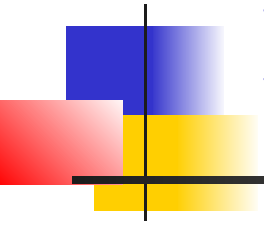
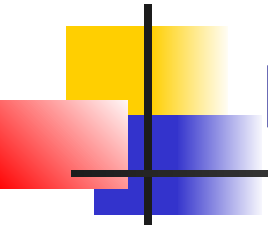


Programmation en Java

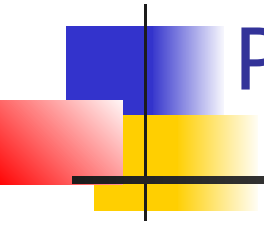
Mohammed Lahmer
m.lahmer@umi.ac.ma

2002-2023

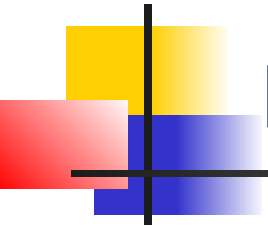


Plan Général

- Principe de la POO
- Présentation de Java
- Syntaxe de base
- Classes et Objets
- L'héritage
- Les collections
- Les entrées / Sorties
- L'interface graphique (Awt/Swing)
- L'accès aux bases des données

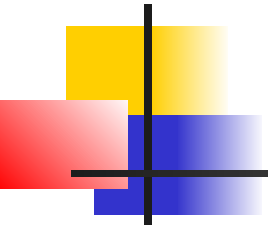


Principe de la P.O.O



Plan P. O. O

- Émergence du génie logiciel
- Approche fonctionnelle
- L’approche Entité/Relation
- L'analyse O. O
- Les messages
- L'héritage
- Typologies des L. O. O
- Persistance des Objets

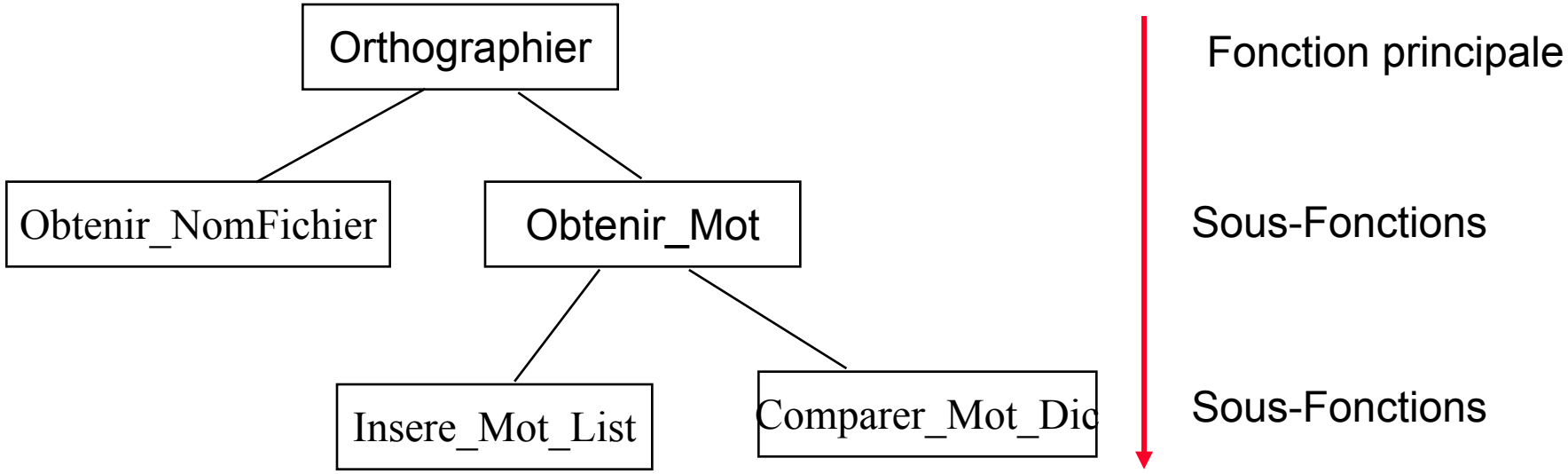


Émergence du génie logiciel

- Début des années 70, concentration des efforts sur l'activité de codage
  - Langage de haut niveau et programmation structurée (Pascal, Cobol, C…)
  - Programme= algorithme + structure de donnée
- Apparition des méthodes structurées
  - Méthode d'analyse
    - DFD (Data Flow Diagramme)
  - Méthodes de conception
    - SADT (Structured Analysis Design Technique)
    - JSD (Jackson System Development)

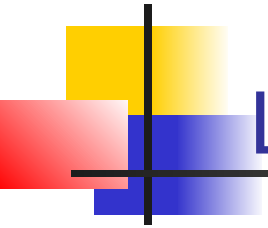
Approche fonctionnelle

- Le logiciel est composé d'une hiérarchie de fonctions, qui ensemble, fournissent les services désirés
- Chaque fonction est identifiée, puis décomposée en sous-fonctions jusqu'à l'obtention d'élément simple représentable par un langage de programmation (par les fonction et les procédures)



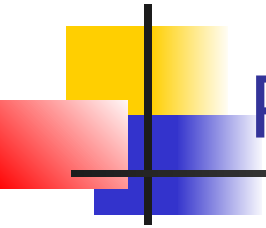
Problèmes de l'approche fonctionnelle

- Les méthodes s'intéresse essentiellement à l'interface externe du système (Orientées traitement)
- La programmation structurées ne prend pas en compte l'aspect évolutif du système
  - Les modules développés sont spécifiques à un besoin ponctuel
- L'approche descendante ordonne et hiérarchise les tâches
  - Centralisation du contrôle
  - Ordonnancement des modules figés trop tôt
- Les données et les opérations sont séparées : maintenance complexe en cas d'évolution



L'approche Entité/Relation

- Utilisée principalement dans le domaine de l'informatique de gestion
- Représenter un système à l'aide d'un ensemble d'entités et décrire leurs inter-relation : le modèle conceptuel
  - Représentation orientée données
  - Modèle E/R de Chen 76 : pas de point de vue dynamique
  - Une méthode d'analyse et de conception fournit :
    - Des concepts pour modéliser le système
    - Un processus aidant l'analyste dans sa démarche



Problèmes de l'approche E/R

- MERISE
  - Plus de concentration sur les données que les traitements
  - Schéma conceptuel sépare données et traitement
    - MCD et MCT
  - L'analyse et la conception n'utilisent pas des concepts homogènes
  - La coupure entre le modèle logique et physique est importante

L'approche Objet

- Fonder l'architecture du système sur des entités regroupant fonctions et données
  - Un objet est une abstraction du monde réel décrit en termes de propriétés (données) et de comportement (actions capable d'effectuer)
  - **Objet = données + traitements**
- Le système est vue comme une collection d'objets en interaction
- L'approche objet tire sa force de sa capacité à regrouper ce qui a été séparé à construire le complexe à partir de l'élémentaire, et surtout à intégrer statiquement et dynamiquement les constituants d'un système

Analyse Orienté Objet

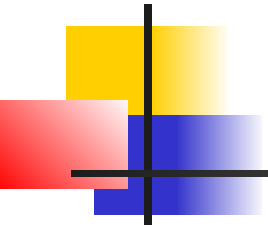
- L'analyse orienté objet s'appuie sur les principes suivants :
  - **Abstraction** : décrire formellement les données et les traitements en ignorants les détails
  - **Encapsulation** : masquer les données et les traitements en définissant une interface
    - L'interface et la vue externe de l'objet, elle définit les services offerts à l'utilisateur de l'objet
    - Faciliter l'évolution d'une application : On peut modifier les attributs d'un objet sans modifier sans interface
  - **Héritage** : reprendre intégralement tout ce qui a déjà été fait et de pouvoir l'enrichir : spécialisation
  - **Messages** : le seul moyen de communication entre les objets et l'envoi de messages

Les méthodes Objets

- OMT (James Rumbaugh) : vues statiques, dynamiques et fonctionnelles d'un système 91
  - Issue du centre de R&D de General Electric.
  - Notation graphique riche et lisible.
- OOD (Grady Booch) : vues logiques et physiques du système 91
  - Définie pour le DOD, afin de rationaliser le développement d'applications ADA, puis C++.
  - Ne couvre pas la phase d'analyse dans ses 1ères versions (préconise SADT).
  - Introduit le concept de package (élément d'organisation des modèles).
- OOSE (Ivar Jacobson) : couvre tout le cycle de développement 92
  - Issue d'un centre de développement d'Ericsson, en Suède.
  - La méthodologie repose sur l'analyse des besoins des utilisateurs

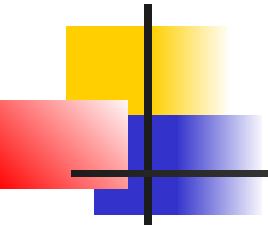
L'OMG (Object Managment Group)

- Avril 89 : Création de l'OMG
  - Regroupe plus de 200 membres
    - SunSoft, IBM, Bull, 3Com, HP, ObjectDesign, O2
  - Harmoniser les environnements O.O
  - Définir des standards pour l'industrie
- Les actions de l'OMG
  - Définition d'un modèle objet " Core Object Modèle"
  - Bases de données Objet "ODMG"
  - Inter-opérabilité des systèmes à objets : distribution (CORBA)
  - Normalisation de bibliothèque de réutilisation (OMA)



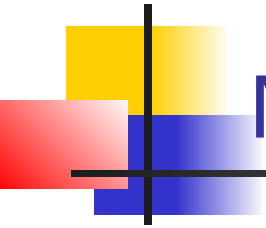
Les langages Objets

- Pourquoi Des Objets dans un langage ?
  - Traçabilité : continuité de représentation entre le schéma conceptuel et phisique
  - Langage de très haut niveau qui "capture" la sémantique de l'application
  - Unifier ce que la programmation traditionnelle a séparé (données et procédures)
- Emergence des langages objets
  - SIMULA : Norwegian Computing Center, 1965
  - SMALLTALK : Xerox Parc, 1972-1980
  - C++ : AT&T : 1983
  - EIFFEL : Meyer, 1989
  - JAVA : Sun, 1993-1997



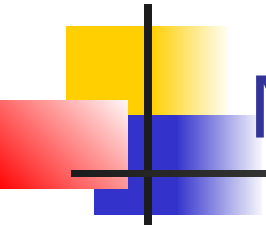
Objet / Classe

- Un objet peut être envisagé de façon pragmatique comme constitué de deux parties :
  - Une partie de stockage physique (Objet)
  - Une partie descriptive (Classe)
- L'Objet peut être apparenté à une variable de type structure en C (record en pascal, ligne d'une table dans une BDRelationnelle)
- Une classe n'est pas destinée au stockage des données
- C'est un type de données abstrait, caractérisé par des propriétés (attributs et méthodes) communes à des objets et permettant de créer des objets possédant ces propriétés.
  - Pile, File, ...
- Un objet est une représentation concrète d'une classe On parle d'**instance** d'une classe



Message

- Pour utiliser un service d'un objet particulier on lui envoie des messages
- Un message spécifie l'opération désirée mais ne dit rien sur la manière dont cette opération est mise en oeuvre
- Les données d'un objet ne peuvent être manipulées qu'à travers l'envoi de messages.
- **Un objet peut accepter un certain nombre de messages** : L'interface décrit les services fournis par l'objet
- Un message, une fois traité par le destinataire peut ou non rendre un résultats



Message vue de l'OMG

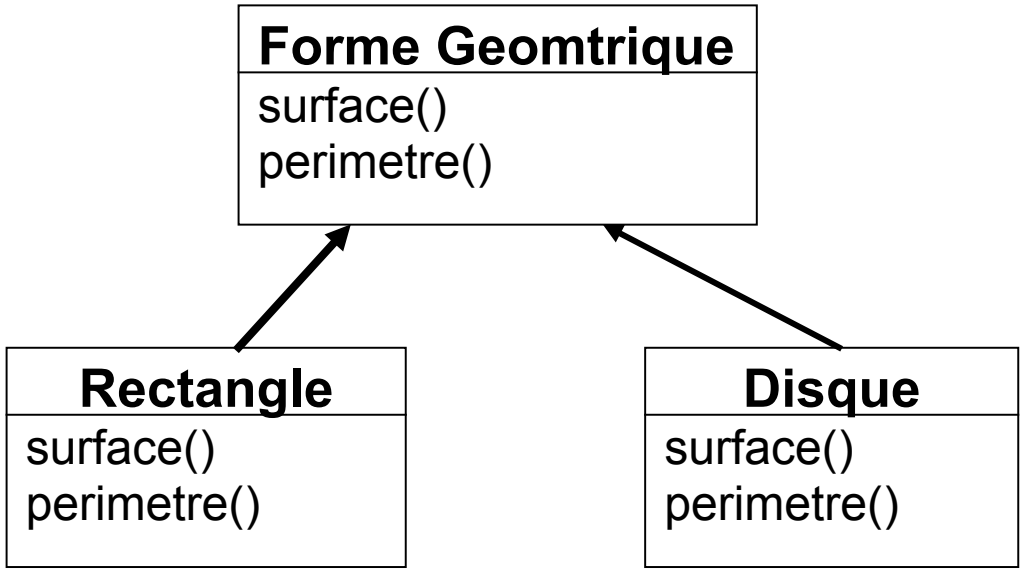
- Les effets de bord éventuels de l'exécution d'un message se manifestent par un changement d'état de l'objet
- Si une condition anormale est détectée lors de l'exécution d'un message alors une exception est retournée
- Une méthode est décrite par sa signature :
  - Nom de la méthode
  - Le type des arguments
  - " Le type de retour "
- Le type de résultat d'un message peut être :
  - Référence d'objet
  - Valeur atomique
  - Valeur structurée (tableau, Union, struct)

L'Héritage

- Soit une classe **A** déjà écrite que l'on veut réutiliser, et dont on a le code compilé
- Une classe **B** peut créer des instances de **A** et les utiliser (leur demander des services)
- On dit que la classe **B** est une classe cliente de la classe **A.** La classe Rectangle utilise la classe point
- Souvent, cependant, on souhaite modifier en partie le code de **A** avant de le réutiliser
- Problèmes si on veut copier, puis modifier le code source de **A** dans des classes **B1**, **B2**,... :
  - on n'a pas toujours le code source de **A**
- **L'héritage** permet de réutiliser dans une classe **B** le code d'une classe **A** sans toucher au code source initial

L'Héritage vue sémantique

- certaines classes ont des comportements communs



- surface(), perimetre() définissent le même comportement pour la classe Disque et la classe Rectangle
- **La généralisation** consiste à factoriser les éléments communs d'un ensemble de classes dans une classe plus générale appelée super-classe.

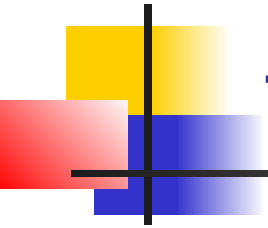
L'Héritage vue sémantique

- La **généralisation** indique toujours une relation de type "*est un*" ou « *est une sorte de* »
- La **généralisation** est une démarche assez difficile car elle demande une bonne capacité d'abstraction.
- L'héritage est la possibilité de pouvoir reprendre intégralement tout ce qui a déjà été fait (**réutilisation**)et de pouvoir l'enrichir : **spécialisation**
- Plutôt que modifier une classe existante, la dériver pour inclure de nouvelles fonctionnalités : **extensibilité**
- La relation d'héritage provoque la transmission de la partie descriptive d'une classe à une autre classe

Polymorphisme résultat de l'héritage

- Les noms des méthodes surface() et perimetre() sont similaires dans la classe rectangle et disque, mais surface d'un disque != surface d'un rectangle
- **Polymorphisme=** un même nom, plusieurs implémentations

```
Forme Geometrique forme[3] ={
Rectangle(2,3), Rectangle(1,5), Cercle(4)};
// Calcule de la somme des surface
float S=0;
for (int i = 0; i < 3; i++)
{
  S= S+ forme[i].surface();
}
```

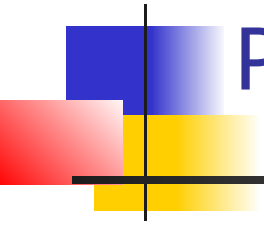


Typologie des L.O.O

- Langage compilé ou interprété
  - C++ : compilé
  - SMALLTALK : interprété
  - Java : semi compilé
- Présence ou absence de méta-classe
- Héritage simple ou multiple
  - C++ : multiple
  - Java : simple
- Aspect séquentiel ou non de l'exécution des messages

La persistance des objets

- Les objets non persistants sont dits transitoires ou éphémères. Par défaut, les objets ne sont pas considérés comme persistants.
- Dans leur ensemble, les langages de programmation objet ne proposent pas de support direct pour assurer la persistance des objets.
- Les constructeurs de bases de données fournissent des solutions pour la sauvegarde des objets, soit totalement objet, soit hybride.
- La gestion de la persistance est une notion intrinsèque des SGBD, elle permet le stockage des données et leur restitution à la demande.
  - ObjectStore, O2
- Contrairement aux SGBDR, les SGBDOO autorisent l'existence conjointe de variables persistantes et non persistantes.

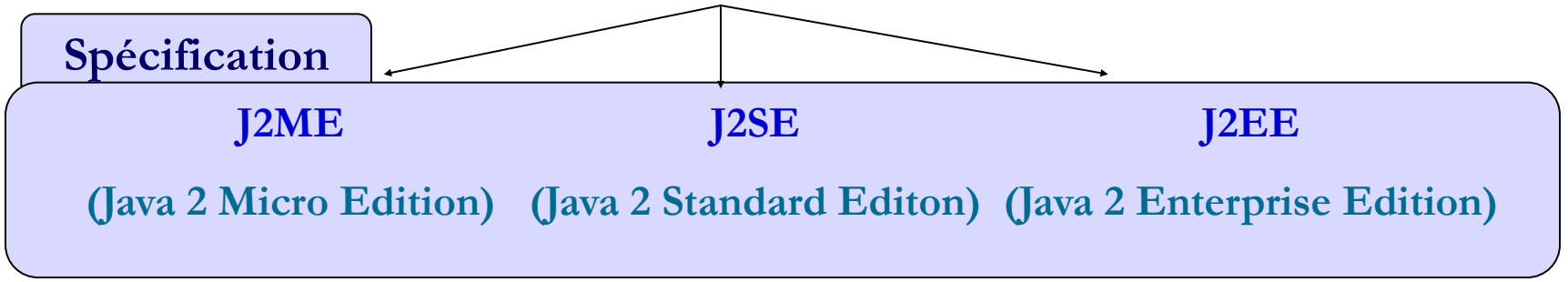


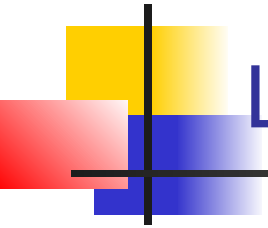
Présentation de Java

Bref historique

- **1993** : projet Oak (langage pour l'électronique grand public)
- **1995** : Java / HotJava
- **Mai 95** : Netscape prend la licence
- **Sept. 95** : JDK 1.0 b1
- **Déc. 95** : Microsoft se dit intéressé
- **Janv. 96** : JDK 1.0.1
- **Eté 96** : Java Study Group ISO/IEC JTC 1/SC22
- **Fin 96** : RMI, JDBC, JavaBeans, ...
- **Fév. 97** : JDK 1.1

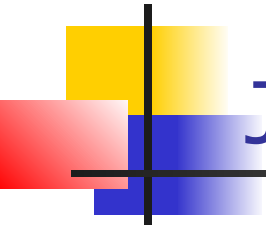
Décembre 98 **Java 2**





Les caractéristiques du langage Java

- Orienté objets
- Interprété
- Portable
- Simple
- Robuste
- Sécurisé
- Multi-threads
- Distribué

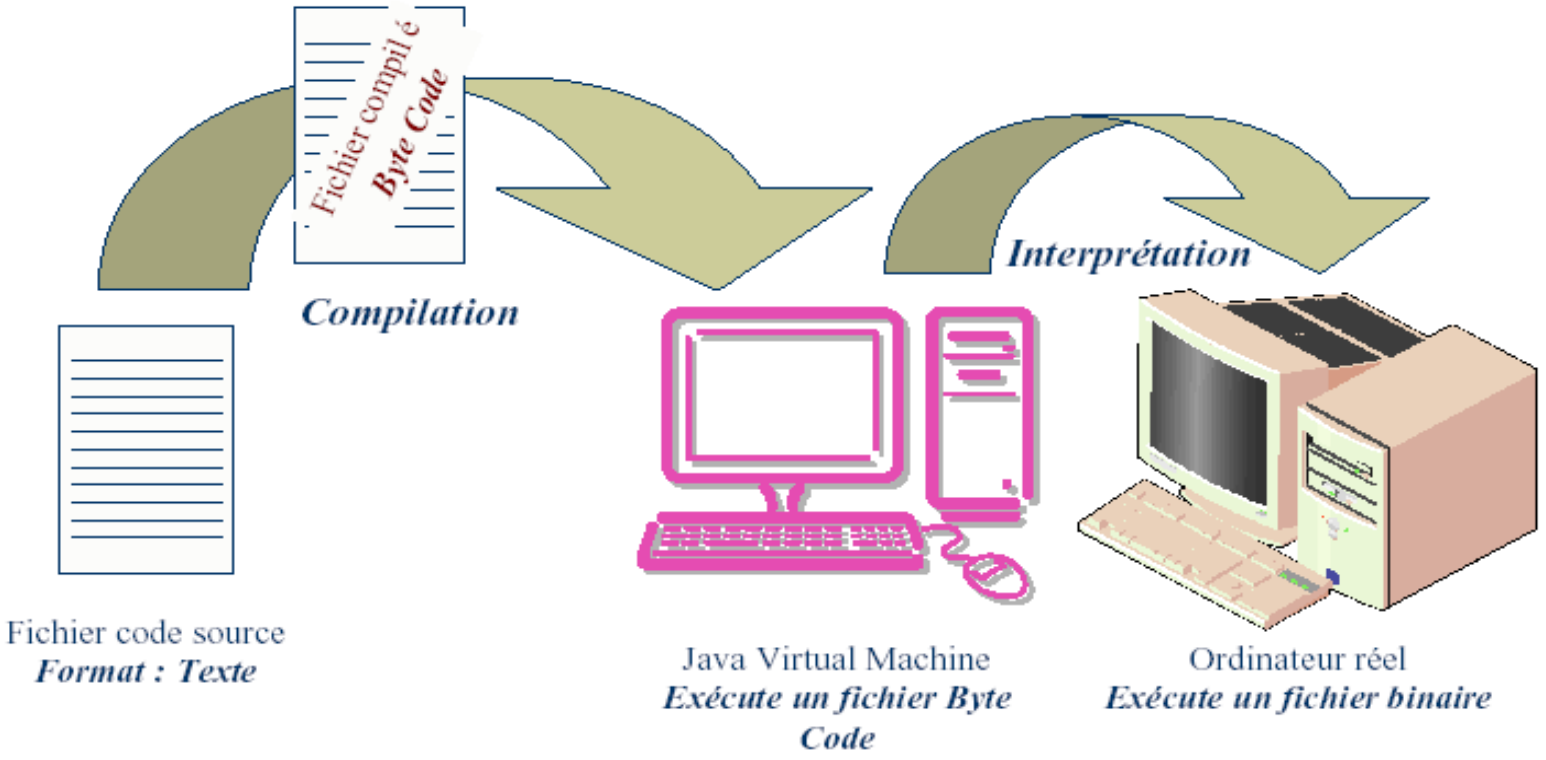


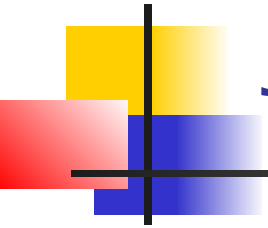
Java est un langage orienté objets

- Tout est classe (pas de fonctions à l'extérieure d'une classe) sauf les types primitifs (int, float, double, ...) et les tableaux
- Toutes les classes dérivent de java.lang.Object
- Héritage simple pour les classes
- Héritage multiple pour les interfaces
- Les objets se manipulent via des références
- Une API objet standard est fournie
- La syntaxe est proche de celle de C

Langage compilé et interprété

- Un langage compilé est un langage qui permet de créer un fichier exécutable à partir d'un programme source
- Un langage interprété est un langage ou le code source doit être lu par un interpréteur pour s'exécuter.



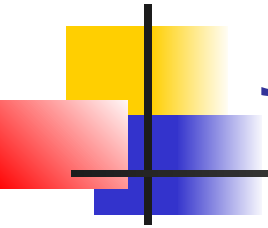


Java est portable

- Les programmes java (à part certains cas très particuliers) sont indépendants de la plate-forme.
- Il est d'abord traduit dans un langage appelé « **bytecode** » qui est le langage d'une machine virtuelle (JVM ; *Java Virtual Machine*) dont le langage a été défini par *Sun*
- La *Java Virtual Machine* (JVM) est présente sur Unix, Linux,Win32, Mac, OS/2, Netscape, IE, ...
- La taille des types primitifs est indépendante de la plate-forme.
- Java supporte un code source écrit en Unicode. Facilement internationalisable.

Java est robuste

- Gestion de la mémoire par un *garbage collector*. *Libère la mémoire d'un objet non utilisé*. Évite la surcharge
- Pas d'accès direct à la mémoire. Sécurité
- Mécanisme d'exception des erreurs système pris en charge par la JVM.(Accès à une référence null exception.)
- Compilateur contraignant (erreur si exception non gérée, si utilisation d'une variable non affectée, ...).
- Tableaux = objets (taille connue, débordement : exception)
- Contrôle des *cast* à l'exécution. Seules les conversions sûres sont automatiques.



