



INF 2421

Programmation Orientée Objet 1

Licence Informatique – Semestre 4

Département informatique

UFR des Sciences et technologies

Université de Thiès

Présentation générale

- **Unité d'Enseignement**
 - Titre : INFORMATIQUE
 - Sigle : INF 242
- **Élément constitutif**
 - Titre : Programmation Orientée Objet 1
 - Sigle : INF 2421
- **Autres éléments constitutifs de l'UE (1/OPTION)**
 - Analyse et Conception des Systèmes Orientés Objet
(INF 2421)

Volume horaire & Notation

- **CM** : 30H (*lundi & mercredi 08H-12H*)
- **TD/TP** : 30H (*jeudi 08H-14H : G₁, G₂*)
- **TPE** : 60H
- **Coefficient de l'UE** : 4
- **Crédits de l'UE** : 11
- **Evaluation**
 - **Contrôle des connaissances** : 40%
 - **Examen écrit** : 60%

Responsables

- **Magistral**

Pr. Mouhamadou THIAM

Maître de conférences en Informatique

Intelligence Artificielle : Web Sémantique

Email : mthiam@univ-thies.sn

- **Travaux dirigés et pratiques**

M. Papa DIOP

Ingénieur Systèmes Informatiques et Bases de Données

Diplômé de l'Université Gaston Berger de Saint-Louis

Email : papaddiop@gmail.com

Objectifs

- Comprendre la notion d'objet
- Connaître
 - la notion POO
 - les bases du langage JAVA
 - les types de base du langage
 - Les structures de contrôle
 - Les tableaux
 - Chaînes de caractères

Plan

1. Programmation Orientée Objet
2. Introduction à JAVA
3. Techniques de base du langage
4. Types primitifs
5. Structures de contrôle
6. Tableaux
7. Chaînes de caractères

Plan

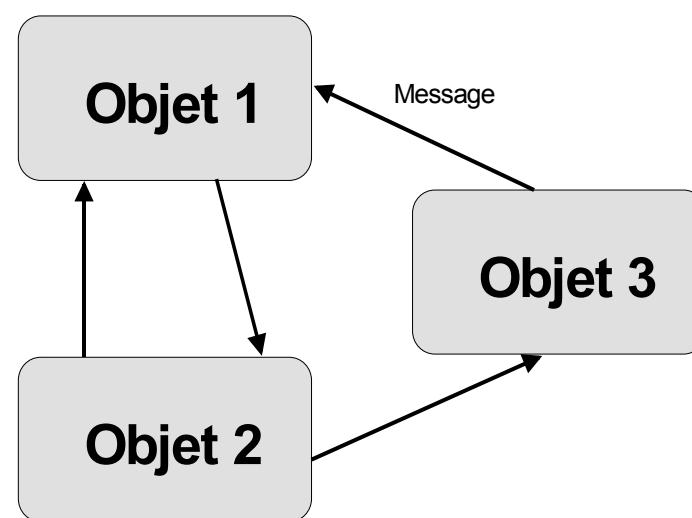
- 1. Programmation Orientée Objet**
2. Introduction à JAVA
3. Techniques de base du langage
4. Types primitifs
5. Structures de contrôle
6. Tableaux
7. Chaînes de caractères

Programmation Orientée Objet

POO

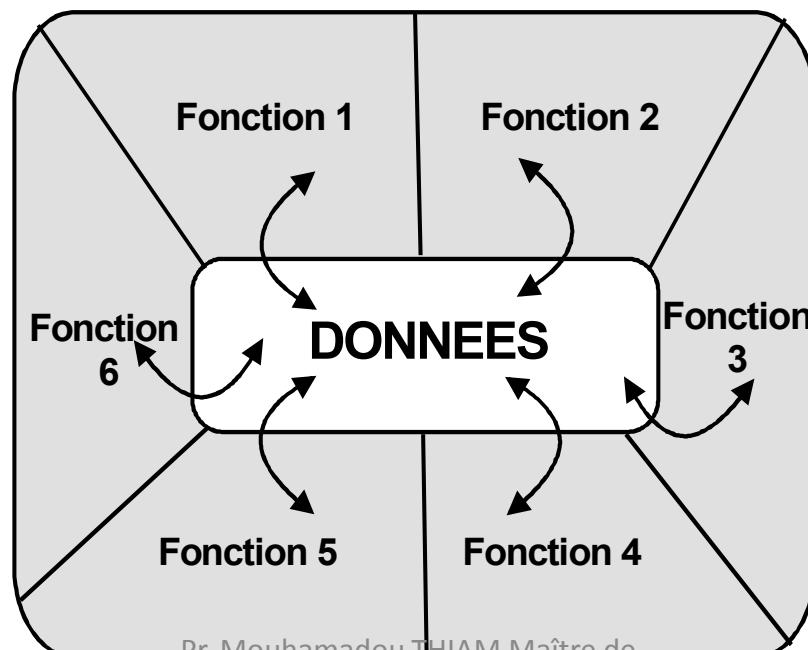
Programmation Orientée Objet (POO)

- **Qu'est ce qu'un Programme Orientée Objet ?**
 - Ensemble d'objets **autonomes** et **responsables** qui s'entraident pour résoudre un problème final en s'envoyant des messages.



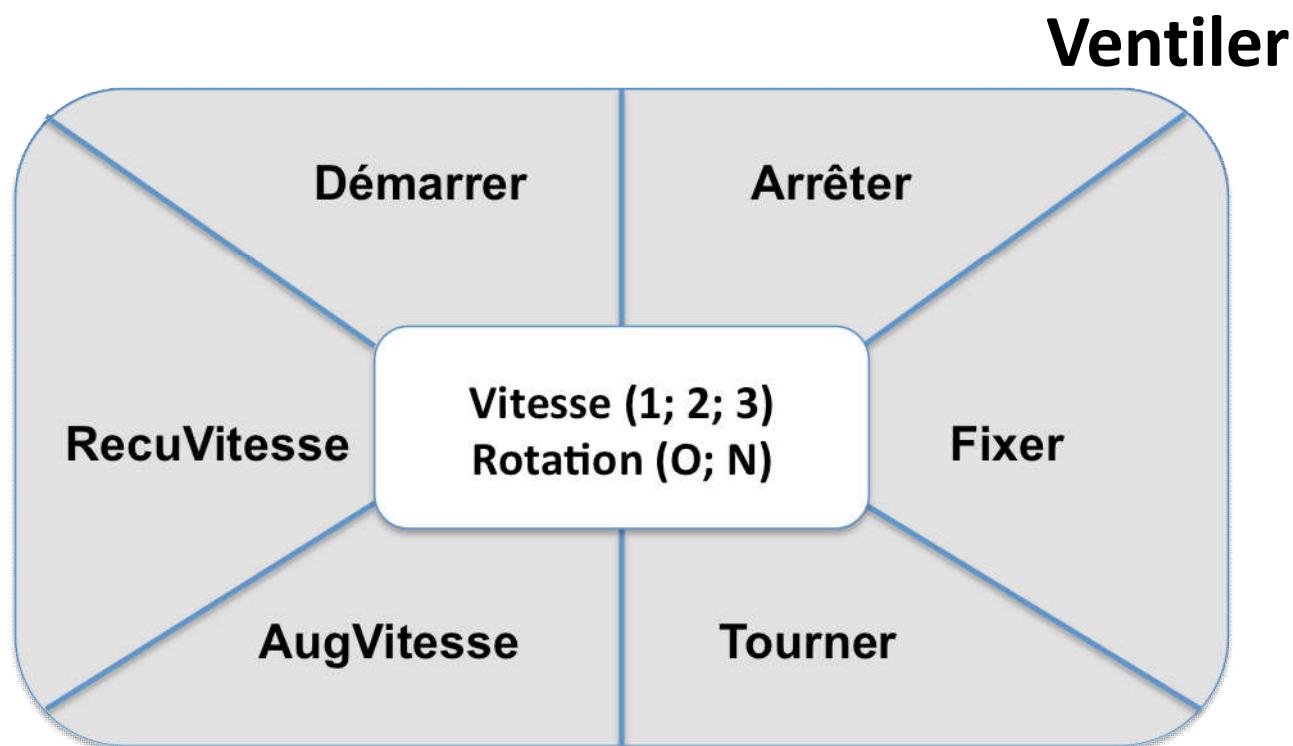
Programmation Orientée Objet

- Qu'est ce qu'un objet
 - Objet = Données + Méthodes (Fonctions Membres)



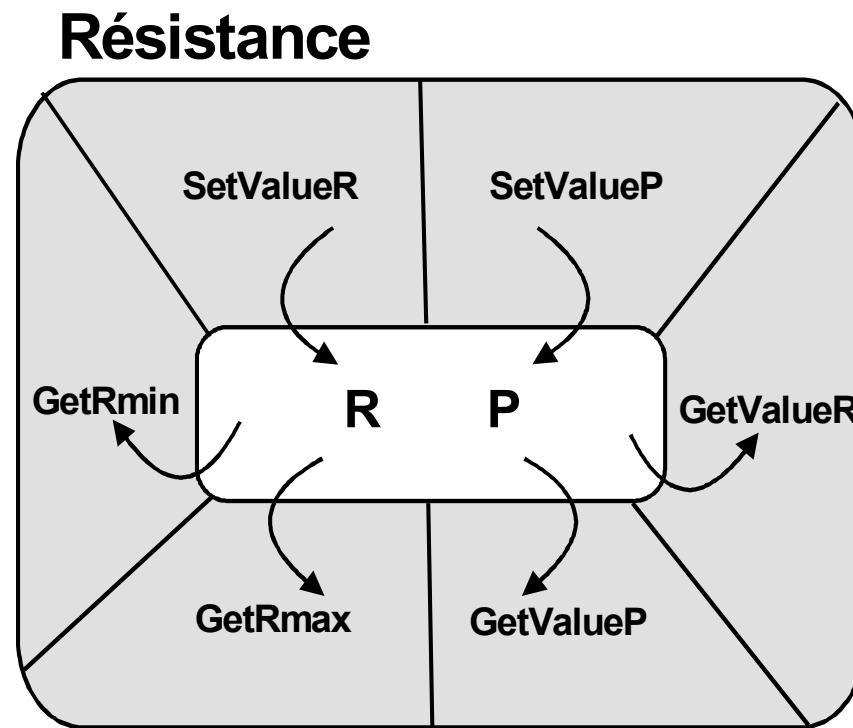
Programmation Orientée Objet

- Exemple d'un objet



Programmation Orientée Objet

- Exemple d'un objet



Pourquoi?

- Problème du logiciel:
 - Taille
 - Coût : développement et maintenance
 - Fiabilité
 - Solutions :
 - Modularité
 - **Réutiliser le logiciel**
 - Certification
- Comment?**

Typage...

- **Histoire:**
 - Fonctions et procédures (60 Fortran)
 - Typage des données (70) Pascal Algol
 - Modules: données + fonctions regroupées (80)
ada
 - Programmation objet: classes, objets et héritage

Principes de base de la POO

- Objet et classe:
 - Classe = définitions pour des données (**variables**)
+ fonctions (**méthodes**) agissant sur ces données
 - Objet = élément d'une classe (**instance**) avec un état
 - (une **méthode** ou une **variable** peut être
 - de classe = commune à la classe ou
 - d'instance = dépendant de l'instance

Principes de bases (suite)

- Concept de Classe
 - Type réunissant une description d'une
 - collection de données membres hétérogènes ayant un lien entre elles
 - collection de méthodes (fonctions membres) servant à manipuler les données membres
 - Généralisation du type *structure* ou *record* des langages non OO
 - Possibilité de fixer la visibilité externe des différents constituants de la classe

Principes de bases (suite)

- Instance d'une classe
 - Objet initialisé à partir de la description figée d'une classe
- Fonctions membres publiques
 - Les constructeurs
 - Les destructeurs
 - Les accesseurs
 - Les modificateurs

Principes de bases (suite)

- **Encapsulation** des données
 - Accès aux données des objets règlementé
 - privées → uniquement par les fonctions membres
 - publiques → accès direct par l'instance de l'objet
 - Conséquences
 - objet visible que par ses spécifications
 - modification interne sans effet sur le fonctionnement général du programme
 - Meilleure réutilisation de l'objet
 - Exemple de la résistance
 - Les données R et P ne peuvent être utilisées que par les fonctions membres

Principes de bases (suite)

- **Constructeurs**
 - Initialiser l'objet lors de sa création
 - copie des arguments vers les données membres
 - initialisation de variables dynamiques à la création de l'objet
 - Sont appelés de manière automatique à la création de l'objet
 - Peuvent être surchargés → On peut en définir plusieurs de même nom (arguments différents)
 - Possèdent le même nom que la classe

Principes de bases (suite)

- Destructeur
 - Il est **unique**
 - Appelé automatiquement lors de la destruction de l'objet
 - Sert généralement à éliminer les variables dynamiques de l'objet (créées en général par le constructeur)
 - Il a pour nom le nom de la classe précédé du symbole ~

Principes de bases (suite)

- **Accesseurs**
 - Fonctions membres qui permettent l'accès et l'utilisation des informations privées contenues dans l'objet
- **Modificateurs**
 - Fonctions membres qui permettent de modifier l'état des données internes (publiques ou privées) de l'objet

Principes de bases (suite)

- Héritage
 - Définir les bases d'un nouvel objet à partir d'un objet existant
 - Hériter des propriétés d'un ancêtre et recevoir de nouvelles fonctionnalités
- Avantages
 - Meilleures réutilisations des réalisations antérieures

Principes de bases (suite)

- **Polymorphisme** (Grec → plusieurs formes)
 - Un nom (de fonction, d'opérateur) peut être associé à plusieurs classe mais différentes utilisations
 - Exemple
 - $\sqrt{36} = ????$
 - $\sqrt{3 + 4i} = ?????$
 - Si **a** est un nombre complexe, **sqrt(a)** appellera si elle existe la fonction adaptée au type de **a**.

Principes de bases (suite)

- **Polymorphisme** (Grec → plusieurs formes)
 - Un nom (de fonction, d'opérateur) peut être associé à plusieurs mais différentes utilisations
 - Exemple
 - Si langage non OO, il aurait fallu connaître le nom de deux fonctions distinctes (selon que **a** soit complexe ou réel)
 - Le système choisit selon le type de l'argument ou de l'opérande

Principes de bases (POO2)

- **Polymorphisme:** exemple
 - Si une classe **A** est une extension d'une classe **B**:
 - **A** peut *redéfinir* certaines méthodes (disons **f()**)
 - Un objet **a** de classe **A** doit pouvoir être considéré comme un objet de classe **B**
 - On doit donc accepter :
 - **B b;**
 - **b=a; (a a toutes les propriétés d'un B)**
 - **b.f()**
 - ✓ Doit appeler la méthode redéfinie dans **A!**
 - ✓ C'est le *transtypage*
 - (exemple: méthode **paint** des interfaces graphiques)

Principes de bases (POO2)

- **Polymorphisme:**
 - Ici l'association entre le nom 'f()' et le code (code de A ou code de B) a lieu dynamiquement (=à l'exécution)
Liaison dynamique
 - On peut aussi vouloir « **paramétriser** » une classe (ou une méthode) par une autre classe.
Exemple: Pile d'entiers

Dans ce cas aussi un nom peut correspondre à plusieurs codes, mais ici l'association peut avoir lieu de façon statique (au moment de la compilation)

Réutilisation du logiciel?

- Type abstrait de données
 - définir le type par ses propriétés (spécification)
- Interface, spécification et implémentation
 - Une interface et une spécification (=les propriétés à assurer) pour définir un type
 - Une (ou plusieurs) implémentation du type abstrait de données
 - Ces implementations doivent vérifier la spécification

Réutilisation du logiciel? (suite)

- Pour l'utilisateur du type abstrait de données
 - Accès uniquement à l'interface (pas d'accès à l'implémentation)
 - Utilisation des propriétés du type abstrait telles que définies dans la spécification.
 - (L'utilisateur est lui-même un type abstrait avec une interface et une spécification)



Réutilisation du logiciel? (fin)

- Mais en utilisant un type *abstrait* l'utilisateur n'en connaît pas l'implémentation
 - il sait uniquement que la spécification du type abstrait est supposée être vérifiée par l'implémentation.
- Pour la réalisation *concrète*, une implémentation particulière est choisie
- Il y a naturellement polymorphisme

Notion de contrat (Eiffel)

- Un *client* et un *vendeur*
- Un *contrat* lie le vendeur et le client (*spécification*)
- Le client ne peut utiliser l'objet que par son *interface*
- La réalisation de l'objet est cachée au client
- Le contrat est conditionné par l'utilisation correcte de l'objet (*pré-condition*)
- Sous réserve de la pré-condition le vendeur s'engage à ce que l'objet vérifie sa spécification (*post-condition*)
- Le vendeur peut déléguer: l'objet délégué doit vérifier au moins le contrat (*héritage*)

Un exemple...

- Pile abstraite et diverses implémentations
 - Pile de brique
 - Pile de papiers
 - Pile d'habits
 - Pile de crêpes
 - Pile de chaises
 - Pile de planchers
 - Etc.

