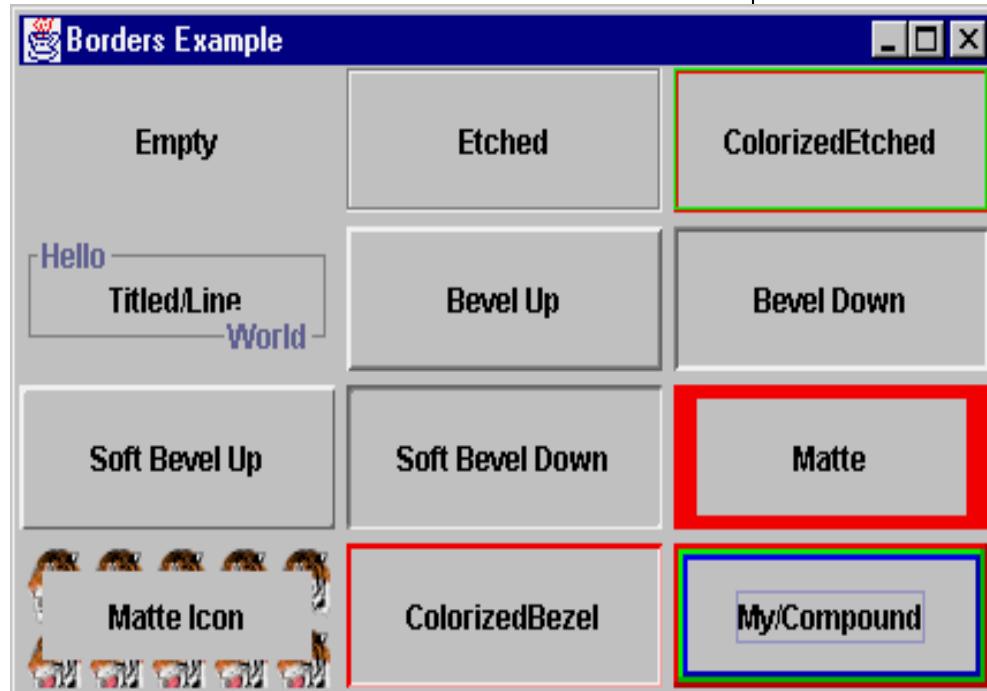
```
SoftBevelBorder(SoftBevelBorder.LOWERED);
```

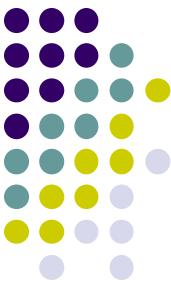
```
MatteBorder(5, 10, 5, 10, Color.red);
```

```
Icon icon = new ImageIcon ("file.gif");
new MatteBorder(10, 10, 10, 10, icon));
```

```
BevelBorder(BevelBorder.RAISED, Color.red,
Color.pink));
```

```
CompoundBorder(
    new MyBorder(Color.red),
    new CompoundBorder (new
        MyBorder(Color.green),
        new MyBorder(Color.blue)));
```





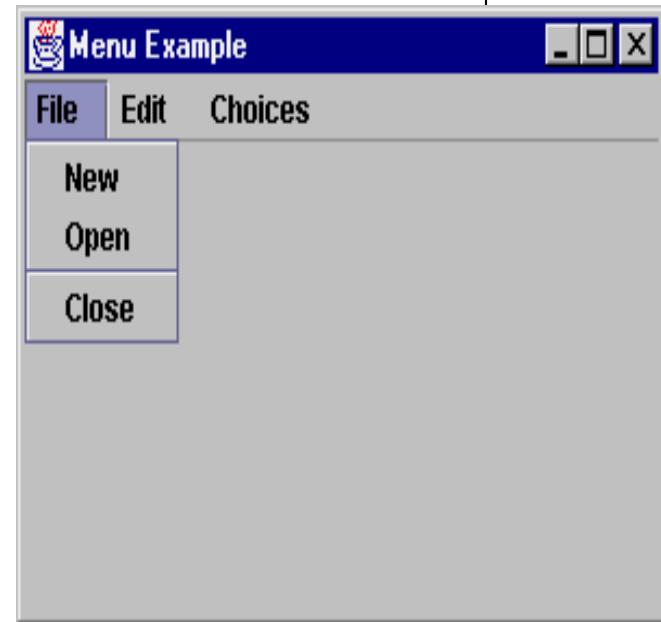
Menus

```
JMenuBar jmb = new JMenuBar();
JMenu file = new JMenu ("File");
file.addMenuListener (new MenuListener() {
    public void menuSelected (MenuEvent e) { ... }
    public void menuDeselected (MenuEvent e) { ... }
    public void menuCanceled (MenuEvent e) { ... }
});
```

```
JMenuItem item;
file.add (item = new JMenuItem ("New"));
file.add (item = new JMenuItem ("Open"))
file.addSeparator();
file.add (item = new JMenuItem ("Close"));
jmb.add (file);
```