Java est sécurisé

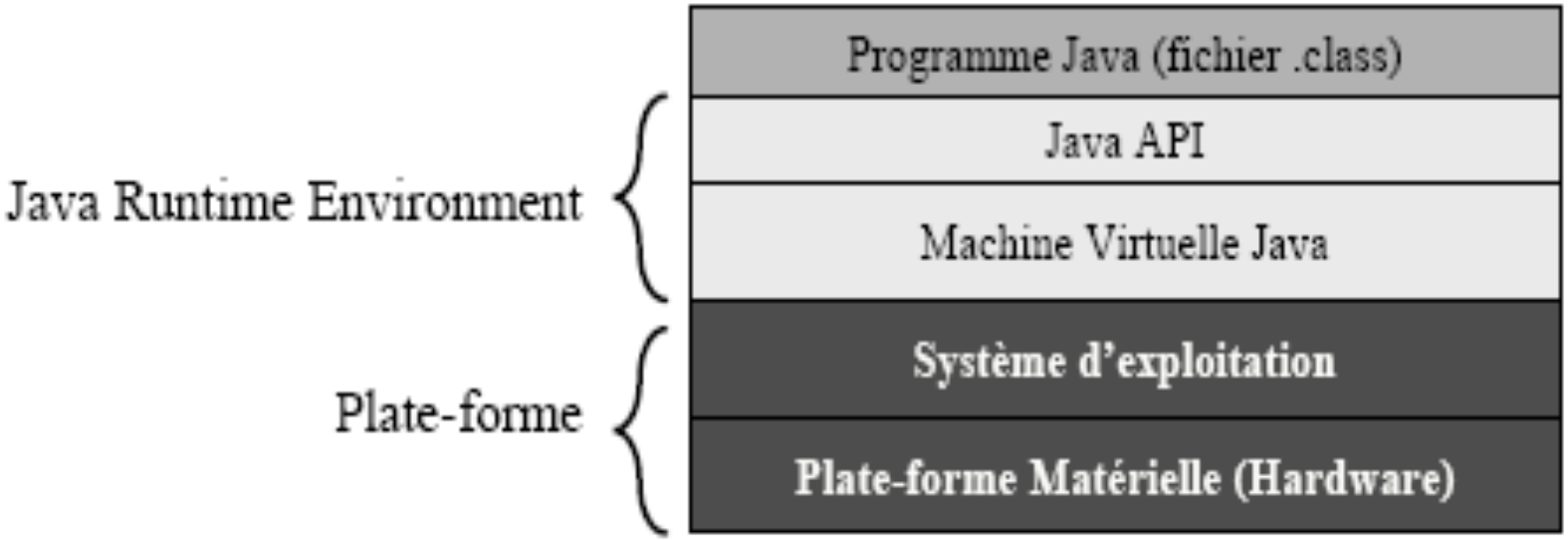
- Indispensable avec le code mobile (Applications répartis).
- Pris en charge dans l'interpréteur : Trois couches de sécurité :
  - *Verifier* : vérifie le *byte code*.
  - *Class Loader* : responsable du chargement des classes. (vérifier les autorisations private, protected, public)
  - *Security Manager* : accès aux ressources.
- Possibilité de certifier le code par une clé. (mécanismes de cryptographie et d'authenticité
- Pas d'accès directe à la mémoire

Java est multi-thread et Distibué

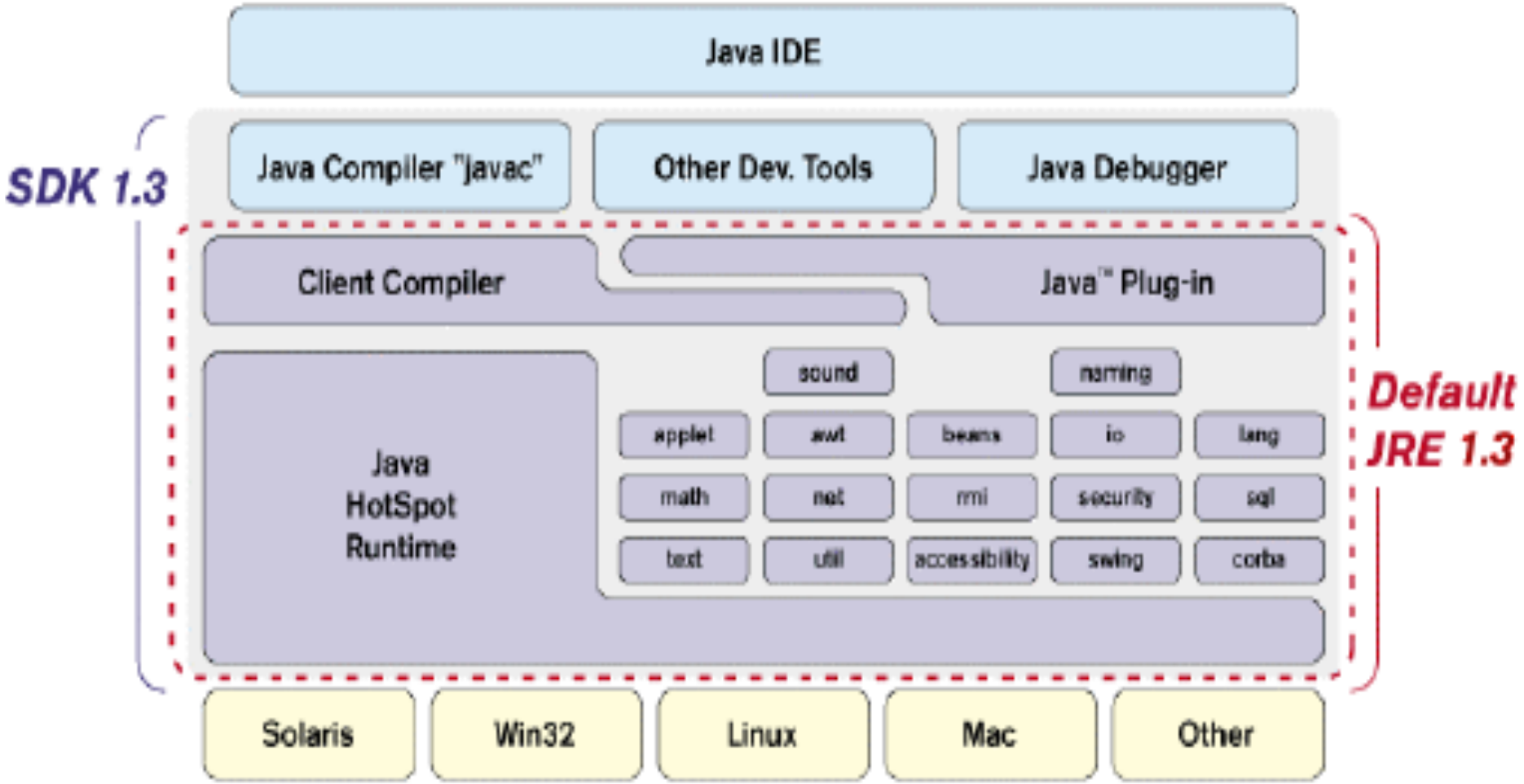
- Java est multi-thread
  - Intégrés au langage et aux API
  - Accès concurrents à objet gérés par un *monitor*.
  - Implémentation propre à chaque JVM->Difficultés pour la mise au point et le portage.
- Java est distribué
  - API réseau (java.net.Socket, java.net.URL, ...).
  - Chargement / génération de code dynamique.
  - Applet.
  - Servlet.
  - Remote Method Invocation.
  - JavaIDL (CORBA).

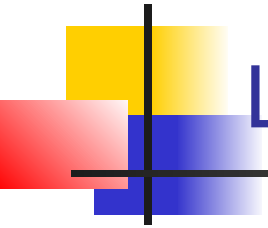
JVM / JRE

- Les machines virtuelles java sont fournies gratuitement par Sun Microsystems.
- Il sont fournis sous forme de JRE (Java Runtime Environment) comprenant la machine virtuelle et les package du noyau de java (APIs).



Structure JSDK



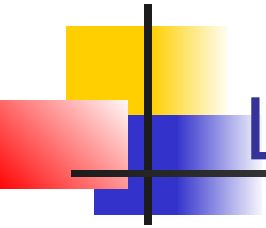


Les core API

- **java.lang** : Types de bases, Threads, ClassLoader, Exception, Math, ...
- **java.util** : Hashtable, Vector, Stack, Date, ...
- **java.applet**
- **java.awt** : Interface graphique portable
- **java.io** : accès aux i/o par flux, wrapping, filtrage
- **java.net** : Socket (UDP, TCP, multicast), URL, ...
- **java.lang.reflect** : introspection sur les classes et les objets
- **java.beans** : composants logiciels réutilisables
- **java.sql** (JDBC) : accès homogène aux bases de données
- **java.security** : signature, cryptographie, authentification
- **java.serialisation** : sérialisation d'objets
- **java.rmi** : *Remote Method Invocation*
- **java.idl** : interopérabilité avec CORBA

Les autres API

- **Java Server :** JSP/ servlets
- **Java Commerce*** : JavaWallet
- **JCA (Java Connector Architecture) :** pour la communication avec les systèmes existants tels que SAP, CICS/COBOL, Siebel,…
- **Java Média** : 2D*, 3D, Média Framework, Share, Animation, Telephony
- **Java Card:**Pour la programmation des cartes à puces.
- **Java Mail :**Modèle de messagerie indépendantes de la plate-forme et du protocole.
- **EmbeddedJava.** API pour les systèmes embarqués automobiles, usines intégrées
- **JDMK (**Java Dynamic Management Kit) : le gestion réseau



Les outils

- **javac** : compilateur de sources java
- **java** : interpréteur de *byte code*
- **appletviewer** : interpréteur d'applet
- **javadoc** : générateur de documentation (HTML, MIF)
- **javah** : générateur de *header* pour l'appel de méthodes natives
- **javap** : désassembleur de *byte code*
- **jdb** : debugger
- **javakey** : générateur de clés pour la signature de code
- **rmic** : compilateur de *stubs* RMI
- **rmiregistry** : "*Object Request Broker*" RMI

Premier programme Java

```
public class HelloWorld {
public static void main(String[] args){
System.out.println("Hello world"); // Afficher sur Ecran
}}
```

La classe Helloworld est **public**, donc le fichier qui la contient doit s'appeler (en tenant compte des majuscules et minuscules) **HelloWorld.java**

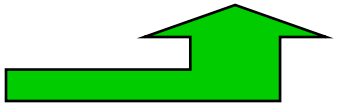
- **Compilation avec javac →** **javac HelloWorld.java**
  - crée un fichier « **HelloWorld.class** » qui contient le *bytecode*, situé dans le même répertoire que le fichier « **.java** »
- **Execution avec java →** **java HelloWorld**
  - interprète le bytecode de la méthode **main()** de la classe **HelloWorld**
  - **HelloWorld.class** doit être dans le répertoire courant ou dans un des emplacements indiqués par une option **-classpath** ou par la variable **CLASSPATH**

Complément Compilation exécution

- Ecrire un programme java qui calcule la distance entre deux points

```
/** Une classe point*/
public class Point {
private int x, y;
 // un constructeur
public Point(int x1, int y1) {
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);
Point p2 = new Point(5, 1);
System.out.println("Distance : " + p1.distance(p2));
}
}
```



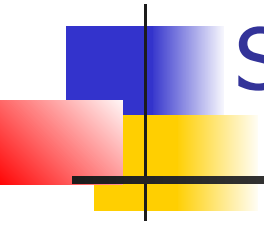
Complément compilation exécution

Fichier Point.java

```
/** 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, Point q)
{
return Math.sqrt((q.x-p.x)*(q.x-p.x) +
(q.y-p.y)*(q.y-p.y));
}}
```

Fichier TestPoint.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,p1)); } }
```



Structure du Langage

Structure du langage Java
Les types primitifs

- **boolean**(true/false), **byte** (1 octet), **char** (2 octets), **short** (2 octets), **int** (4 octets), **long** (8 octets), **float** (4 octets), **double** (8 octets).
- Les caractères sont codés par le codage Unicode (et pas ASCII) **'\u03a9'**
- Les variables peuvent être déclarées n'importe où dans un bloc.
- Les affectations non implicites doivent être castées (sinon erreur à la compilation).

```
int i = 258;
long l = i;   // ok
byte b = i;   // error: Explicit cast needed to convert int to byte
byte b = 258; // error: Explicit cast needed to convert int to byte
byte b = (byte)i; // ok mais b = 2
```

Structure du langage Java
la conversion de type

- En Java, **2 seuls cas** sont autorisés pour les *casts* :
  - entre types primitifs,
  - entre classes mère/ancêtre et classes filles
- Un *cast* entre types primitifs peut occasionner une perte de données Par exemple, la conversion d'un **int** vers un **short**
- Un *cast* peut provoquer une simple perte de précision. Par exemple, la conversion d'un **float** vers un **int**
- Attention, dans le cas d'un *cast* explicite, la traduction peut donner un résultat aberrant sans aucun avertissement ni message d'erreur :

```
int i = 130;
b = (byte)i; // b = -126 !
int c = (int)1e+30; // c = 2147483647
int d = (int)1.99; // d = 1
```

Structure du langage Java
Les types primitifs

- Valeurs par défaut
  - Si elles ne sont pas initialisées, les variables d'instance ou de classe (pas les variables locales d'une méthode) reçoivent par défaut les valeurs suivantes :

  boolean **false**

  char **'\u0000'**

  Entier (**byte short int long**) **0 0L**

  Flottant (**float double**) **0.0F 0.0D**

  Référence d'objet **null**
- Les constantes
  - Une constante « entière » est de type **long** si elle est suffixée par « L » et de type **int** sinon
  - Une constante « flottante » est de type **float** si elle est suffixée par « F » et de type **double** sinon : **.123587E-25F // de type float**

Structure du langage Java
Les identificateurs et les commentaires

- Un identificateur Java
  - est de longueur quelconque
  - commence par une lettre Unicode (caractères ASCII recommandés)
  - peut ensuite contenir des lettres ou des chiffres ou le caractère souligné _
  - ne doit pas être un mot-clé ou les constantes **true**, **false** et **null**
- Les commentaires
  - Sur une seule ligne **int prime = 1500; // prime fin de mois**
  - Sur plusieurs lignes :**/* Première ligne du commentaire**

    suite du commentaire */

Structure du langage Java
les commentaires pour la Doc

- Les zone de commentaires spécialisées sont délimitées par /** et */. 3 types de commentaire :

```
/** les commentaires de classe */
public class docClass {
/** les commentaires de variable */
public int i;
/** les commentaires de méthode */
public void m() {}
}
```

- L'outil pour extraire les commentaires est appelé *javadoc*.
- La sortie de javadoc est un fichier HTML qui peut être visualisé avec un browser Web.

Structure du langage Java
les commentaires pour la Doc

- Il faut noter que **javadoc** ne traite les commentaires de documentation que pour les membres **public** et **protected**.
- Il y a deux principales façons d'utiliser javadoc : du HTML intégré ou des « onglets doc ».
- Les onglets doc sont des commandes qui commencent avec un '@' et qui sont placées au début d'une ligne de commentaire

  @see classname

  @version version-information

  @author author-information

  @since depuis quelle version

  @param parameter-name description

  @return description

  @throws fully-qualified-class-name description

  @deprecated pourquoi?

Structure du langage Java
Exemple avec l'outils javadoc

```
import java.util.*;
/** Exemple de programme pour Génération de doc Java.
* Affiche la date du jour.
* @author Mohammed Lahmer
* @version 2.0
*/
public class HelloDate {
/** Unique point d'entrée de la classe et de l'application
* @param args tableau de paramètres sous forme de chaînes de
  caractères
* @return Pas de valeur de retour
**/
public static void main(String[] args) {
System.out.println("Hello, it's: ");
System.out.println(new Date());
}
```

- **javadoc -d c:\user\doc HelloDate.java**

Structure du langage
Les structures de contrôle et expressions

- Essentiellement les mêmes qu'en C
  - if, switch, for, while, do while
  - ++, +=, &&, &, <<, ?:,!,~,^…
- Plus les blocs labellisés

```
UN: while(...) {
    DEUX: for(...) {
        TROIS: while(...) {
        if (...) continue UN;// Reprend sur la première boucle while
        if (...) break DEUX;  // Quitte la boucle for
        continue;// Reprend sur la deuxième boucle while
        }}}
```

Structure du langage
Portée d'une variable

```
{
int x = 12;
/* seul x est accessible */
{
int q = 96;
/* x & q sont tous les deux accessibles
   */
}
/* seul x est accessible */
/* q est « hors de portée » */
}
```

```
{
int x=12;
{
int x = 96; /* illegal */
}
}
/* en C et C++ c est
autorisé*/
```

Structure du langage
Les tableaux

- **Déclaration**

  int[] array_of_int; // équivalent à : int array_of_int[];

  Color rgb_cube[][][];

- **Création et initialisation**

  array_of_int = **new** int[42];

  rgb_cube = **new** Color[256][256][256];

  int[] primes = {1, 2, 3, 5, 7, 7+4};

  array_of_int[0] = 3

- **Utilisation**

  int l = array_of_int.length; // l = 42

  l'indice commence à **0** et se termine à **length - 1**

  int e = array_of_int[50]; /* Lève une ArrayIndexOutOfBoundsException*/

Structure du langage
Les tableaux

- Copier une partie d'un tableau dans un autre
  - la classe **System** fournit une méthode **static** performante
  - **public static void arraycopy(Object src, int src_position,Object dst, int dst_position, int length)**
- Comparer 2 tableaux
  - **java.util.Arrays.equals()**
  - par exemple, **java.util.Arrays.equals(double[] a1, double[] a2)**
- Paramètres de ligne de commande

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

- **java Arguments toto titi** **//affichera toto puis titi**

Les constructeurs

- Chaque classe a un ou plusieurs constructeurs qui servent à
  - créer les instances
  - initialiser l'état de ces instances
- Un constructeur
  - a le même nom que la classe
  - n'a pas de type retour
- Lorsque le code d'une classe ne comporte pas de constructeur, un constructeur sera automatiquement ajouté par Java
- Pour une classe **Classe**, ce constructeur par défaut sera :

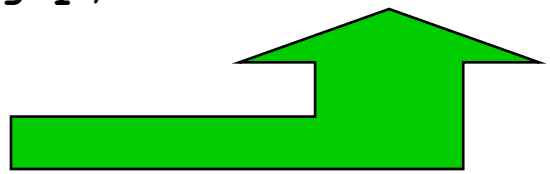
  [public] Classe() { }

Les constructeurs

```
public class Employe {

private String nom, prenom;

private double salaire;

// Ce constructeur appelle l'autre
constructeur

public Employe(String n, String p) {

this(n, p,0);

}

public Employe(String n, String p,
double s) {

nom = n;

prenom = p;

salaire = s;}. . .
```

```
e1 = new
Employe("Dupond",
"Pierre");

e2 = new
Employe("Durand",
"Jacques", 15000);
```

Surcharge de méthodes



Les méthodes

- Quand la méthode renvoie une valeur, on doit indiquer le type de la valeur renvoyée dans la déclaration de la méthode :

  double calculSalaire(int indice, double prime)

- Le pseudo-type **void** indique qu'aucune valeur n'est renvoyée :

  void setSalaire(double unSalaire)

```
public class Employe {
private double salaire;
public void setSalaire(double
unSalaire) {
if (unSalaire >= 0.0)
salaire = unSalaire;
}
public double getSalaire() {
return salaire;
}}
```

Modificateur

Accesseur

Surcharge d'une méthode

- **Signature d'une méthode** : nom de la méthode et ensemble des types de ses paramètres
- Signature de la méthode **main()** : **main(String[])**
- En Java, on peut surcharger une méthode, c'est-à-dire, ajouter une méthode qui a le même nom mais pas la même signature qu'une méthode existante :
  - **calculerSalaire(int)**
  - **calculerSalaire(int, double)**
- En Java, il est interdit de surcharger une méthode en changeant le type de retour
- Par exemple, il est interdit d'avoir ces 2 méthodes dans une classe :
  - **int calculerSalaire(int)**
  - **double calculerSalaire(int)**

Passage des paramètres

- Le mode de passage des paramètres dans les méthodes dépend de la nature des paramètres :
  - par référence pour les objets
  - par copie pour les types primitifs

```
public class C {
    void methode1(int i, StringBuffer s) { i++; s.append("d");}

    void methode2() {
        int i = 0;
        StringBuffer s = new StringBuffer("abc");
        methode1(i, s);
        System.out.println(i=" + i + ", s=" + s); // i=0, s=abcd
    }}
```

Classes / Objets
méthodes et variables de classe

- **Variables de classe**
  - Certaines variables peuvent être partagées par toutes les instances d'une classe. Ce sont les variables de classe (modificateur **static** en Java)
  - Si une variable de classe est initialisée dans sa déclaration, cette initialisation est exécutée une seule fois quand la classe est chargée en mémoire
  - Les variables **static** sont communes à toutes les instances de la classe.

```
public class Employe {

private static int nbEmployes = 0;

// Constructeur

public Employe(String n, String p) {

nom = n;prenom = p;nbEmployes++;}
```

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

Employe e1=new
Employe("Toto","Titi");

System.out.println("Nombre
Employer
est :"+Employe.nbEmployer);}}
```

méthodes et variables de classe

- **Méthodes de classe**
  - Une méthode de classe (modificateur **static** en Java) exécute une action indépendante d'une instance particulière de la classe
  - Elle ne peut utiliser de référence à une instance courante (**this**)
  - **static double tripleSalaire() {return salaire * 3;} // NON**
  - La méthode **main()** est nécessairement **static.**Pourquoi ?
  - Pour désigner une méthode **static** depuis une autre classe, on la préfixe par le nom de la classe :
    - **int n = Employe.nbEmploye();**
  - On peut aussi la préfixer par une instance quelconque de la classe (à éviter car cela nuit à la lisibilité : on nevoit pas que la méthode est **static**) :
    - **int n = e1.nbEmploye();**

méthodes et variables de classe

- Les blocs **`static`** permettent d'initialiser les variables **`static`** trop complexes à initialiser dans leur déclaration :
- Ils sont exécutés **une seule fois**, quand la classe est chargée en mémoire

```
class UneClasse {
private static int[] tab = new int[25];
static {
for (int i = 0; i < 25; i++) {
tab[i] = -1;
}
} // fin du bloc static

.  .  .
```

Variable d'état final

- Le modificateur **final** indique que la valeur de la variable (d'état ou locale) ne peut être modifiée :
- On pourra lui donner une valeur une seule fois dans le programme (à la déclaration ou ensuite)
- Une variable de classe **static final** est vraiment constante dans tout le programme ;

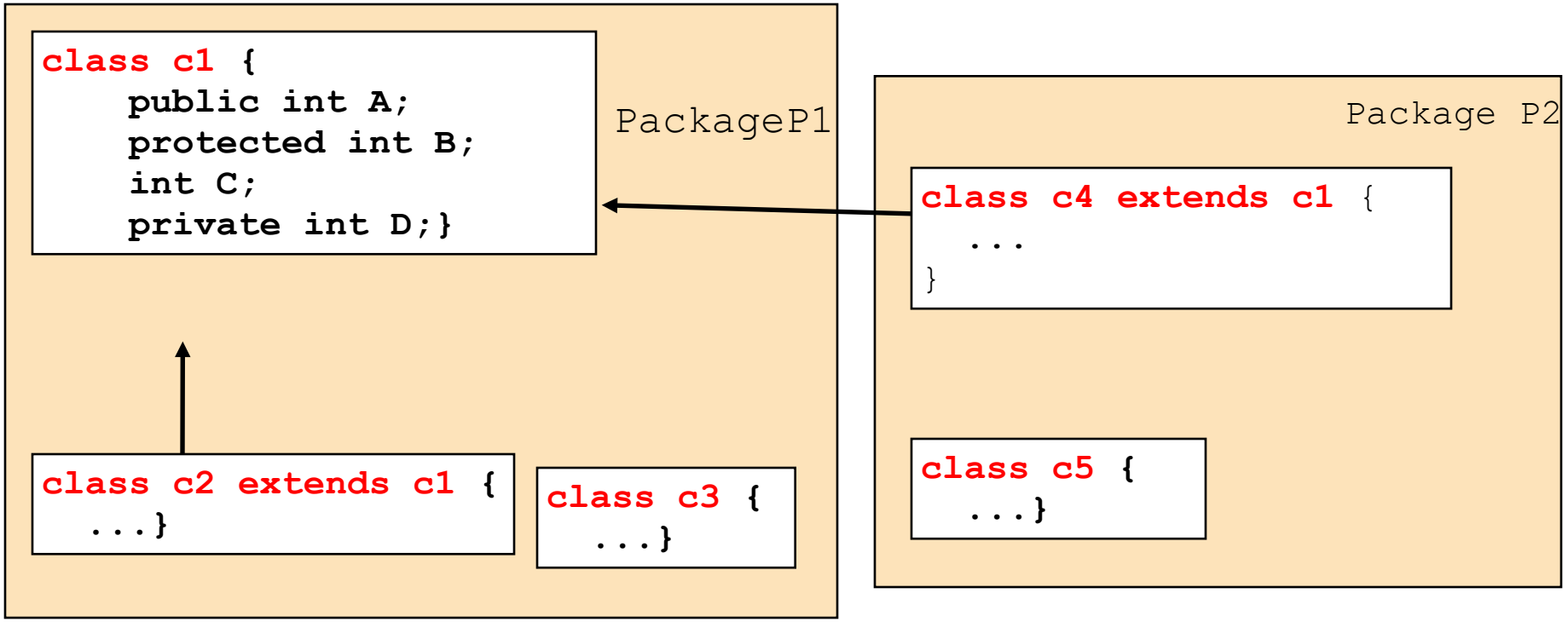
```
static final double PI = 3.14;
```

- Une variable *d'instance* (pas **static**) **final** est constante pour chaque instance ; mais elle peut avoir 2 valeurs différentes pour 2 instances
- Une variable d'instance **final** peut ne pas être initialisée à sa déclaration mais elle doit avoir une valeur à la sortie de tous les constructeurs

Les packages

- Un package regroupe un ensemble de classes sous un même espace de 'nomage'.
- Les noms des packages suivent le schéma : name.subname ...
- Une classe employe appartenant au package societe.travail doit se trouver dans le fichier **societe\travail\employe.class**
- L'instruction package indique à quel package appartient la ou les classe(s) de l'unité de compilation (le fichier).
- Les répertoires contenant les packages doivent être présents dans la variable d'environnement CLASSPATH
- En dehors du package, les noms des classes sont : **packageName.className**
- L'instruction **import packageName** permet d'utiliser des classes sans les préfixer par leur nom de package.