Type abstrait de données

NOM

pile[X]

FONCTIONS

vide : pile[X] -> Boolean

nouvelle : -> pile[X]

empiler : X x pile[X] -> pile[X]

dépiler : pile[X] -> X x pile[X]

PRECONDITIONS

dépiler(s: pile[X]) \Leftrightarrow (not vide(s))

AXIOMES

forall x in X, s in pile[X]

vide(nouvelle())

not vide(empiler(x,s))

dépiler(empiler(x,s))=(x,s)

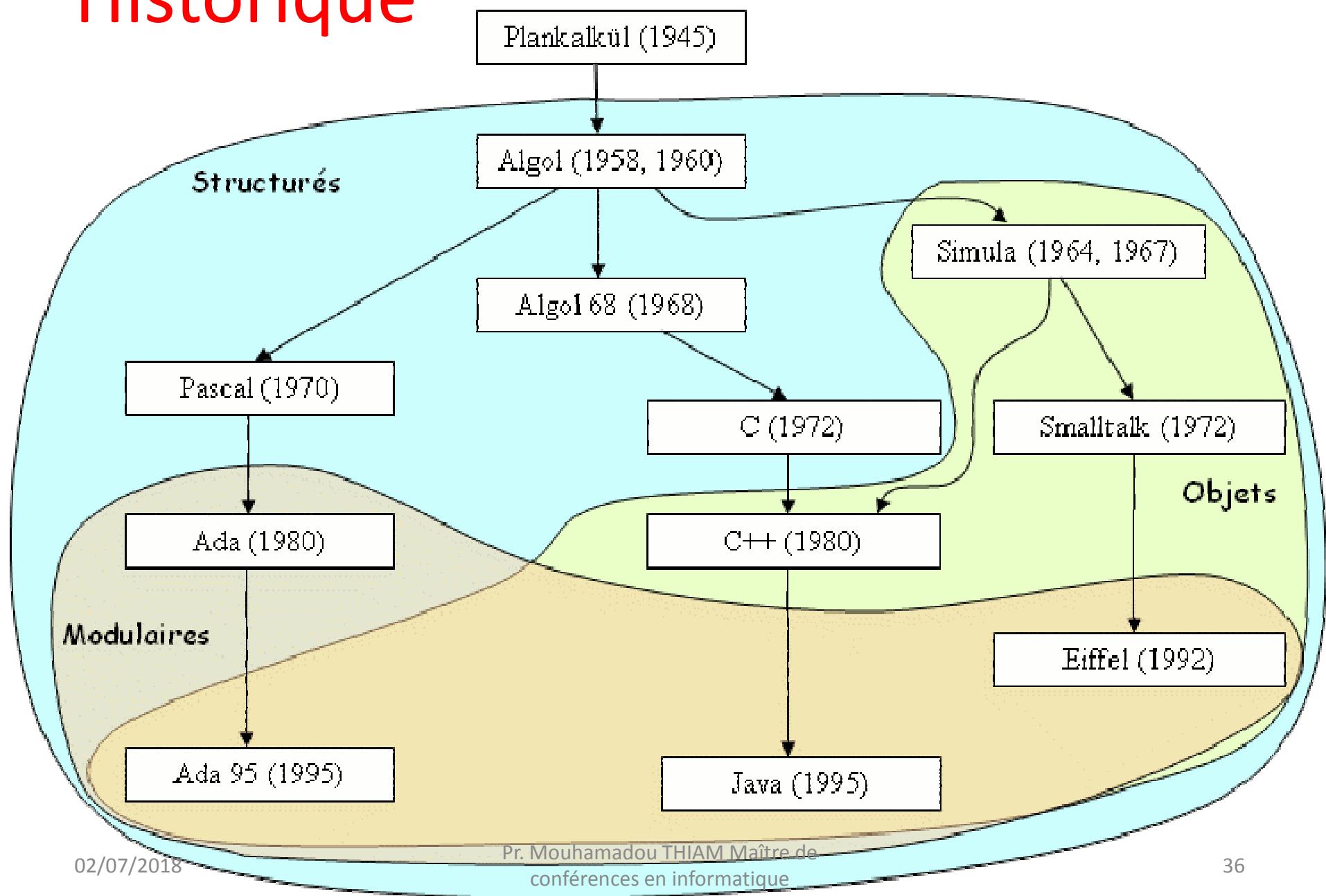
Remarques

- Type paramétré par un autre type (Poo2)
- Axiomes correspondent aux pré conditions
- Pas de représentation
- Vérification que cette définition caractérise bien une pile au sens usuel du terme (c'est possible)

Plan

1. Programmation Orientée Objet
2. **Introduction à JAVA**
3. Techniques de base du langage
4. Types primitifs
5. Structures de contrôle
6. Tableaux
7. Chaînes de caractères

Historique



Généralités...

- Un peu plus qu'un langage de programmation:
 - Créé par James Gosling et Patrick Naughton, Sun Microsystems
 - Présenté officiellement 23 Mai 1995 au SunWord
 - Depuis 2009 Oracle bouffe Sun et maintient Java
 - “gratuit”!
 - Indépendant de la plateforme
 - *Langage interprété et byte code*
 - Portable

Généralités...

- Un peu plus qu'un langage de programmation:
 - Syntaxe à la C
 - Orienté objet (classe, héritage, polymorphisme, etc.)
 - Nombreuses bibliothèques
 - Pas de pointeurs! (ou que des pointeurs!)
 - Ramasse-miettes
 - Multi-thread
 - Distribué (WEB) applet, servlet, etc...
 - url
 - <http://java.sun.com>
 - <http://java.sun.com/docs/books/tutorial/index.html>

Java interprété

- Source
 - compilé en pseudo code ou bytecode
 - exécuté par un interpréteur Java : la Java Virtual Machine (**JVM**).
- Concept base du slogan de Sun pour Java :
 - **WORA** (Write Once, Run Anywhere : écrire une fois, exécuter partout).
 - **Bytecode** un code indépendant à une plate-forme
 - Quasiment mêmes résultats sur toutes les machines disposant d'une **JVM**.

Java portable

- Pas de compilation spécifique par plate forme.
- Code indépendant de la machine sur laquelle il s'exécute.
- Possibilité d'exécuter sur tout environnement qui possède une **JVM**
- Indépendance assurée au niveau du code source grâce à Unicode et au niveau du bytecode.

Java orienté objet

- Fichier source = définition d'une ou plusieurs classes utilisées les unes avec les autres pour former une application.
- Java pas complètement objet car définit types primitifs (entier, caractère, flottant, booléen,...).

Java simple

- Auteurs abandonnent des éléments mal compris ou mal exploités des autres langages
 - notion de pointeurs (éviter incidents en manipulant directement la mémoire),
 - héritage multiple
 - surcharge des opérateurs, ...

Java fortement typé

- Toute variable typée
- Pas de conversion automatique
 - Risque perte de données.
 - Conversion : obligatoirement utiliser un **cast** ou une **méthode statique** fournie en standard

Java gère la mémoire

- Allocation mémoire automatique pour un objet à sa création
- Récupération automatique mémoire inutilisée **garbage collector**
- Restitution zones de mémoire laissées libres suite à la destruction des objets.

Java est sûr

- Sécurité intégrée au système d'exécution et au compilateur.
 - Programme planté menace pas le SE
 - Pas d'accès direct à la mémoire.
 - Accès disque dur règlementé dans une applet.

Java : restrictions des applets

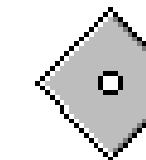
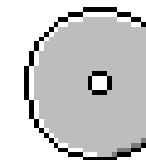
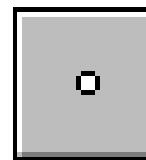
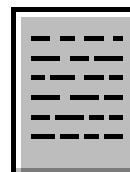
- programme ne peut
 - ouvrir, lire, écrire ou effacer un fichier coté client
 - lancer un autre programme coté client
 - se connecter à d'autres sites Web que celui dont ils proviennent
- fenêtre créée par le programme
 - clairement identifiée comme étant fenêtre Java (interdire création fausse fenêtre demandant mot de passe)

Java est ...

- Économe
 - Pseudo-code de taille relativement petite
 - Bibliothèques de classes requises ne sont liées qu'à l'exécution.
- Multitâche
 - utilisation de threads (unités d'exécutions isolées).
 - JVM utilise plusieurs threads.

Le monde sans Java

Fichier Source

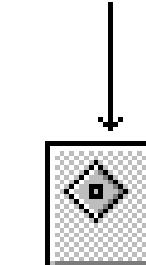
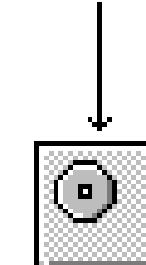
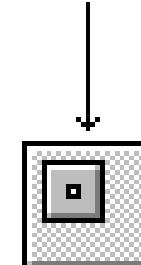


Compilateurs

Intel

PowerPC

Sparc



Binaire
Intel

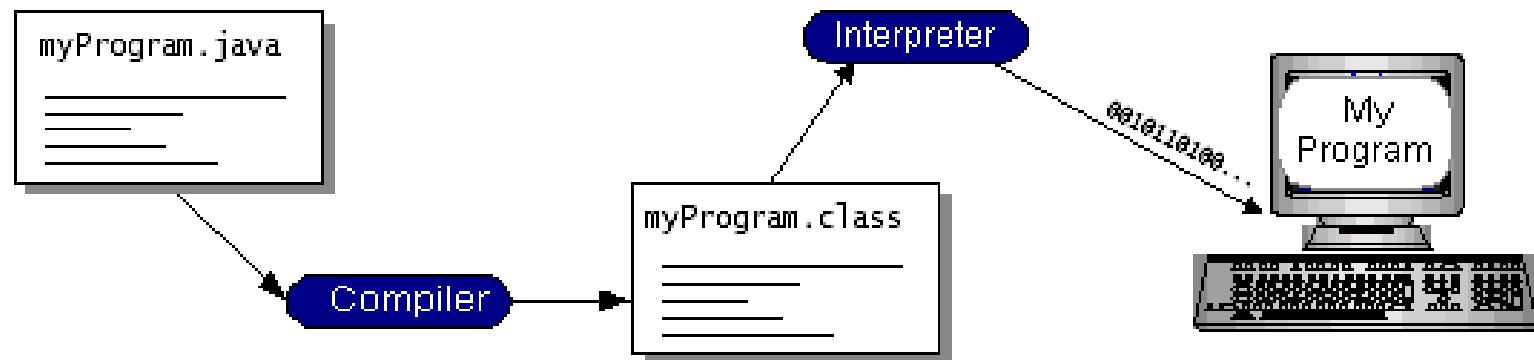
Binaire
Mac

Binaire
Sun

Plateforme Java (1/2)

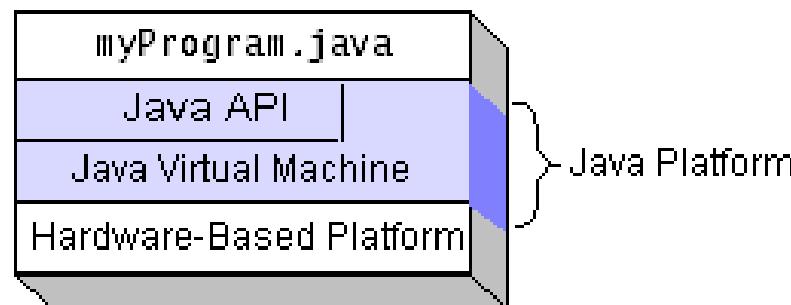
- compilation génère un **.class** en bytecode
- langage intermédiaire indépendant de la plateforme
- bytecode est interprété par un interpréteur Java JVM

Compilation javac
interprétation java



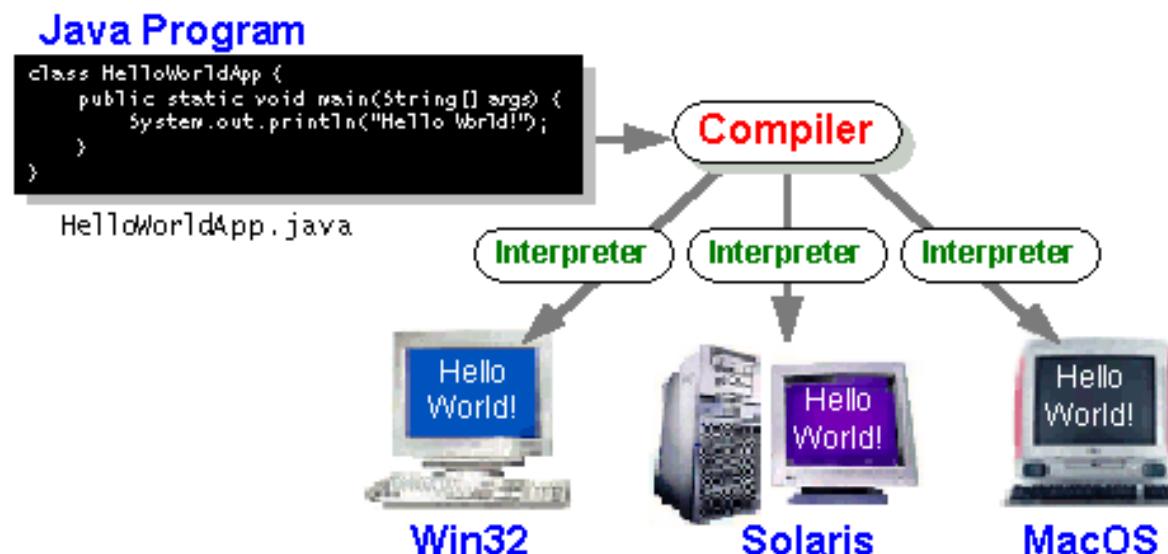
Plateforme Java (2/2)

- La plateforme java: **software** au-dessus d'une **plateforme** exécutable sur un **hardware** (exemple MacOs, linux ...)
- Java VM
- Java **A**pplication **P**rogramming **I**nterface (Java API):



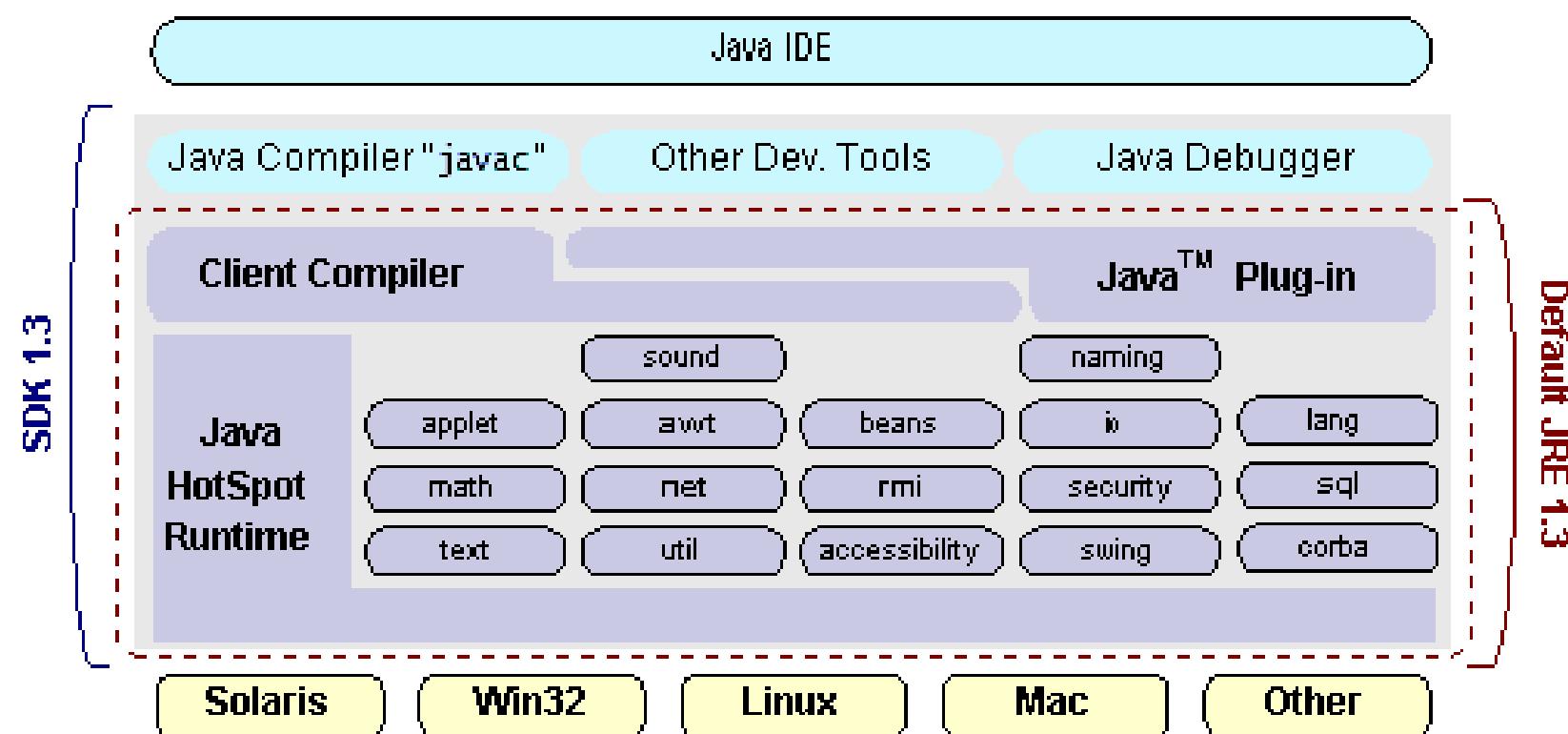
Langage intermédiaire et Interpréteur...

- Avantage: indépendance de la plateforme
 - Échange de byte-code (applet)
- Inconvénient: efficacité



Tout un environnement...

- Java 7 sdk: JRE (Java Runtime Environment + outils de développement: compilateurs, débogueurs, etc...)



Machines virtuelles Java

- Navigateurs Web, Stations de travail, Network Computers
- WebPhones
- Téléphones portables
- Cartes à puces
- ...



Télécharger outils ...

- <http://java.com:80/en/download/manual.jsp>
- <http://www.jcreator.org/download.htm>
- Machines à GAZ
- Eclipse
- Netbeans
- Etc ...

Java évolution

- Java 1.0

Références

- C. Delannoy. – **Programmer en Java (9e édition)**. N°14007, 2014, 940 pages.
- C. Delannoy. – **Exercices en Java (4e édition)**. N°14009, 2014, 360 pages.
- C. Delannoy. – **Programmer en Java (6e édition)**. *Java 5 et 6*. N°13443, 2012, 788 pages (format semi-poche).
- J.-B. Boichat. – **Apprendre Java et C++ en parallèle (4e édition)**. N°12403, 2008, 600 pages + CD-Rom.
- A. Patrício. – **Java Persistence et Hibernate**. N°12259, 2008, 364 pages.
- E. PuyBaret. – **Bien programmer en Java 7**. N°12794, 2012, 428 pages.
- R. Fleury. – **Les Cahiers du programmeur Java/XML**. N°11316, 2004, 218 pages.
- P. Haggar. – **Mieux programmer en Java. 68 astuces pour optimiser son code**. N°9171, 2000, 256 pages.
- J.-P. retaillé. – **Refactoring des applications Java/J2EE**. N°11577, 2005, 390 pages.

Références

- C. Delannoy. – **Programmer en Fortran.** *Fortran 90 et ses évolutions - Fortran 95, 2003 et 2008.*
- C. Delannoy. – **Le guide complet du langage C.** N°14012, 2014, 844 pages.
- C. Delannoy. – **S'initier à la programmation et à l'orienté objet.** *Avec des exemples en C, C++, C#, Python, Java et PHP.* N°14067, 2014, 382 pages.
- G. Dowek *et al.* – **Informatique et sciences du numérique.** *Manuel de spécialité ISN en terminale.* N°13676, 2013, 354 pages.
- G. Dowek *et al.* – **Informatique pour tous en classes préparatoires aux grandes écoles.** *Manuel d'algorithmique et programmation structurée avec Python.* N°13700, 2013, 408 pages.

Plan

1. Programmation Orientée Objet
2. Introduction à JAVA
- 3. Techniques de base du langage**
4. Types primitifs
5. Structures de contrôle
6. Tableaux
7. Chaînes de caractères

TECHNIQUES DE BASE

Java: éléments pratiques ...