...

```
setJMenuBar (jmb);
```





Callbacks sur menu items

```
menuItem.addActionListener(this);
```

```
...
```

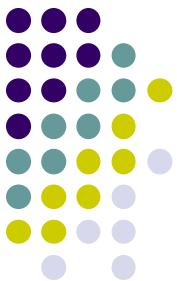
```
//JRadioButtonMenuItem:
```

```
rbMenuItem.addActionListener(this);
```

```
...
```

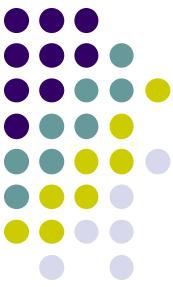
```
//JCheckBoxMenuItem:
```

```
cbMenuItem.addItemListener(this);
```



Sous menus

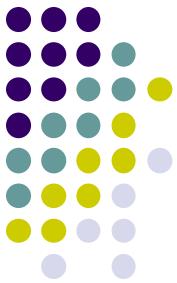
```
//submenu  
submenu = new JMenu("A submenu");  
submenu.setMnemonic(KeyEvent.VK_S);  
  
menuItem = new JMenuItem("dans le sous menu");  
menuItem.setAccelerator(KeyStroke.getKeyStroke(  
    KeyEvent.VK_2, ActionEvent.ALT_MASK));  
submenu.add(menuItem);  
  
...  
menu.add(submenu);
```



JPopupMenu

```
public class PopupPanel extends JPanel {  
    JPopupMenu popup = new JPopupMenu ();  
    public PopupPanel() {  
        popup.add (new JMenuItem ("Cut"));  
        ...  
        popup.setInvoker (this);  
  
        addMouseListener (new MouseAdapter() {  
            public void mousePressed (MouseEvent e) {  
                if (e.isPopupTrigger()) {  
                    popup.show (e.getComponent(), e.getX(), e.getY());  
                } }  
            public void mouseReleased (MouseEvent e) {  
                if (e.isPopupTrigger()) {  
                    popup.show (e.getComponent(), e.getX(), e.getY());  
                } } }); } }
```

Tooltips



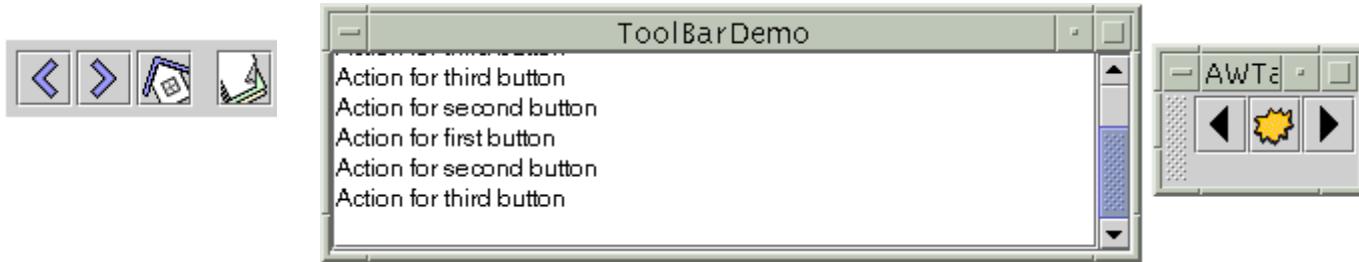
```
public class TooltipPanel extends JPanel {  
  
    public TooltipPanel() {  
  
        JButton myButton = new JButton("Hello");  
  
        myButton.setToolTipText ("World");  
  
        add(myButton);  
    }  
}
```