Classes / Objets
L'encapsulation des membres



	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	-	-	-

Classes particulières

Les chaînes de caractères

- 2 classes :
  - **String** pour les chaînes constantes
  - **StringBuffer** pour les chaînes variables
- On utilise le plus souvent **String**, sauf si la chaîne doit être fréquemment modifiée
- **L'affectation** d'une valeur littérale à un **String** (pas à un **StringBuffer**) s'effectue par :
  - **chaine = "Bonjour";**
- La spécification de Java impose que
  - **chaine1 = "Bonjour";**
  - **chaine2 = "Bonjour";**
  - crée un seul objet **String** (référencé par les 2 variables)

Classes particulières
Les chaînes de caractères

- **Concaténation**
  - **String s = "Bonjour" + " les amis";**
  - **int x = 5;**
  - **s = "Valeur de x + 1 = " + x + 1;**
  - S'il y a plusieurs **+** dans une expression, les calculs se font de gauche à droite
  - s contient **Valeur de x + 1 = 51**
- Attention, la concaténation de **String**s est une opération coûteuse (elle implique en particulier la création d'un **StringBuffer**)
- Passer explicitement par un **StringBuffer** si la concaténation doit se renouveler. Une seule création d'un **StringBuffer** pour toutes les concaténations

```
c = new StringBuffer(t[0]);

for (int i = 1; i < t.length; i++)
c.append(t[i]);
```

Classes particulières
Les chaînes de caractères

- **Égalité**
- La méthode **equals()** teste si 2 instances de **String** contiennent la même valeur :
- **String s1, s2;**
- **s1 = "Bonjour ";s2 = "les amis";**
- **if ((s1 + s2).equals("Bonjour les amis"))**
- **System.out.println("Egales");**
- **equalsIgnoreCase()** pas de différence entre majuscule et minuscule
- **s.compareTo(t)** renvoie (« signe » de **s-t**)
  - 0 en cas d'égalité de s et de t,
  - un nombre entier positif si **s** suit **t** dans l'ordre lexicographique
  - un nombre entier négatif sinon
- **compareToIgnoreCase(t)**

Classes particulières

Les chaînes de caractères

- Les caractères d'une **String** sont numérotés de **0** à **length()-1**
- Convertir un objet en **String toString()**
- Le ième caractère : **charAt(i)**
- Extraire une sous-chaîne : **substring(int debut,int nbchar)**
- Rechercher la position des sous-chaînes :
  - **indexOf(String sousChaine)**
  - **indexOf(String sousChaine, int debutRech)**
  - idem pour **lastIndexOf()**
- Exercice
  - Ecrire un programme Java qui permet de récupérer le protocole, la machine, le répertoire et le nom du fichier une adresse Web **http://www.iam.ma/rep1/rep2/fichier.html**

Classes particulières

Les classes enveloppes

- Le paquetage **java.lang** fournit des classes pour envelopper les types primitifs (les objets sont constants) : **Byte, Short, Integer,Long, Float, Double, Boolean,Character.**
- les classes enveloppes offre des méthodes utilitaires (le plus souvent **static**) pour faire des conversions entre types primitifs (et avec la classe **String**)
  - **Int→ new Integer** (constructeur de **Integer**) : **new Integer(int i)**
  - **Integer→int** : **int intValue()**
  - **String→ Integer** : **static Integer valueOf(String ch [,int base])**
  - **Integer → String** : **String toString()**
  - **String→ int** : **static int parseInt(String ch [,int base]) de Integer**
  - **int→ String** (méthode de la classe **String**) : **static String valueOf(int i)**
- String s = String.valueOf(2 * 3.14159**);** **// s = "6.28318"**
- Integer i = Integer.valueOf("123"); **// i = 123**

Classes particulières
Classe System

- Cette classe est très utile pour diverses opérations d'entrée et sortie, d'émission d'erreurs, d'accès aux propriétés du système, et bien d'autres fonctionnalités relatives à des tâches système.
- **System.getProperty(NomPropriete)** : renvoie la valeur de la propriété system
- **System.gc()** : appel le Garbage collector
- **System.runFinalization()** : appel finalize sur tous les objets non utilisés
- **System.exit(code retour)** : arrêt immédiat de la JVM

/*Recuperation des variables environnements*/

- Nom Utilisateur : System.getProperty("user.name")
- Répertoire de travail : System.getProperty("user.home")
- Nom OS : System.getProperty("os.name")
- Version OS : System.getProperty("os.version");

Classes particulières
Classe System

- **Ramasse Miettes (garbage Collector)**
- Une variable dont le type est une classe contient une référence à un objet.
- Lorsque l'objet n'est plus référencé, un « **ramasse-miettes** » (garbage collector) libère la mémoire qui lui a été allouée.
- Le ramasse-miettes (garbage collector) est une tâche(job) qui
  - travaille en arrière-plan
  - libère la place occupée par les instances non référencées
- Il intervient quand le système a besoin de mémoire ou, de temps en temps, avec une priorité faible (voir Thread)
- Une destruction manuelle des objets reste possible par appel explicite du ramasse miette. Classe System
  - **Méthodes static** : gc(), runFinalization()

Classes particulières
Classe System

- Avant la suppression d'un objet par le ramasse miettes, le système appelle la méthode **finalize()** de la classe de l'objet (si elle existe)

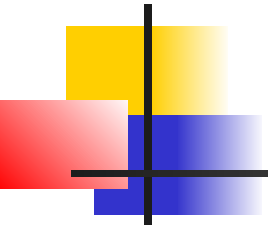
```
class garbage{
  static boolean gcok=false; // devient true lorsque le GC
  est appelé
  static int count;
  public  garbage(){count++;}//  Compter  le  nombre
  dinstances crées
  public void finalize(){
  if(gcok==false)  System.out.println("Appel GC:\n"+
  "Nombre d'instance crees/supprimées : "+count);
  gcok=true;}
  public static void main(String[] args)
  {   while(!gcok) {new garbage();}}}
```

Classes particulières

Les exceptions (1)

- Elles permettent de séparer un bloc d'instructions de la gestion des erreurs pouvant survenir dans ce bloc.

```
try {
/* Code pouvant lever des IOException ou des
   SecurityException*/
}

catch (Exception e){
   // Gestion de toutes les autres exceptions}
finally{
    // Code qui sera exécuté dans tous les cas}
```

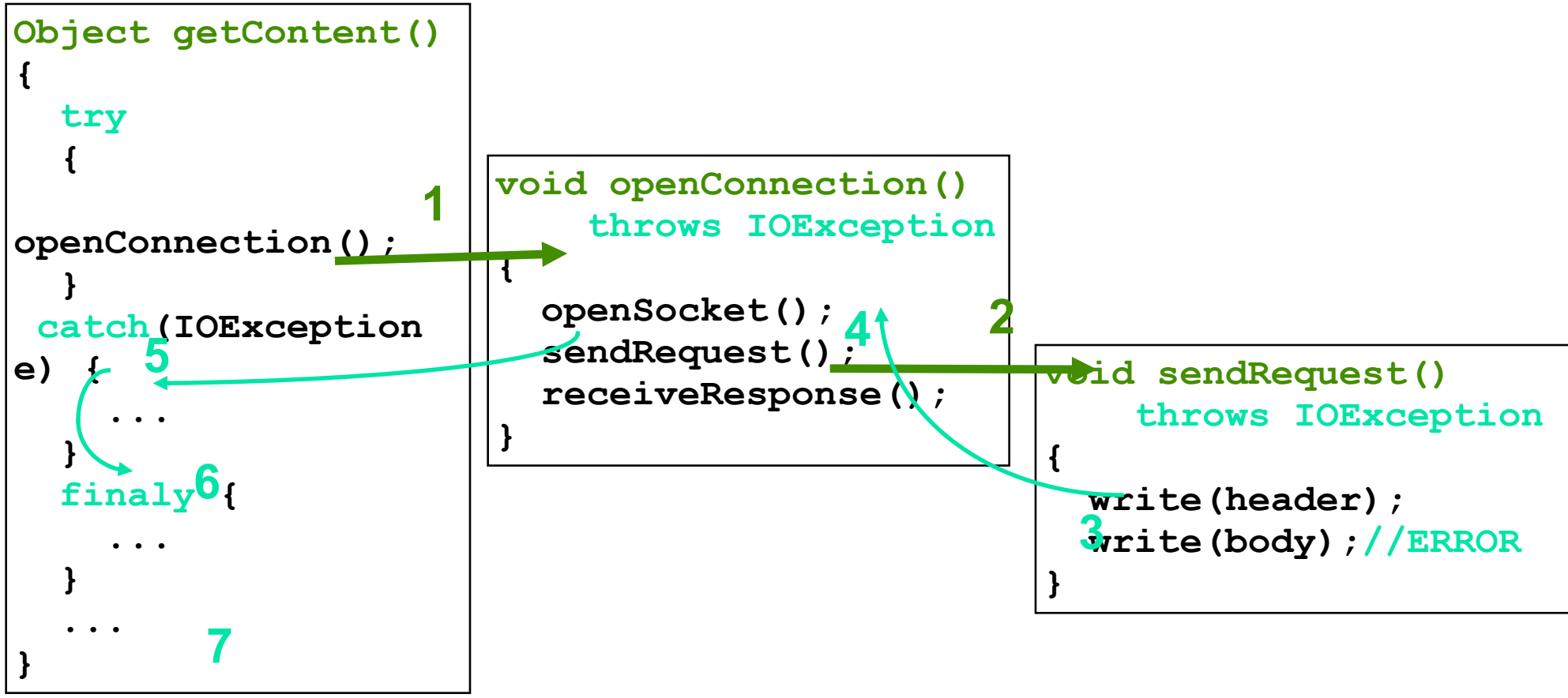


Classes particulières

Les exceptions (2)

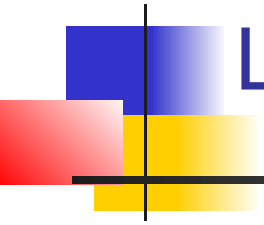
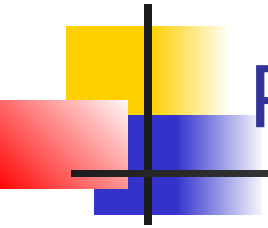
- Ce sont des instances de classes dérivant de **java.lang.Exception**
- La levée d'une exception provoque une remontée dans l'appel des méthodes jusqu'à ce qu'un bloc catch acceptant cette exception soit trouvé. Si aucun bloc catch n'est trouvé, l'exception est capturée par l'interpréteur et le programme 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
  - void fonction throws IOException { // . . . }

Classes particulières
Les exceptions (3)



Classes particulières

Définir ses propres exceptions

- ```
  public class MonException extends Exception
  { public MonException(String text)
  { super(text); }}
  ```
- On utilise le mot clé **throw** pour lever une exception en instanciant un objet sur une classe dérivée de la classe Exception.
- **throw new** MonException("blabla");
- Exemple de classe d'Exception
  - **IOException**
  - **NullPointerException**
  - **OutOfMemoryException**
  - **ArithmeticException**
  - **ClassCastException**
  - **ArrayIndexOutOfBoundsException**
  ```

Principe d'identification de type en Java

- Comment Java découvre dynamiquement des informations sur les objets et les classes?
- Les objets sont crées comme des instances d'une classe (new). Mais comment trouver les information associée à la classe?
- Java utilise un objet **Class (**meta-classe**)** pour chacune des classes d'un programme.
- Durant l'exécution, lorsqu'un nouvel objet d'une classe doit être créé, la MachineVirtuelle Java (*JVM*) vérifie d'abord si l'objet **Class** associé est déjà chargé.
- Si non, la JVM le charge en cherchant un fichier **.class** du même nom.
  - `try {`
  - `Class.forName("MaClasse");} catch(ClassNotFoundException e) {}`
  - `Équivalent à Maclass.class`

Principe d'identification de type en Java

- La méthode newInstance() permet de créer une nouvelle instance de type Object
  - `Maclasse c=null;`
  - `try {`
  - `c=(MaClasse)Class.forName("MaClasse").newInstance();}`
  - `catch(ClassNotFoundException e) {}`
  - `catch(InstantiationException e) {}`
  - `catch(IllegalAccessException e) {}`
- Pour vérifier le type d'un objet on utilise la méthode isInstance() qui fait appel directement au mot clef instanceof
  - `If (Class.forName("MaClasse").isInstance(c)) {}`
- La méthode **getInterfaces()** retourne un tableau d'objets Class représentant les interfaces qui sont contenues dans l'objet en question.
- La méthode **getSuperclass()** retourne un objet de type Class représentant la classe mère de l'objet

Principe d'identification de type en Java

- Le type d'un objet doit être connu à la compilation afin que vous puissiez le détecter en utilisant le RTTI
- Que faire si vous ne connaissez pas le type de l'objet que vous chercher (une classe téléchargée de l'Internet)
- La classe **Class** supporte le concept de ***réflexion***, et une bibliothèque additionnelle, **java.lang.reflect**, contenant les classes **Field**, **Method**, et **Constructor**
- On peut alors utiliser les constructeurs pour créer de nouveaux objets
- les méthodes **get()** et **set()** pour lire et modifier les champs associés à des objets **Field.**
- De plus, on peut utiliser les méthodes très pratiques **getFields()**, **getMethods()**, **getConstructors()**
  - `Class c = Class.forName("MaClasse");`
  - `Method[] m = c.getMethods();`
  - `Constructor[] ctor = c.getConstructors();`



L'héritage



Plan Héritage

- Généralités sur l'héritage
- Le polymorphisme
- Les classes abstarites
- Les Interfaces
- Réutiliser des classes (Composition + Délégation)
- Classes Internes
- Clonnage des objets

Réutilisation par l'héritage

- Il minimise les modifications à effectuer : on indique seulement ce qui a changé dans **B** par rapport au code de **A** ; on peut par exemple
  - ajouter de nouvelles méthodes ou attributs
  - modifier certaines méthodes
- La classe **A** s'appelle une classe mère, classe parente ou **super-classe**
- La classe **B** qui hérite de la classe **A** s'appelle une classe fille ou **sous-classe**
- Exemples d'héritage
  - La classe mère **Vehicule** peut avoir les classes filles **Velo**, **Voiture** et **Camion**
  - La classe **Employe** peut hériter de la classe **Personne**
  - La classe **Image** peut avoir comme classe fille **ImageGIF** et **ImageJpeg**

L'héritage en Java

- En Java, chaque classe a une et une seule classe mère (pas d'héritage multiple) dont elle hérite les variables et les méthodes
- Le mot clef **extends** indique la classe mère :**class RectangleColore extends Rectangle**
- Par défaut (pas de **extends** dans la définition d'une classe), une classe hérite de la classe **Object**
- La classe qui hérite peut
  - **ajouter** des variables, des méthodes et des constructeurs
  - **redéfinir** des méthodes (exactement la même signature et le même type retour)
  - **surcharger** des méthodes (même nom mais pas même signature) (possible aussi à l'intérieur d'une classe)
- Mais elle ne peut retirer aucune variable ou méthode
- Si « **B extends A** », le grand principe est que tout **B** est un **A**

Exemple Réutilisation par l'héritage

```
//super-classe
public class Rectangle {
private int x, y; // point en haut
   à gauche
private int largeur, hauteur;
// Constructeurs
// Méthodes getX(), setX(int),…
// getLongueur(), getLargeur(),
// setLongueur(int),
   setLargeur(int),
// contient(Point),
   intersecte(Rectangle),…
// translateToi(Vecteur),
   toString()
public void dessineToi(Graphics g)
   {
   g.drawRect(x, y, largeur,
   hauteur);}}
```

```
//sous-classe
public class RectangleColore
   extends Rectangle {
private Color couleur; // nouvelle
   variable
// Constructeurs
. . .
// Nouvelles Méthodes
public getCouleur() { return
   this.couleur; }
public setCouleur(Color c)
   { this.couleur = c; }
// Méthodes modifiées
public void dessineToi(Graphics g)
   {
   g.setColor(couleur);
   g.fillRect(getX(),
   getY(),getLargeur(),
   getHauteur());}}
```

Complément sur le constructeur

- La première instruction (interdit de placer cet appel ailleurs !) d'un constructeur peut être un appel
  - à un constructeur de la classe mère : **super(...)**
  - ou à un autre constructeur de la classe : **this(...)**

```
public class RectangleColore extends Rectangle {

private Color couleur;

public RectangleColore(int x, int y, int largeur, int hauteur,
Color couleur) {

super(x, y, largeur, hauteur);

this.couleur = couleur;}

public RectangleColore(int x, int y, int largeur, int hauteur)
{

this(x, y, largeur, hauteur, Color.black);}}
```

Polymorphisme

- Supposons que **B** hérite de **A** et que la méthode **m()** de **A** soit redéfinie dans **B** Quelle méthode **m()** sera exécutée dans le code suivant, celle de **A** ou celle de **B** ?
  - **A a = new B();** **//a est un objet de la classe B**
  - **a.m();**
- La méthode appelée ne dépend que du type réel (**B**) de l'objet **a** (et pas du type déclaré, ici **A**). C'est la méthode de la classe **B** qui sera exécutée
- **Le polymorphisme** est le fait qu'une même écriture peut correspondre à différents appels de méthodes ;

Polymorphisme

```
public class Figure {
public void dessineToi() { }
}
public class Rectangle extends
    Figure {
public void dessineToi() {
. . .
}
public class Cercle extends Figure.
    {
public void dessineToi() {
...
}
```

```
public class Dessin { // dessin composé de
            //plusieurs figures
private Figure[] figures;
. . .
public void afficheToi() {
for (int i=0; i < nbFigures; i++)
figures[i].dessineToi();
}
public static void main(String[] args) {
Dessin dessin = new Dessin(30);
// création des points centre, p1, p2
dessin.ajoute(new Cercle(centre, rayon))
dessin.ajoute(new Rectangle(p1, p2));
dessin.afficheToi();
...
}
```

Polymorphisme

- Bien utilisé le polymorphisme évite les codes qui comportent de nombreux embranchements et tests ; sans polymorphisme, la méthode **dessineToi()** aurait dû s'écrire :

```
for (int i=0; i < figures.length; i++) {
if (figures[i] instanceof Rectangle) {
. . .// dessin d'un rectangle}
else if (figures[i] instanceof Cercle) {
. . .// dessin d'un cercle}}
```

- Le polymorphisme **facilite l'extension** des programmes : on peut créer de nouvelles sous classes sans toucher aux programmes déjà écrits
- Par exemple, si on ajoute une classe **Losange**, code de **afficheToi()** sera toujours valable

Conversion de classes

- On parle de *upcast* et de *downcast* en faisant référence au fait que la classe mère est souvent dessinée au-dessus de ses classes filles
- **Upcast** : un objet est considéré comme une instance d'une des classes ancêtres de sa classe réelle. Le *upcast* est souvent implicite (polymorphisme): **figures[0] = new Cercle(p1, 15);**
- **Downcast** : un objet est considéré comme étant d'une classe fille de sa classe de déclaration Un *downcast* doit toujours être explicite
- Le *downcast* est utilisé pour appeler une méthode de la classe fille qui n'existe pas dans une classe ancêtre
  - **Vehicule c12 = new Camion();**
  - **((Camion)c12).charge(livraison);**

Classes final et autres final

- Classe **final** : ne peut avoir de classes filles (**String** et java.util.**Vector** sont **final**)
- Méthode **final** : ne peut être redéfinie
- Variable (locale ou d'état) **final** : la valeur ne pourra être modifiée après son initialisation
- Paramètre **final** (d'une méthode) : la valeur (éventuellement une référence) ne pourra être modifiée dans le code de la méthode

```
Class GeometriePlane {
public static final double PI=22/7;
void methode(){
PI=3.14; // interdits
}….}
```

Classes et méthodes abstraites

- Une méthode est abstraite (modificateur **abstract**) lorsqu'on la déclare, sans donner son implémentation
- Une classe doit être déclarée abstraite (**abstract class**) si elle contient une méthode abstraite
- Il est interdit de créer une instance d'une classe abstraite
- Une méthode **static** ne peut être abstraite (car on ne peut redéfinir une méthode **static**)

```
abstract class Geometrie{
//Methode abstraite
abstract double perimetre();
…}
```

```
class Rectangle extends
geometrie {
double L,l;
double perimetre(){
return L*l;
…}
```

Les interfaces

- Une interface est une « classe » purement abstraite dont toutes les méthodes sont abstraites et publiques (les mots-clés **abstract** et **public** sont optionnels dans la déclaration des méthodes)
- En fait, une interface correspond à un **service rendu** qui correspond aux méthodes de l'interface

```
public interface Figure {
void dessineToi();
void deplaceToi(int x, int
y);
Position getPosition();}
```

```
class rectangle implements Figure{
void dessineToi(){
    /*Dessiner Rectangle*/}
    void deplaceToi(int x, int y){
    /*Deplacer Rectangle*/}
    Position getPosition(){
    /*Retourner la position du
    rectangle*/}
    }
```

Les interfaces

- **public class C implements I { … }** 2 seuls cas possibles :
- la classe **C** doit implémenter toutes les méthodes de I sinon, la classe **C** doit être déclarée **abstract**
- Une classe peut implémenter une ou plusieurs interfaces (et hériter d'une classe) :
- **public class Cercle implements Figure, Coloriable {}**
- Une interface ne peut contenir que
  - des méthodes **abstract** et **public**
  - des définitions de constantes publiques (« **public static final** »)
- Une interface ne peut contenir de méthodes **static**, **final**, **synchronized** ou **native**
- Une interface peut remplacer une classe pour déclarer une variable

Les classes Internes

- Depuis la version 1.1, Java permet de définir des classes à l'intérieur d'une classe
- Il y a 2 types de classes internes :
  - classes définies à l'extérieur de toute méthode (au même niveau que les méthodes et les variables d'instance ou de classe)
  - classes définies à l'intérieur d'une méthode
- 2 types de classes internes définies à l'extérieur d'une méthode
  - Classes non **static** : une instance d'une telle classe est liée à une instance de la classe englobante
  - Classes **static** : les instances ne sont pas liées à une instance de la classe englobante
- Une classe interne ne peut avoir le même nom qu'une classe englobante (quel que soit le niveau d'imbrication)

Les classes internes non incluses dans une méthode

- Une telle classe peut avoir les mêmes degrés d'accessibilité que les membres d'une classe :
  - **private**, *package*, **protected**, **public**
- Elle peut aussi être **abstract** ou **final**
- La classe englobante (**ClasseE**) fourni un espace de nommage pour une classe interne (**ClasseI**). Son nom est de la forme
  - **ClasseE.ClasseI**
- On peut importer une classe interne : **import ClasseE.ClasseI;**
- On peut aussi importer toutes les classes **import ClasseE.*;**

Classe interne static

- Une classe interne **static** joue à peu près le même rôle que les classes non internes
- La différence est qu'elle a accès à toutes les variables **static** de la classe englobante, même les variables **static private**
- Elle n'a pas accès aux variables d'instance
- En définissant une telle classe, le programmeur indique que la classe interne n'a de sens qu'en relation avec la classe externe
- A l'extérieur de la classe englobante, on peut créer une instance de la classe interne par :

ClasseE.ClasseI x = new ClasseE.ClasseI(…);

Exemple de Classe interne static

- On veut récupérer les valeurs minimale et maximale d'un tableau, variable d'instance d'une classe **ClassExterne**
- Pour cela on écrit une méthode qui renvoie une paire de nombres (pour éviter un double parcours du tableau)
- On crée une classe interne **static ClassInterne** pour contenir cette paire de nombres

```
public class ClassExterne {

// tableau dont on veut le Min et
Max

   private int valeurs[];

public ClassExterne(int t[]){

valeurs=new int[t.length];