- Un source avec le suffixe .java
- Une classe par fichier source (en principe) même nom pour la classe et le fichier source (sans le suffixe .java)
- Méthode

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

– main est le point d'entrée
- Compilation génère un .class
- Exécution en lançant la machine java

2 types de programmes

- Applets
 - Application chargée par un navigateur et exécutée sous le contrôle d'un plug in de ce dernier
 - pas de méthode **main** (appelée par la machine virtuelle pour exécuter une application)
 - peuvent pas être testées avec **l'interpréteur**
 - doivent être testées avec **l'applet viewer** ou intégrées à une page HTML (visualisée par navigateur disposant d'un plug in Java)
- Applications
 - application autonome (*stand alone program*) s'exécute sous le contrôle direct du SE

Algorithme (1)

- Ne faire qu'une chose à la fois
- Ex : Comment faire un café chaud non sucré ?
 - non sucré est aussi important que café et chaud
 - ingrédients et ustensiles nécessaires pas cités dans l'énoncé
 - cafetière électrique ? Faisons des choix?
 - Nous avons : café moulu, filtre, eau, pichet, cafetière électrique, tasse, électricité, table

Algorithme (2)

- Environnement = base de travail établi
- Liste des opérations
 - Verser l'eau dans la cafetière, le café dans la tasse, le café dans le filtre.
 - Remplir le pichet d'eau.
 - Prendre du café moulu, une tasse, de l'eau, une cafetière électrique, un filtre, le pichet de la cafetière.
 - Brancher, allumer ou éteindre la cafetière électrique.
 - Attendre que le café remplisse le pichet.
 - Poser la tasse, la cafetière sur la table, le filtre dans la cafetière, le pichet dans la cafetière.

Algorithme (3)

- Langage minimal pour réaliser un café:
- Verbes
 - Prendre, Poser, Verser, Faire, Attendre, etc.
- Objets
 - Café moulu, Eau, Filtre, Tasse, etc.
- Taille langage dépend de l'environnement
 - Contrat à la SENELEC
 - Planter une graine de café
- **Café sucré: Que faut-il rajouter?**

Algorithme (4)

- Pour boire du café → ordonner la liste
 1. Prendre une cafetière électrique.
 2. Poser la cafetière sur la table.
 3. Prendre un filtre.
 4. Poser le filtre dans la cafetière.
 5. Prendre du café moulu.
 6. Verser le café moulu dans le filtre.
 7. Prendre le pichet de la cafetière.
 8. Remplir le pichet d'eau.

Algorithme (5)

- Pour boire du café → ordonner la liste
 9. Verser l'eau dans la cafetière.
 10. Poser le pichet dans la cafetière.
 11. Brancher la cafetière.
 12. Allumer la cafetière.
 13. Attendre que le café remplisse le pichet
 14. Prendre une tasse
 15. Poser la tasse sur la table.
 16. Éteindre la cafetière.
 17. Prendre le pichet de la cafetière
 18. Verser le café dans la tasse.

Trois exemples de base

- Une application
- Une applet
- Une application avec interface graphique

Application:

- Fichier Appli.java:

```
/**  
 * Une application basique...  
 */  
class Appli {  
    public static void main(String[] args) {  
        System.out.println("Bienvenue en L3 si...");  
        //affichage  
    }  
}
```

Compiler, exécuter...

- Créer un fichier `Appli.java`
- Compilation:
 - `javac Appli.java`
- Création de `Appli.class` (bytecode)
- Interpréter le byte code:
 - `java Appli`
- Attention aux suffixes!!!
 - (il faut que `javac` et `java` soient dans `$PATH`)
`Exception in thread "main" java.lang.NoClassDefFoundError:`
 - Il ne trouve pas le main -> vérifier le nom!
 - Variable `CLASSPATH` ou option `-classpath`

Remarques

- Commentaires /* ... */ et //
- Définition de classe
 - une classe contient des méthodes (=fonctions) et des variables
 - Pas de fonctions ou de variables globales (uniquement dans des classes ou des instances)
- Méthode main:
 - public static void main(String[] arg)
 - public
 - static
 - Void
 - String
 - Point d'entrée

Remarques

- Classe **System**
 - **out** est une variable de la classe *System*
 - **println** méthode de *System.out*
 - **out** est une variable de classe qui fait référence à une instance de la classe *PrintStream* qui implémente un flot de sortie.
 - Cette instance a une méthode **println**

Remarques...

- Classe: définit des méthodes et des variables (déclaration)
- Instance d'une classe (objet)
 - Méthode de classe: fonction associée à (toute la) classe.
 - Méthode d'instance: fonction associée à une instance particulière.
 - Variable de classe: associée à une classe (globale et partagée par toutes les instances)
 - Variable d'instance: associée à un objet (instancié)
- Patience...

Applet:

- Applet et WEB
 - Client (navigateur) et serveur WEB
 - Le client fait des requêtes html, le serveur répond par des pages html
 - Applet:
 - Le serveur répond par une page contenant des applets
 - Applet: byte code
 - Code exécuté par le client
 - Permet de faire des animations avec interfaces graphiques sur le client.
 - Une des causes du succès de java

Exemple applet

- Fichier **MonApplet.java**:

```
/**  
 * Une applet basique...  
 */  
import java.applet.Applet;  
import java.awt.Graphics;  
public class MonApplet extends Applet {  
    public void paint(Graphics g){  
        g.drawString("Bienvenue en en L3 SI...", 50, 25);  
    }  
}
```

Remarques:

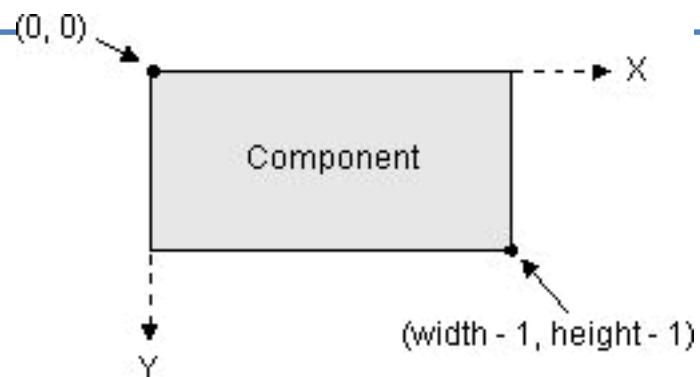
- import et package:
 - Un package est un regroupement de classes.
 - Toute classe est dans un package
 - Package par défaut (sans nom)
 - classpath
- import **java.applet.*;**
 - Importe le package java.applet
 - Applet est une classe de ce package,
 - Sans importation il faudrait `java.applet.Applet`

Remarques:

- La classe **Applet** contient ce qu'il faut pour écrire une applet
- ... **extends Applet**:
 - La classe définie est une extension de la classe **Applet**:
 - Elle contient tout ce que contient la classe **Applet**
 - (et peut redéfinir certaines méthodes (paint))
 - Patience!!

Remarques...

- Une Applet contient les méthodes `paintstart` et `init`. En redéfinissant `paint`, l'applet une fois lancée exécutera ce code redéfini.
- `Graphics g` argument de `paint` est un objet qui représente le contexte graphique de l'applet.
 - `drawString` est une méthode (d'instance) qui affiche une chaîne,
 - 50, 25: affichage à partir de la position (x,y) à partir du point (0,0) coin en haut à gauche de l'applet.



Pour exécuter l'applet

- L'applet doit être exécutée dans un navigateur capable d'interpréter du bytecode correspondant à des applet.
- Il faut créer un fichier HTML pour le navigateur.

Html pour l'applet

- Fichier Bienvenu.html:

```
<HTML>
<HEAD>
<TITLE>Une petite applet </TITLE>
<BODY>
<APPLET CODE='MonApplet.class' WIDTH=200
 Height=50>
</APPLET>
</BODY>
</HTML>
```

Html

- Structure avec balises:
- Exemples:
 - <HTML></HTML>
 - url:
 - page de hf
- Ici:

```
<APPLET CODE='MonApplet.class' WIDTH=200  
Height=50>  
</APPLET>
```

Exemple interface graphique (POO2)

Fichier MonSwing.java:

```
/**  
 * Une application basique... avec interface graphique  
 */  
import javax.swing.*;  
public class MonSwing {  
    private static void creerFrame() {  
        //Une formule magique...  
        JFrame.setDefaultLookAndFeelDecorated(true);  
        //Creation d'une Frame  
        JFrame frame = new JFrame("MonSwing");  
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);  
        //Afficher un message  
        JLabel label = new JLabel("Bienvenue en L3...");  
        frame.getContentPane().add(label);  
        //Afficher la fenêtre  
        frame.pack();  
        frame.setVisible(true);  
    }  
    public static void main(String[] args) {  
        creerFrame();  
    }  
}
```



Remarques

- Importation de packages
- Définition d'un conteneur top-level **JFrame**, implémenté comme instance de la classe **JFrame**
- Affichage de ce conteneur
- Définition d'un composant **JLabel**, implémenté comme instance de **JLabel**
- Ajout du composant **JLabel** dans la **JFrame**
- Définition du comportement de la **JFrame** sur un click du bouton de fermeture
- Une méthode **main** qui crée la **JFrame**

Pour finir...

- Java 1.5 (5) et 1.6 (6) annotations, types méthodes paramétrés par des types
- Java 1.7 (7) diffusé juillet 2011
- Java 1.8 (8) diffusé en mars 2014
- Très nombreux packages
- Nombreux outils de développement (gratuits)
 - eclipse, netbeans..

Plan

1. Programmation Orientée Objet
2. Introduction à JAVA
3. Techniques de base du langage
- 4. Types primitifs**
5. Structures de contrôle
6. Tableaux
7. Chaînes de caractères

TYPES PRIMITIFS

Types primitifs

- Comme la plupart des langages récents, Java est orienté objet.
- Chaque fichier source contient la définition d'une ou plusieurs classes
- Utilisées les unes avec les autres pour former une application.
- Java pas complètement objet
- types primitifs (entier, caractère, flottant, booléen,...).

Booléen : boolean

- C'est l'ensemble des deux valeurs
 - true
 - false

Caractère : char

- Ensemble des valeurs unicode
- De '\u0000' à '\uffff' avec 4 chiffres obligatoires après '\u')
- Les 128 premiers caractères sont les codes ASCII et se notent entre apostrophes
- Exemple : 'a', '1', '\'', '\n'.

Nombres Entiers (1/2)

- Se distinguent par la taille de leur représentation.
 - byte : (1 octet) - entre **-128** et **+127** (-2^7 et 2^7-1)
 - short : (2 octets) - entre **-32768** et **+32767** (-2^{15} et $2^{15}-1$)
 - int : (4 octets) entre **-2147483648** et **+2147483647** (-2^{31} et $2^{31}-1$)
 - long : (8 octets) entre **-9223372036854775808** et **+9223372036854775807** (-2^{63} et $2^{63}-1$)

Nombres Entiers (2/2)

- Codage en binaire pour ≥ 0 et en binaire en complément à deux pour ≤ 0 :
- 9 est codé sur un octet en 00001001
- -9 est codé sur un octet en 11110111
- +1 est codé sur un octet en 00000001
- -1 est codé sur un octet en 11111111
- -128 est codé sur un octet en 10000000
- -128 est codé sur un octet en 01111110
- 127 est codé sur un octet en 01111111

Types flottants (1/2)

- Se distinguent par
 - taille de leur représentation et
 - précision et l'étendue des valeurs.
- Représentation en virgule flottante :
 - signe, exposant et significande (norme IEEE754)
 - signe : 0 pour ≥ 0 et 1 pour ≤ 0
 - exposant : un nombre entier
 - significande : un nombre entier

Types flottants (2/2)

Type de nombre	S	Exposant	Significande
Float	1	8	23
Double	1	11	52

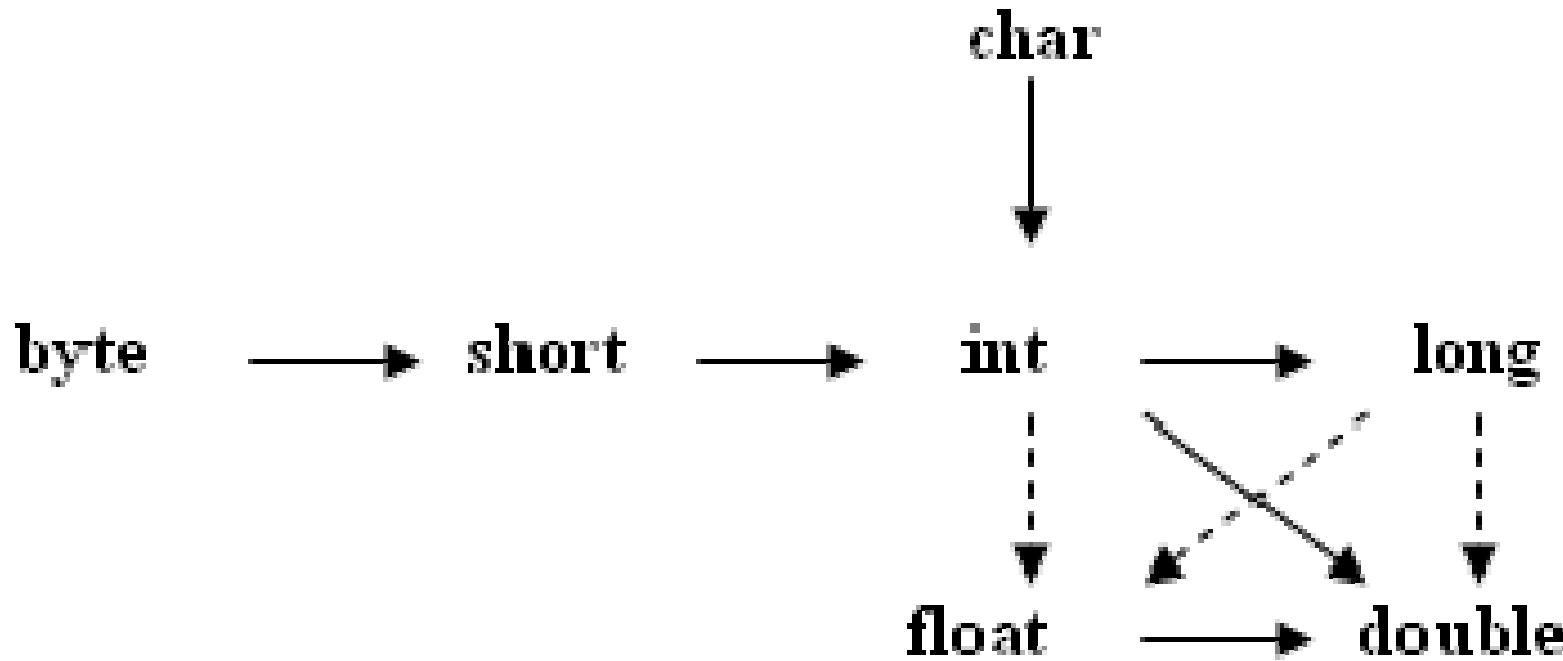
- Représentation
 - entier.{entier}{E{+, -}entier} ou
 - .entier{E{+, -}entier}
 - {} facultatif.
 - Par défaut un littéral flottant est de type double.
 - suffixant avec fou F → Float
 - suffixant avec dou D → Double

Exemples

- float A=3.1F,B=2.3F,C=1.5F;
- float X,Y;
- X=(A*B)*C;
- Y=A*(B*C);
- System.out.println(X);
- System.out.println(Y);
- System.out.println();
- System.out.println(4096.1F-4095.1F);
- System.out.println(4096.1-4095.1);

- les sorties console sont les suivantes :
- 10.695
- 10.694999
- 1.0
- 1.000000000004547

conversion



Les autres conversions peuvent entraîner une perte d'information, et doivent être spécifiées par un opérateur de cast.

Exemples : **double d = 12345.6;**
 byte b = (byte)d; // b vaut 57

Variables constantes

- variable doit être déclarée.
- déclaration = type suivi du nom variable.
- nom suivi d'1 initialisation (= valeur).
- Déclaration constante comme de variable précédée du mot **final**.
- nom variable ou constante constitué de
- lettres, chiffres, '_', et symbol monaie ('\$', '£', '€', etc...)
- peut pas commencer par un chiffre.
- Character.isJavaIdentifierStart(Charc)
- Character.isJavaIdentifierPart(Charc)

Mots réservés

- | | | | | |
|-------------------|-------------------|---------------|--------------------------|---------------------|
| • abstract | continue | for | new | switch |
| • assert | default | goto | package | synchronized |
| • boolean | implements | if | private | this |
| • Break | protected | do | double | throw |
| • byte | import | else | public | throws |
| • case | instanceof | enum | return | transient |
| • catch | extendsint | short | | try |
| • char | interface | static | void | final |
| • class | finally | long | strictfp volatile | |
| • Const | native | float | super | while |

Best practices

- Par convention les
 - noms de variable,
 - attributs,
 - paramètres
- commencent par une minuscule,
 - les nom de classe,
 - d'interface et
 - d'énumération
- commencent par une majuscule.
- Identificateur composé de plusieurs mots, tous les mots sauf le premier commencent par une majuscule.

Expressions

- Expressions construites à partir de
 - littéraux,
 - variables,
 - constantes,
 - opérateurs
 - et parenthèses.
- Toute expression retourne une valeur.
- Les expressions (sauf **&&** et **||**) sont évaluées en
 - Évaluant d'abord l'opérande de gauche,
 - Puis l'opérande de droite,
 - puis en réalisant l'opération.