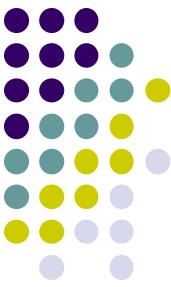
JToolBar

Une JToolBar est une barre de menu



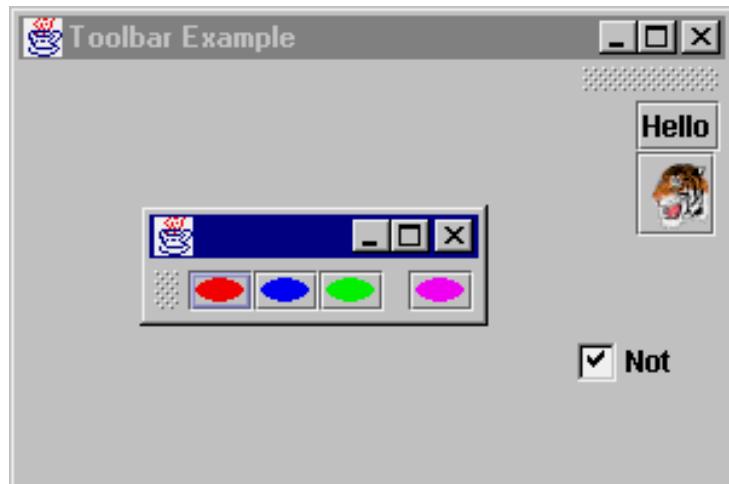
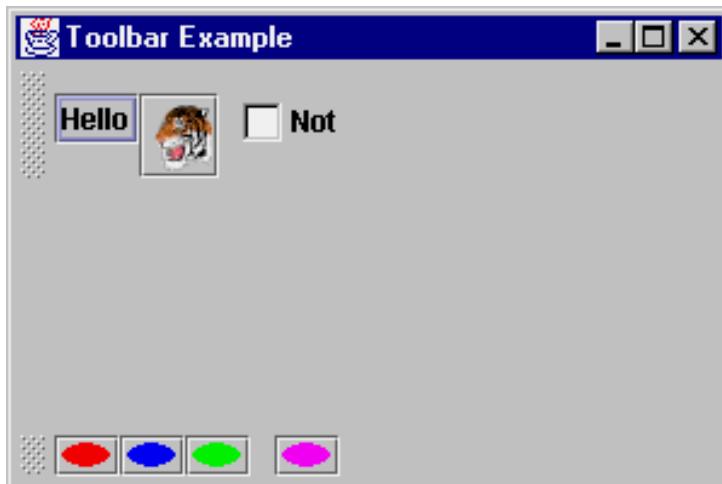
Quelques Méthodes :

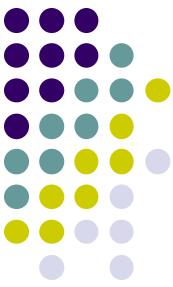
```
public JToolBar();
public Component add(Component c);
public void addSeparator();
```



Toolbars

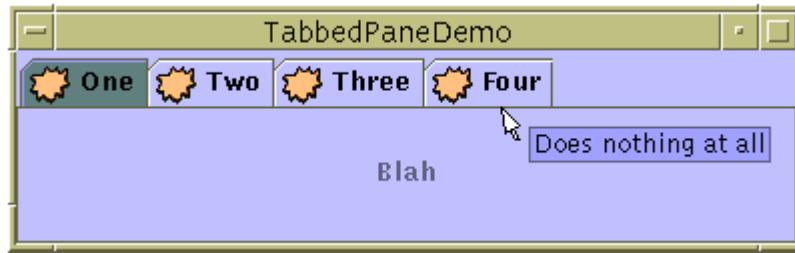
- JToolbar est un container qui permet d'afficher des toolbars déplaçables, éventuellement dans d'autres containers que celui d'origine. L'affichage du toolbar passe de vertical à horizontal suivant son emplacement.
- On peut désactiver la possibilité de rendre les toolbars flottantes.
`aToolBar.setFloatable (false);`