System.arraycopy(t,0,valeurs,0,t.l
ength);   }
```

```
public static class  ClassInterne
{

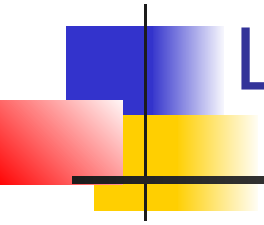
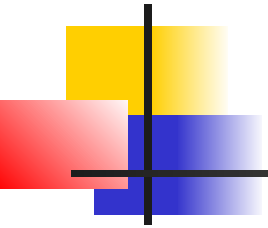
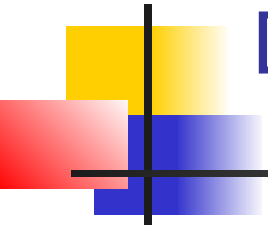
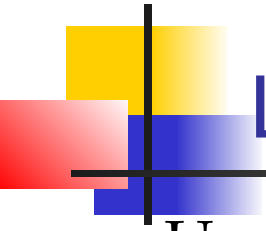
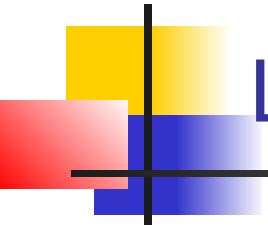
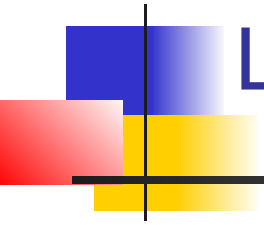
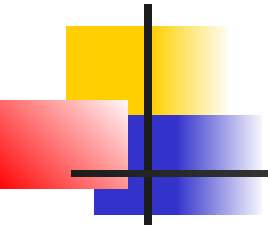
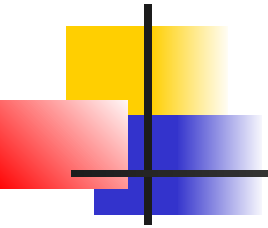
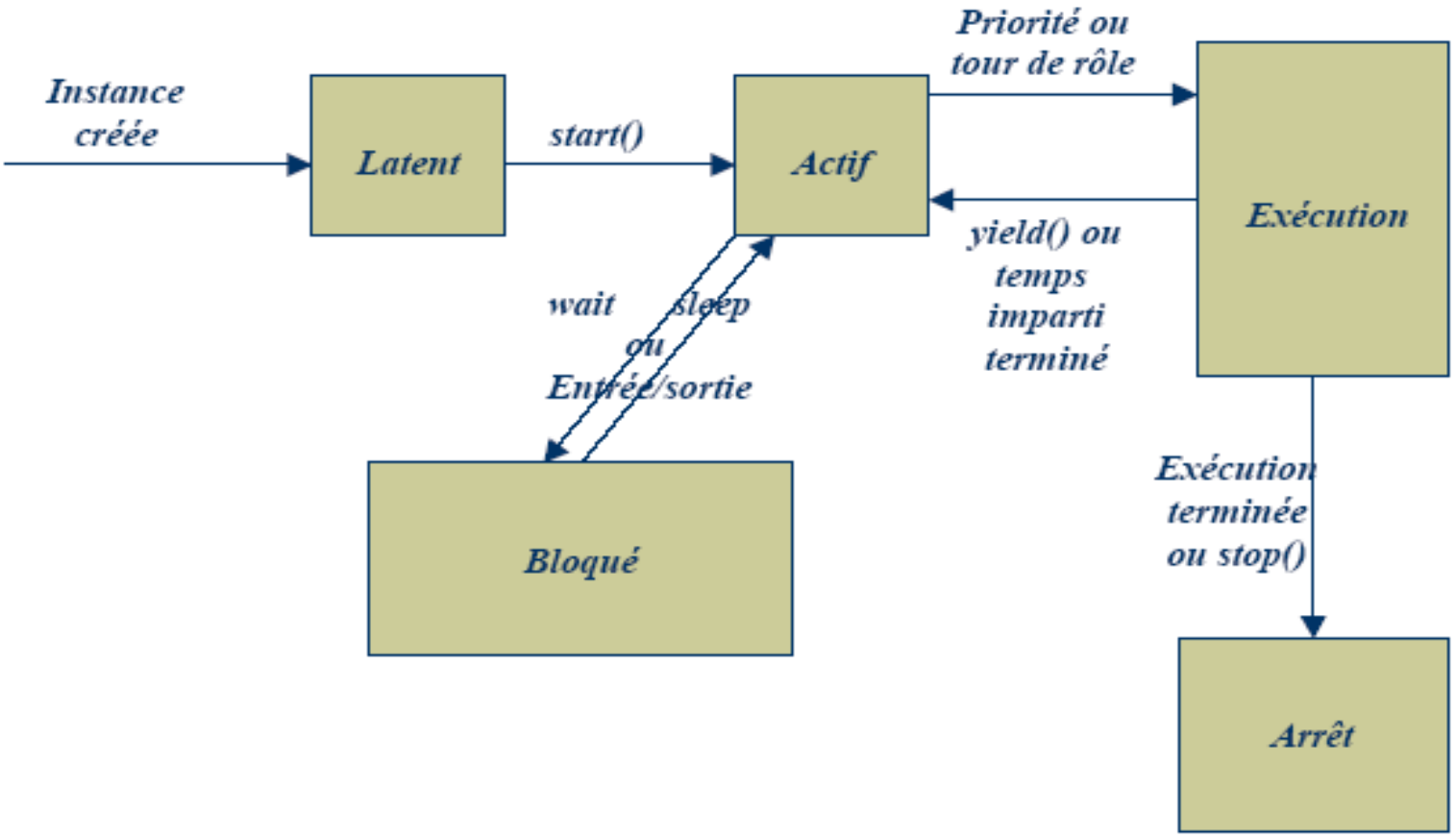
 private int x, y;

private ClassInterne(int x,int y)
{ this.x = x;

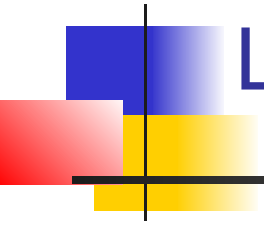
 this.y = y;}

public int getX() { return x; }

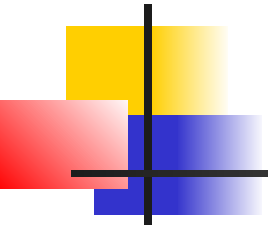
public int getY() { return y; }}
```

Exemple de Classe interne static

```
//Methode retourne un objet de type ClassInterne Contenant MiN //
   MAX
   public ClassInterne getMinMax () {
   if (valeurs.length == 0) return null;
    int min = Integer.MAX_VALUE;
    int max = Integer.MIN_VALUE;
   for (int i = 0; i < valeurs.length; i++) {
    if (valeurs[i] < min) min = valeurs[i];
    if (valeurs[i] > max) max = valeurs[i];}
   return new ClassInterne(min, max);}
public static void main(String[] args)
   {   int[] t={5,4,6,8,2,1};
   ClassExterne ce=new ClassExterne(t);
   ClassExterne.ClassInterne ci= ce.getMinMax();
   System.out.println("Min = " + ci.getX());   //affiche 1
   System.out.println("Max = " + ci.getY());  //affiche 8
   }}  // Ferme la classe ClassExterne
```

Classe interne non static

- Une instance d'une classe interne non **static** ne peut exister que « à l'intérieur » d'une instance de la classe englobante
- Les classes internes non **static** ne peuvent avoir de variables **static**
- Une classe interne non **static** partage tous les membres (même privés) avec la classe dans laquelle elle est définie :
  - la classe interne a accès à tous les membres de la classe englobante
  - la classe englobante a accès à tous les membres de la classe interne
- On crée une instance de **ClasseI** interne à **instanceClasseE** dans le code d'une autre classe par **instanceClasseE. new ClasseI(...)**

Clonage des Objets

```
class point implements Cloneable{
   private int x,y;
   public point(int x1, int y1)
   {x=x1;y=y1;}
//Redéfinition Obligatoire de
   clone
public Object clone() {
Object o=null;
try{o=super.clone();}
catch(CloneNotSupportedException
   e){}
return o;}}
```

```
class Clone{

public static void main(String[]args)
   {
 point p1,p2,p3;
 p1=new point(1,2);
 p2=(point)p1.clone();
 p3=p1;
 if(p1 == p2)
 System.out.println("p1 == p2");
  else
 System.out.println("p1 != p2");
 if(p1 == p3)
 System.out.println("p1 == p3");
  else
System.out.println("p1 != p3");
          }}
```



Les Collections



Plan

- Généralités sur les collections
- Collections
  - classe vector
  - classe la plus utilisée : **ArrayList**
  - interface **Iterator** (énumérer les éléments d'une collection)
- Maps
  - classe la plus utilisée : **HashMap**
- Utilitaires : trier une collection et rechercher une information dans une liste triée



Définitions

- Une collection est un objet qui contient d'autres objets
- Par exemple, un tableau est une collection
- Le JDK fournit d'autres types de collections sous la forme de classes et d'interfaces
- Ces classes et interfaces sont dans le paquetage **java.util**
- Des interfaces dans 2 hiérarchies principales :
  - **Collection**
  - **Map**
- **Collection** correspond aux interfaces des collections proprement dites
- **Map** correspond aux collections indexées par des clés ; un élément d'une *map* est retrouvé rapidement si on connaît sa clé

Les méthodes d'une collection

boolean add(Object)	Ajoute un objet dans une collection
void clear()	Supprime tous les éléments du conteneur
boolean contains(Object)	Vérifie si l'objet argument est contenu dans la collection
boolean isEmpty()	Vrai si si le conteneur ne contient pas d'éléments.
boolean remove(Object)	Supprime un objet de la collection
int size()	Renvoie le nombre d'éléments dans le conteneur.

Exemples de Collection / Map

```
// création d'une Liste l
List l = new ArrayList();
// ajouter les objets ds l
l.add("Ahmed");
l.add("Mohamed");
l.add("Mustapha");
l.add("Taib");
// trier les objets de l
Collections.sort(l);
// afficher le contenu de l
System.out.println(l);
```

```
Map frequences = new HashMap();
for (int i = 0; i < args.length;
  i++) {
Integer freq =
(Integer)frequences.get(args[i]);
if (freq == null)
freq = new Integer(1);
else
freq = new Integer(freq.intValue()
  + 1);
frequences.put(args[i], freq);
}
System.out.println(frequences);
```



La classe ArrayList

- Une instance de la classe **ArrayList** est une sorte de tableau qui peut contenir un nombre quelconque d'instances de la classe **Object**
- Les emplacements sont repérés par des nombres entiers (à partir de 0)
- Les principales méthodes en plus de celle de Collection
  - **Object get(int indice)**
  - **int indexOf(Object obj)**
  - **void set(int indice, Object obj)**
- Le type retour de la méthode **get()** est **Object** ; on doit donc souvent *caster* les éléments que l'on récupère dans un A**rrayList** si on veut leur envoyer des messages

L'interface Iterator

- Un itérateur (instance d'une classe qui implante l'interface **Iterator**) permet d'énumérer les éléments contenus dans une collection
- L'interface **Collection** fournit la méthode **Iterator iterator()** qui renvoie un itérateur pour parcourir les éléments de la collection
- Les méthodes de Iterator
  - **boolean hasNext()**
  - **Object next()**

```
List le = new ArrayList();
Employe e = new Employe("Ahmed");
le.add(e);
. . . // ajoute d'autres employés dans le
Iterator it = le.iterator();
// le 1er next() fournira le 1er élément
while (it.hasNext()) {
System.out.println( ((Employe)it.next()).getNom());
```



L'interface Map

- L'interface **Map** correspond à un groupe de couples clés-valeurs
- Une clé repère une et une seule valeur
- 2 clés égales au sens de **equals()** repèrent la même valeur
- Deux implémentations possibles
  - **HashMap**, table de hachage ; garantit un accès en temps constant
  - **TreeMap**, arbre ordonné suivant les valeurs des clés avec accès en log(n) ;
  - La comparaison utilise l'ordre naturel (interface **Comparable**) ou une instance de **Comparator**
- On peut (entre autres)
  - ajouter et enlever des couples clé – valeur
  - récupérer une référence à un des éléments en donnant sa clé
  - savoir si une table contient une valeur
  - savoir si une table contient une clé

La classe HashMap (Table de hachage)

- Une table de hachage range des éléments en les regroupant dans des sous-ensembles pour les retrouver plus rapidement
- Le regroupement dans les sous-ensembles dépend de la valeur d'un calcul effectué sur les éléments
- Pour retrouver un élément, on effectue d'abord ce calcul qui va déterminer dans quel sous-ensemble il est rangé
- Pour que la recherche soit efficace,
  - le calcul doit être rapide
  - répartir les éléments uniformément sur les sous ensembles
- Le type retour de la méthode **get()** est **Object.** Il sera donc souvent nécessaire de *caster* les éléments récupérés dans une **HashMap** lorsque l'on voudra leur envoyer un message

La classe HashMap (Table de hachage)

```
Map hm = new HashMap();
Employe e = new Employe("Dupond");
e.setMatricule("E125");
hm.put(e.matricule, e);
. .
/*crée et ajoute les autres employés dans la table de
  hachage*/
Employe e2 = (Employe)hm.get("E369");
Collection elements = hm.values();
Iterator it = elements.iterator();
//Afficher tous les employes
while (it.hasNext()) {
System.out.println(
((Employe)it.next()).getNom());}
```

Tri et Recherche dans des collections

- **Classe Collections**
- Cette classe ne contient que des méthodes **static**, utilitaires pour travailler avec des collections :
  - tris (sur listes)
  - recherches (sur listes)
  - copies
  - minimum et maximum
- Si **l** est une liste, on peut trier **l** par : **Collections.sort(l);**
- la méthode **sort()** ne fonctionnera que si tous les éléments de la liste sont d'une classe qui implante l'interface **java.lang.Comparable**
- Toutes les classes du JDK qui enveloppent les types primitifs (**Integer** par exemple) implantent l'interface **Comparable**
- Il en est de même pour les classes du JDK **String**, **Date**, **BigInteger**, **BigDecimal**

Tri et Recherche dans des collections

- **Interface Comparator**
- si les éléments de la collection n'implantent pas l'interface **Comparable**,
- ou si on ne veut pas les trier suivant l'ordre donné par **Comparable** ?
- on construit une classe qui implante l'interface **Comparator**, qui permettra de comparer 2 éléments de la collection
- on passe en paramètre une instance de cette classe à la méthode **sort()**
- Elle comporte une seule méthode :
- **int compare(Object o1, Object o2)** qui doit renvoyer
  - un entier positif si **o1** est « plus grand » que **o2**
  - 0 si **o1** a la même valeur (au sens de **equals**) que **o2**
  - un entier négatif si **o1** est « plus petit » que **o2**

Tri et Recherche dans des collections

```
//Critère de comparaison
public class CompareSalaire implements Comparator {
public int compare(Object o1, Object o2) {
double s1 = ((Employe) o1).getSalaire();
double s2 = ((Employe) o2).getSalaire();
return (int)(s1 - s2);
}}
..
List employes = new ArrayList();
// On ajoute les employés . . .
Collections.sort(employes,new CompareSalaire());
System.out.println(employes);
```

Tri et Recherche dans des collections

- Pour rechercher la position d'un objet dans une liste :
- **int binarySearch(List l, Object obj)**
- **int binarySearch(List l, Object obj, Comparator c)**
- La liste doit être triée en ordre croissant :
  - suivant l'ordre naturel (interface **Comparable**)
  - ou suivant le comparateur, pour la 2ème méthode
- La classe **Arrays** contient de nombreuses méthodes **static** utilitaires pour travailler avec des tableaux d'objets ou de types primitifs, en particulier pour :
  - trier (**sort**)
  - chercher dans un tableau trié (**binarySearch**)



Les Threads



Plan

- Les Threads
- Utilisation des Threads
- Les méthodes d'un Thread
- Les états d'un Thread
- Ordonnancement des Threads
- Gestion de la concurrence
- Notion de rendez-vous



Les Threads

- Un thread (appelée aussi processus léger ou activité) est une suite d'instructions à l'intérieur d'un process. Les programmes qui utilisent plusieurs threads sont dits multithreadés.
- Les threads peuvent être créés
  - comme instance d'une classe dérivée de la classe Thread.
  - Comme instance de la classe Thread sur un objet qui implémente Runnable
- Elles sont lancées par la méthode **start(),** qui demande à l'ordonanceur de thread de lancer la méthode **run()** du thread
- Cette méthode **run()** doit être implantée dans le programme.

Utiliser les thread en java

- Créer un contrôleur de thread avec une classe fille de la classe **Thread**
  - `class ThreadTache extends Thread {`
  - `public void run() {`
  - `// Code qui sera exécuté par le thread . . .} }`
  - `ThreadTache threadTache = new ThreadTache(…);`
- Créer un contrôleur de thread avec l'interface **Runnable**
  - **class Tache implements Runnable {**
  - **public void run() {**
  - **// Code qui sera exécuté par le thread. . .}}**
  - **Tache tache = new Tache(…);**
  - **Thread t = new Thread(tache) ;**

Les méthodes d'un thread

- **void start()** Lance l'exécution d'un thread t.start()
- **static void sleep(long mis) throws InterruptedException** Fait dormir le thread pendant un certains temps en ms
- **void join() throws InterruptedException** Attend la fin de l'exécution du thread t
- **void interrupt()** Permet d'interompre un thread en attente (sleep, wait, join)
- **String getName()** Retourne le nom associé au thread
- **boolean isAlive()** Retourne true si le thread est vivant false si non
- **exit() ou stop()//**fin d'exécution
- **static void yield()** Permet de passer la main à un thread de priorité supérieure ou égale

Les états d'un thread



Exemple de threads en Java

```
class SimpleThread extends
   Thread {
public SimpleThread(String str)
   {
super(str);
}
public void run()
{
for (int i = 0; i < 10; i++)
   {
System.out.println(i + " "
   +getName());
   }
System.out.println("DONE! " +
   getName());
}
}
```

```
public class TestThread {
public static void main
   (String[] args) {
SimpleThread rabat = new
   SimpleThread("Rabat");
rabat.start();
SimpleThread casablanca = new
   SimpleThread("casablanca");
casablanca.start();}
}
```

Ordonnancement des threads

- A toute thread est associée une priorité appartenant à l'intervalle Thread.MIN_PRIORITY Thread.MAX_PRIORITY
- Par défaut un thread est crée avec une priorité égale à 5
- Une thread adopte la priorité (méthode getPriority()) de son processus créateur. setPriority(int p) permet de changer celle-ci.
- L'ordonnancement des threads dépendant des plate-formes d'exécution
- Sous Windows 95(JDK1.1.4), et MacOS (JDK 1.0.2), un tourniquet pour les threads de même priorité est installé.

Concurrence, synchronisation en Java

- Un des problèmes centraux du traitement multithread est la gestion d'accès simultané à une ressource partagée
- Pour éviter de telle situation Java offre le mot clé synchronized
  - synchronized(obj){ //code atomique}
  - synchronized void p(){ } qui est équivalente à void p() { synchronized (this) { }}
  - Lorsqu'un thread exécute cette méthode sur un objet, un autre thread ne peut pas l'exécuter pour le même objet.
  - En revanche, il peut exécuter cette méthode pour un autre objet.

Exemple de code avec verrou

```
// section critique en Java
class AnObject {

public synchronized void
sectionCritique(String str) {
// code en section critique
System.out.println(str+" debut
   section critique");
for (int i=0; i<10000; i++);
System.out.println(str+" fin
   section critique");
    }
}
```

```
class SimpleThread implements
   Runnable {
     String name;
     AnObject o;
SimpleThread (String str,
   AnObject object) {
      name = str;
      o = object;}
public void run() {
   int j;
for (int i = 0; i < 10; i++) {
       try {
Thread.sleep((long)(Math.random()
   * 1000));
o.sectionCritique(name);}
catch (InterruptedException e) {}
   }}
```

Rendez-vous en Java

- Un thread est parfois obligé de faire une attente sur la libération d'une ressource.
- Java offre les méthodes wait() et notify(). Ce sont des méthodes de la classe Object et pas de la classe Thread.
- Pour endormir un thread sur le moniteur, il vous faut utiliser la méthode wait.
- Pour réveiller des threads endormis, vous pouvez utiliser les méthodes notify et notifyAll
- Ne peuvent être appelées que dans un code synchronisé
- notify() avertit un des threads qui attend sur un wait() sur l'objet sur lequel a été lancé notify() et débloque le wait() sur cet objet.
  - ```
    Class ObjetPartage{
      boolean write;
      syncronized void read(){ if (!write) { ..wait();..}
       notify();}}
    ```
    ```



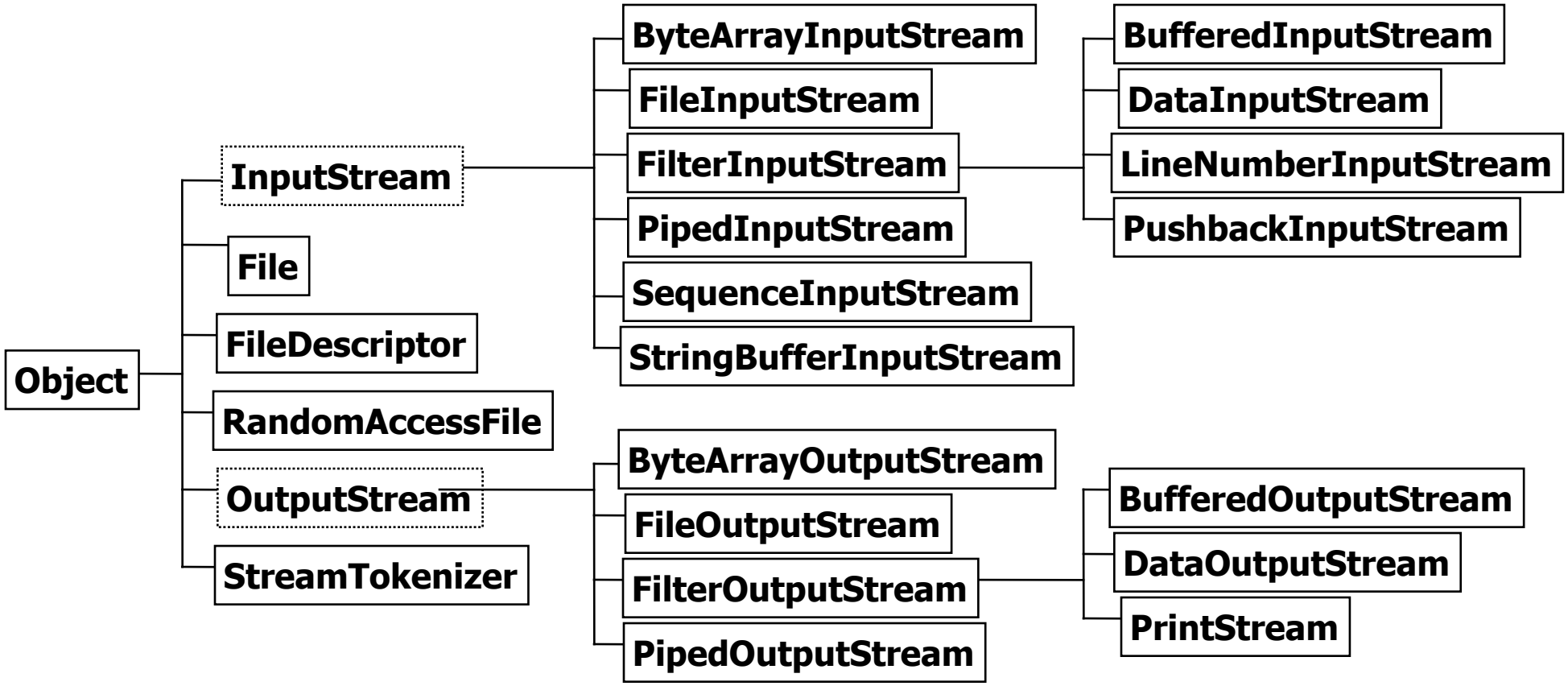
Les entrées sorties en Java

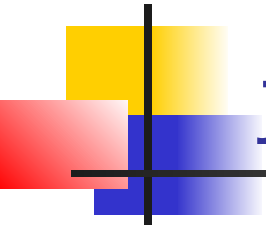


Plan

- Présentation de java.io
- La classe File
- Les filtres
- Entrées / Sorties Formatées DataXXXputStream
- Lecture / Ecriture à partire d'un fichier
- Sérialisation des objets
  - Objetc(Input|Output)Stream

java.io.*


Object
InputStream
File
FileDescriptor
RandomAccessFile
OutputStream
StreamTokenizer
ByteArrayInputStream
FileInputStream
FilterInputStream
PipedInputStream
SequenceInputStream
StringBufferInputStream
BufferedInputStream
DataInputStream
LineNumberInputStream
PushbackInputStream
ByteArrayOutputStream
FileOutputStream
FilterOutputStream
PipedOutputStream
BufferedOutputStream
DataOutputStream
PrintStream



java.io.File

- Cette classe fournie une définition *plateform-independent* des fichiers et des répertoires.
- Les principaux constructeurs et méthodes de cette classe sont :
  - File(File rep, String nom)
  - File(String chemin)
  - File(String rep, String nom)
- **Les principales méthodes**

```
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.length()); // --> 11345
```

- **Les répertoires**

```
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]);
```

Les filtres sur java.io.File

- public String[] list(FilenameFilter filtre) de la classe File qui contient comme argument un filtre.
- C'est un objet d'une classe qui implémente l'interface FilenameFilter.
- Cette classe doit donner une définition de la méthode public boolean accept(File rep, Stringnom) et seuls les noms de fichiers qui renvoie true par cette méthode seront affichés.

```
import java.io.*;
class Filtre implements FilenameFilter {
public boolean accept(File rep, String nom) {
if (nom.endsWith(".java")) return true;
return false;}
//dans une classe de test
String nomFics[ ] = (new File(rep)).list(new
Filtre());}
```

Data (Input|Output)putStream

- Ce sont ces classes qui possèdent des écritures ou lectures de plus haut niveau que de simples octets (des entiers, des flottants, ...)
- dans DataInputStream ,

```
public final boolean readBoolean() throws
  IOException
public final char readChar() throws IOException
public final int readInt() throws IOException
...
```

- dans DataOutputStream,

public final void writeBoolean(boolean) throws IOException,

public final void writeChar(int) throws IOException,

public final void writeInt(int) throws IOException

java.io.File(Input|Output)Stream

- Ces classes permettent d'accèder en lecture et en écriture à un fichier.
  FileInputStream fis = new **FileInputStream**("source.txt");
  fis.read(data);// Lecture binaire
  FileOutputStream fos = new **FileOutputStream**("cible.txt");
  fos.write(data);// Ecriture binaire
- Pour lire et d'écrire des types primitifs et des lignes sur des fichiers on utilise Data(Input|Output)Stream
  FileInputStream fis = new FileInputStream("source.txt");
  DataInputStream dis = new **DataInputStream**(fis);
  int i = **dis.readInt**();
  double d = **dis.readDouble**();
  String s = **dis.readLine**();
  FileOutputStream fos = new FileOutputStream("cible.txt");
  DataOutputStream dos = new **DataOutputStream**(fos);
  dos.writeInt(123);
  dos.writeDouble(123.456);
  dos.writeChars("Une chaine");

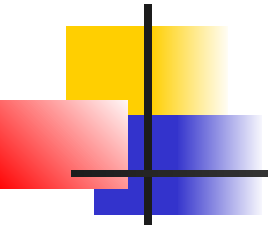
java.io.PrintStream

- Cette classe permet de manipuler un OutputStream au travers des méthode print() et println().

```
PrintStream ps = new PrintStream(new FileOutputStream ("cible.txt"));
ps.println("Une ligne");
ps.println(123);
ps.print("Une autre ");
ps.print("ligne");
ps.flush();// vider le buffer
ps.close();
```

- Pour Lire du texte à partir du clavier (JDK1.1)

```
import java.io.*;
try {
InputStreamReader isr = new InputStreamReader (System.in);
BufferedReader br = new BufferedReader (isr);
String ligne = br.readLine();} catch (IOException e) {}
```



Sérialisation des objets

- En Java, seules les données (et les noms de leur classe ou type) sont sauvegardées (pas les méthodes ou constructeurs).
- Si un objet contient un champ qui est un objet, cet objet inclus est aussi sérialisé et on obtient ainsi un arbre (un graphe) de sérialisation d'objets.
- Des objets de certaines classes ne peuvent pas être sérialisés : c'est le cas des Threads, des FileXXXputStream, …
- Pour indiquer que les objets d'une classe peuvent être persistants on indique que cette classe implémente l'interface Serializable.
- Certaines classes sont par défaut sérializable (String et Date)

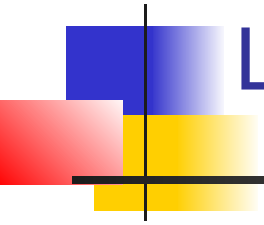
java.io.Object(Input|Output)Stream (1)

- Ces classes permettent de lire et d'écrire des objets, implémentant java.io.serializable, sur des flux.