Exemples

- `int i = 0 ;`
- `t[++i]=++i ; // affectation de 2 à t[1]`
- Certaines instructions sont des instructions expressions et peuvent se trouver partout où on peut trouver une instruction :
 - Affectation
 - pré et post incrément ou décrément

Opérateurs (1)

Opérateur	Sémantique	Priorité	associativité
[] . () appel de méthode	Sélection dans un tableau, sélection d'un attribut ou méthode de classe, et paramètre d'un appel de méthode	0	gauche vers droite
++ --	Pré ou post incrémentation ou décrémentation	1	Droite
+ - unaire		1	Droite
!	Négation booléenne	1	Droite
~	Négation bit à bit	1	Droite
() cast	Transtypage	1	Droite
	Décalage à droite, le remplissage se fait avec de 0. $32 >> 3 = 4$ ET $-32 >>> 3 = 536870908$		

Opérateurs (2)

Opérateur	Sémantique	Priorité	Associativité
*	Multiplication	2	Gauche
/	Division : division réelle si un des deux opérandes est un réel, et quotient de la division si les deux opérandes sont des entiers. Une division par 0 lève une exception.	2	Gauche
%	Reste de la division. (opérandes entiers ou réels)	2	Gauche
+ - binaire	addition et soustraction.	3	Gauche
<< >>	Décalage à gauche signé, complété par des 0, décalage à droite signé, le remplissage dépend du signe. $32 >> 3 = 4$ ET $-32 >> 3 = -4$	4	Gauche
>>>	Décalage à droite, le remplissage se fait avec 0. $32 >> 3 = 4$ ET $-32 >>> 3 = 536870908$		

Opérateurs (3)

Opérateur	Sémantique	Priorité	associativité
<code><<= >>=</code>	Opérateurs de comparaison.	5	Gauche
<code>== !=</code>	Opérateurs de comparaison.	6	Gauche
<code>&</code>	<ul style="list-style-type: none">•Et bit à bit pour les opérandes entiers.•Et évaluant les 2 opérandes pour des opérandes booléens.	7	Gauche
<code>^</code>	<ul style="list-style-type: none">•Ou exclusif bit à bit pour des opérandes entiers.•Ou évaluant les 2 opérandes pour des opérandes booléens.	8	Gauche
<code> </code>	<ul style="list-style-type: none">•Ou inclusif bit à bit pour des opérande sentiers.•Ou inclusif évaluant les 2 opérandes pour des opérandes booléens.	9	Gauche

Opérateurs (4)

Opérateur	Sémantique	Priorité	associativité
&&	Et logique. le premier opérande est évalué : s'il vaut false la valeur de l'expression est false, sinon c'est la valeur du second opérande.	10	gauche
	Ou logique. le premier opérande est évalué : s'il vaut true la valeur de l'expression est true, sinon c'est la valeur du second opérande.	11	Gauche
? :	<p>Opérateur ternaire a ? b : c</p> <p>a est évalué</p> <ul style="list-style-type: none"> •a vaut true la valeur de l'expression est la valeur de b •a vaut false la valeur de l'expression est la valeur de c •b est une expression quelconque, mais c ne peut être une affectation que si elle est entre parenthèses. 	12	Gauche

Opérateurs (5)

Opérateur	Sémantique	Priorité	associativité
= += -= &= *=/= %=<<=>>=>>>= & = ^=	Opérateurs d'affectation : la partie gauche doit être une variable. Les opérateurs composés ont le sens suivant : $a \text{ op} = b$ est équivalent à $a = (T) (a \text{ op} (b))$ où T est le type de a . L'expression b est évaluée avant d'évaluer $a \text{ op} b$; String s; $s += 'a'+'b';$ est équivalent à $s = s + ('a' + 'b')$.	13	Droite vers gauche
instanceof	a pour valeur true si l'opérande de gauche a pour type l'opérande de droite et false sinon.	14	Gauche
cast		1	Gauche
new		1	Gauche

Remarque

- Les opérations sur les nombres réels ne lèvent pas d'exceptions, mais produisent des valeurs NaN, POSITIVE_INFINITY ou NEGATIVE_INFINITY
- `float a = 1000f;`
- `System.out.println(a/0); // affichage de Infinity`
- `a = a*a*a*a;`
- `System.out.println(a); // affichage de 1.0E12`
- `a = a*a*a*a;`
- `System.out.println(a); // affichage de Infinity`
- `System.out.println(a/a); // affichage de NaN`

STRUCTURES DE CONTROLE

La structure de bloc.

Pseudo-code

```
debut  
instruction(s)  
fin
```

Java

```
{  
instruction(s);  
}
```

La structure d'alternative (sialorsfsi)

Pseudo-code

```
si condition alors  
instruction(s);  
fsi
```

Java

```
if (condition) {  
instruction(s);  
}
```

La structure d'alternative (sialorssinonfsi)

Pseudo-code

```
si condition alors
    instruction(s)_1
sinon
    instruction(s)_2
fsi
```

Java

```
if (condition) {
    instruction(s)_1;
} else {
    instruction(s)_2;
}
```

La structure d'alternative (sialorssinonsisinonfsi)

Pseudo-code

```
si condition_1 alors
    instruction(s)_1
Sinon si condition_2 alors
    instruction(s)_2
sinon
    instruction(s)_3
fsi
```

Java

```
if (condition_1) {
    instruction(s)_1;
} else if (condition_2) {
    instruction(s)_2;
} else {
    instruction(s)_3;
}
```

La structure d'alternative (si imbriqué)

Pseudo-code

```
si condition_1 alors
    instruction(s)_1
sinon
    si condition_2 alors
        instruction(s)_2
    sinon
        instruction(s)_3
fsi
fsi
```

Java

```
if (condition_1) {
    instruction(s)_1;
} else {
    if (condition_2) {
        instruction(s)_2;
    } else {
        instruction(s)_3;
    }
}
```

La structure de répétitive (le tantque)

Pseudo-code

```
tq condition_1  
instruction(s)  
ftq
```

Java

```
while (condition) {  
    instruction(s);  
}
```

La structure de répétitive (le faire tantque)

Pseudo-code

```
faire  
instruction(s)  
tq condition
```

Java

```
do {  
instruction(s)  
} while (condition)
```

La boucle Pour (pas de +1)

Pseudo-code

```
pour i de d à f faire  
instruction(s)  
fpour
```

Java

```
for (int i = d ; i<= f ; i++) {  
instruction(s);  
}
```

La boucle Pour (pas de -1)

Pseudo-code

```
pour i de d à f faire  
instruction(s)  
fpour
```

Java

```
for (int i = d ; i<= f ; i --) {  
instruction(s);  
}
```

La boucle Pour (pas de +p)

Pseudo-code

```
pour i de d à f pas p faire  
instruction(s)  
fpour
```

Java

```
for (int i = d ; i<= f ; i +=p) {  
instruction(s);  
}
```

La boucle Pour (pas de -p)

Pseudo-code

```
pour i de d à f pas p faire  
instruction(s)  
fpour
```

Java

```
for (int i = d ; i<= f ; i -=p) {  
instruction(s);  
}
```

Le selon (ou cas – Extension du si alors sinon fsi)

Pseudo-code

```
selon variable  
cas valeur_1  
    instruction(s)_1  
cas valeur_2  
    instruction(s)_2  
...  
cassinon  
instruction(s)_sinon  
fselon
```

Java

```
switch (variable) {  
    case valeur_1  
        instruction(s)_1;  
        break;  
    case valeur_2  
        instruction(s)_2;  
        break;  
    ...  
    default :  
        instruction(s)_sinon  
}
```

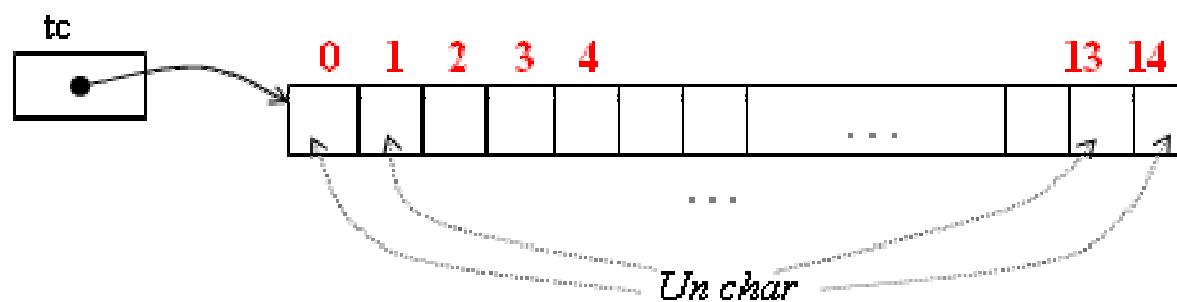
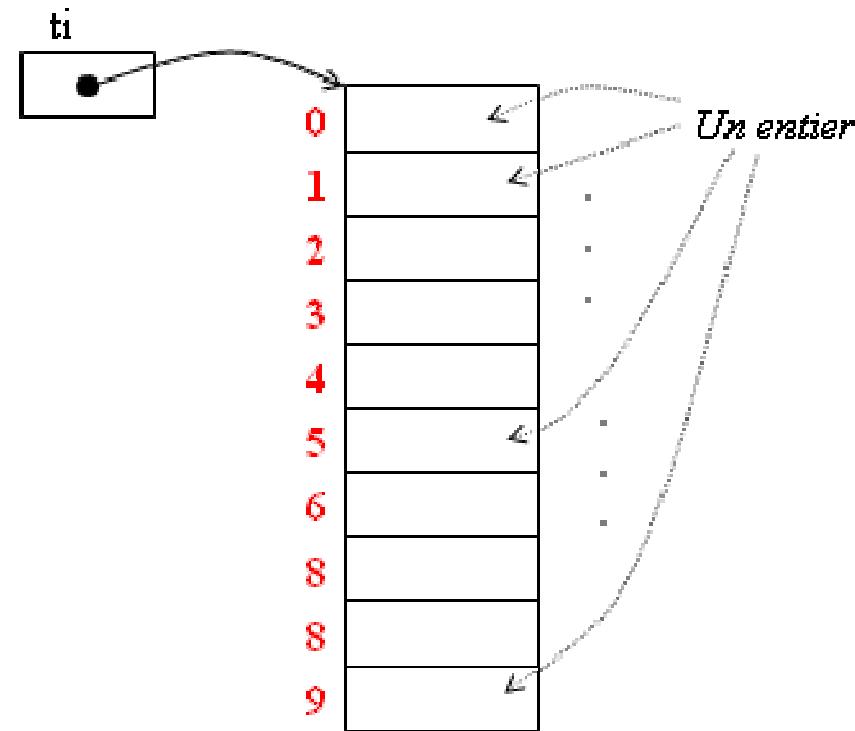
TABLEAUX

Définition

- Tableau = structure de données contenant un **groupe** d'éléments **tous** du **même type**.
- Type des éléments peut être un type **primitif** ou une **classe**.
- Lors de la définition d'un tableau les **[]** spécifiant qu'il s'agit d'un tableau peuvent être placés **avant** ou **après** le nom du tableau.

Exemples

- // ti est un tableau à une dimension de int
 - `int ti [];`
- // tc est un tableau à une dimension de char
 - `char[] tc;`
- Un tableau peut être initialisé :
 - `int ti [] = { 1, 2, 3 , 4};`
 - `char[] tc = {'a', 'b', 'c'};`



Allouer l'espace au tableau

- Il faut utiliser **new** :
 - // ti est un tableau à une dimension de 10 int
 - **ti = new int [10];**
 - // tc est un tableau à une dimension de 15 char
 - **tc = new char [15];**
- L'utilisation d'un tableau pour lequel l'espace n'a pas été alloué provoque la levée d'une exception **NullPointerException**

Accès aux éléments

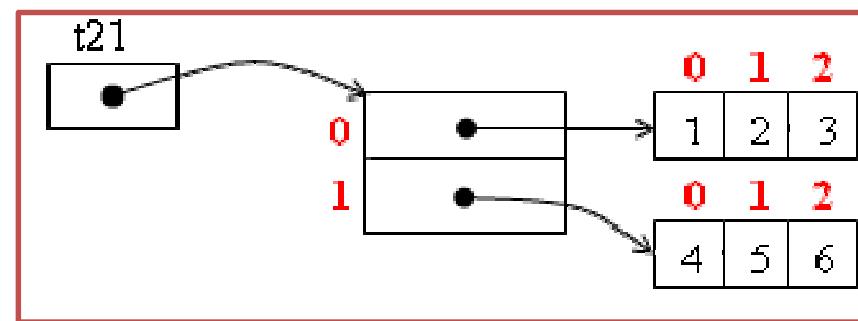
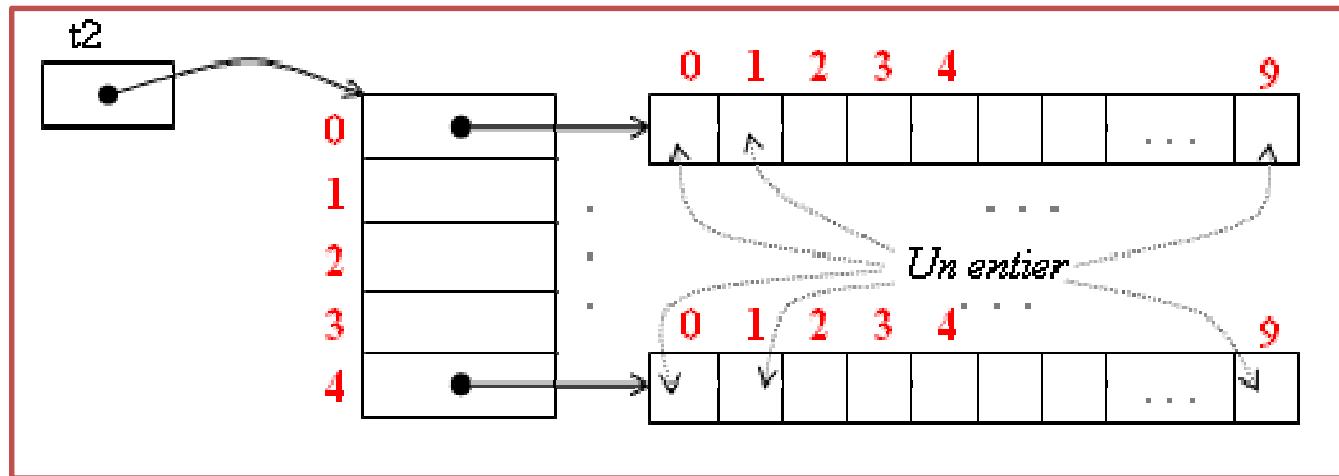
- Les éléments d'un tableau sont indicés à partir de 0.
- Chaque élément peut être accédé individuellement en donnant le nom du tableau suivi de l'indice entre []
 - $ti[0]$;// *premier élément du tableau ti*
 - $ti[9]$;// *dernier élément du tableau ti*

Taille de tableau

- L'accès à un élément du tableau en dehors des bornes provoque la levée d'une exception **ArrayIndexOutOfBoundsException**
- Un tableau possède un attribut *length* qui permet de connaître le nombre d'éléments d'un tableau.
- `ti.length` vaut 10
- `tc.length` vaut 15

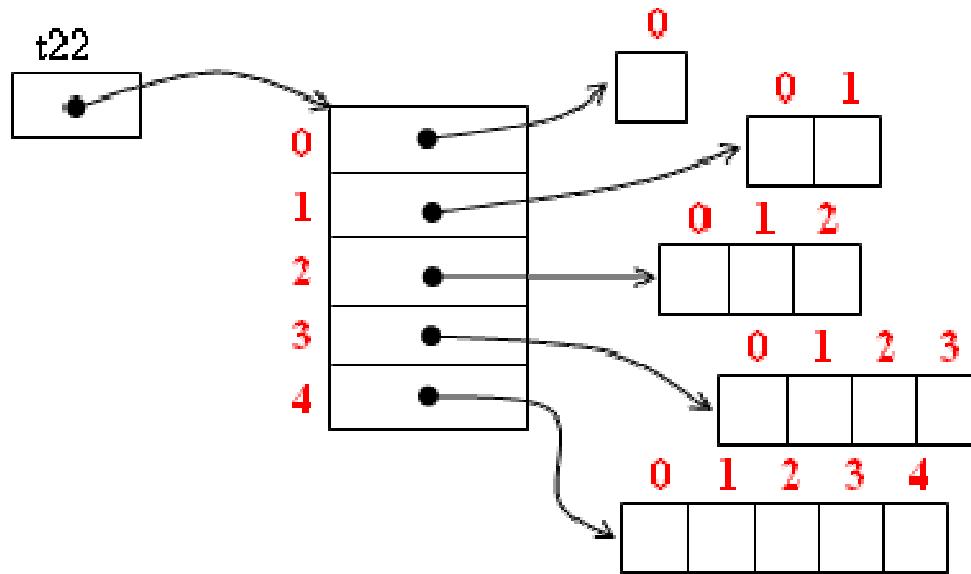
Plusieurs dimensions

- Lors de sa définition nombre de crochets = nombre de dimensions.
 - `int t2[][] = new int[5][10];`
- définit un tableau à 2 dimensions 5 lignes sur 10 colonnes (ou l'inverse).
- `t2[i]` désigne le `ième` tableau à une dimension d'entiers.
- Un tableau à plusieurs dimensions peut être initialisé :
 - `int t21[][] = { {1, 2, 3}, {4, 5, 6} };`
- définit un tableau dont la première dimension va de l'indice 0 à l'indice 1 et la deuxième dimension de l'indice 0 à l'indice 2.



Remarques

- Lignes d'un tableau à 2 dimensions n'ont pas forcément le même nombre d'éléments :
 - `int t22[][] ;`
 - `t22 = new int[5][];`
 - `for(int i = 0; i < t22.length; ++i){`
`t22[i] = new int [i+1];`
`}`
 - `for(int i = 0; i < t22.length; ++i){`
`for(int j = 0; j < t22[i].length; ++j){`
`//accès à t22[i][j]`
`}`
`}`



En paramètre pas de dimensions dans la spécification du tableau

Méthodes sur les tableaux

- static Object get(Object t, int index)
- static xxx getXxx(Object t, int index)
- static void set(Object t, int index, Object valeur)
- static void setXxx(Object t, int index, xxx z)
- Les méthodes précédentes peuvent lever les exceptions suivantes :
 - **NullPointerException** si l'objet au rang index vaut null.
 - **IllegalArgumentException** si le premier argument n'est pas un tableau.
 - **ArrayIndexOutOfBoundsException** si index est en dehors des bornes du tableau

Algorithme de calcul

- $T=[12, 4, 0, -4, 6, 19]$
 1. Max=12, min=12, som=12;
 2. Max=12, min=4, som=16
 3. Max=12, min=0, som=16
 4. Max=12, min=-4, som=12
 5. Max=12, min=-4, som=18
 6. Max=19, min=-4, som=37

CHAINES DE CARACTERES

Chaîne de caractères

- Trois classes
 - String, StringBuffer et StringBuilder.
 - String → objets immuables : ils ne peuvent pas changer de valeur.
 - Les classes StringBuffer et StringBuilder → objets qui peuvent changer de valeur.
 - Deux classes avec exactement les mêmes méthodes.
 - Les méthodes de la classe StringBuffer sont synchronisées, et peuvent être utilisées par plusieurs thread sur une même chaîne de caractères.
 - Les méthodes de la classe StringBuilder ne sont pas synchronisées, elles sont donc plus rapides que celle de la classe StringBuffer, mais ne sont pas «thread safe».