JTabbedPane

Un JTabbedPane permet d'avoir des onglets



Quelques méthodes :

```
public JTabbedPane();
```

```
public void addTab(String s, Icon i, Component c, String s);
```

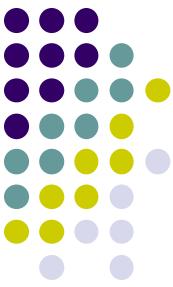
```
public Component getSelectedComponent();
```



JTabbedPane

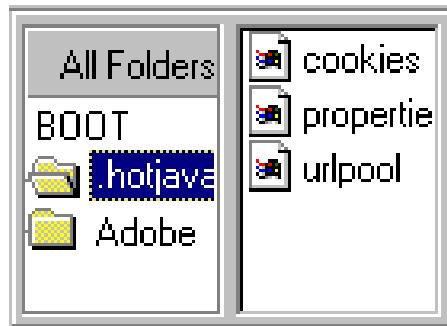
- JTabbedPane permet de réaliser des interfaces à onglets.
- On ajoute les onglets (des "cards") avec addTab().
Une des versions permet l'affichage d'un tooltip
- N'importe quel Component peut être affiché dans un onglet
 - addTab(String title, Component component)
 - addTab(String title, Icon icon, Component component)
 - addTab(String title, Icon icon, Component component, String tip)





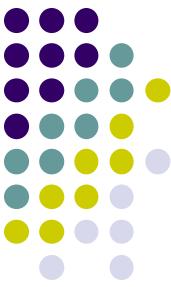
JSplitPane

Un JSplitPane est un panel coupé en deux par une barre de séparation. Un JSplitPane accueille deux composants.



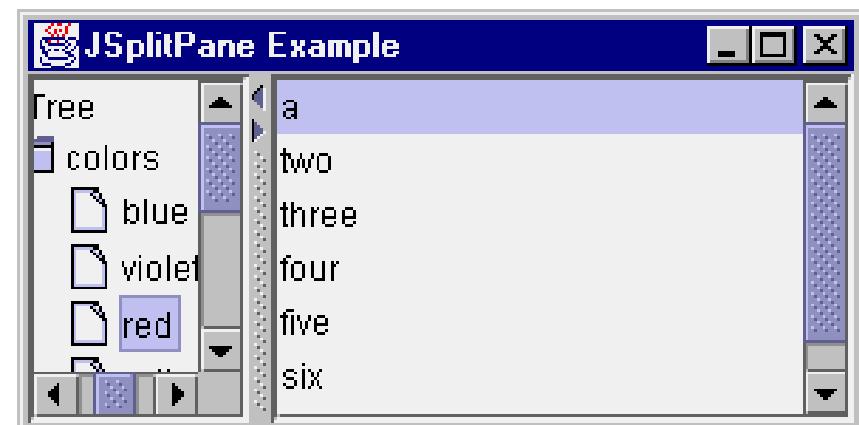
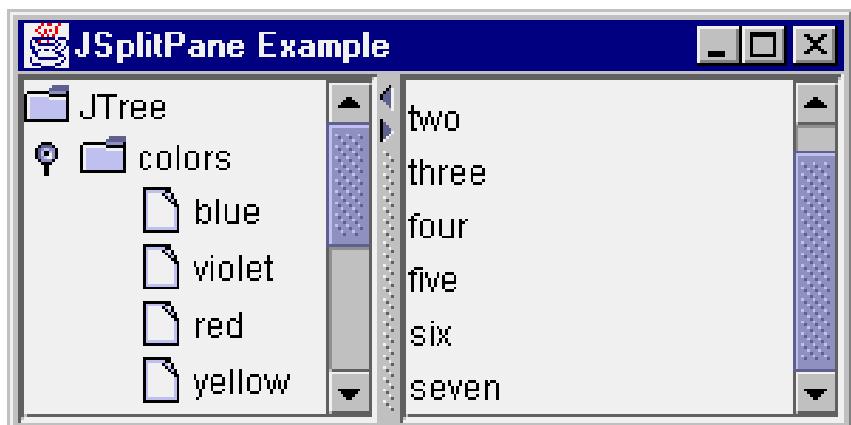
Quelques Méthodes :