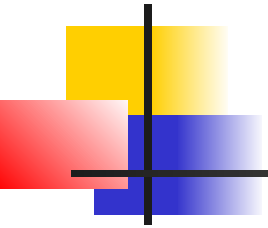
```
// Ecriture
FileOutputStream fos = new FileOutputStream("tmp");
ObjectOutputStream oos = new ObjectOutputStream(fos);
oos.writeObject("Today");
oos.writeObject(new Date());
oos.flush();

// Lecture
FileInputStream fis = new FileInputStream("tmp");
ObjectInputStream ois = new ObjectInputStream(fis);
String today = (String)ois.readObject();
Date date = (Date)ois.readObject();
```

java.io.Object(Input|Output)Stream (2)

- Par défaut, tous les champs d'un objets sont sérialisés (y compris private)
- Cela peut poser des problèmes de sécurité
- 3 solutions :
  - Ne pas implémenter Serializable
  - Réécrire les méthodes writeObjet() et readObject()
  - Le mot clé transient permet d'indiquer qu'un champs ne doit pas être sérialisé.
  - private transient passwd;



Les swing

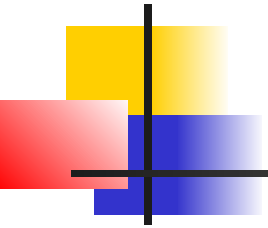


Plan

- Les APIs graphiques awt / swings
- Premier programme Graphique
- Composants lourds / légers
- Les conteneurs
- Composantes graphiques de base
- Gestion de Mise en forme
  - FlowLayout, BorderLayout, GridLayout
- Gestion des événements
  - Écouteur / Adaptateur
- Composantes graphiques supplémentaires

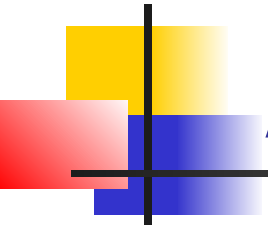
Interface graphique et programmation par événement

- Une interface graphique est formée d'une ou plusieurs fenêtres qui contiennent divers composants graphiques (*widgets*) tels que
  - boutons
  - listes déroulantes
  - menus
  - champ texte, etc...
- Les interfaces graphiques sont souvent appelés GUI d'après l'anglais ***G****raphical* ***U****ser* ***I****nterface*
- L'utilisation d'interfaces graphiques impose une programmation « conduite par les événements »



Solution Java: Les écouteurs

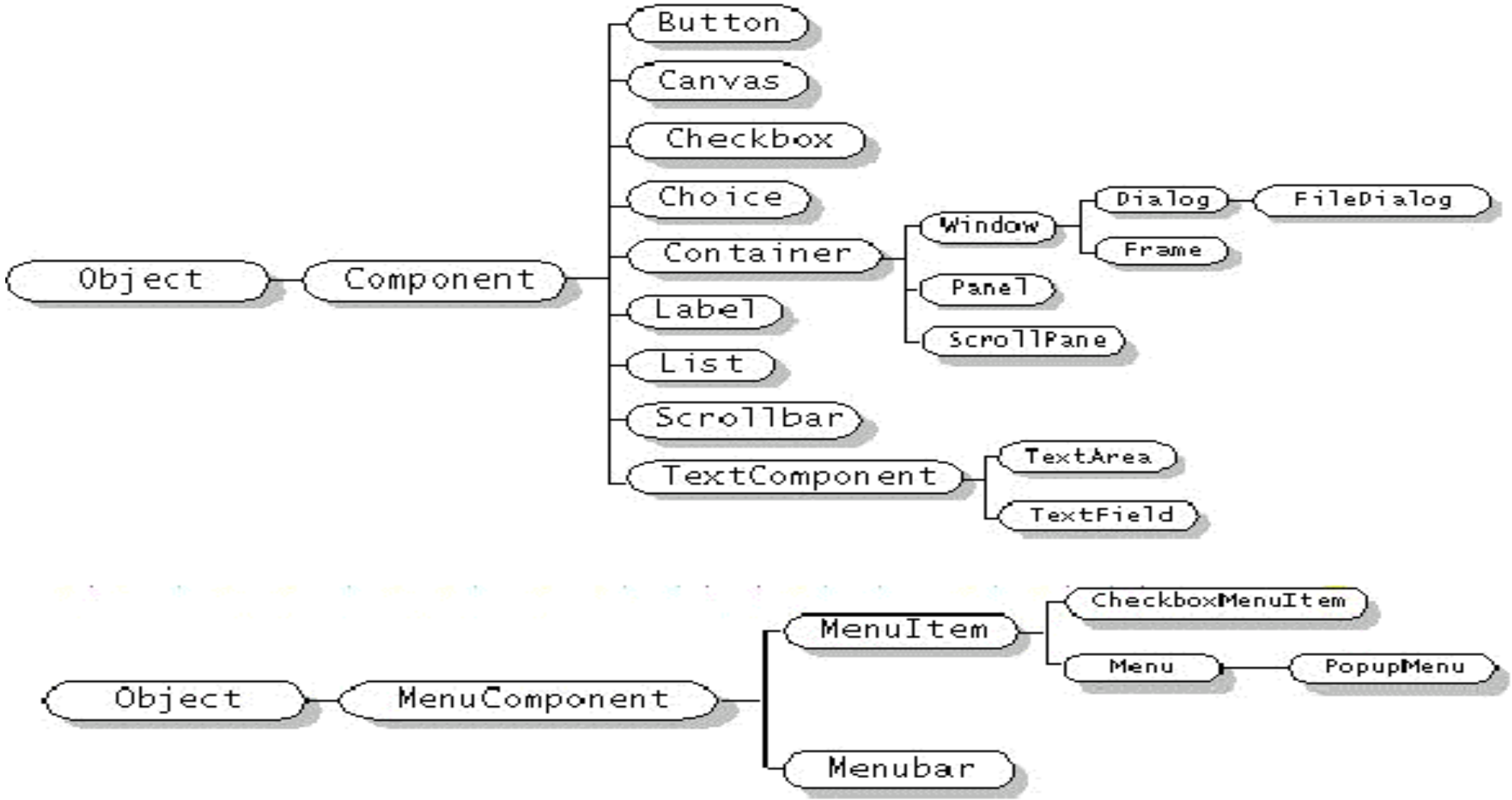
- Le JDK utilise une architecture de type « observateur - observé »
- Les composants graphiques (comme les boutons) sont les observés
- Chacun des composants graphiques a ses observateurs (ou écouteurs, listeners), objets qui s'enregistrent (ou se dés-enregistrent) auprès de lui comme écouteur d'un certain type d'événement
- Ces écouteurs sont prévenus par le composant graphique dès qu'un événement qui les concerne survient sur ce composant
- Le code de ces écouteurs exécute les actions à effectuer en réaction à l'événement



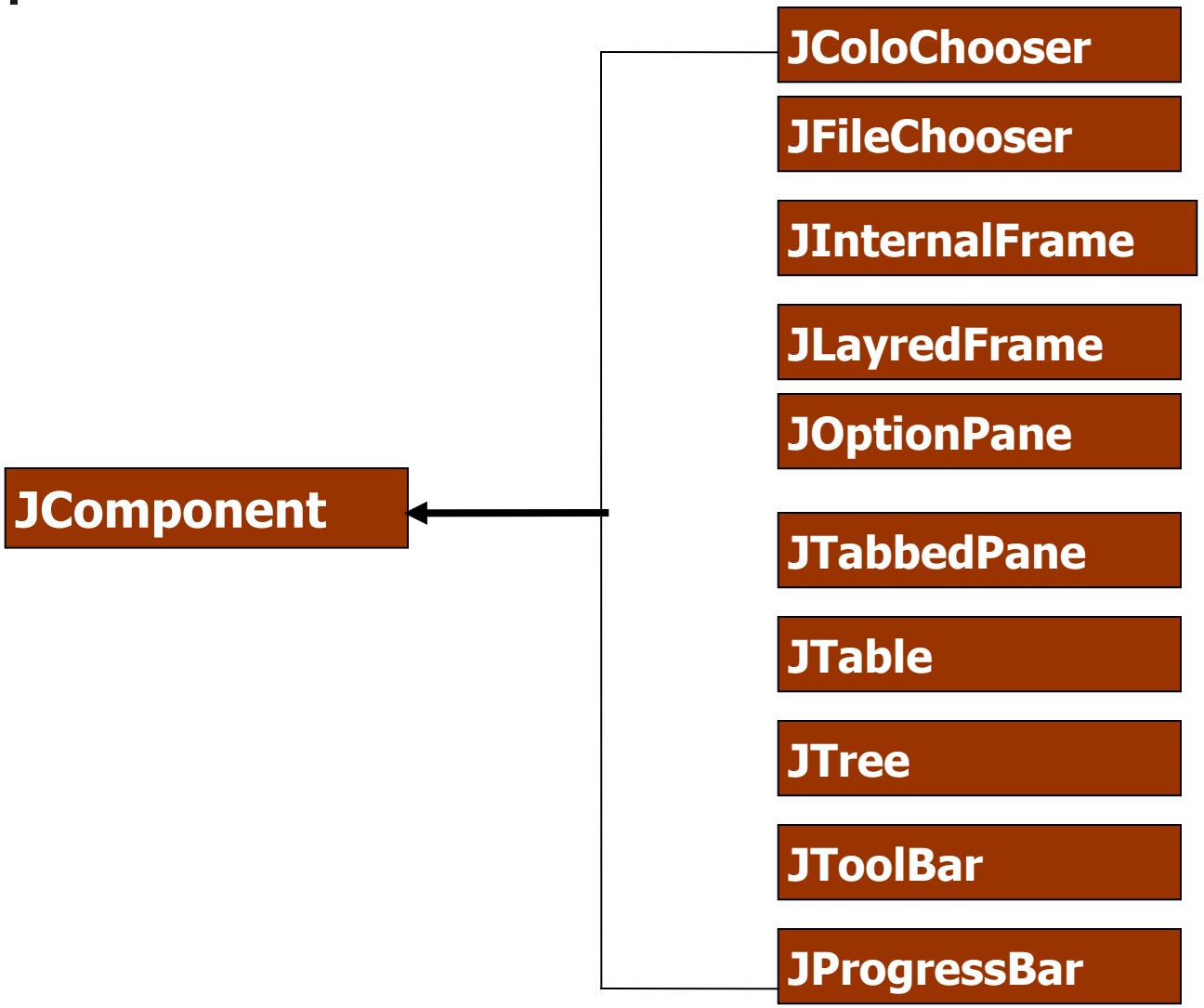
APIs pour GUI

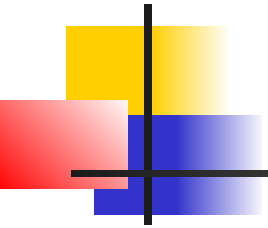
- 2 bibliothèques :
  - AWT (*Abstract Window Toolkit*, JDK 1.1)
  - Swing (JDK 1.2)
- Swing et AWT font partie de JFC (*Java Foundation Classes*) qui offre des facilités pour construire des interfaces graphiques
- Swing est construit au-dessus de AWT
  - même gestion des événements
  - les classes de *Swing* héritent des classes de AWT
- Tous les composants de AWT ont leur équivalent dans Swing : en plus joli et avec plus de fonctionnalités
- Swing offre de nombreux composants qui n'existent pas dans AWT

Les principaux composants awt/swing



Les extensions swing





Package javax.swing

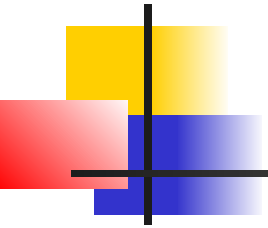
- **javax.swing**
  - utilisation des composantes, des adaptateurs, des interfaces et des models par défauts.
- **javax.swing.colorchooser**
  - Utilisation de la palette de couleurs
- **javax.swing.filechooser**
  - Utilisation de la boite de dialogue gestion de fichier
- **javax.swing.event** : Traitement des événements
- **javax.swing.undo** : intégration de annuler/répéter
- **javax.swing.text** : traitement du texte HTML, rtf, parser
- **javax.swing.table** resp. **javax.swing.tree** : intégration d'une table resp. arbre dans une GUI

awt vs swing(1)

Conteneur AWT	Conteneur Swing	Remarques sur les conteneurs Swing
Canevas	JPanel ou JLabel	Dépend des besoins de l'application
Dialog	JDialog ou JOptionPane	Les JDialog ne peuvent pas contenir directement les composants, il faut passer par un JPanel
FileDilog	**JFileChooser**	JFileChooser n'est pas une boite de dialogue, mais un composant qu'on peut aisément ajouter à un conteneur
Frame	**JFrame**	On ne rajoute pas directement les composants au JFrame, on passe par un JPanel
Panel	**JPanel**	On peut rajouter des bord sans réécrire paint

awt vs swing(2)

Conteneur AWT	Conteneur Swing	**Remarques sur les conteneurs Swing**
Label	**JLabel**	JLabel peut inclure du le texte et les icônes
Button	**JButton**	Les JButton peuvent contenir des icônes
PopupMenu	**JPopupMenu**	Les composants des menus descendent de la classe JComponent,
ScrollPane	**JScrollPane**	On peut ajouter des décorations, telles que des titres de lignes ou de colonnes
Window	JWindow ou JToolTip	On peut utiliser un JLayeredPane comme conteneur principal pour avoir plusieurs couches superposées sans rajouter de fenêtre



awt vs swing(3)

Conteneur AWT	Conteneur Swing	Remarques sur les conteneurs Swing
Scrollbar	JScrollPane ou Jslider ou JProgressBar	Notez : B est majuscule dans JProgrssBar
TextArea	**JTextArea**	Attention aux retours de fin de ligne : utiliser setLineWrap(true) -par défaut, c'est false!
TextField	JTextField ou JPasswordField	N'utilise plus les TextEvent

Premier programme Swing

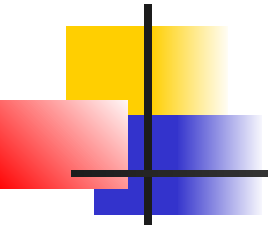
```
import javax.swing.*;
public class HelloWorldSwing {
//Création et initiation de la fenêtre.
    JFrame frm = new JFrame("HelloWorldSwing");
//création d'une étiquette "Hello World".
    JLabel lbl = new JLabel("Hello World");
public  void GUI() {
//être sure d'avoir une bonne décoration v1.4. JFrame.setDefaultLookAndFeelDecorated(true);
//Fermer l'application dès qu'on clique sur le bouton fermer
    frm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
//ajouter
frm.add(lbl);
//Afficher la fenêtre.
    frm.pack();
frm.setVisible(true); }
}
 class Test{
public static void main(String[] args) {
      ( new HelloWorldSwing()). GUI();   }
}
```

Taille d'une fenêtre

- **pack()** donne à la fenêtre la taille nécessaire pour respecter les tailles préférées des composants de la fenêtre
- Taille ou un emplacement précis sur l'écran (en pixels) :
  - **setLocation(int xhg, int yhg)** (ou **Point** en paramètre)
  - **setSize(int largeur, int hauteur)** (ou **Dimension** en paramètre)
  - **setBounds(int x, int y, int largeur, int hauteur) (**ou **Rectangle** en paramètre)
- Les sous-classes de la classe abstraite **java.awt.Toolkit** implantent la partie de AWT qui est en contact avec le système d'exploitation hôte
- Quelques méthodes publiques :
- **getScreenSize**, **getScreenResolution**, **getDefaultToolkit**, **getImage**

Taille d'une fenêtre

- **//Centrage de la fenêtre**
  - Toolkit tk = Toolkit.getDefaultToolkit();
  - Dimension d = tk.getScreenSize();
  - int hauteurEcran = d.height;
  - int largeurEcran = d.width;
- **//Ajuster la taille et la position de la fenêtre.**
  - frame.setSize(largeurEcran/2, hauteurEcran/2);
  - frame.setLocation(largeurEcran/4,hauteurEcran/4);
- **//une autre manière pour fermer la fenêtre à condition d'implémenter WindowListener**
  - windowClosing(WindowEvent e){System.exit(0) ;}
    windowOpened(WindowEvent e){}
    windowClosed(WindowEvent e){}
    windowDeiconified(WindowEvent e){}
    windowDeactivated(WindowEvent e){}
    windowActivated(WindowEvent e){}
    windowIconified(WindowEvent e){}

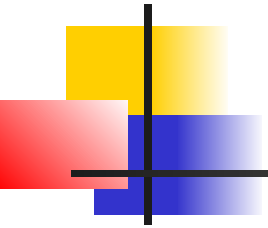


Composants lourds/ légers (1)

- Pour afficher des fenêtres (instances de **JFrame**), Java s'appuie sur les fenêtres fournies par le système d'exploitation hôte dans lequel tourne la JVM
- On dit que les **JFrame** sont des composants lourds
- Au contraire de AWT qui utilise les widgets du système d'exploitation pour tous ses composants graphiques (fenêtres, boutons, listes, menus, etc.), Swing ne les utilise que pour les fenêtres de base « top-level »
- Les autres composants, dits légers, sont dessinés dans ces containers lourds, par du code « pur Java »

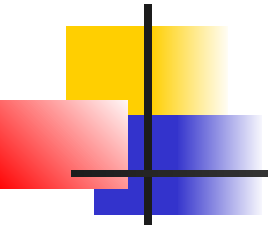
Composants lourds/ légers (2)

- Il y a 3 sortes de containers lourds (un autre, **JWindow**, est rarement utilisé) :
  - **JFrame** fenêtre pour les applications
  - **JApplet** pour les applets
  - **JDialog** pour les fenêtres de dialogue
- Pour construire une interface graphique avec Swing, il faut créer un (ou plusieurs) container lourd et placer à l'intérieur les composants légers qui forment l'interface graphique
- La plupart des widgets de Swing sont des instances de sous-classes de la classe **Jcomponent**
- Les instances des sous-classes de **JComponent** sont de composants « légers »



Les conteneurs (1)

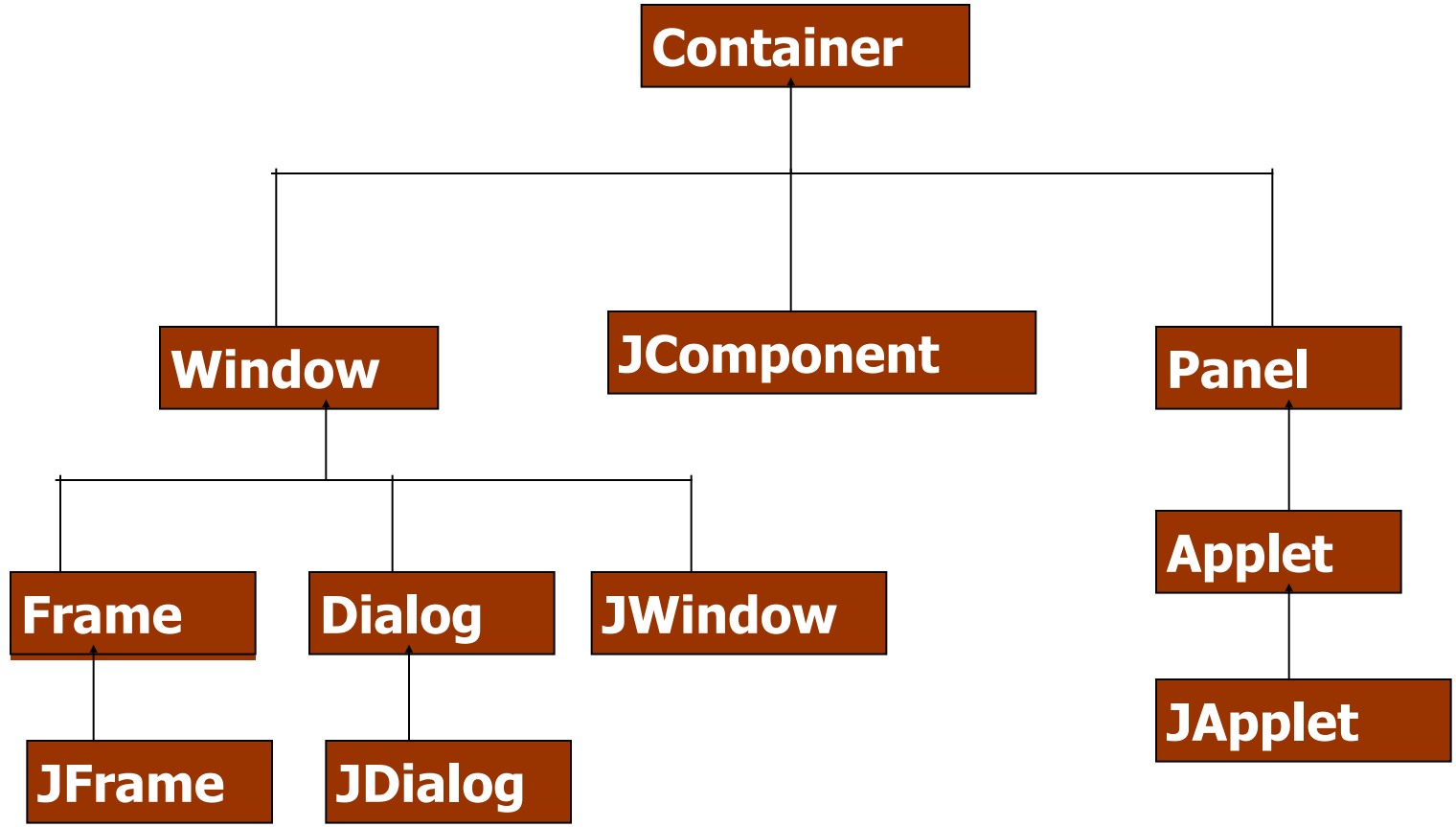
- Tous les composants légers des sous-classes de **JComponent** héritent de la classe **Container** et peuvent donc contenir d'autres composants
- Des composants sont destinés spécifiquement à recevoir d'autres éléments graphiques :
- les containers « top-level » lourds **JFrame**, **JApplet**, **JDialog**, **JWindow**
- Les containers « intermédiaires » légers **JPanel**, **JScrollPane**, **JSplitPane**, **JTabbedPane**, **Box** (ce dernier est léger mais n'hérite pas de **JComponent**)



Les conteneurs (2)

- Les containers « top-level » ne peuvent contenir directement d'autres composants
- Ils sont associés à un autre container, le « content pane » dans lequel on peut ajouter les composants
- On obtient ce content pane par **Container contentPane = topLevel.getContentPane();**
- **topLevel** est un container lourd quelconque ; **JFrame** par exemple
- **JPanel** est la classe mère des containers intermédiaires les plus simples ; il sert à regrouper des composants dans une zone d'écran
- Avec un Jpanel opaque on peut créer notre propre ContentPane et l'associer à un topLevel

Hiérarchie d'héritage des composants lourds



Placer les composants dans un Conteneur

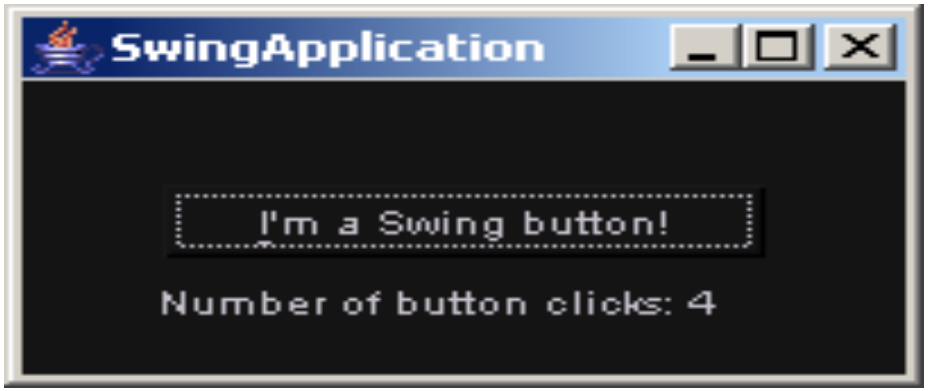
- Les conteneurs sont des éléments graphiques capables de contenir d'autres composants graphiques.
- **public component add(Component comp)** : Cette méthode est utilisée pour ajouter le composant spécifié à la fin du conteneur. Elle renvoie l'objet ajouté.
- **public component add(Component comp, int index)** : Ajoute le composant spécifié à la position spécifié par index. Elle renvoie l'objet ajouté.
- **public void remove(int index)** : Supprime le composant situé à la position index.
- **public void remove(Component comp)** : Supprime le composant spécifié.
- **public void removeAll()** : Supprime tous les composants contenus dans ce conteneur.