La classe String

- chaînes de caractères littérales de Java (exemple "abc") sont instances de cette classe.
- **String** est une classe spéciale :
 - chaînes de caractères peuvent se concaténer à l'aide de l'opérateur +, ou à l'aide de la méthode concat. Ces deux concaténations ne sont pas équivalentes lorsque l'opérande de droite vaut null.
 - les instances peuvent ne pas être créées explicitement **String s = "abc" ;** au lieu de **String s = new String("abc") ;**

String : constructeurs

String()	Construit la chaîne vide
String (byte[] bytes)	Construit une chaîne de caractères à partir d'un tableau d'octets
String (byte[] bytes, int offset, int length)	Construit une chaîne de caractères à partir d'une partie de tableau d'octets
String (byte[] bytes, int offset, int length, String enc)	Construit une chaîne de caractères à partir d'une partie de tableau d'octets, et d'un encodage
String (byte[] bytes, String enc)	Construit une chaîne de caractères à partir d'un tableau d'octets, et d'un encodage
String(char[] value)	Construit une chaîne de caractères à partir d'un tableau de caractères
String (char[] value, int offset, int count)	Construit une chaîne de caractères à partir d'une partie de tableau de caractères
String (String value)	Construit une chaîne à partir d'une autre chaîne.
String (StringBuffer buffer)	Construit une chaîne à partir d'une autre chaîne de type StringBuffer .

String : méthodes(1)

- concaténation → concat(String s).
- length() → longueur (nombre de caractères) de la chaîne.
- Plein d'autres méthodes
- Comparaison
 - int compareTo(Object o) → <0; ==0;>0
 - int compareTo(String anotherString)
 - int compareIgnoreCase(String str)

String : méthodes (2)

- Comparaison
 - boolean equals(Object anObject)
 - boolean equalsIgnoreCase(Object anObject)
 - boolean regionMatches(boolean ignoreCase, int toffset, String other, int ooffset, int len)
 - boolean regionMatches(int toffset, String other, int ooffset, int len)

String : méthodes (3)

- Caractère et sous-chaîne
 - char charAt(int i)
 - String substring(int d)
 - String substring(int d, int f)
 - boolean startsWith(String prefix)
 - boolean startsWith(String prefix, int i)
 - boolean endsWith(String suffix)

String : méthodes (4)

- Conversion
 - String toLowerCase()
 - String toLowerCase(Locale locale)
 - String toString()
 - String toUpperCase()
 - String toUpperCase(Locale locale)
 - String trim()
 - String replace(char ac, char nc)

String : méthodes (5)

- Recherche
 - int indexOf(int ch)
 - int indexOf(int ch, int fromIndex)
 - int indexOf(String str)
 - int indexOf(String str, int fromIndex)
 - int lastIndexOf(int ch)
 - int lastIndexOf(int ch, int fromIndex)
 - int lastIndexOf(String str)
 - int lastIndexOf(String str, int fromIndex)

LECTURE DE NOMBRES ET AUTRES

Class Scanner (1)

- classe injustement méconnue du JDK est Scanner (depuis la version 5 de Java)
- fonctionnalités très intéressantes pour parser des chaînes de caractères, et en extraire et convertir les composants.
- Scanner peut se brancher sur à peu près n'importe quelle source : InputStream, Readable (et donc Reader), File... et bien sûr une simple String.

Class Scanner (2)

- Ensuite utiliser les méthodes de type
 - hasNext...()
 - next...(),
- ou alors les méthodes de type
 - find...()
 - match()
 - group().

```
Scanner sc = new Scanner(System.in);  
int i = sc.nextInt();
```

Class Scanner (3)

- Méthode `hasNext()` / `next()`
 1. découper la chaîne de caractères en *tokens* grâce à un délimiteur ; il s'agit par défaut d'un caractère "blanc" (espace, tabulation, retour à la ligne...), mais il est évidemment possible de fournir sa propre expression via la méthode **`useDelimiter(expression)`**.
 2. utiliser les méthodes de type `hasNext...()` et `next...()` pour parcourir, récupérer et convertir ces tokens.
 3. Les méthodes de type `hasNext...()` (`hasNextInt()`, `hasNextFloat(...)`) fonctionnent sur le même principe qu'un `Iterator`.

Class Scanner(4)

- A l'aide de ces méthodes, il est très facile de parser une chaîne dont vous maîtrisez parfaitement le format, par exemple un fichier .csv :

```
String s =  
"Dalton;Joe;1.4\n" +  
"Dalton;Jack;1.6\n" +  
"Dalton;William;1.8\n" +  
"Dalton;Averell;2.0";  
  
Scanner scan = new Scanner(s);  
scan.useDelimiter(";" + "\n");  
scan.useLocale(Locale.US); // Pour les floats  
while(scan.hasNextLine()) {  
    System.out.printf("%2$s %1$s : %3$.1f m \n",  
        scan.next(), scan.next(), scan.nextFloat());  
}
```

ENCAPSULATION & CONTRÔLE D'ACCÈS

Encapsulation (1)

- Notion d'encapsulation :
 - données et méthodes de manipulation regroupées dans une même entité, l'objet.
 - détails d'implémentation cachés, accès aux données de extérieur par l'intermédiaire d'un **interface** de l'objet (ensemble d'opérations)
 - programmeur pas se soucier de la représentation physique des entités utilisées et peut raisonner en termes d'abstractions.

Encapsulation (2)

- Programmation dirigée par les données :
 - pour traiter une application, le programmeur commence par définir les classes d'objets appropriées, avec leurs opérations spécifiques.
 - chaque entité manipulée dans le programme est un représentant (ou instance) d'une de ces classes.
 - L'univers de l'application est par conséquent composé d'un ensemble d'objets qui détiennent, chacun pour sa part, les clés de leur comportement.

Contrôle d'accès (1)

- Attribut et méthode d'une classe peut être :
 - visible depuis les instances de toutes les classes d'une application. En d'autres termes, son nom peut être utilisé dans l'écriture d'une méthode de ces classes. Il est alors **public**.
 - visible uniquement depuis les instances de sa classe. En d'autres termes, son nom peut être utilisé uniquement dans l'écriture d'une méthode de sa classe. Il est alors **privé**.
- Les mots réservés sont :
 - **public**
 - **private**

Contrôle d'accès (2)

- En toute rigueur, il faudrait toujours que :
 - les attributs ne soient pas visibles,
 - Les attributs ne devraient pouvoir être lus ou modifiés que par l'intermédiaire de méthodes prenant en charge les vérifications et effets de bord éventuels.
 - les méthodes "utilitaires" ne soient pas visibles,
 - seules les fonctionnalités de l'objet, destinées à être utilisées par d'autres objets soient visibles.
 - C'est la notion d'encapsulation
- Nous verrons dans la suite que l'on peut encore affiner le contrôle d'accès

Contrôle d'accès (3)

```
public class Parallelogramme
{
    private int longueur = 0; // déclaration + initialisation explicite
    private int largeur = 0; // déclaration + initialisation explicite
    public int profondeur = 0; // déclaration + initialisation explicite
    public void affiche ( )
    {System.out.println("Longueur= " + longueur + " Largeur = " + largeur +
                       " Profondeur = " + profondeur);
}
```

```
public class ProgPpal
{
    public static void main(String args[])
    {
        Parallelogramme p1 = new Parallelogramme();
        p1.longueur = 5; // Invalide car l'attribut est privé
        p1.profondeur = 4; // OK
        p1.affiche( ); // OK
    }
}
```

Variables de classe (1)

- Il peut s'avérer nécessaire de définir un attribut dont la valeur soit partagée par toutes les instances d'une classe. On parle de variable de classe.
- Ces variables sont, de plus, stockées une seule fois, pour toutes les instances d'une classe.
- Mot réservé : **static**
- Accès :
 - depuis une méthode de la classe comme pour tout autre attribut,
 - via une instance de la classe,
 - à l'aide du nom de la classe.

Variables de classe (2)

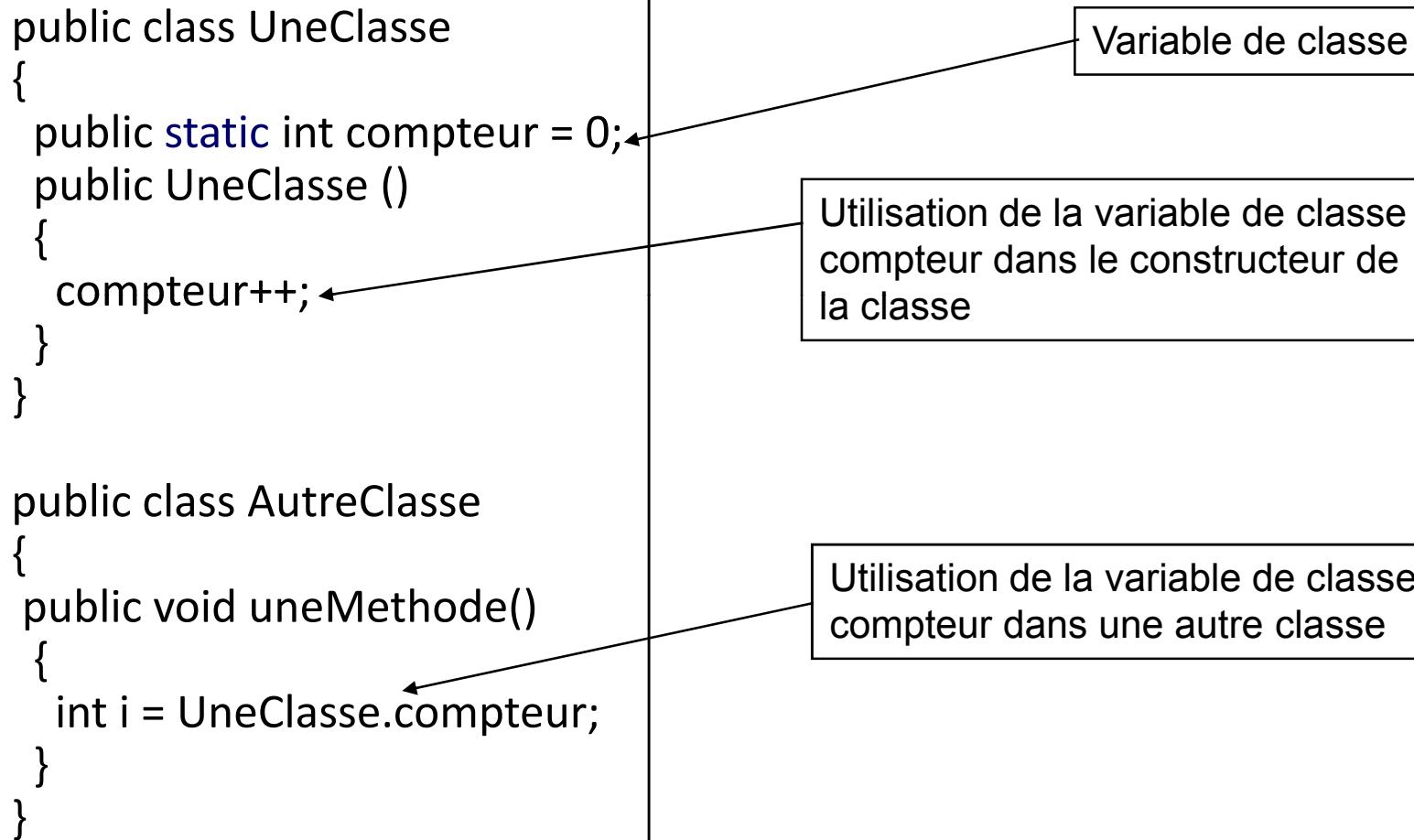
```
public class UneClasse
{
    public static int compteur = 0;
    public UneClasse ()
    {
        compteur++;
    }
}

public class AutreClasse
{
    public void uneMethode()
    {
        int i = UneClasse.compteur;
    }
}
```

Variable de classe

Utilisation de la variable de classe compteur dans le constructeur de la classe

Utilisation de la variable de classe compteur dans une autre classe



Méthodes de classe (1)

- Il peut être nécessaire de disposer d'une méthode qui puisse être appelée sans instance de la classe. C'est une méthode de classe.
- On utilise aussi le mot réservé **static**
- Une méthode de classe pouvant être appelée sans qu'il existe une instance, elle ne peut pas accéder à des attributs non statiques. Elle ne peut accéder qu'à ses propres variables et à des variables de classe.

Méthodes de classe (2)

```
public class UneClasse
{
    public int unAttribut;
    public static void main(String args[])
    {
        unAttribut = 5; // Erreur de compilation
    }
}
```

La méthode main est une méthode de classe donc elle ne peut accéder à un attribut non lui-même attribut de classe

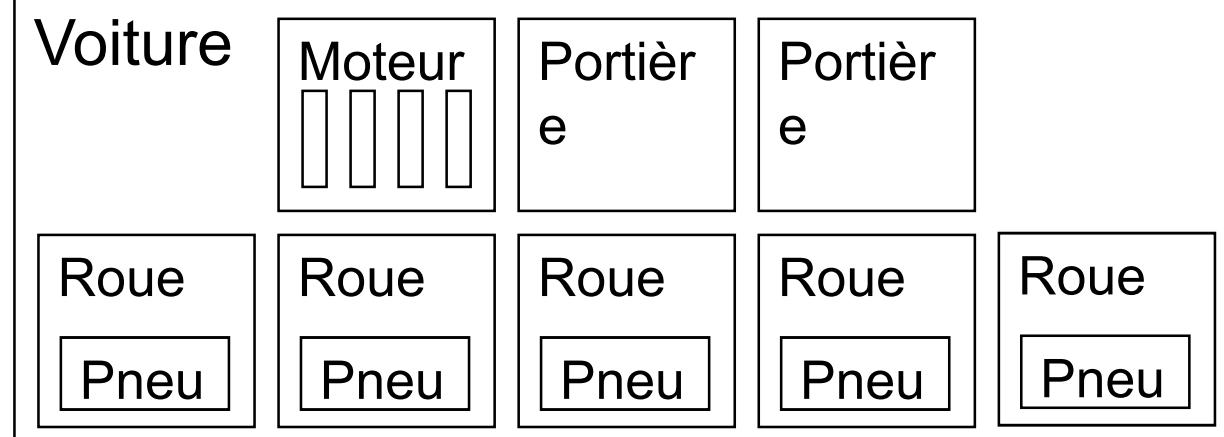
Autres exemples de méthodes de classe courantes

Math.sin(x);

String String.valueOf (i);

Composition d'objets (1)

- Un objet peut être composé à partir d'autres objets
Exemple : Une voiture composée de
 - 5 roues (roue de secours) chacune composée
 - d'un pneu
 - d'un moteur composé
 - de plusieurs cylindres
 - de portières
 - Etc...



Chaque composant est un attribut de l'objet composé

Composition d'objets (2)

```
class Pneu {  
    private float pression ;  
    void gonfler();  
    void degonfler();  
}
```

```
class Roue {  
    private float diametre;  
    Pneu pneu ;  
}
```

```
class Voiture {  
    Roue roueAVG,roueAVD, roueARG, roueARD , roueSecours ;  
    Portiere portiereG, portiereD;  
    Moteur moteur;  
}
```

Composition d'objets (3)

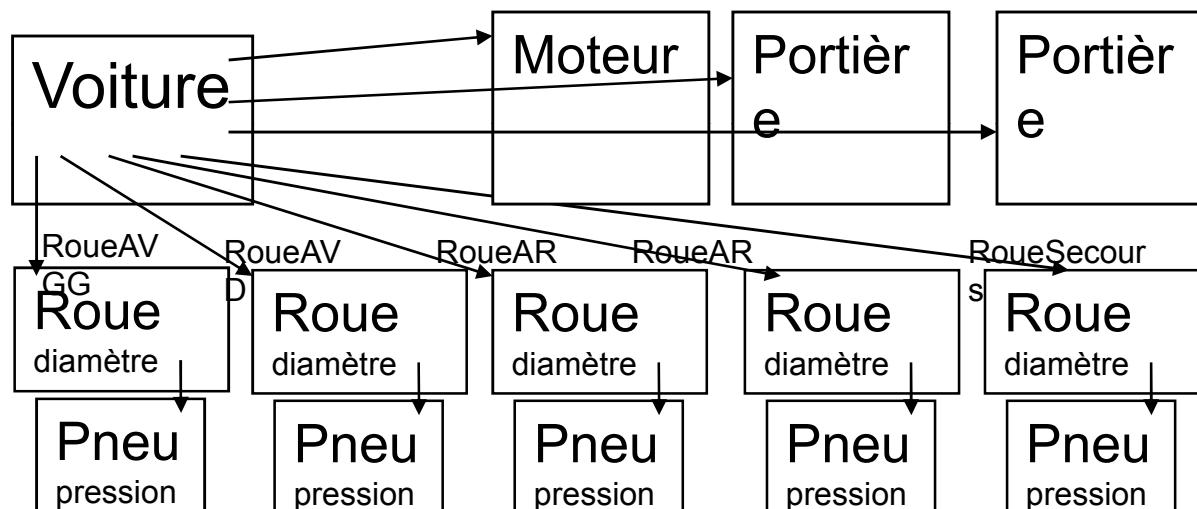
- Généralement, le constructeur d'un objet composé appelle le constructeur de ses composants

```
public Roue () {  
    pneu = new Pneu();  
}
```

```
public Voiture () {  
    roueAVG = new Roue();  
    roueAVD = new Roue();  
    roueARG = new Roue();  
    roueARD = new Roue();  
    portiereG = new Portiere();  
    portiereD = new Portiere();  
    moteur = new Moteur();  
}
```

Composition d'objets (4)

- L'instanciation d'un objet composé instancie ainsi tous les objets qui le composent



HERITAGE

02/07/2018

Pr. Mouhamadou THIAM Maître de
conférences en informatique

160

Introduction

- Pour raccourcir les
 - temps d'écriture et de
 - mise au point du code d'une application,
- il est intéressant de pouvoir réutiliser du code déjà écrit.
- Réutilisation ...