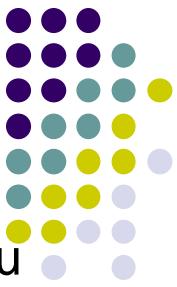
```
public JSplitPane(int ori, Component comp, Component c);  
public void setDividerLocation(double pourc);
```



JSplitPane

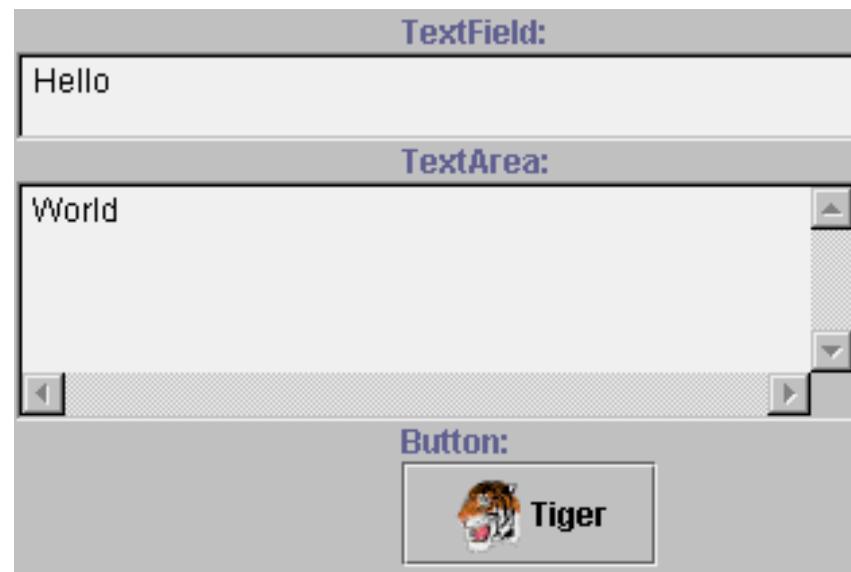
- JSplitPane permet le redimensionnement réciproque de deux fenêtres
- On peut placer un JSplitPane dans un autre : c'est un moyen de faire des interfaces compliquées sans gérer les layouts
- setContinuousLayout permet de voir le redimensionnement en direct
- On peut changer par programme la "dividerLocation"





BoxLayout

- Le BoxLayout layout arrange les composants selon l'axe horizontal ou vertical, mais plus intelligemment que le grid layout : les épaisseurs ou largeurs peuvent varier
- Il centre les composants ne pouvant pas être redimensionnés
- `setLayout(new BoxLayout(this, BoxLayout.Y_AXIS));`
- Le premier paramètre spécifie le container (JPanel par ex), et le second l'axe du BoxLayout. On ajoute les composants comme d'habitude
- `add(myComponent);`





Conclusion

- C'était un point de départ pour Javax Swing
- De la pratique maintenant.
- Doc en ligne (référence et exemples) sur
<http://www.java.com>

Aperçu sur le dessin avec Swing (1)



En AWT :

- Un appel à **repaint()** provoquant d'abord un appel à **update()**, qui est ensuite suivi d'un appel à **paint()** par défaut, les composants lourds peuvent redéfinir **update()** pour réaliser du « dessin incrémental » (le dessin incrémental n'est pas possible avec les composants légers)
- Les classes dérivées de **java.awt.Container** qui redéfinissent **paint()** devrait toujours invoquer dans le corps de cette méthode **super.paint()** pour s'assurer que tous ses composants sont redessinés.

Aperçu sur le dessin avec Swing (2)



- Comme pour l'AWT, en Swing on va utiliser les méthodes paint() et repaint(). Swing supporte de plus quelques propriétés additionnelles
 - le double buffering
 - le chevauchement des composants

Aperçu sur le dessin avec Swing (3)



- Pour tenir compte des spécificités des composants Swing, la méthode `paint()` va appeler 3 méthodes distinctes dans l'ordre suivant
 - **protected void paintComponent(Graphics g)**
 - **protected void paintBorder(Graphics g)**
 - **protected void paintChildren(Graphics g)**
- Par conséquent, un programme Swing devrait redéfinir
 - **paintComponent()** et non **paint()**
 - Généralement, il n'est pas nécessaire de redéfinir les 2 autres méthodes.