Quelques méthodes des conteneurs

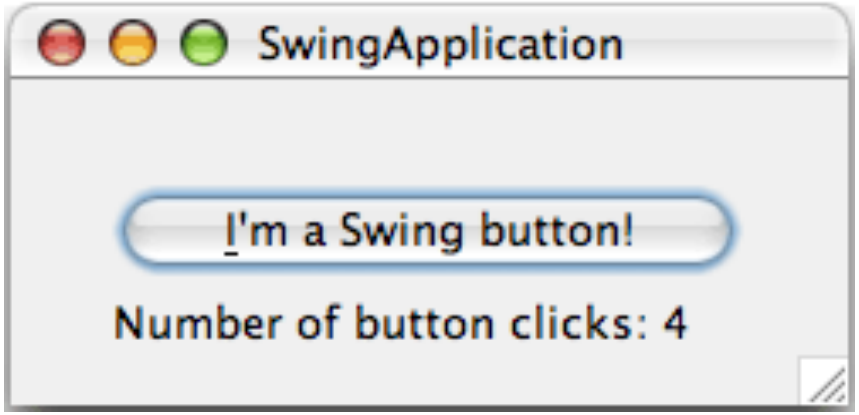
- **public LayoutManager getLayout()** : Renvoie le Layout de ce conteneur.
- **public void setLayout(LayoutManager mgr)** : Définit le Layout de ce conteneur comme étant le layout spécifié par mgr.
- **public void paint(Graphics g) :** Masque la même méthode de la classe Component. Dessine le conteneur et fait appel à la méthode paint(Graphics) de tout composant contenu dans ce conteneur.
- **public void repaint() :** Masque la même méthode de la classe Component. Met à jour le dessin du conteneur et fait appel à la méthode update(Graphics) de tout composant contenu dans ce conteneur.

Look and Feel (1)

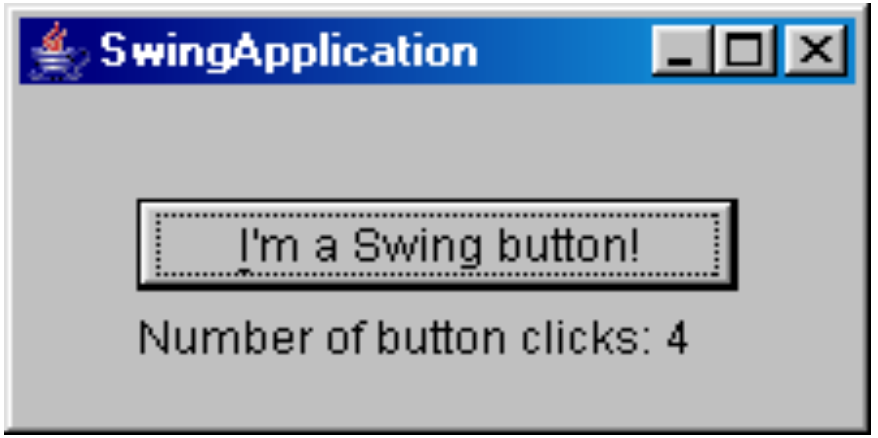
Java L&F



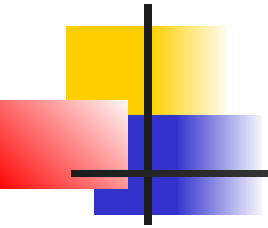
GTK L&F



Mac L&F



Windows L&F



Look and Feel (2)

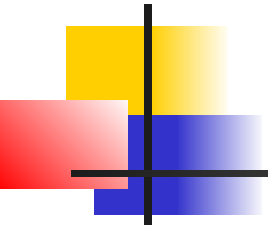
- Pour programmer un Look and Feel particulier utiliser la méthode **UIManager.setLookAndFeel (String lf) throws Exception**
- Récupérer le L&F du système

  UIManager. getSystemLookAndFeelClassName()
- GTK L&F "com.sun.java.swing.plaf.gtk.GTKLookAndFeel" à partir de JDK v1.4.2
- Windows L&F "com.sun.java.swing.plaf.windows.WindowsLookAndFeel"
- Java L&F "javax.swing.plaf.metal.MetalLookAndFeel" ou

  UIManager.getCrossPlatformLookAndFeelClassName()
- Unix L&F "com.sun.java.swing.plaf.motif.MotifLookAndFeel"

Récupérer les L&F installés

```
import javax.swing.*;
public class ListPlafs {
public static void main (String args[]) {
//retourne un tableau de L&F
UIManager.LookAndFeelInfo plaf[] =
UIManager.getInstalledLookAndFeels();
for (int i=0, n=plaf.length; i<n; i++) {
System.out.println("Nom LF : " + plaf[i].getName());
System.out.println("Nom de la classe : " + plaf[i].getClassName());
}
System.exit(0);}
}
```

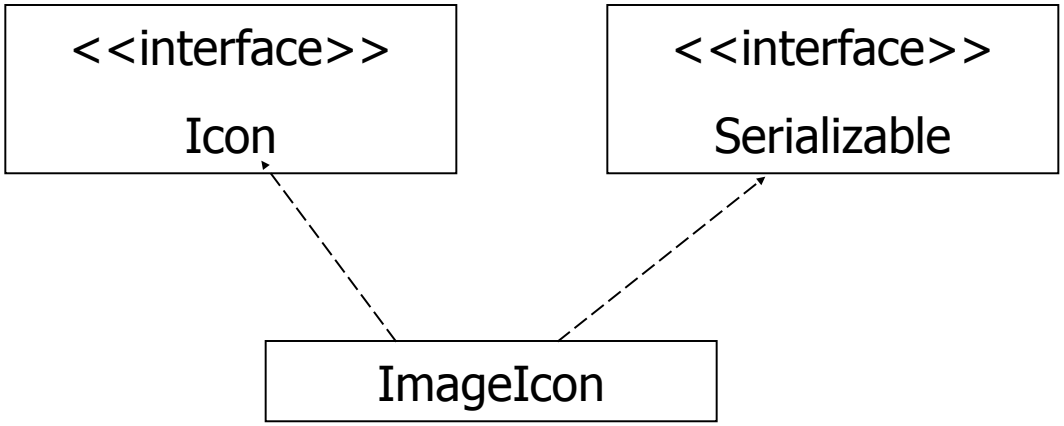


Les images

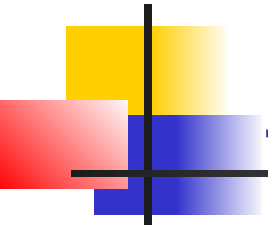
- Elles sont représentées en Java par 3 classes
- **java.awt.Image** : une image de base
- **java.awt.image.BufferedImage** : permet de manipuler les pixels de l'image
- **javax.swing.ImageIcon** : correspond à une image de taille fixe utilisée pour décorer un composant
- Contrairement à Image, ImageIcone est sérialisable
- En Swing, on peut l'inclure dans un **JLabel** (le plus simple) mais aussi dans un **JPanel** ou un **JComponent**
- Avec AWT, on inclut les images dans un **Canvas**
- De base, Java sait travailler avec les images GIF, JPEG ou PNG, et les GIF animés

ImageIcone

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

int getIconWidth();

int getIconHeight(); }
```



- Il existe des constructeurs de **ImageIcon** pour créer une instance à partir de :
  - un nom de fichier absolu ou relatif ; le séparateur est « / » quel que soit le système
  - un URL
  - une image (**Image**)
  - un tableau de byte (**byte[]**)
- **Icon icone = new ImageIcon("images/image.gif");**



JLabel

- Avec un Jlabel, on peut créer un texte « étiquette » et/ou une image
- On peut également intégrer du HTML
- Spécifier la position du texte par rapport à l'image
- Spécifier l'emplacement par rapport à son conteneur
- Quelques méthodes publiques
  - **JLabel(String, Icon, int) // Création d'un JLabel**
  - **void setText(String) // Modifie le texte du JLabel**
  - **String getText() // retourne le texte du Jlabel**
  - **void setIcon(Icon) et Icon getIcon()**
  - **// spécifier la position du texte par rapport à l'icone**
  - **void setHorizontalTextPosition(int)**
  - **void setVerticalTextPosition(int)**
  - **void setToolTipText(String) //associe un info bulles**

Jlabel Exemple

```
public class LabelPanel extends JPanel {
  public LabelPanel() {
    JLabel testLabel = new JLabel("Icon Big Label");
    testLabel.setToolTipText("a Label with Icone");
    // Instantiate a Font object to use for the label
    Font serif32Font =    new Font("Serif", Font.BOLD | Font.ITALIC, 32);
    // Associate the font with the label
    testLabel.setFont(serif32Font);
    // Create an Icon
    Icon soundIcon = new ImageIcon("images/sound.gif");
    // Place the Icon in the label
   testLabel.setIcon(soundIcon);
    // Align the text to the right of the Icon
    testLabel.setHorizontalTextPosition(JLabel.RIGHT);
    // Add to panel
    add(testLabel);}}
```

JButton

- **//Création d'un Bouton avec icone**

JButton precedent = new **JButton**("Precedent",leftButtonIcon);

- **//Création d'un Bouton avec texte HTML**

new JButton("<html><h1>Ligne 1</h1></html>");

//positionnement du texte / à l'Icone

precedent.**setVerticalTextPosition**(**AbstractButton**.CENTER);

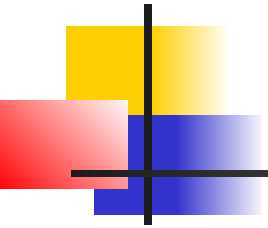
- **//Associer un raccourcis clavier au bouton ici p**

precedent.**setMnemonic**(KeyEvent.VK_P);

- **//associer une action à un bouton**

precedent.**setActionCommand**("disable");

- Quelques méthodes utiles
  - **void setText(String) et String getText()**
  - **void setIcon(Icon)et Icon getIcon()**
  - **char getMnemonic() String getActionCommand()**
  - **void setEnabled(Boolean)**



Les Layouts

- Les Layouts ou Layout Managers sont des classes qui implémentent une interface LayoutManager, et représentent des modèles de positionnement des composants.
- Par exemple, le FlowLayout arrange les composants de gauche à droite, jusqu'à ce qu'il n'y ait plus de place dans la même ligne, alors il commence à arranger les composants suivants sur la ligne suivante.
- Le BorderLayout arrange les composants selon des positions prédéfinies qui sont NORTH, WEST, CENTER, EAST, et SOUTH. Il redimensionne chaque composant de façon à ce que tout l'espace soit rempli.
- Pour utiliser un LayoutManager, il faut le lier à un conteneur. Cela s'effectue en faisant appel à la méthode setLayout de la classe Container.

FlowLayout

- Quand un conteneur utilise FlowLayout, les composants sont arrangés dans l'ordre de

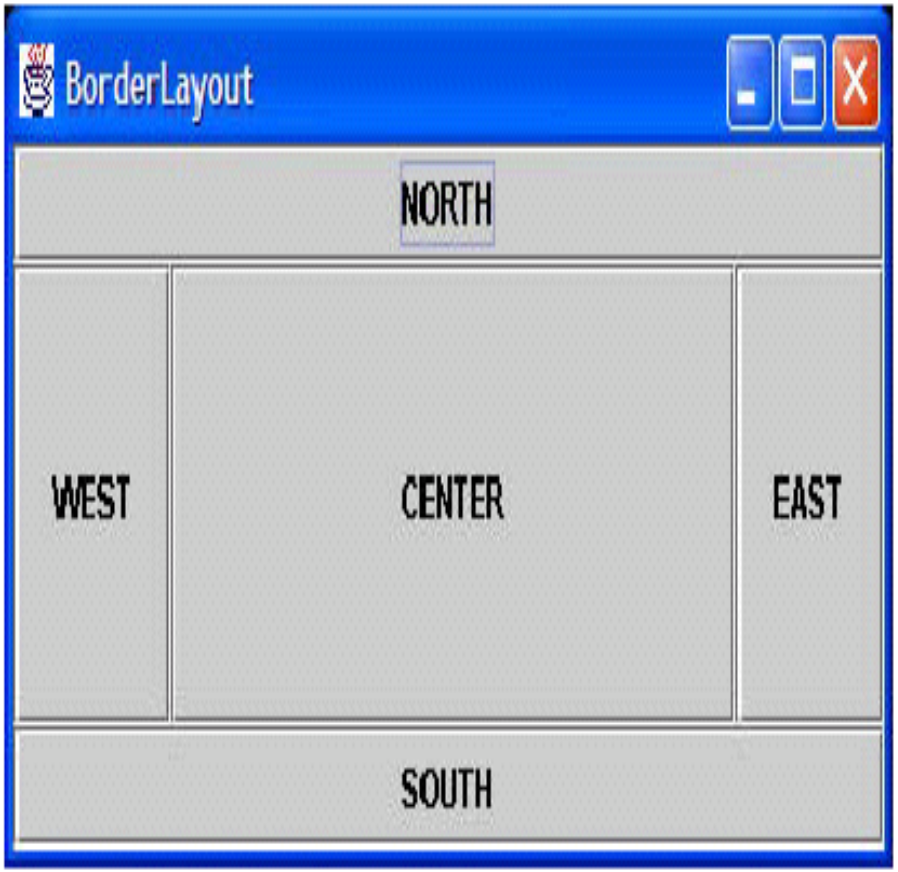
  leur ajout dans le conteneur, depuis le haut, de gauche à droite,
- Ne passant à la prochaine ligne que quand l'espace restant sur la ligne n'est plus suffisant pour contenir le composant.
- Il est utilisé par défaut dans les applets

```
Panel p = new Panel();
p.setLayout(new FlowLayout());
p.add(new Button("Bouton 1"));
p.add(new Button("Bouton Large 2"));
p.add(new Button("Bouton 3"));
p.add(new Button("4"));
p.add(new Button("Bouton encore plus
    large 5"));
p.add(new Button("Bouton 6"));
p.add(new Button("Bouton 7"));
p.add(new Button("Bouton 8"));
p.add(new Button("Bouton 9"));
add(p);
```

BorderLayout

- Quand on ajoute un composant à un contenaire qui utilise BorderLayout, on spécifie la région dans laquelle on veut le positionner.
- Si la région n'est pas spécifié, il le positionne automatiquement dans le centre.
- Les 5 régions utilisés par un BorderLayout sont :NORTH, WEST, CENTER, EAST et SOUTH

- JPanel p = new JPanel();
- p.setLayout(new BorderLayout());
- p.add(new JButton("NORTH"), BorderLayout.NORTH);

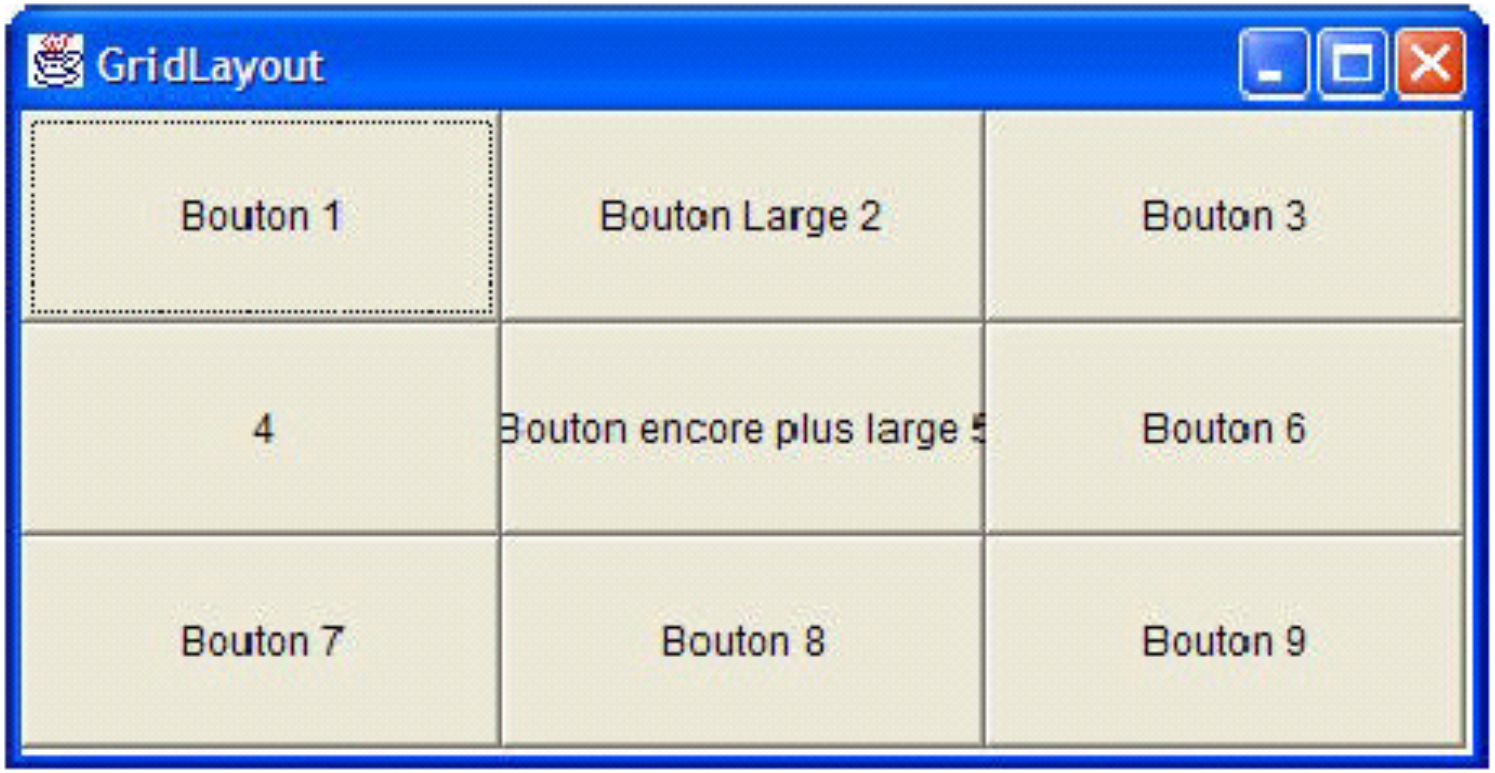


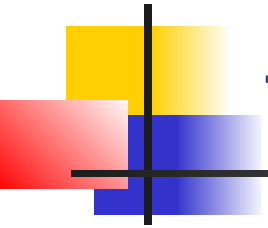
GridLayout

- Le GridLayout arrange les composants dans un nombre donné de colonnes et de lignes.
- Il commence en haut à gauche, et passe à la colonne suivante, et ainsi de suite, jusqu'à la dernière colonne, puis passe à la deuxième ligne de la première colonne, … etc.
- Le nombre de colonnes et de lignes est spécifié dans les arguments du constructeur :
- **new GridLayout(nombre_ligne, nombre_colonnes)**
- Le nombre de colonnes et de lignes ne peuvent pas être tous les deux des zéros. Si l'un ou l'autre est un zéro, il est interprété comme à déterminer par le nombre des composants.

```
JPanel p = new JPanel();
p.setLayout(new GridLayout(0,3));
p.add(new JButton("Bouton 1"));
p.add(new JButton("Bouton Large 2"));
p.add(new JButton("Bouton 3"));
p.add(new JButton("4"));
p.add(new JButton("Bouton encore plus
    large 5"));
p.add(new JButton("Bouton 6"));
p.add(new JButton("Bouton 7"));
p.add(new JButton("Bouton 8"));
p.add(new JButton("Bouton 9"));
add(p);
```

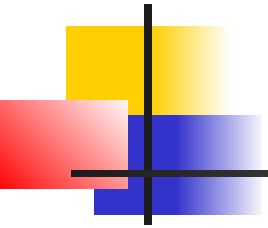
GridLayout l'exemple





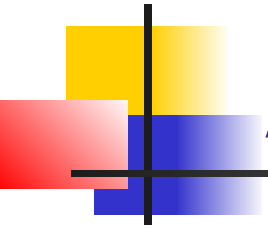
Traitement des événements

- L'utilisateur utilise le clavier et la souris pour intervenir sur le déroulement du programme
- Le système d'exploitation engendre des événements à partir des actions de l'utilisateur
- Le programme doit lier des traitements à ces événements
- On distingue deux types d'événement
- Bas Niveau :
  - appui sur un bouton de souris ou une touche du clavier
  - relâchement du bouton de souris ou de la touche
  - déplacer le pointeur de souris
- Logique
  - frappe d'un A majuscule
  - clic de souris
  - choisir un élément dans une liste



Classes d' événements

- Les événements sont représentés par des instances de sous-classes de **java.util.EventObject**
- Événements liés directement aux actions de l'utilisateur : **KeyEvent**, **MouseEvent**
- Événements de haut niveau : **FocusEvent**, **WindowEvent**, **ActionEvent**, **ItemEvent**, **ComponentEvent**
- Chacun des composants graphiques a ses observateurs (ou écouteurs, listeners)
- Un écouteur doit s'enregistrer auprès des composants qu'il souhaite écouter (addXXXListener) et il peut ensuite se désenregistrer (removeXXXListener)
- La méthode « **public Object getSource()** » de la classe **EventObject** permet d'obtenir l'objet d'où est parti l'événement, par exemple



ActionEvent

- Cette classe décrit des événements de haut niveau très utilisés qui correspondent à un type d'action de l'utilisateur qui va le plus souvent déclencher un traitement (une action) :
  - clic sur un bouton
  - return dans une zone de saisie de texte
  - choix dans un menu
- Ces événements sont très fréquemment utilisés et ils sont très simples à traiter
- Un objet ecouteur intéressé par les événements de type « action » (classe **ActionEvent**) doit appartenir à une classe qui implémente l'interface **java.awt.event.ActionListener**

Traitement de ActionEvent

```
//Fenetre est un écouteur de ActionEvent
class Fenetre extends JFrame implements ActionListener{
int nbClique=0;
public Fenetre(){
JButton ok=new JButton(”OK”+nbClique);
//inscription du bouton ok auprès de l’écouteur
Ok.addActionListener(this);
getContentPane().add(ok);}
//message déclenché lorsque l’utilisateur clique sur le bouton
public void actionPerformed(ActionEvent e){
nbClique++;
Object source = e.getSource();
If (source==ok) Ok.setText(”OK”+nbClique);
}}
```

Classe écouteur

- // Classe interne de Fenetre

```
class Fenetre extends JFrame{
...
EcouteurBouton eb= new EcouteurBouton();
Ok.addActionListener(eb);
class EcouteurBouton implements ActionListener {
public void actionPerformed(ActionEvent e) {
String commande = e.getActionCommand();
if (commande.equals("ok")) {. . .}
}}
```

- //Classe Anonyme