Par des classes clientes

- Soit une classe **A** dont on a le code compilé
- Une classe **C** veut réutiliser la classe **A**
 - Elle peut créer des instances de **A** et leur demander des services
 - On dit que la classe **C** est une classe cliente de la classe **A**

Avec modifications

- Souvent, cependant, on souhaite modifier en partie le comportement de **A** avant de le réutiliser
- Le comportement de **A** convient, sauf pour des détails qu'on aimeraient changer
- Ou alors, on aimeraient ajouter une nouvelle fonctionnalité à **A**

Avec modifications du code source

- On peut copier, puis modifier le code source de **A** dans des classes **A1, A2,...**
- **Problèmes:**
 - on n'a pas toujours le code source de **A**
 - les améliorations futures du code de **A** ne seront pas dans les classes **A1, A2,...**

Par l'héritage (1)

- L'héritage existe dans tous les langages objet à classes
- L'héritage permet d'écrire une classe **B**
 - qui se comporte dans les grandes lignes comme la classe **A**
 - mais avec quelques différences sans toucher au code source de **A**
 - On a seulement besoin du code compilé de **A**

Par l'héritage (2)

- Le code source de **B** ne comporte que ce qui a changé par rapport au code de **A**
- On peut par exemple
 - ajouter de nouvelles méthodes
 - modifier certaines méthodes

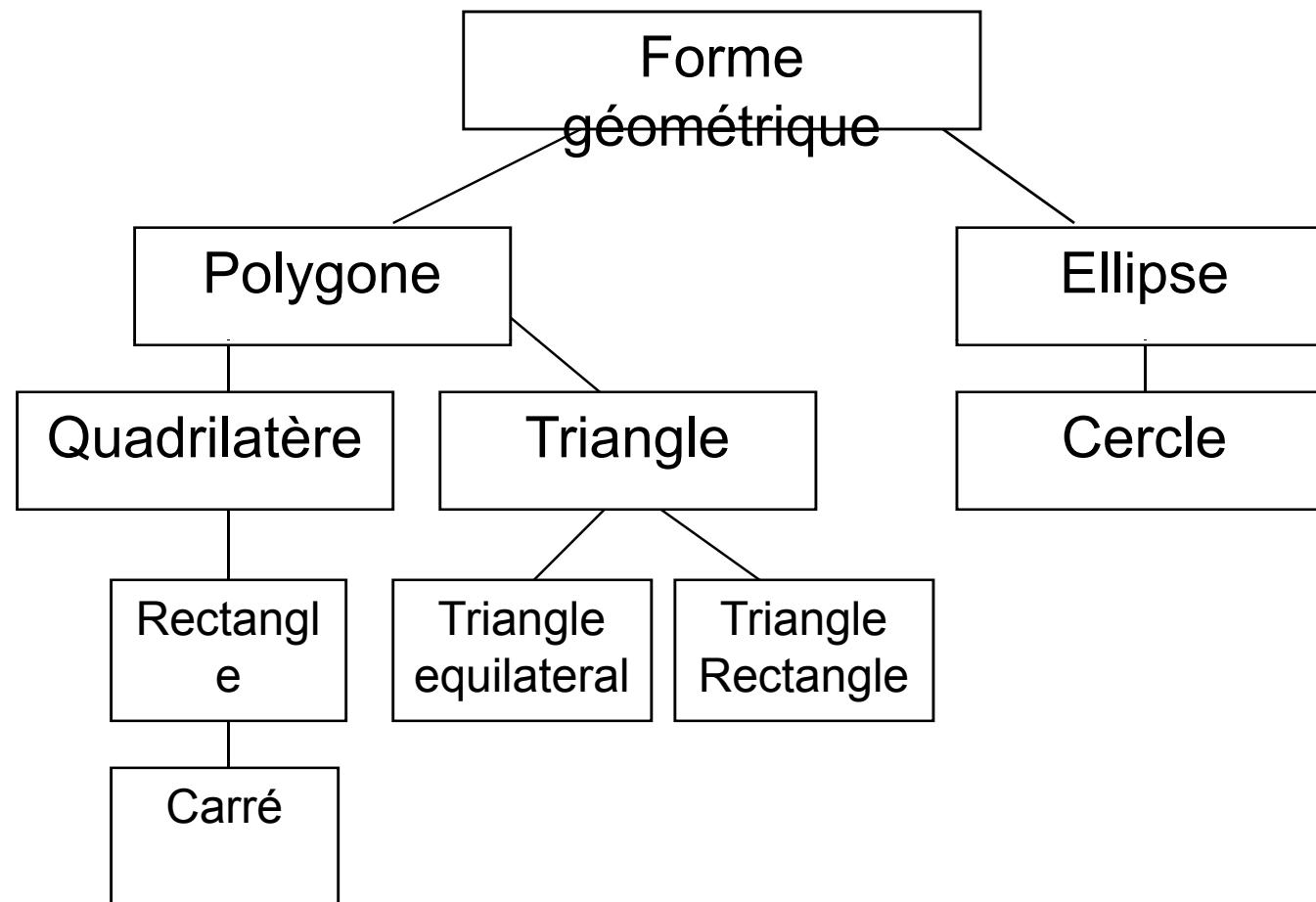
Vocabulaire

- 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**

L'héritage : Concept

- Modéliser le monde réel nécessite classification objets le composant
- Classification = distribution systématique en catégories selon des critères précis
- Classification = hiérarchie de classes
- Exemples :
 - classification des éléments chimiques
 - classification des êtres vivants

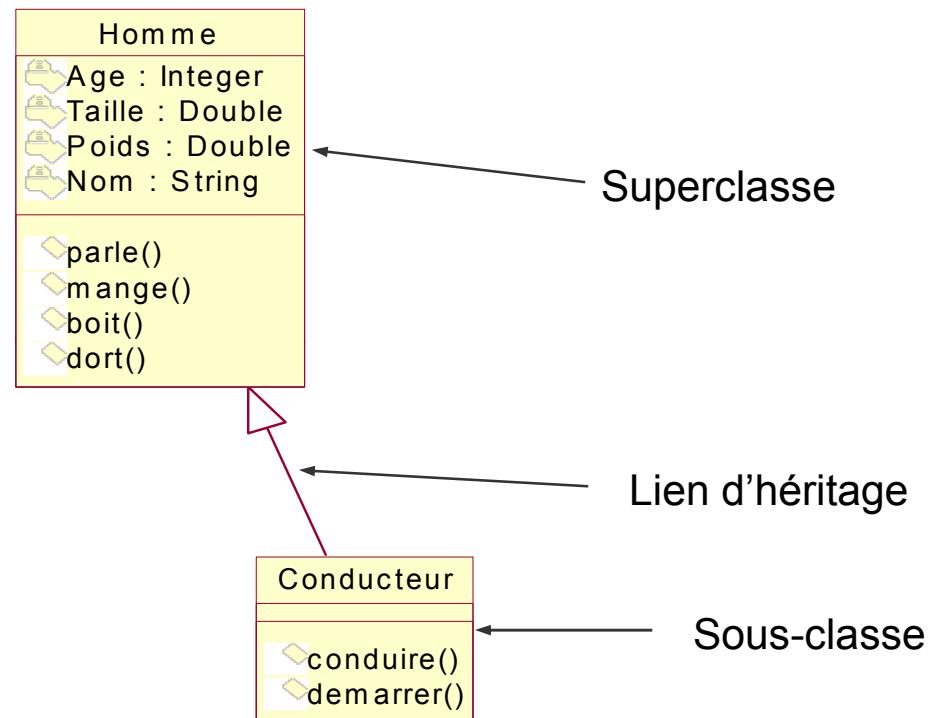
L'héritage : exemple



L'héritage : définition

- Héritage : mécanisme permettant **partage** et **réutilisation de propriétés** entre objets.
- Relation d'héritage : relation de généralisation / spécialisation.
- La classe parente est la **superclasse**.
- La classe qui hérite est la **sous-classe**.

L'héritage : représentation graphique



Représentation avec UML d'un héritage (simple)

EN JAVA

02/07/2018

Pr. Mouhamadou THIAM Maître de
conférences en informatique

172

Héritage en Java (1)

- Java implémente mécanisme d'héritage simple qui permet de
- "factoriser" de l'information grâce à une relation de généralisation / spécialisation entre deux classes.
- Pour le programmeur, il s'agit d'indiquer, dans la sous-classe, le nom de la superclasse dont elle hérite.

Héritage en Java (2)

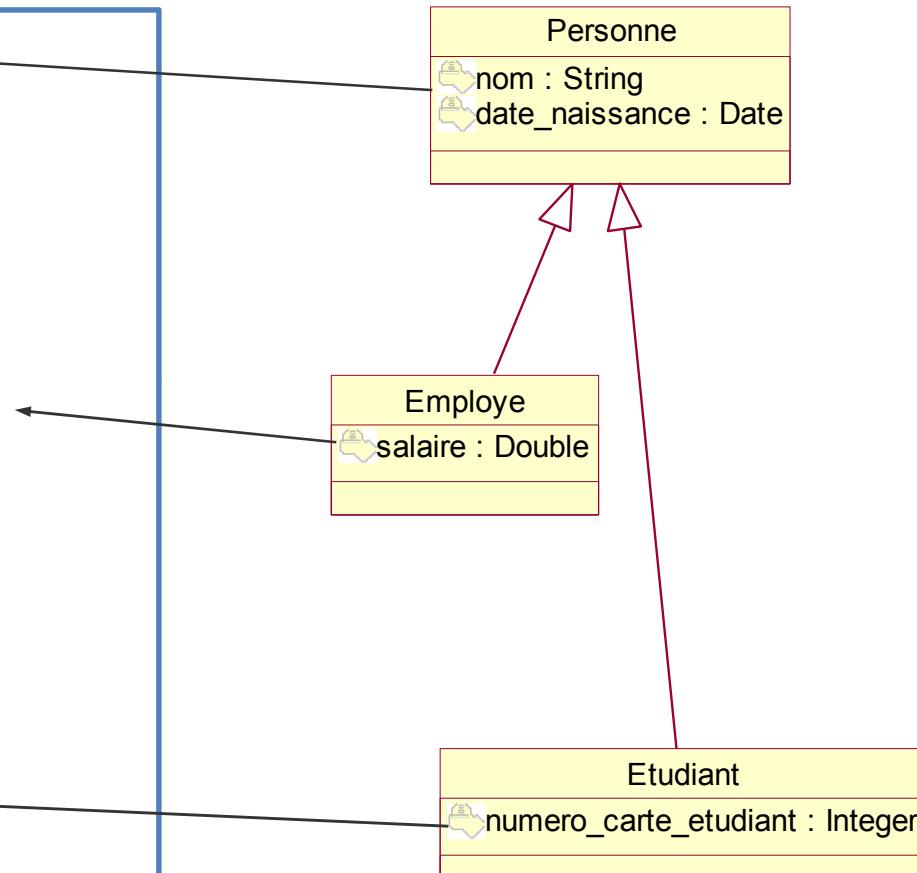
- Par défaut toutes classes Java hérite de la classe **Object**
- L'héritage **multiple** n'existe pas en Java.
- Mot réservé : **extends**

Héritage en Java (3)

```
class Personne
{
    private String nom;
    private Date date_naissance;
    // ...
}

class Employe extends Personne
{
    private float salaire;
    // ...
}

class Etudiant extends Personne
{
    private int numero_carte_etudiant;
    // ...
}
```



HERITAGE ET CONSTRUCTEURS

Constructeurs et héritage (1)

- Par défaut constructeur sous-classe appelle constructeur "**par défaut**" (celui qui ne reçoit pas de paramètres) de la superclasse.
- Attention : *constructeur sans paramètre doit exister toujours dans la superclasse...*

Constructeurs et héritage (2)

- Pour forcer l'appel d'un constructeur précis, utiliser le mot réservé **super**
- Cet appel doit être **première instruction** du constructeur.

Exemple

```
public class Employe extends Personne
{
    public Employe () {}
    public Employe (String nom,
                   String prenom,
                   int anNaissance)
    {
        super(nom, prenom, anNaissance);
    }
}
```

Appel explicite à ce constructeur
avec le mot clé super

```
public class Personne
{
    public String nom, prenom;
    public int anNaissance;
    public Personne()
    {
        nom=""; prenom="";
    }
    public Personne(String nom,
                    String prenom,
                    int anNaissance)
    {
        this.nom=nom;
        this.prenom=prenom;
        this.anNaissance=anNaissance;
    }
}
```

Cas de la classe objet

```
public class Personne
{
    public String nom, prenom;
    public int anNaissance;
    public Personne()
    {
        nom=""; prenom="";
    }
    public Personne(String nom,
                    String prenom,
                    int anNaissance)
    {
        this.nom=nom;
        this.prenom=prenom;
        this.anNaissance=anNaissance;
    }
}
```

```
public class Object
{
    public Object()
    {
        ... / ...
    }
}
```

Appel par défaut dans le constructeur de Personne au constructeur par défaut de la superclasse de Personne, qui est Object

REDÉFINITION DE MÉTHODES

Redéfinition de méthodes

- sous-classe peut redéfinir des méthodes existant dans une de ses superclasses (directe ou indirectes), à des fins de spécialisation.
 - Le terme anglophone est "overriding". On parle aussi de masquage.
 - La méthode redéfinie **doit avoir la même signature**.

Exemple

```
class Employe extends Personne
{
    private float salaire;
    public calculePrime( )
    {
        // ...
    }
}
```

Redéfinition

```
class Cadre extends Employe
{
    public calculePrime()
    {
        // ...
    }
    // ...
}
```

Recherche dynamique des méthodes (1)

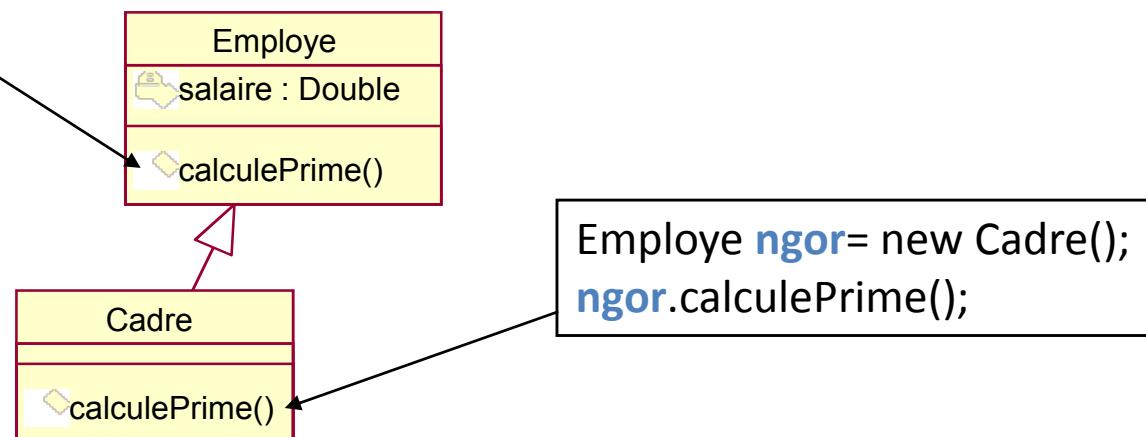
- Le polymorphisme
 - Capacité pour une entité de prendre plusieurs formes.
 - En Java, toute variable désignant un objet est potentiellement polymorphe (héritage).
 - Polymorphisme dit « d'héritage »

Recherche dynamique des méthodes (2)

- Mécanisme de "**lookup**" dynamique :
 - Appel de la méthode la plus spécifique d'un objet
 - Méthode correspondant au type réel de l'objet
 - Détermination à l'exécution uniquement (et non le type de la référence, seul type connu à la compilation, qui peut être plus générique).
 - Dynamicité permet d'écrire du code plus générique

RDM : Exemple

```
Employe ngor= new Employe();  
ngor.calculerPrime();
```



```
Employe ngor= new Cadre();  
ngor.calculerPrime();
```

SURCHARGE DE MÉTHODES

Surcharge de méthodes (1)

- Dans une même classe, plusieurs méthodes
 - peuvent posséder le même nom,
 - diffèrent en nombre et/ou type de paramètres.
- On parle de
 - surdéfinition ou
 - surcharge, on encore
 - overloading en anglais.

Surcharge de méthodes (2)

- Le choix de la méthode à utiliser dépend des paramètres passés à l'appel.
 - Ce choix est réalisé de façon statique (c'est-à-dire à la compilation).
- Très souvent **constructeurs** sont surchargés
 - plusieurs constructeurs prenant des paramètres différents et initialisant de manières différentes les objets

Opérateur *instanceof*

- Confère aux instances une capacité d'introspection
- Permet de savoir si une instance est instance d'une classe donnée.
- Renvoie une valeur booléenne

Exemple avec *instanceof*

```
if ( ... )
    Personne ngor = new Etudiant();
else
    Personne ngor = new Employe();

//...

if (ngor instanceof Employe)
    // discuter affaires
else
    // proposer un stage
```

FORÇAGE DE TYPE : TRANSTYPAGE

Transtypage (1)

- Quand **référence** du type d'une classe désigne une instance d'une **sous-classe**,
- Il faut forcer le type de la référence pour accéder aux attributs spécifiques à la sous-classe.

Transtypage (2)

- Sinon
 - compilateur ne peut déterminer le type réel de l'instance
 - ce qui provoque une erreur de compilation.
- On utilise également le terme de **transtypage**
- Similaire au « **cast** » en C

Transtypage : exemple

```
class Personne
{
    private String nom;
    private Date date_naissance;
    // ...
}
```

```
class Employe extends Personne
{
    public float salaire;
    // ...
}
```

```
Personne ngor = new Employe ();
float i = ngor.salaire; // Erreur de compilation
float j = ( (Employe) ngor ).salaire; // OK
```

A ce niveau pour le compilateur dans la variable « **ngor** » c'est un objet de la classe **Personne**, donc qui n'a pas d'attribut « **salaire** »

On « force » le type de la variable « **ngor** » pour pouvoir accéder à l'attribut « **salaire** ». On peut le faire car c'est bien un objet **Employe** qui est dans cette variable

AUTORÉFÉRENCE

Autoréférence : this (1)

- Le mot réservé **this**, utilisé dans une méthode, désigne
 - référence de l'instance à laquelle le message a été envoyée
 - donc celle sur laquelle la méthode est « exécutée »

Autoréférence : this (2)

- Il est utilisé principalement :
 - lorsqu'une référence à l'instance courante doit être passée en paramètre à une méthode,
 - pour lever une ambiguïté,
 - dans un constructeur, pour appeler un autre constructeur de la même classe.

L'autoréférence : exemple

```
class Personne
{
    public String nom;
    Personne (String nom)
    {
        this.nom=nom;
    }
}
```

Pour lever l'ambiguïté sur le mot « nom » et déterminer si c'est le nom du paramètre ou de l'attribut

```
public MaClasse(int a, int b) {...}

public MaClasse (int c)
{
    this(c,0);
}

public MaClasse ()
{
    this(10);
}
```

Appelle le constructeur
MaClasse(int a, int b)

Appelle le constructeur
MaClasse(int c)

RÉFÉRENCE À LA SUPERCLASSE

Référence à la superclasse : super

Le mot réservé **super** permet de faire référence au constructeur de la superclasse directe mais aussi à d'autres informations provenant de cette superclasse.

```
class Employe extends Personne
{
    private float salaire;
    public float calculePrime()
    {
        return (salaire * 0,05);
    }
    // ...
}
```

Appel à la méthode calculPrime()
de la superclasse de Cadre

```
class Cadre extends Employe
{
    public float calculPrime()
    {
        return (super.calculePrime() / 2);
    }
    // ...
}
```

FIN PREMIÈRE PARTIE POO1

Classes abstraites (1)

- Il peut être nécessaire au programmeur de créer une classe déclarant une méthode sans la définir (c'est-à-dire sans en donner le code). La définition du code est dans ce cas laissée aux sous-classes.
- Une telle classe est appelée classe abstraite.
- Elle doit être marquée avec le mot réservé **abstract**.
- Toutes les méthodes de cette classe qui ne sont pas définies doivent elles aussi être marquées par le mot réservé **abstract**.
- Une classe abstraite ne peut pas être instanciée.