```
JButton ok = new JButton("OK");
ok.addActionListener(new ActionListener() {
public void actionPerformed(ActionEvent e) {} });
```

Événement, Interface et Écouteur

Event, listener interface and add- and remove-methods	Components supporting this event
ActionEvent **ActionListener** **addActionListener( )** **removeActionListener( )**	**JButton, JList, JTextField, JMenuItem** and its derivatives including **JCheckBoxMenuItem, JMenu,** and **JpopupMenu.**
AdjustmentEvent **AdjustmentListener** **addAdjustmentListener( )** **removeAdjustmentListener( )**	**JScrollbar** and anything you create that implements the **Adjustable interface.**
ComponentEvent **ComponentListener** **addComponentListener( )** **removeComponentListener( )**	**Component** and its derivatives, including **JButton, JCanvas, JCheckBox, JComboBox, Container, JPanel, JApplet, JScrollPane, Window, JDialog, JFileDialog, JFrame, JLabel, JList, JScrollbar, JTextArea,** and **JTextField.**

Événement, Interface et Écouteur

ContainerEvent **ContainerListener** **addContainerListener( )** **removeContainerListener( )**	**Container** and its derivatives, including **JPanel, JApplet, JScrollPane, Window, JDialog, JFileDialog,** and **JFrame.**
FocusEvent **FocusListener** **addFocusListener( )** **removeFocusListener( )**	**Component** and **derivatives*.**
KeyEvent **KeyListener** **addKeyListener( )** **removeKeyListener( )**	**Component** and **derivatives*.**
MouseEvent (for both clicks and motion) **MouseListener** **addMouseListener( )** **removeMouseListener( )**	**Component** and **derivatives*.**

Événement, Interface et Écouteur

MouseEvent (for both clicks and motion) **MouseMotionListener** **addMouseMotionListener( )** **removeMouseMotionListener( )**	**Component** and **derivatives*.**
WindowEvent **WindowListener** **addWindowListener( )** **removeWindowListener( )**	**Window** and its derivatives, including **JDialog, JFileDialog, and JFrame.**
ItemEvent **ItemListener** **addItemListener( )** **removeItemListener( )**	**JCheckBox, JCheckBoxMenuItem, JComboBox, JList,** and anything that implements the **ItemSelectable interface.**
TextEvent **TextListener** **addTextListener( )** **removeTextListener( )**	Anything derived from **JTextComponent,** including **JTextArea** and **JTextField.**

Ecouteur vs Adaptateur

- Pour éviter au programmeur d'avoir à implanter toutes les méthodes d'une interface « écouteur », AWT fournit des classes (on les appelle des adaptateurs), qui implantent toutes ces méthodes
- Le code des méthodes ne fait rien. Ça permet au programmeur de ne redéfinir dans une sous-classe qui l'intéressent méthodes
- Les classes suivantes du paquetage **java.awt.event** sont des adaptateurs : **KeyAdapter**, **MouseAdapter**, **MouseMotionAdapter**, **FocusAdapter**, **ComponentAdapter**, **WindowAdapter**
- **Exemple**
  - **addWindowListener(new WindowAdapter() {**
  - **public void windowClosing(WindowEvent e) {**
  - **System.exit(0);}});**

Interface Listener vs Adapter

Listener interface w/ adapter	Methods in interface
ActionListener	actionPerformed(ActionEvent)
AdjustmentListener	adjustmentValueChanged(AdjustmentEvent)
ComponentListener ComponentAdapter	componentHidden(ComponentEvent) componentShown(ComponentEvent) componentMoved(ComponentEvent) componentResized(ComponentEvent)
ContainerListener ContainerAdapter	componentAdded(ContainerEvent) componentRemoved(ContainerEvent)
FocusListener FocusAdapter	focusGained(FocusEvent) focusLost(FocusEvent)

Interface Listener vs Adapter

KeyListener **KeyAdapter**	**keyPressed(KeyEvent)** **keyReleased(KeyEvent)** **keyTyped(KeyEvent)**
MouseListener **MouseAdapter**	**mouseClicked(MouseEvent)** **mouseEntered(MouseEvent)** **mouseExited(MouseEvent)** **mousePressed(MouseEvent)** **mouseReleased(MouseEvent)**
MouseMotionListener **MouseMotionAdapter**	**mouseDragged(MouseEvent)** **mouseMoved(MouseEvent)**

Interface Listener vs Adapter

WindowListener **WindowAdapter**	**windowOpened(WindowEvent)** **windowClosing(WindowEvent)** **windowClosed(WindowEvent)** **windowActivated(WindowEvent)** **windowDeactivated(WindowEvent)** **windowIconified(WindowEvent)** **windowDeiconified(WindowEvent)**
ItemListener	**itemStateChanged(ItemEvent)**

Évènements clavier

- **Utilisation du clavier : KeyListener**
  - **public interface KeyListener extends EventListener {**
  - **//Action sur une seule touche clavier**
  - **void keyPressed(KeyEvent e);**
  - **void keyReleased(KeyEvent e);**
  - **// Action sur plusieurs touches clavier ou caractère unicode**
  - **void keyTyped(KeyEvent e); }**
- Si on n'est intéressé que par une des méthodes, on peut hériter de la classe **KeyAdapter**
- Dans ces méthodes, on peut utiliser les méthodes **getKeyChar()** et **getKeyCode()** pour récupérer le caractère tapé.
- **Exemple :**
  - `class EcouteCaracteres extends KeyAdapter {`
  - `public void keyTyped(KeyEvent e) {`
  - `if (e.getKeyChar() == 'q') quitter();}}`

JCheckBox zone à cocher

- **//Création d'un checkBox avec icône**

JCheckBox cbn**= new JCheckBox(**"No Choose Me", false**);**
JCheckBox cbc**= new JCheckBox(**"Choose Me");

- **//selectionner cbi par defaut**

cbc**.setSelected**(true);

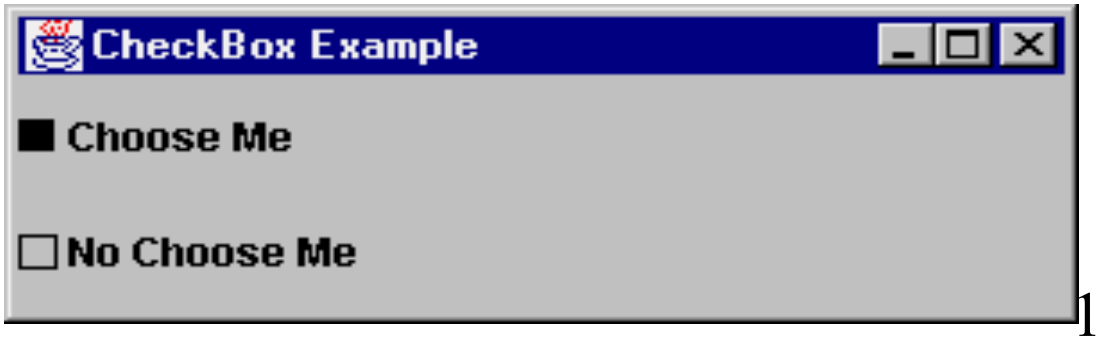
- **//recuperer l etat d un checkbox**

boolean etat=cbc**.isSelected()**

- **// associer un raccourcis clavier**

cbc**.setMnemonic(c);**

- **public void itemStateChanged(ItemEvent e) { //Ecouteur**
- **JCheckBox choix=(JCheckBox)e.getItemSelectable();}**
- **Exemple avec Icone**



Exemple de code

```
Icon unchecked = new ToggleIcon
    (false);
Icon checked = new ToggleIcon (true);
JCheckBox cb1 = new
    JCheckBox("Choose Me", true);
cb1.setIcon(unchecked);
cb1.setSelectedIcon(checked);
JCheckBox cb2 = new
    JCheckBox( "No Choose Me",
    false);
cb2.setIcon(unchecked);
cb2.setSelectedIcon(checked);
```

```
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

- Dans awt, les boutons radio sont des checkboxs qui appartiennent au même CheckboxGroup
- Swing offre un nouveau composante qui est JRadioButton
- Chaque JRadioButton est ajouté à un ButtonGroup
- Comme CheckboxGroup, le ButtonGroup est un composant logique qui n'a aucun effet visuel
- Exemple d'utilisation
  - **ButtonGroup** rbg = **new ButtonGroup();**
  - **JRadioButton** Bird = **new JRadioButton("Bird");**
  - rbg.**add** (Bird); **// ajouter le bouton au groupe**
  - Bird.**setSelected(true);** Bird.**setMnemonic**(b);

Les différents composants texte

- **JLabel** : une ligne de texte non modifiable par l'utilisateur
- **JTextField** : entrer d'une seule ligne
  - **JTextField** tf**= new JTextField();**
  - tf**.setText("nouveau Texte");**
- **JPasswordField** : entrer un mot de passe non affiché sur l'écran
  - **JPasswordField** textfield **= new JPasswordField (20);**
  - textfield**.setEchoChar('#');//* par defaut**
  - **char[]** input **=** textfield**.getPassword();**

- **JTextArea** : quelques lignes de texte avec une seule police de caractères
- **JTextPane** : documents avec plusieurs polices et des images et composants inclus (pas étudié dans cette partie du cours)

Les différents composants texte

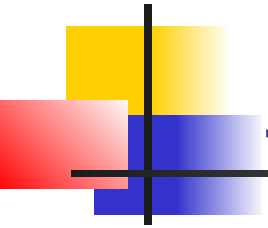
- Les méthodes communes composants texte
  - String getText() et void setText(String)
  - void setEditable(boolean) et boolean isEditable()
  - int getColomns() et void setColumns(int)
  - String getSelectedText(), int getSelectionEnd() et int getSelectionStart()
  - void select(int start, int end) et selectAll()
    void copy() et past()
- **JTextArea**
  - **textArea = new JTextArea(5, 40);//lignes, colonnes**
  - **append(String)** ajouter du texte à la fin
  - **setLineWrap(boolean)** indique si les lignes trop longues sont coupées ou affichées sur la ligne suivante
  - **replaceRange**(String str, int start, int end)
  - **insert**(String str, int pos)

Ecouteur pour **JTextArea**

- Si on veut traiter tout de suite les différentes modifications introduites dans le texte :
  - **textArea = new JTextArea(5, 40);**
  - JScrollPane scrollPane = new JScrollPane(textArea, JScrollPane.VERTICAL_SCROLLBAR_ALWAYS, JScrollPane.HORIZONTAL_SCROLLBAR_ALWAYS);
  - **textArea.getDocument ().addDocumentListener(new**
  - **MyDocumentListener());**
  - **. . .**
  - **class MyDocumentListener implements DocumentListener {**
  - **public void insertUpdate(DocumentEvent evt) {}**
  - **public void removeUpdate(DocumentEvent evt) {}**
  - **public void changedUpdate(DocumentEvent evt) {}}**

JComboBox Zone de Liste modifiable

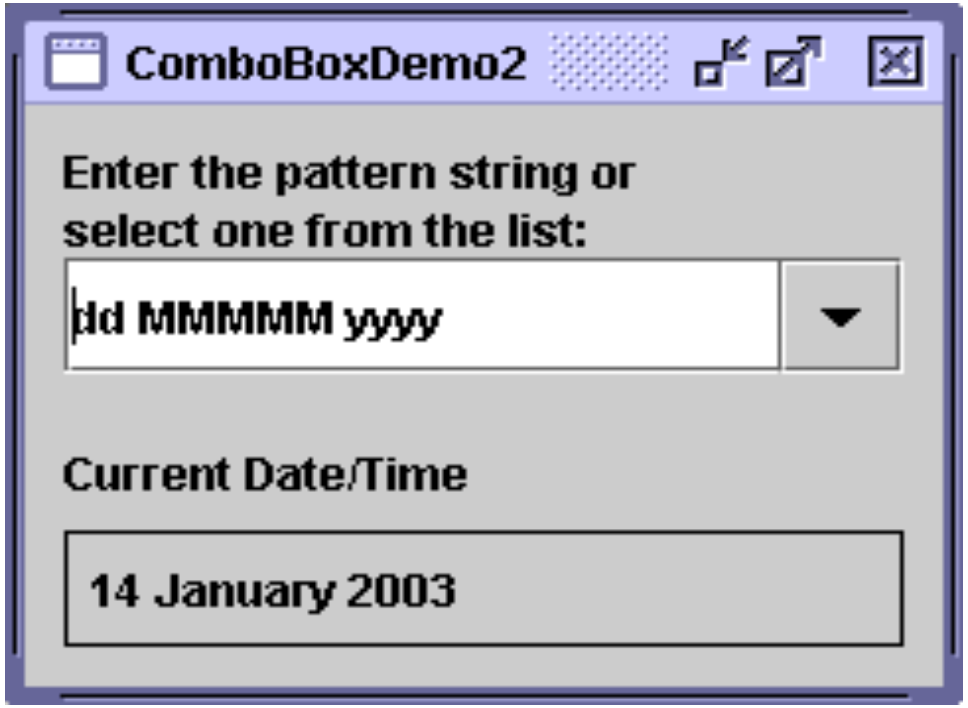
- Même fonctionnement que le composant Choice avec quelques comportement supplémentaires
- //crérer un combobox
  - String[] Villes = {"Rabat", "Casa", "Oujda", "Layoune"};
  - JComboBox cbVilles= new JComboBox() ;
- //Ajouter les éléments à la liste
  - for (int i=0;i<Villes.length;i++) { cbVilles.addItem (Villes[i]);}
- // On peut initialiser la liste au moment de sa création
  - JComboBox cbVilles= new JComboBox(Villes) ;
- // choisir un element par defaut
  - combo2.setSelectedItem("Rabat");//setSelectedIndex(0)



JComboBox Méthodes supplémentaires

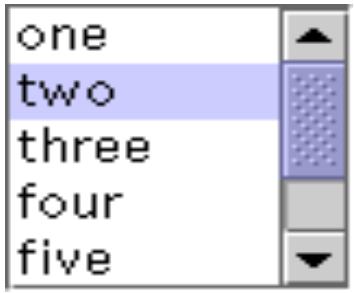
- Ajouter / Récupérer un élément sélectionné ou dans une position spécifique
  - insertItemAt(Objetc , int)/ Object getItemAt(int)
  - Object getSelectedItem()
- Supprimer un ou plusieurs éléments
  - removeAllItems() void removeItemAt(int)
  - void removeItem(Object)
- Méthodes divers
  - void setEditable(boolean) boolean isEditable()
  - int getItemCount()
- Écouteurs
  - ActionListener
  - ItemListener

Exemple de JComboBox



- String[] patternExamples = { "dd MMMMM yyyy", "dd.MM.yy", "MM/dd/yy", "yyyy.MM.dd G 'at' hh:mm:ss z", "EEE, MMM d, ''yy", "h:mm a", "H:mm:ss:SSS", "K:mm a,z", "yyyy.MMMMM.dd GGG hh:mm aaa" };
- JComboBox patternList = new JComboBox(patternExamples); **patternList.setEditable(true);** patternList.addActionListener(this);
- Date today = new Date(); SimpleDateFormat formatter = new SimpleDateFormat(currentPattern); try { String dateString = formatter.format(today);} catch (IllegalArgumentException iae)

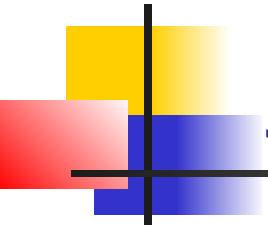
JList liste non modifiable




```
Jlist list = new JList(data); //data de
type Object[]
list.setSelectionMode(ListSelectionModel
.SINGLE_SELECTION);
```

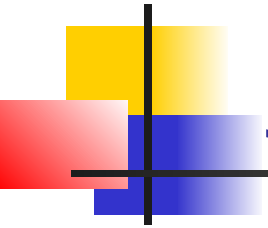

```
list.setSelectionMode(ListSelectionModel.S
INGLE_INTERVAL_SELECTION);
```


```
list.setSelectionMode(ListSelectionModel.
MULTIPLE_INTERVAL_SELECTION);
```



JList liste non modifiable

- //Récupérer ou modifier le nombre d'éléments visible
  - **void setVisibleRowCount(int) int getVisibleRowCount()**
- //Récupérer ou modifier le mode de séléction
  - **void setSelectionMode(int) int getSelectionMode()**
- //Récupérer les infos sur la séléction
  - **int getSelectedIndex() int getMinSelectionIndex()**
    int getMaxSelectionIndex()
  - **int[] getSelectedIndices()**
    Object getSelectedValue()
  - **Object[] getSelectedValues()**
- //Tester ou initialiser la séléction
  - **void clearSelection() boolean isSelectionEmpty()**



JList liste non modifiable

- Il est rare d'écrire un écouteur de liste. L'utilisation standard d'une liste est de demander à l'utilisateur de cliquer sur un bouton lorsqu'il a fini de faire ses choix dans la liste
- On récupère alors la sélection de l'utilisateur par une des méthodes **getSelectedValue()** ou **getSelectedValues()** dans la méthode **actionPerformed()** de l'écouteur du bouton
- La classe d'un écouteur de liste doit implémenter l'interface **ListSelectionListener** qui contient la méthode
  - **void valueChanged(ListSelectionEvent e)**

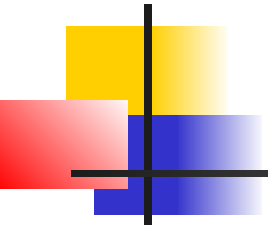
Modèle de Liste

- Une liste modifiable est associée à un modèle qui fournit les données affichées par la liste :
  - **public JList(ListModel dataModel)**
- **public interface ListModel {**
- **int getSize();**
- **Object getElementAt(int i);**
- **void addListDataListener(ListDataListener l);**
- **void removeListDataListener(ListDataListener l);}**
- Pour écrire une classe qui implémente **ListModel**, le plus simple est d'hériter de la classe abstraite **AbstractListModel** qui implémente les 2 méthodes de **ListModel** qui ajoutent et enlèvent les écouteurs
- Les plus simples ont un modèle de la classe **DefaultListModel** qui hérite de la classe **AbstractListModel**

Modèle de Liste

- **ListModel pays = new DefaultListModel();**
- **pays.addElement("Maroc");**
- **pays.addElement("France");**
- **pays.addElement("Italie");**
- **pays.addElement("Espagne");**
- **liste = new JList(pays);**

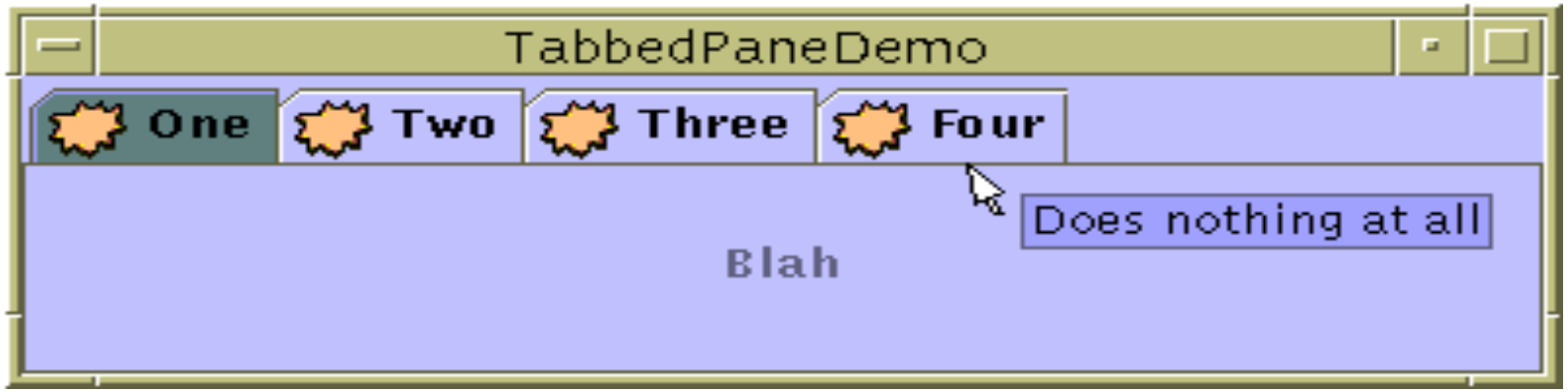
- Pour le modèle on doit créer une classe qui héritent de la classe **AbstractListModel**
- **/** Les 1000 premiers entiers composent le modèle de données de la liste */**
  - **class Entiers1000 extends AbstractListModel {**
  - **public int getSize() { return 1000; }**
  - **public Object getElementAt(int n) { return new Integer(n + 1); }}**
  - **JList liste = new JList(new Entiers1000());**



ProduitModèle

```
import java.util.*;
class ProduitModel extends AbstractListModel
{
   public Vector produits;
   public ProduitModel()
   {     produits = new Vector();}
   public int getSize(){   return produits.size();    }
   public Object getElementAt(int j)
   {     return produits.elementAt(j);}
   public void addElement(Object a)
   {     produits.addElement(a);
   this.fireIntervalAdded(this,0,getSize()-1);
   }
}
```

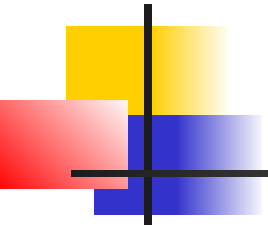
Les onglets JTabbedPane



- JTabbedPane tabbedPane = new JTabbedPane();
- ImageIcon icon = createImageIcon("images/middle.gif");
- JPanel panel4=new JPanel()
- tabbedPane**.addTab("Four", icon, panel4, "Does nothing at all");**
- tabbedPane**.setMnemonicAt(0, KeyEvent.VK_1);**

Les onglets JTabbedPane

- **// ajouter supprimer un élément**
  - **insertTab(String, Icon, Component, String, int)**
  - **void removeTabAt(int) void removeAll()**
- **//Récupérer un élément**
  - **int indexOfComponent(Component|String|Icon)**
  - **int getSelectedIndex() Component getSelectedComponent()**
- **//changer la couleur de l'arrière plan et du style d'écriture**
  - **void setBackgroundAt(int, Color) Color getBackgroundAt(int) void setForegroundAt(int, Color) Color getForegroundAt(int)**
- **//Activer désactiver un onglet**
  - **void setEnabledAt(int, boolean) boolean isEnabledAt(int)**



les boîtes de dialogues

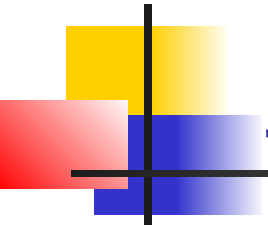
- **JOptionPane** est un composant léger, classe fille de **Jcomponent**
  - elle permet d'avoir très simplement les cas les plus fréquents de fenêtres de dialogue
  - message d'information avec bouton OK (**showMessageDialog**)
  - demande de confirmation avec boutons Oui, Non et Cancel (**showConfirmDialog)**
  - possibilité de configurer les boutons avec **showOptionDialog**
  - saisie d'une information sous forme de texte, **showInputDialog**
- **Jdialog** (composant lourd) est utilisée pour les cas non prévues par **JOptionPane**

JOptionPane Le constructeur

- Les arguments complets du constructeur de JoptionPane
  - Component frame **// associée à la boite de dialogue**
  - Object message **// à afficher sur la boite de dialogue vous pouvez // le mettre sue plusieurs lignes**

    "message1:\n \« message2\""
  - String titre **// le titre de la boite de dialogue**
  - int TypeBouton **// spécifier les boutons à faire apparaître**
  - int TypeIcone **// détermine l'icône à afficher**
  - Object[] options **// le message à afficher sur chaque bouton**
  - Object initialValue **// le bouton sélectionné par défaut**
- **Exemple**

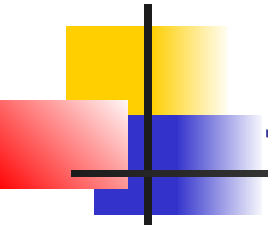
```
JOptionPane optionPane = new JOptionPane( "Quitter", "voulez vous vraiment Quitter",
JOptionPane.QUESTION_MESSAGE,JOptionPane.YES_NO_OPTION);
```



JOptionPane Types de messages

- On peut indiquer un type de message qui indiquera l'icône affichée en haut, à gauche de la fenêtre (message d'information par défaut)

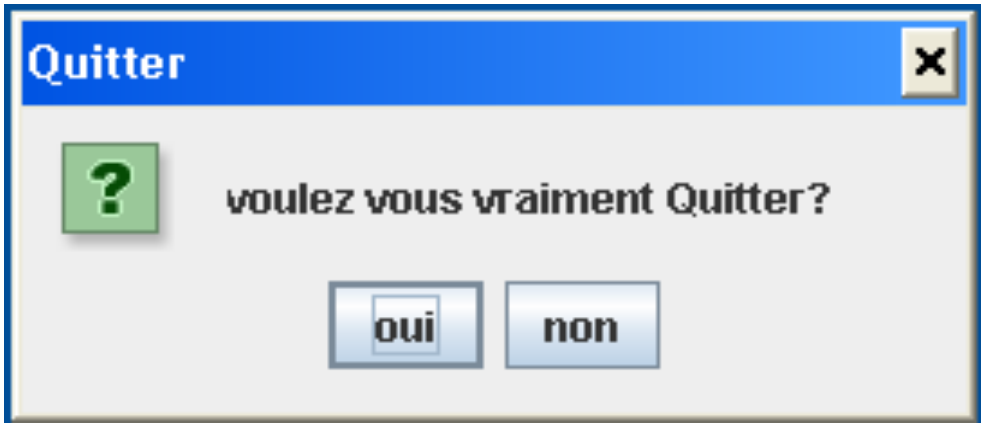
  - **JOptionPane.INFORMATION_MESSAGE**
  - **JOptionPane.ERROR_MESSAGE**
  - **JOptionPane.WARNING_MESSAGE**
  - **JOptionPane.QUESTION_MESSAGE**
  - **JOptionPane.PLAIN_MESSAGE (pas d'icône)**



JOptionPane Types de boutons

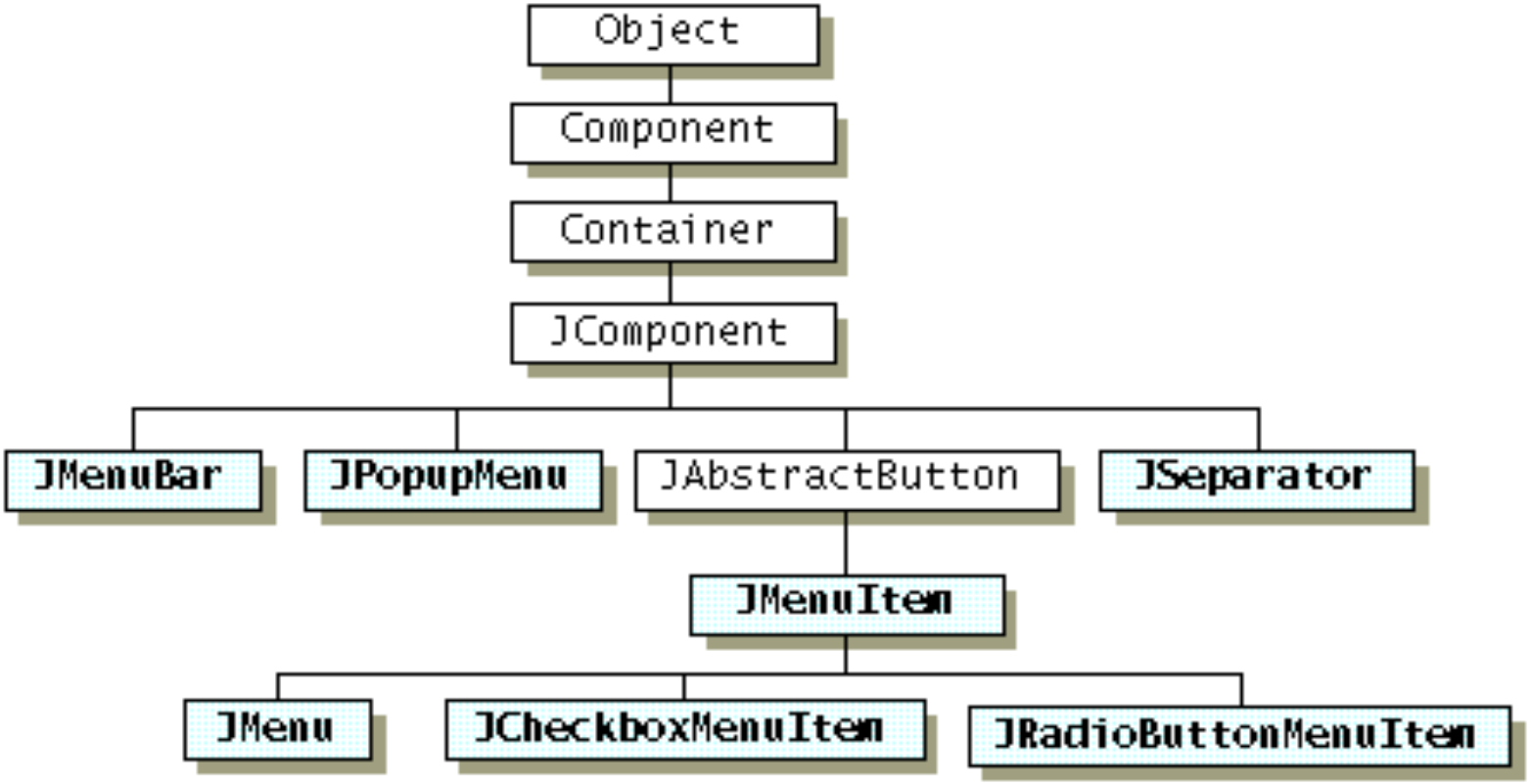
- Ils dépendent des méthodes appelées :
  - **showMessageDialog** : bouton Ok
  - **showInputDialog** : Ok et Cancel
  - **showConfirmDialog** : dépend du paramètre passé ; les différentes possibilités sont :
    - **DEFAULT_OPTION**
    - **YES_NO_OPTION**
    - **YES_NO_CANCEL_OPTION**
    - **OK_CANCEL_OPTION**
- **showOptionDialog** : selon le tableau d'objet passé en paramètres (vous pouvez franciser le texte à afficher sur les boutons)