Classes abstraites (2)

- Par contre, il est possible de déclarer et d'utiliser des variables du type de la classe abstraite.
- Si une sous-classe d'une classe abstraite ne définit pas toutes les méthodes abstraites de ses superclasses, elle est abstraite elle aussi.

```
public abstract class Polygone
{
    private int nombreCotes = 3;
    public abstract void dessine (); // methode non definie
    public int getNombreCotes()
    {
        return(nombreCotes);
    }
}
```

Introduction

- Pour raccourcir les
 - temps d'écriture et de
 - mise au point du code d'une application,
- il est intéressant de pouvoir réutiliser du code déjà écrit.
- Réutilisation ...

Notions importantes

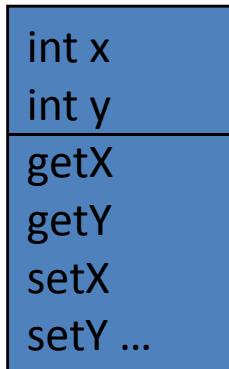
- Classe:
 - Une déclaration ~ un modèle à objet
 - Un type de donnée
 - (Aucun objet concret existe après la déclaration)
- Objet, Instance
 - Une instance d'une classe
 - (Une donnée concrète)

Comparaison

- Notion de **Cours** dans une université
 - Une vue abstraite
 - Une définition correspondant aux caractéristiques générales d'un 'cours'
 - On ne peut pas manipuler un **cours** abstrait
- **Instance**
 - Une instance concrète de cours (e.g. **POO**)
 - Contient des données concrètes (étudiants, prof, livre, ...)
 - Manipulable (inscrire un étudiant dans un cours, ...)
 - (En générale, on appelle une méthode à partir d'un objet/instance: **POO.inscrire(Etudiant)**)
- Relation: **POO** appartient à la classe **Cours**

Exemple

- public class Point{
 private int x, y;
 public Point(int x, int y) {this.x = x, this.y = y;}
 public int getX() {return x;}
 public int getY() {return y;}
 public void setX(int x) {this.x = x;}
 public void setY(int y) {this.y = y;}
}
- Point p = new Point(1,2);



Classe: rappel

- Déclaration d'une classe
 - Spécifier ce dont on a besoin pour une classe par ses comportements
 - Requêtes envoyées à un objet de cette classe pour obtenir des valeurs (accessors)
 - Commandes pour changer l'état de l'objet ou pour exécuter une tâche
 - (On ne spécifie pas les attributs à cette étape)
 - Implantation
 - Définir les attributs (variables, champs) à utiliser

Exemple

- public class Point
{
 private int x, y;

 public int getX() {return x;}
 public int getY() {return y;}
 public void setX(int x) {this.x = x;}
 public void setY(int y) {this.y = y;}
}

Réutilisation Par des classes clientes

- Soit une classe **A** dont on a le code compilé
- Une classe **C** veut réutiliser la classe **A**
- Elle peut créer des instances de **A** et leur demander des services
- On dit que la classe **C** est une classe cliente de la classe **A**

Avec modifications

- Souvent, cependant, on souhaite modifier en partie le comportement de **A** avant de le réutiliser
- Le comportement de **A** convient, sauf pour des détails qu'on aimeraient changer
- Ou alors, on aimeraient ajouter une nouvelle fonctionnalité à **A**

Avec modifications du code source

- On peut copier, puis modifier le code source de **A** dans des classes **A1, A2,...**
- **Problèmes:**
 - on n'a pas toujours le code source de **A**
 - les améliorations futures du code de **A** ne seront pas dans les classes **A1, A2,...**

Par l'héritage (1)

- L'héritage existe dans tous les langages objet à classes
- L'héritage permet d'écrire une classe **B**
 - qui se comporte dans les grandes lignes comme la classe **A**
 - mais avec quelques différences sans toucher au code source de **A**
 - On a seulement besoin du code compilé de **A**

Par l'héritage (2)

- Le code source de **B** ne comporte que ce qui a changé par rapport au code de **A**
- On peut par exemple
 - ajouter de nouvelles méthodes
 - modifier certaines méthodes

Vocabulaire

- 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**

Exemple Java – classe mère

```
public class Rectangle {  
    Private int x, y; // point en haut à gauche  
    Private int largeur, hauteur;  
    // La classe contient des constructeurs,  
    public int getX(){return x;} public void setX(int x){this.x=x;}  
    public int getY(){return y;} public void setY(int y){this.y=y;}  
    public int getHauteur() {return hauteur;} public void setHauteur(int h){hauteur = h;}  
    public int getLargeur() {return largeur;} public void setLargeur(int l){largeur = l;}  
    // contient(Point), intersecte(Rectangle)  
    // translateToi(Vecteur), toString(), ...  
    ...  
    public void dessineToi(Graphics g) {  
        g.drawRect(x, y, largeur, hauteur);  
    }  
}
```

Exemple Java – classe fille

```
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());  
    }  
}
```

Code des classes filles

- Quand on écrit la classe **RectangleColore**, on doit seulement
 - écrire le code (variables ou méthodes) lié aux nouvelles possibilités ; on ajoute ainsi une variable **couleur** et les méthodes qui y sont liées
 - redéfinir certaines méthodes ; on redéfinit la méthode **dessineToi()**

Redéfinition et surcharge

Ne pas confondre redéfinition et surcharge des méthodes : on

- **redéfinit** une méthode quand une nouvelle méthode a la même signature qu'une méthode héritée de la classe mère
- **surcharge** une méthode quand une nouvelle méthode a le même nom, mais pas la même signature, qu'une autre méthode de la même classe

Rappel: Signature d'une méthode (nom de la méthode + ensemble des types de ses paramètres)

Héritage : 2 façons de voir

- Particularisation-généralisation:
 - un rectangle coloré *est un* rectangle mais un rectangle particulier
 - la notion de figure géométrique est une généralisation de la notion de polygone
 - Une classe fille offre de nouveaux services ou enrichit les services rendus par une classe : la classe **RectangleColore** permet de dessiner avec des couleurs et pas seulement en « noir et blanc »

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 RectangleColoreextends Rectangle`
- Par défaut (pas de **extends** dans la définition d'une classe), une classe hérite de la classe **Object**

Exemples d'héritages

- class Voiture **extends** Vehicule
- class Velo **extends** Vehicule
- class VTT **extends** Velo
- class Employe **extends** Personne
- class ImageGIF **extends** Image
- class PointColore **extends** Point

Ce que peut faire une classe fille

La classe qui hérite peut

- ajouter des variables, des méthodes et des constructeurs
- redéfinir des méthodes (exactement les mêmes types de paramètres)
- surcharger des méthodes (même nom mais pas même signature) (**possible aussi à l'intérieur d'une classe**)

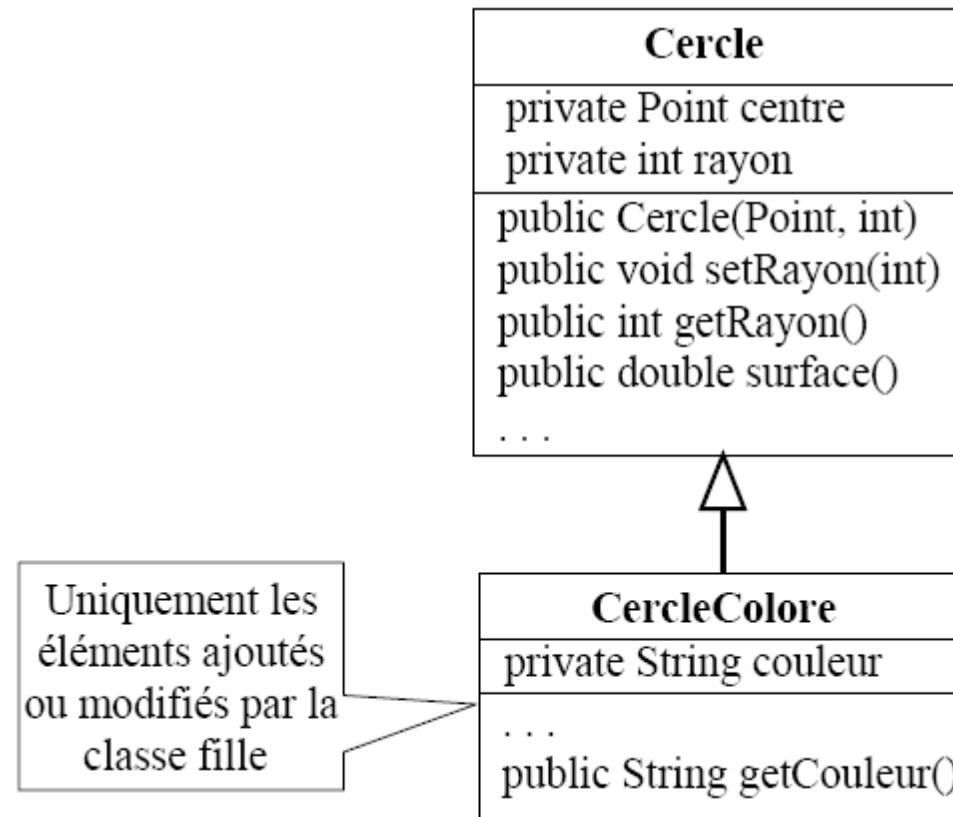
Principe important de la notion d'héritage

- Si « **B extends A** », le grand principe est que tout **B** est un **A**
- Par exemple,
 - un rectangle coloré *est un* rectangle ;
 - un poisson *est un* animal ;
 - une voiture *est un* véhicule
- En Java, on évitera d'utiliser l'héritage pour réutiliser du code dans d'autres conditions

Sous-type

- **B** est un sous-type de **A** si on peut ranger une expression de type **B** dans une variable de type **A**
- Les sous-classes d'une classe **A** sont des sous types de **A**
- En effet, si **B** hérite de **A**, tout **B** est un **A** donc on peut ranger un **B** dans une variable de type **A**
- Par exemple,
A a = new B(...); est autorisé

L'héritage en notation UML



Constructeurs : quoi de neuf?

- 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(...)

Constructeur de la classe mère

```
public class Rectangle {  
    private int x, y, largeur, hauteur;  
  
    public Rectangle(int x, int y, int largeur, int hauteur) {  
        this.x= x;  
        this.y= y;  
        this.largeur= largeur;  
        this.longueur= longueur;  
    }  
    . . .  
}
```

Constructeurs de la classe fille

```
public class RectangleColore extends Rectangle {  
    privateColor 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);  
    }  
    ...  
}
```

Appel implicite du constructeur de la classe mère (1)

- Si la première instruction d'un constructeur n'est ni **super(...)**, ni **this(...)**, le compilateur ajoute un appel implicite au constructeur sans paramètre de la classe mère (erreur s'il n'existe pas)

=> Un constructeur de la classe mère est toujours exécuté avant les autres instructions du constructeur

Appel implicite du constructeur de la classe mère (2)

- Mais la première instruction d'un constructeur de la classe mère est l'appel à un constructeur de la classe « grand-mère », et ainsi de suite...
- Donc la toute, première instruction qui est exécutée par un constructeur est le constructeur (sans paramètre) par défaut de la classe **Object** !
- (D'ailleurs c'est le seul qui sait comment créer un nouvel objet en mémoire)

La classe Cercle(1)

```
public class Cercle {  
    // Constante  
    public static final double PI = 3.14;  
    // Variables  
    private Point centre;  
    Private int rayon;  
    // Constructeur  
    public Cercle(Point c, int r) {  
        centre = c;  
        rayon = r;  
    }  
}
```

Ici pas de constructeur sans paramètre

Appel implicite du constructeur **Object()**

La classe Cercle(2)

// Méthodes

```
public double surface() {
    return PI * rayon * rayon;
}

public Point getCentre() {
    return centre;
}

public static void main(String[] args) {
    Point p = new Point(1, 2);
    Cercle c = new Cercle(p, 5);
    System.out.println("Surface du cercle: " + c.surface());
}
```

La classe CercleColore (1)

```
public class CercleColoreextendsCercle {  
    private String couleur;  
  
    public CercleColore(Point p, int r, String c) {  
        super(p, r);  
        couleur = c; Que se passe-t-il si on enlève  
cette instruction?  
    }  
  
    public voidsetCouleur(String c) {  
        couleur = c;  
    }  
  
    public String getCouleur() {  
        return couleur;  
    }  
}
```

Attention – débutant !

```
public class CercleColore extends Cercle {  
    Private Point centre;  
    Private int rayon;  
    Private String couleur;  
    public CercleColore(Point p, int r, String c) {  
        centre = p;  
        rayon = r;  
        couleur = c;  
    }  
    ...  
}
```

Que se passe-t-il ici?

Accès aux attributs (1)

```
public class Animal {  
    String nom; // remarquer que nom n'est pas private  
    public Animal() {  
    }  
    public Animal(String unNom) {  
        nom = unNom;  
    }  
    public void set Nom(String unNom) {  
        nom = unNom;  
    }  
    public String toString() {  
        return "Animal " + nom;  
    }  
}
```

Accès aux attributs (2)

```
public class Poisson extends Animal {  
    private int profondeurMax;  
  
    public Poisson(String nom, int uneProfondeur) {  
        this.nom= nom; // Et si nom est private ?  
        profondeurMax= uneProfondeur;  
    }  
    public void setProfondeurMax(int uneProfondeur) {  
        profondeurMax= uneProfondeur;  
    }  
    public String toString() {  
        return "Poisson " + nom + " ; plonge jusqu'à "+ profondeurMax + "  
               mètres";  
    }  
}
```

Résolution par encapsulation

```
public class Poisson extends Animal {  
    Private int profondeurMax;  
    public Poisson(String unNom, int uneProfondeur) {  
        super(unNom); // convient même si nom est private  
        profondeurMax= uneProfondeur;  
    }  
    public void setProfondeurMax(intuneProfondeur) {  
        profondeurMax= uneProfondeur;  
    }  
    public String toString() {  
        return "Poisson " + getNom()  
            + " plonge jusqu'à " + profondeurMax  
            + " mètres";  
    }  
}
```

De quoi hérite une classe ?

- Si une classe **B** hérite de **A** (**B extends A**), elle hérite automatiquement et implicitement de tous les membres de la classe **A** (mais pas des constructeurs)
- Cependant la classe **B** peut ne pas avoir accès à certains membres dont elle a implicitement hérité de **A** (par exemple, les membres **private**)
- Ces membres sont utilisés pour le bon fonctionnement de **B**, mais **B** ne peut pas les nommer ni les utiliser explicitement

Redéfinition des attributs

- Soit une classe **B** qui hérite d'une classe **A**
- Dans une méthode d'instance **m()** de **B**,
super. sert à désigner un membre de **A** :
 - en particulier, **super.m()** désigne la méthode de **A** qui est donc en train d'être redéfinie dans **B** :

```
intm(int i) {  
    return 500 + super.m(i);  
}
```

Classe Animal

```
public class Animal {  
    private String proprietaire;  
    private String nom;  
    private boolean vaccine = false;  
  
    public Animal(String p, String n){  
        proprietaire = p;  
        nom = n;  
    }  
  
    public void vacciner() {  
        vaccine = true;  
    }  
  
    public String toString() {  
        return "Proprietaire : " + proprietaire + "\nNom : " + n  
            "\nVaccine : " + vaccine;  
    }  
}
```

Rajouter une classe fille chat avec une méthode "miauler"

```
public class Chien extends Animal{
    private String race;

    public Chien(String r, String p, String n){
        super(p,n);
        race = r;
    }

    public void aboyer() {
        System.out.println("ouah ouah");
    }

    public String toString() {
        String s = super.toString();
        s = s + "\nRace : " + race;
        return s;
    }

    public static void main(String args[]){
        Chien c = new Chien("Caniche","Joe Blow", "Fifi");
        System.out.println(c);
        c.aboyer();
        c.vacciner();
        System.out.println(c);
    }
}
```

Méthode statique

- On ne peut utiliser **super.m()** dans une méthode **static m()** ;
 - une méthode **staticne** peut être redéfinie
 - **super.i**désigne la variable cachée **i** de la classe mère (ne devrait jamais arriver) ; dans ce cas, **super.i**est équivalent à **((A)this).i**

Méthode statique : exemple

```
public class C{
    int i;
}

public class D extends C{
    private int i;

    public void test() {
        i = 0; // D
        this.i = 0; // D
        super.i = 1; // C
        ((C)this).i = 1; // C
    }

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

Accès protected

- **Protected** joue sur l'accessibilité des membres (variables ou méthodes) par les classes filles
- Un membre **protected** de la classe A peut être manipulé par les classes filles de A sans que les autres classes non filles de A ne puisse les manipuler

Exemple

```
public class Animal {  
    protected String nom;  
    ...  
}
```

```
public class Poisson extends Animal {  
    private int profondeurMax;  
    public Poisson(String unNom, int uneProfondeur) {  
        nom = unNom; // utilisation de nom de la classe mère  
        profondeurMax = uneProfondeur;  
    }
```

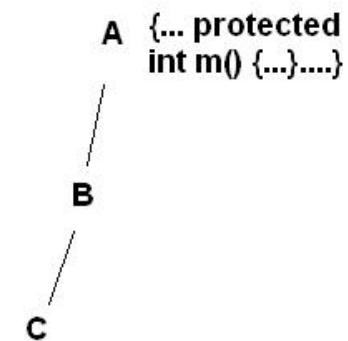
Autre règle

- **Protected** autorise également l'utilisation par les classes du même paquetage que la classe où est définie le membre ou le constructeur

Exemple

- Du code de **B** peut accéder à un membre **protected** de **A** (méthode **m()** ci-dessous) dans une instance de **B** ou d'une sous-classe **C** de **B** mais pas d'une instance d'une autre classe (par exemple, de la classe est **A** ou d'une autre classe fille **D** de **A**) ; voici du code de la classe **B** :