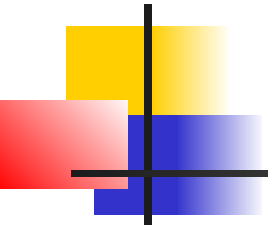
Exemple showOptionDialog




```
String [] options={"oui","non"};
int n =
JOptionPane.showOptionDialog(this,
    "voulez vous vraiment Quitter?","Quitter",
JOptionPane.YES_NO_OPTION,JOptionPane.QUESTION_MESSAGE,
    null,options,options[0]);
    if(n==1) this.dispose();
```

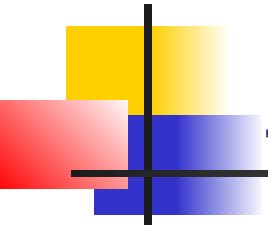
Les Menus


Object
Component
Container
JComponent
JMenuBar
JPopupMenu
JAbstractButton
JSeparator
JMenuItem
JMenu
JCheckboxMenuItem
JRadioButtonMenuItem



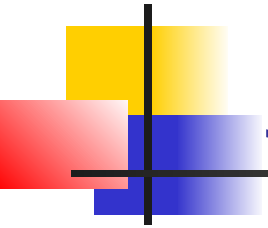
Les Menus

- Le modèle d'utilisation des menu swing est à peut prêt identique à celui des awt.
  - Les classes de menu (**JMenuItem, JCheckBoxMenuItem**, **JMenu, et JMenuBar**) sont toutes des sous classe de Jcomponent. Donc on peut placer un JMenuBar dans conteneur par **setMenuBar()**
  - Une nouvelle classe de menu JRadioButtonMenuItem
  - On peut associer une icône à un JMenuItem
- **JPopupMenu** Les pop-up sont des menus qui apparaissent à la demande (click bouton droit de la souris)



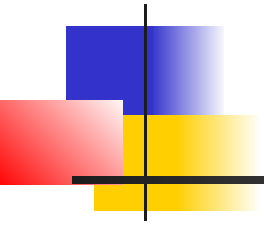
JMenuBar, Jmenu, JMenuItem

- Création d'une barre de menu
  - JMenuBar jmb = new JMenuBar();
- Création d'un menu fichier
  - JMenu fichier = new JMenu ("Fichier");
- Ajouter des éléments au menu Fichier
  - JMenuItem item;
  - fichier.add (item = new JMenuItem ("Nouveau")); fichier.add (item = new JMenuItem ("Ouvrir"));
- Ajout d'un séparateur de menu
  - fichier.addSeparator();
  - fichier.add(item =new JMenuItem ("Fermer"));
- Ajouter le menu fichier à la barre de menu
  - jmb.add (fichier);



JPopupMenu et JToolBar

- Création d'un menu pop-up
  - JPopupMenu popup = new JPopupMenu ();
- Ajouter un élément un menu-pop-up
  - JMenuItem item;
  - popup.add (item = new JMenuItem ("Cut"));
- Création d'une barre d'outils
  - JToolBar toolbar = new JToolBar();
- Ajouter un élément à la barre d'outils
  - JButton ouvrir=new JButton();
  - ouvrir.setIcon(("images/open.gif");
  - toolbar.add(ouvrir);
- Désactiver le déplacement de la barre d'outils
  - toolbar.setFloatable (false);

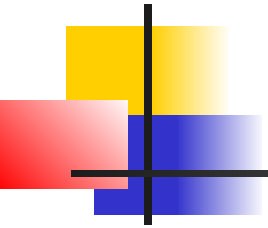


Java Avancé

Net, JDBC, Servlet, JSP

M. Lahmer

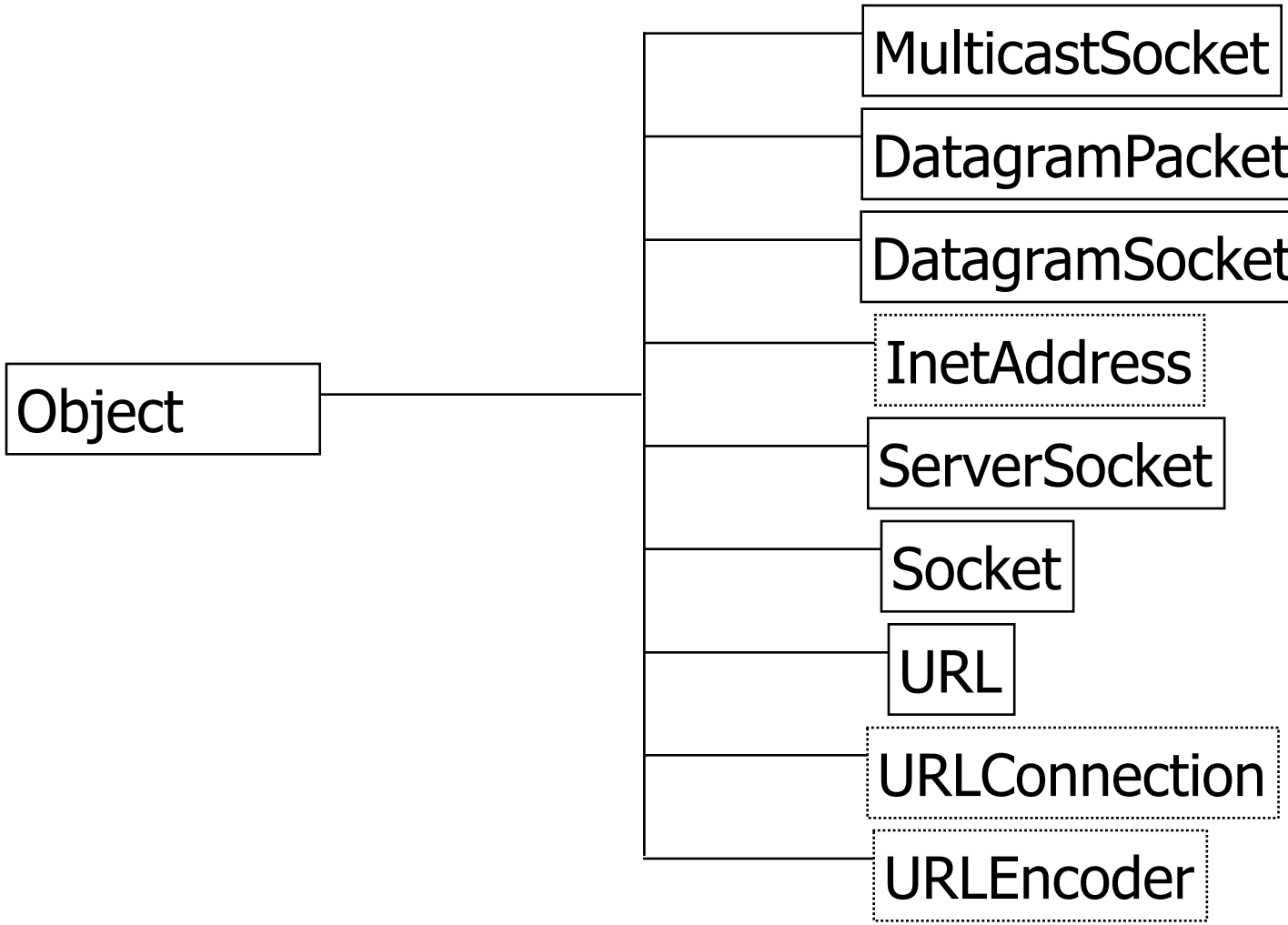
Mohamedlahmer@yahoo.fr



Plan Net

- Présentation de java.net
- Communication sous TCP
  - La classe socket : Le client
    - Constructeurs et méthodes de bases
    - Lecture et écriture à partire d'une socket
  - La classe ServerSocket : Le serveur
    - Constructeurs et méthodes de bases
    - Traitement d'une connexion
- Communication sous UDP
  - Construction d'un datagram(Packet|Socket)
  - Communication par DatagramSocket

java.net.*


Object
MulticastSocket
DatagramPacket
DatagramSocket
InetAddress
ServerSocket
Socket
URL
URLConnection
URLEncoder

La classe Socket **Les constructeurs**

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

```
try{ Socket toyahoo=new Socket(”www.yahoo.fr”,80);}
catch(UnknownHostException e) {System.err.println(e);}
```

- **public Socket(InetAddress host, int port) throws UnknownHostException, IOException**

```
InetAddress adr;
try{ adr=new InetAddress(”localhost”)
Socket tome=new Socket(adr,80);}
catch(UnknownHostException e) {System.err.println(e);}
```

- **public Socket(String hostDest, int portDest, InetAddress host, int port)**
- **public Socket(InetAddress hostDest, int portDest, InetAddress hostlocale, int portlocal)**

La classe Socket **Les informations**

- **public InetAddress getInetAddress()** retourne la machine sur laquelle la socket est connectée
- **public int getPort()** retourne le port de connection distant de la socket
- **public int getLocalPort()** retourne le port de connection local de la socket
- **public InetAddress getLocalAddress()** retourne l'addresse IP de la carte réseau locale associé à la socket

```
try{ Socket tolocal=new Socket(''localhost'',13);
    System.out.println(''Connecté a :''+tolocal.getInetAddress()
      +''sur le port :''+tolocal.getPort()+''Addresse
      Locale :''+tolocal.getLocalAddress()+''Port
      local :''+getLocalPort());
    }
    catch(UnknownHostException e) {System.err.println(e);}
```

La classe Socket **Lecture**

- **public InputStream getInputStream() throws IOException** retourne un flux que vous pouvez utiliser pour lire les données de la socket
- Lire la date à partir du serveur de temps qui écoute sur le port 13

```
Socket s;
DataInputStream theTimeStream;
InetAddress a;
try{
a=InetAddress.getByName("lahmer");
s=new Socket(a,13);
theTimeStream=new DataInputStream(s.getInputStream());
    String Stime = theTimeStream.readLine();
    System.out.println("aujourdhui nous somme le :"+Stime);
}
```

La classe Socket **Ecriture**

- **public OutputStream getOutputStream() throws IOException** retourne un flux que vous pouvez utiliser pour écrire les données dans la socket
- Le serveur echo écoute sur le port 7, il vous renvoie tous ce que vous lui envoyer

```
try{
    a=InetAddress.getByName("lahmer");
    s=new Socket(a,7);
DataInputStream canalLecture=new
DataInputStream(s.getInputStream());
printStream canalEcriture=new printStream(s.getOutputStream());
DataInputStream clavier= new DataInputStream(System.in);
while(true){
   String Line=clavier.readLine();
   if (Line.equals(''fin'')) break;
   canalEcriture.println(Line);
   System.out.println(canalLecture.readLine());}
```

La classe ServerSocket **Les constructeurs**

- **public ServerSocket(int port) throws IOException, BindException**

```
try{ ServerSocket httpd=new ServerSocket(80);}
catch(IOException e) {System.err.println(e);}
```

- **public Socket(int port, int queuelength) throws IOException, BindException**
  - Queuelength représente le nombre maximum de connections qu'on peut stocker
- **Public Socket(int port, int queuelength, InetAddress) throws IOException, BindException**

```
try{
 ServerSocket sd=
newServerSocket(1555,100,InetAddress.getHostByName(''localhost''));}
catch(IOException e) {System.err.println(e);}
```

La classe ServerSocket **Traiter une connexion**

- **public Socket accept() throws IOException**
  - bloque l'exécution jusqu'à ce qu'un client se connecte. Elle retourne un Objet Socket qui représente le client

```
ServerSocket server;
PrintStream p;
try{
    server=new ServerSocket(1350);
Socket client=server.accept();
     p=new PrintStream(client.getOutputStream());
 p.println("Vous etes connecte sur: "+server.getInetAddress());
     client.close();
 } catch(IOException e) {System.err.println(e);}}
```

UDP/Construction de Datagram

- **Public DatagramPacket(byte buffer[], int length) :**buffer contient les données, length est la taille de ces données. length<buffer.length<Max=64Ko=65535 o
  - byte[] buffer=new byte[8029];
  - DatagramPacket dp=new DatagramPacket(buffer, buffer.length)
- **Public DatagramPacket(byte buffer[], int length, InetAddress a, int port) :** permet de construire un datagramme à envoyer vers un host d'adresse a sur le port port
  - String s="Ce ci est un test"; **//s est la chaine à envoyer**
  - Byte[] data=new byte[s.length()];
  - s.getBytes(0,s.length(),data,0**);//formater le contenu de s dans data**
  - DatagramPacket dp =new DatagramPacket(data, data.length,InetAddress.getByName("localhost"),7);

Les méthodes de DatagramPacket

- **Public InetAddress getAddress()** et **Public int getPort() Public int getLength()** retourne le nombre de bytes des données contenues dans le Datagramme
- **Public byte[] getData()** retourne un tableau de bytes contenant les données du datagramme.
- Pour récupérer les données sous forme de chaîne de caractères en utilise :
- **String s= new String(dp.getData(),0,0,dp.getLength());**

```
String s=”Ce ci est un test”; //s est la chaine à envoyer
Byte[] data=new byte[s.length()];
s.getBytes(0,s.length(),data,0); //formater le contenu de s dans data
DatagramPacket dp =new DatagramPacket(data, data.length,InetAddress.getByName(”localhost”),7);
System.out.println(new String(dp.getData(),0,0,dp.getLength());
```

les constructeurs DatagramSocket

- **Public DatagramSocket() throws SocketException :** crée une socdket sur un port anonyme, dans telle situation le port destinataire est intégré dans le datagramme
- **Public DatagramSocket(int port) throws SocketException :** crée une socket UDP qui écoute sur le port spécifié en argument
- **Public DatagramSocket(int port, InetAddress a) throws SocketException :**

```
for (int i=0;i<=1024;i++)
try{
DatagramSocket server=new DatagramSocket(i);
server.close();
} catch(SocketException e){
System.out.println(”Il y a un serveur en écoute sur le port :”+i);}}
```

Envoyer et Recevoir sur DatagramSocket

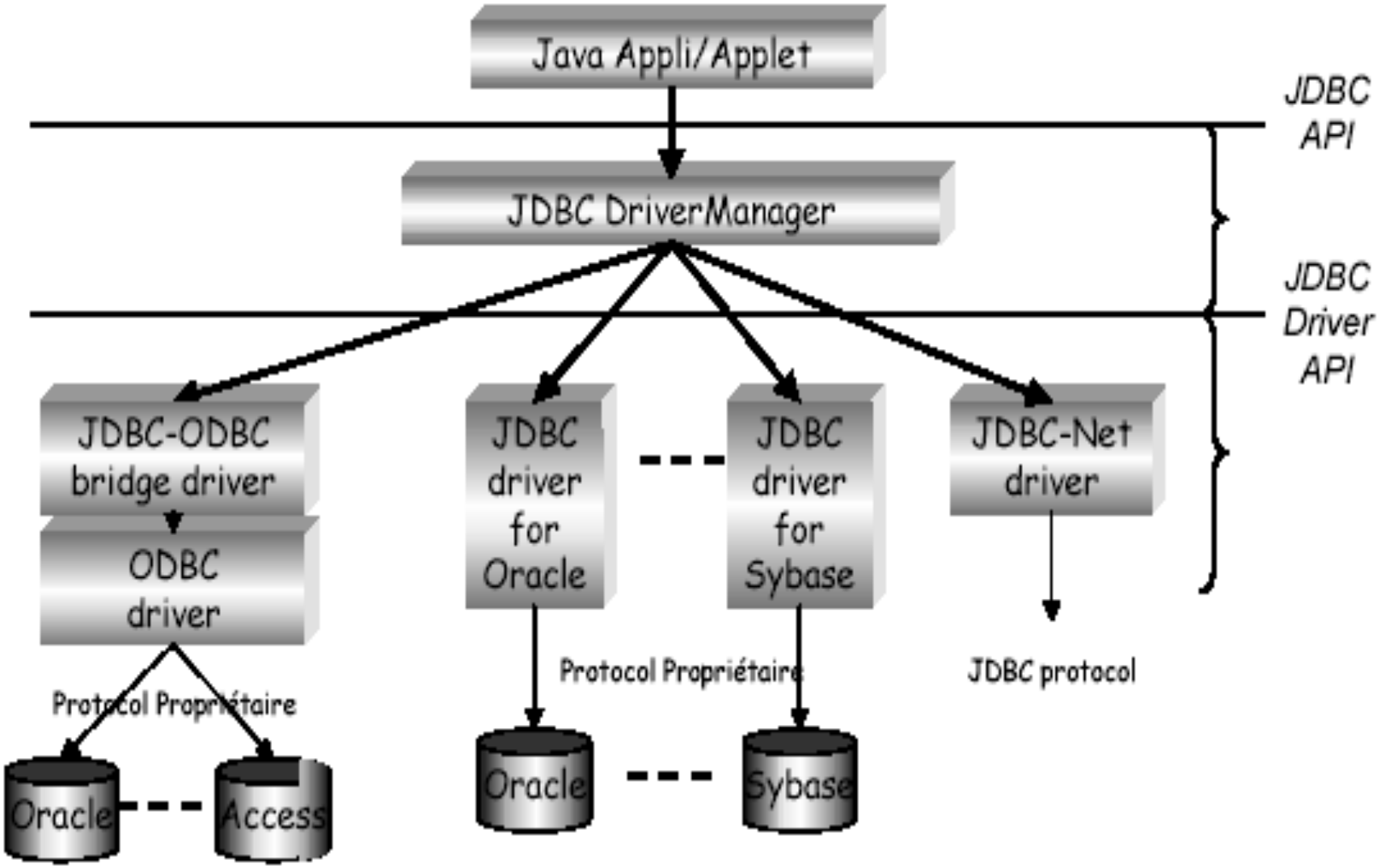
- **Public void send(DatagramPacket dp) throws IOException**
- **Public void receive(DatagramPacket dp) throws IOException**

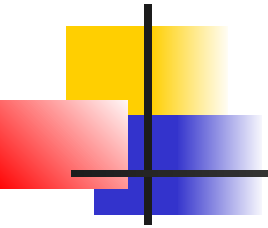
```
// Client
Byte[] data = "un message".getBytes();
InetAddress addr = InetAddress.getByName("falconet.inria.fr");
DatagramPacket packet = new DatagramPacket(data, data.length,
    addr, 1234);
DatagramSocket ds = new DatagramSocket();
ds.send(packet);
ds.close();
// Serveur
DatagramSocket ds = new DatagramSocket(1234);
while(true)
{  DatagramPacket packet = new DatagramPacket(new byte[1024],
    1024);
    s.receive(packet);
    System.out.println("Message: " + packet.getData());}
```

API JDBC Java Data Base Connectivity

- JDBC est une API fournie avec Java permettant l'accès à n'importe quelle base de données à travers un réseau
- Principe de fonctionnement
  - Chaque base de données utilise un pilote (*driver*) qui lui est propre et qui permet de convertir les requêtes JDBC dans le langage natif du SGBDR.
  - Ces drivers dits JDBC (un ensemble de classes et d ’interfaces Java) existent pour tous les principaux constructeurs :
    - Oracle, Sybase, Informix, SQLServer, MySQL, MsAccess

Pilotes JDBC





Mise en oeuvre

- Importer le package **java.sql**
- 1. Charger le driver JDBC
- 2. Établir la connexion à la base de données
- 3. Créer une zone de description de requête
- 4. Exécuter la requête
- 5. Traiter les données retournées
- 6. Fermer les différents espaces

Charger un pilote

- utiliser la méthode de chargement à la demande **forName()** de la classe **Class**
- Exemples :
  - **Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");**
  - **Class.forName("oracle.jdbc.driver.OracleDriver");**
  - **Class.forName(" com.microsoft.jdbc.sqlserver. SQLServerDriver ")**
- try { Class.forName("com.mysql.jdbc.Driver").

  newInstance();

  System.out.println("Tout est OK"); }

  catch (Exception e) { System.out.println("Erreur de drivers JDBC"); }

Structure d'une URL

- <protocol>:<subprotocol>:<subname>
- protocol : jdbc
- subprotocol : driver utilisé (mysql)
- subname : //<host>[:<port>][/<databasename>]
- par exemple **jdbc:mysql://localhost:port (MySQL)/client** ou
- pour une base de donnée locale **jdbc:odbc:NSD**

NSD : source de données associée à la base de données client exemple Access

Connexion à la base

- Une fois l'URL est crée,on fait appel à la méthode **getConnexion()** de **DriverManager** qui renvoie un objet de type **Connexion**.

– 3 arguments :

- l 'URL de la base de données
- le nom de l 'utilisateur de la base
- son mot de passe

– Exemple:

Connection connect =
DriverManager.getConnection(url,user,password);

- Le **DriverManager** essaye tous les drivers qui se sont enregistrés

Exécution des requêtes

- La première étape est de construire un objet ***Statement***
- **Statement** statement = connection.**createStatement();**
- Quand un objet ***Statement*** execute une requête, il retourne un objet ***ResultSet*** (pour les requête de consultation).
- L'exécution de la requête se fait par l'une des trois méthodes suivantes :
  - **executeQuery** : pour les requêtes de consultation. Renvoie un ***ResultSet*** pour récupérer les lignes une par une.
  - **executeUpdate** : pour les requêtes de modification des données (update, insert, delete) ou autre requête SQL (create table, ...). Renvoie le nombre de lignes modifiées.
  - **execute** : si on connaît pas à l'exécution la nature de l'ordre SQL à exécute
- **ResultSet** rs = statement.**executeQuery**("SELECT * FROM acc_acc");

Manipulation d'un ResultSet

- Différentes méthodes vous permettent de récupérer la valeur des champs de l'enregistrement courant
- Ces méthodes commencent toutes par get, immédiatement suivi du nom du type de données (ex: getString)
- chaque méthode accepte soit l'indice de la colonne (à partir de 1) soit le nom de la colonne dans la table
- boolean getBoolean(int); boolean getBoolean(String); byte getByte(int); byte getByte(String); Date getDate(int); Date getDate(String); double getDouble(int); double getDouble(String); float getFloat(int); float getFloat(String); int getInt(int); int getInt(String); long getLong(int); long getLong(String); short getShort(int); short getShort(String); String getString(int); String getString(String);

Manipulation d'un ResultSet

- Exemple de parcours d'un ResultSet
  - String strQuery = "SELECT * FROM T_Users;";
  - **ResultSet** rsUsers = stUsers.exexcuteQuery(strQuery); while(rsUsers.next()) {
  - System.out.print("Id[" + rsUsers.getInt(1) + "]" + "Pass[" + rsUsers.getString("Password")); }
  - rsUsers.close();
- Pour parcourir les enregistrements de n'importe quelle manière
- Il faut passer des paramètres à la méthode **createStatement** de l'objet de connexion.
- st = conn.createStatement(type, mode);

Manipulation d'un ResultSet

- Type ==
  - ResultSet.TYPE_FORWARD_ONLY
  - ResultSet.TYPE_SCROLL_SENSITIVE
  - ResultSet.TYPE_SCROLL_INSENSITIVE
- Mode ==
  - ResultSet.CONCUR_READ_ONLY
  - ResultSet.CONCUR_UPDATABLE
- Pour le type scroll, on a les possibilités de parcours first, last , next et previous
- Modification d'un ResultSet
  - rsUsers.first(); // Se positionne sur le premier enregistrement
  - rsUsers.updateString("Password", "toto"); // Modifie la valeur du // password
  - rsUsers.updateRow(); Applique les modifications sur la base

Récupération des métadata

- JDBC permet de récupérer des informations sur le type de données que l'on vient de récupérer par un SELECT (interface **ResultSetMetaData**),mais aussi sur la base de données elle-même (interface **DatabaseMetaData**)
- Les données que l'on peut récupérer avec DatabaseMetaData dépendent du SGBD avec lequel on travaille

```
ResultSet rs = stmt.executeQuery("SELECT * FROM emp");
ResultSetMetaData rsmd = rs.getMetaData();
int nbColonnes = rsmd.getColumnCount();
for (int i = 1; i <= nbColonnes; i++) {
String typeColonne = rsmd.getColumnTypeName(i);
String nomColonne = rsmd.getColumnName(i);
System.out.println("Colonne " + i + " de nom " + nomColonne +
 " de type " + typeColonne);
```

Récupération des métadata

- **DatabaseMetaData**

```
private DatabaseMetaData metaData;
private java.awt.List listTables = new List(10);
. . .
metaData = conn.getMetaData();
// Récupérer les tables et les vues
String[] types = { "TABLE", "VIEW" };
ResultSet rs = metaData.getTables(null, null, "%", types);
String nomTables;
while (rs.next()) {
nomTable = rs.getString(3);
listTables.add(nomTable);}
```

PreparedStatement

- **PreparedStatement** : Requêtes dynamiques pré-compilées plus rapide qu'un **Statement** classique
- le SGBD n'analyse qu'une seule fois la requête pour de nombreuses exécutions d 'une même requête SQL avec des paramètres variables
- Gérée par le client JDBC.
- La méthode **prepareStatement() throws SQLException** de l 'objet **Connection** crée un **PreparedStatement**