```
A a = new A(); // A classe mère de B
B b = new B();
C c = new C(); // C sous-classe de B
D d = new D(); // D autre classe de A
a.m(); // interdit
b.m(); // autorisé
c.m(); // autorisé
d.m(); // interdit
```



Encore des règles ...

Attention, **protected** joue donc sur

- l'accessibilité par **B** du membre **m** hérité (**B** comme sous-classe de **A**)
- mais pas sur l'accessibilité par **B** du membre **m** des instances de **A** (**B** comme cliente de **A**)

Polymorphisme (1)

- Contexte: soit **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(5);**
 - **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

a est un objet de la classe **B**
mais il est déclaré de la classe **A**

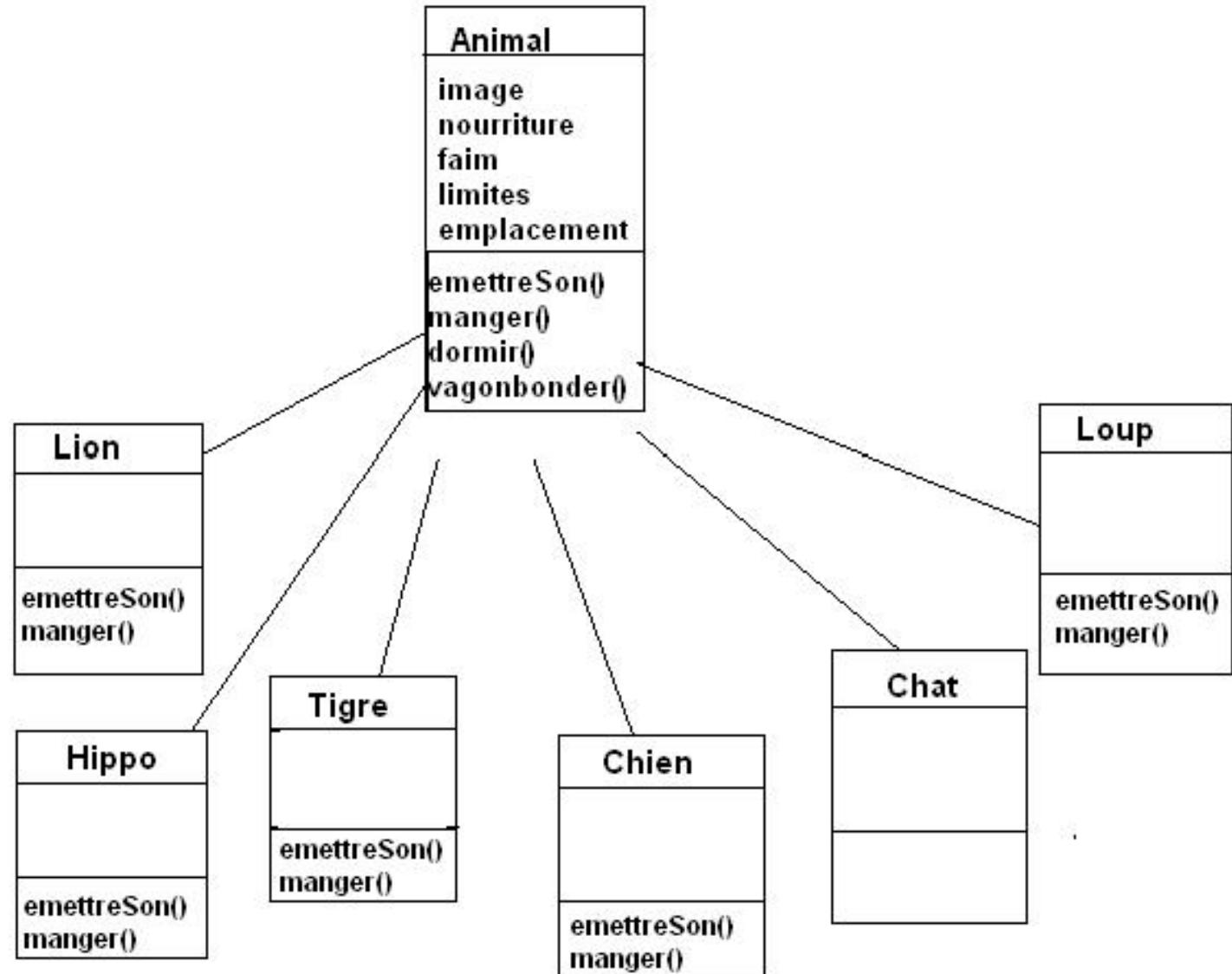
Polymorphisme (2)

- Le polymorphisme est le fait qu'une même écriture peut correspondre à différents appels de méthodes ; par exemple,
 - **A a = x.f();**
 - **a.m();**

f peut renvoyer une instance de A
ou de n'importe quelle sous-classe de A

appelle la méthode **m()** de **A** ou de n'importe quelle sous-classe de **A**
- Ce concept est une notion fondamentale de la programmation objet, indispensable pour une utilisation efficace de l'héritage

Exemple de la jungle



Exemple de la jungle (suite)

- Animal[] animaux=new Animal[5]
 - Animaux[0]= new Chien();
 - Animaux[1]= new Chat();
 - Animaux[2]= new Loup();
 - Animaux[3]= new Hippo();
 - Animaux[4]= new Lion();
- for (int i=0;i<animaux.length;i++) {
 Animaux[i].manger();
 Animaux[i].vagabonder();
}
- 
- Déclare un tableau de type animal
- On peut placer un objet de n'importe quelle sous-classe d'Animal dans le tableau

Utilisation

- Class **Veto** {
 Public void soigner(Animal a) {
 a.emettreSon();
 }
- Class **ProprietaireDAnimaux** {
 Public void start() {
 Veto v = new Veto();
 Chien c = new Chien();
 Hippo h = new Hippo();
 v.soigner(c);
 v.soigner(h);
 }

Remarques

- Le type de la référence peut être la superclasse du type de l'objet réel
- Les arguments et les types de retour peuvent également être polymorphes

Mécanisme du polymorphisme

- Le polymorphisme est obtenu grâce au « *latebinding* » (liaison retardée) : la méthode qui sera exécutée est déterminée
 - seulement à l'exécution, et pas dès la compilation
 - par le type réel de l'objet qui reçoit le message (et pas par son type déclaré)

Un peu de géométrie

```
public class Figure {  
    public void dessineToi() {}  
}
```

```
public class RectangleextendsFigure {  
    public void dessineToi() {}  
...  
}
```

```
public class CercleextendsFigure {  
    public void dessineToi() {}  
...  
}
```

Un peu de géométrie (suite)

```
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();
        ...
    }
}
```

Typage statique et polymorphisme

- En Java, le typage statique doit garantir dès la compilation l'existence de la méthode appelée :
 - la classe déclarée de l'objet qui reçoit le message doit posséder cette méthode
 - Ainsi, la classe **Figure** doit posséder une méthode **dessineToi()**, sinon, le compilateur refusera de compiler l'instruction « **figures[i].dessineToi()** »

Usage du polymorphisme (1)

- 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  
    }  
    elseif (figures[i] instanceof Cercle) {  
        ... // dessin d'un cercle  
    }  
}
```

Usage du polymorphisme (2)

- 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**, le code de **afficheToi()** sera toujours valable
- Sans polymorphisme, il aurait fallu modifier le code source de la classe **Dessin** pour ajouter un nouveau test :

```
if (figures[i] instanceof Losange) {  
    ... // dessin d'un losange  
}
```

Extensibilité

- Avec la programmation objet, une application est dite extensible si on peut étendre ses fonctionnalités **sans toucher au code source déjà écrit**
- C'est possible en utilisant en particulier le polymorphisme comme on vient de le voir

Mécanisme de la liaison retardée

- Soit **C** la classe réelle d'un objet **o** à qui on envoie un message « **o.m()** »
- Si le code de la classe **C** contient la définition (ou la redéfinition) d'une méthode **m()**, c'est cette méthode qui sera exécutée
- Sinon, la recherche de la méthode **m()** se poursuit dans la classe mère de **C**, puis dans la classe mère de cette classe mère, et ainsi de suite, jusqu'à trouver la définition d'une méthode **m()** qui est alors exécutée

Transtypage (*cast*)

- **Définitions**
 - Classe (ou type) réelle d'un objet : classe du constructeur qui a créé l'objet
 - Type déclaré d'un objet : type donné au moment de la déclaration
 - de la variable qui contient l'objet,
 - ou du type retour de la méthode qui a renvoyé l'objet

Cast: conversions de classes

- Le « *cast* » est le fait de forcer le compilateur à considérer un objet comme étant d'un type qui n'est pas le type déclaré ou réel de l'objet
- En Java, les seuls *casts* autorisés entre classes sont les *casts* entre classe mère et classes filles
- On parle de *upcast* et de *downcast* suivant *le fait que le type est forcé de la classe fille vers la classe mère ou inversement*

Syntaxe

- Pourcaster un objet en classe C :
(C) o;
- Exemple :

Velo v = new Velo();

Vehicule v2 = (Vehicule) v;

UpCast: classe fille → classe mère

- *Upcast*: un objet est considéré comme une instance d'une des classes ancêtres de sa classe réelle
- Il est toujours possible de faire un *upcast*: à cause de la relation *est-unde* l'héritage, tout objet peut être considéré comme une instance d'une classe ancêtre
- Le *upcast* est souvent implicite

Utilisation du *UpCast*

- Il est souvent utilisé pour profiter ensuite du polymorphisme :

```
Figure[] figures = new Figure[10];
```

```
figures[0] = new Cercle(p1, 15);
```

```
...
```

```
figures[i].dessineToi();
```

DownCast: classe mère → classe fille

- ***Downcast:*** un objet est considéré comme étant d'une classe fille de sa classe de déclaration
- Toujours accepté par le compilateur Mais peut provoquer une erreur à l'exécution ;
- A l'exécution il sera vérifié que l'objet est bien de la classe fille
- Un ***downcast*** doit toujours être explicite

Utilisation du *DownCast*

- Utilisé pour appeler une méthode de la classe fille qui n'existe pas dans une classe ancêtre
 - **Figure f1 = new Cercle(p, 10);**
 - ...
 - **Point p1 = ((Cercle)f1).getCentre();**

Downcast pour récupérer les éléments d'une liste

```
// Ajoute des figures dans un ArrayList.  
// Un ArrayList contient un nombre quelconque  
// d'instances de Object  
  
ArrayList figures = new ArrayList();  
figures.add(new Cercle(centre, rayon));  
figures.add(new Rectangle(p1, p2));  
...  
// get() renvoie un Object. Cast nécessaire pour appeler  
// dessineToi() qui n'est pas une méthode de Object  
  
((Figure)figures.get(i)).dessineToi();  
...
```

Classe final (et autres au final)

- Classe **final** : ne peut avoir de classes filles (**String** est **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 ou d'un **catch**) : la valeur (éventuellement une référence) ne pourra être modifiée dans le code de la méthode

Réalisation d'un parseur

- Un **parseur** est un **programme** qui **analyse syntaxiquement** un document écrit dans un langage donné (plus souvent de balises). Il permet de récupérer les informations contenues dans les balises d'un document **XML** ou **HTML** par exemple. Cet outil distinguera les informations en fonction de leur contenu et de leur situation dans le document : balise de **début**, balise de **fin**, **texte**, etc. Plus généralement, un parseur peut être assimilé à un **outil d'analyse syntaxique**. C'est d'ailleurs le sens premier du terme anglais **parser**.

FIN PREMIÈRE PARTIE

Objectifs du cours

- Connaître les principaux concepts manipulés en POO
- Connaître les principes de la programmation des interfaces graphiques
- Connaître les principes de gestion des collections
- Etre capables de manipuler des fichiers (entrée/sortie)
- Etre capables de communiquer avec une BD
- Savoir utiliser des API, courbes et processus

Plan

1. Rappels POO et Java
2. Classe abstraite, Interface et Type générique
3. Exceptions Java et Javadoc
4. Swing: Création d'interfaces graphiques
5. Collections
6. Entrées/Sorties (Fichiers Textes et binaires)
7. JDBC
8. Compléments

Plan

- 1. Rappels POO et Java**
2. Classe abstraite, Interface et Type générique
3. Exceptions Java et Javadoc
4. Swing: Création d'interfaces graphiques
5. Collections
6. Entrées/Sorties (Fichiers Textes et binaires)
7. JDBC
8. Compléments

- CF première partie du cours

Plan

1. Rappels POO et Java
2. Classe abstraite, Interface et Type générique
3. Exceptions Java et Javadoc
4. Swing: Création d'interfaces graphiques
5. Collections
6. Entrées/Sorties (Fichiers Textes et binaires)
7. JDBC
8. Compléments

CLASSES ABSTRAITES

Intuition

- Classe **abstraite** est une classe **incomplète**.
- Ensemble d'attributs et de méthodes dont
 - Certaines méthodes ne contiennent pas d'instructions
 - elles devront être définies dans une classe héritant de cette classe abstraite.

Utilité des CA

- En général à définir les **grandes lignes** du comportement d'une classe d'objets **sans forcer l'implémentation** des détails de l'algorithme.
- Pourquoi "abstraite" ?
- Une classe est **abstraite** soit parce que
 - on n'est **pas capable** d'écrire l'implémentation de toutes les méthodes,
 - on ne **veut pas** créer d'instance de cette classe.

Mécanismes de mise en œuvre

- Une sous-classe qui n'implémente pas toutes les méthodes abstraites de sa super-classe est elle-même abstraite. Il faut donc la qualifier d'abstraite.
- Quand on ne peut pas écrire d'implémentation pour une méthode donnée, cette méthode est qualifiée d'**abstraite**. Cela signifie que l'on laisse le soin aux sous-classes d'implémenter cette méthode.
- En java, c'est le mot clef **abstract** qui permet de qualifier d'abstraite une classe ou une méthode

Classe abstraite

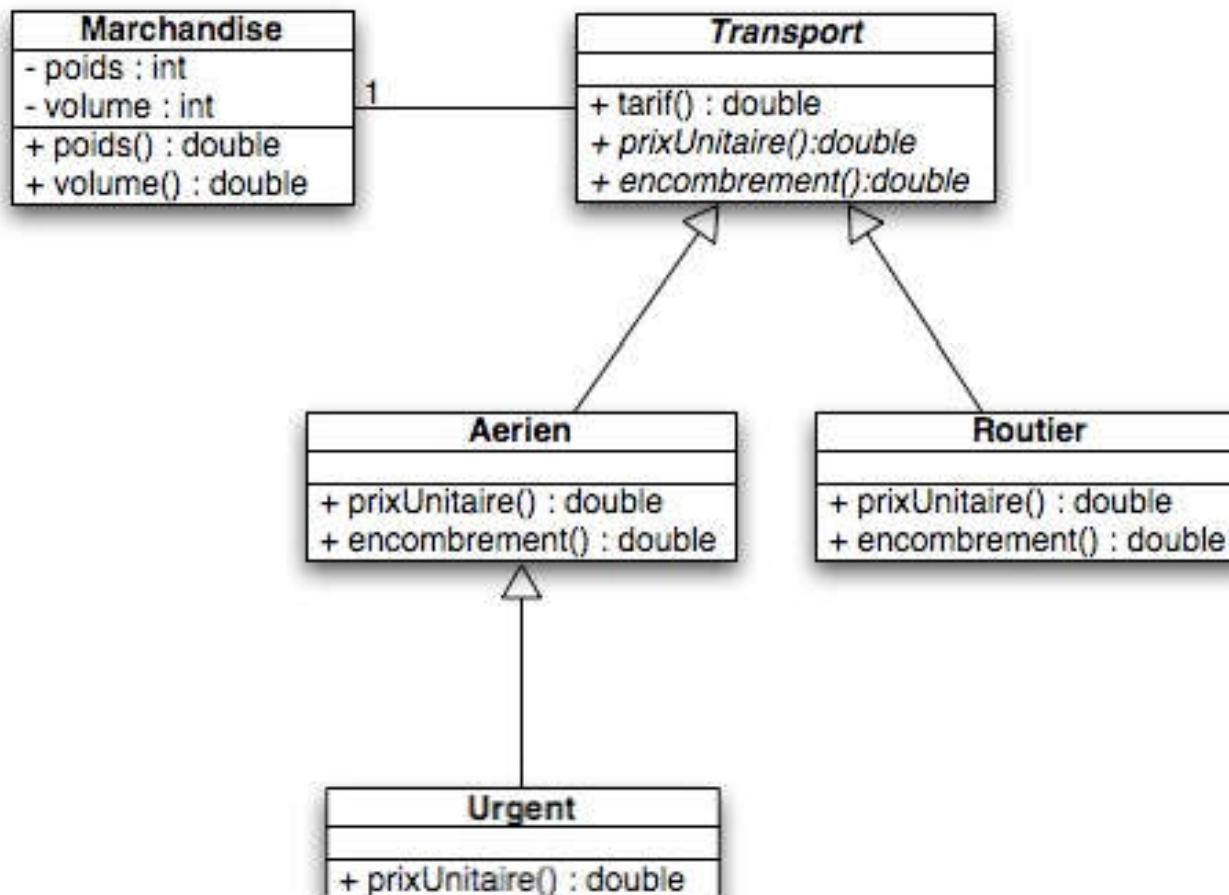
- Définition : Une classe *abstraite* est une classe contenant une (ou plusieurs) méthode(s) *abstraite(s)* :
- Syntaxe :

```
abstract class ClasseAbstraite {  
    abstract void ... méthodeAbstraite(...);  
    ...  
}
```

Exemple

- On modélise la tarification du transport d'une marchandise :
 - Une marchandise comporte un poids et un volume
 - Le tarif d'un transport est toujours le produit d'un prix unitaire par l'encombrement de la marchandise transportée.
 - Pour le transport aérien, l'encombrement est le poids de la marchandise. Le tarif sera 10 fois supérieur à l'encombrement.
 - Si on veut un transport aérien urgent, il faut payer deux fois plus cher que le tarif normal.
 - Pour le tarif routier, l'encombrement est le volume de la marchandise. Le tarif sera 2 fois supérieur à l'encombrement.

Diagramme UML du transport



Classe abstraite : règles (1)

- Méthode *abstraite* est déclarée (**signature** est donnée), mais non définie (**corps** n'est pas écrit).
- Sous-classe classe abstraite est abstraite, à moins qu'elle ait définie **toutes** les *méthodes abstraites* de sa superclasse.
- Classe abstraite n'est pas *classe d'allocation*
Interdit de créer un objet (en utilisant une fonction *new ...()*) dont le type est une classe abstraite.

Classe abstraite : règles (2)

- Doit **forcément être héritée**. Famille de classes possédant mêmes méthodes avec un comportement commun.
- Utilise le **polymorphisme** pour manipuler un ensemble d'objets de cette famille indépendamment de leur type réel.

Classe abstraite : règles (3)

- L'héritage sans classe abstraite met en avant une classe de la famille ayant le même comportement. L'utilisation d'une classe abstraite permet d'éviter cette différenciation.
- **Surtout**, l'utilisation d'une classe abstraite garantit que les méthodes abstraites de cette classe seront définies spécifiquement pour chaque sous-classe.

Classe abstraite : règles (4)

- Héritage sans classe abstraite → définition de méthodes communes dans la superclasse,
- Certaines méthodes doivent être redéfinies dans les sous-classes.
- **On n'a aucune garantie quant à cette redéfinition.**

Classe abstraite : règles (5)

- Oubli de définir une de ces méthodes, la méthode correspondante de la superclasse sera utilisée (à tort).
- Problème ne peut pas se produire si utilisation classe abstraite :
 - Oubli de redéfinir une des méthodes abstraites dans une sous-classe, cette classe ne sera pas une classe d'allocation, et on ne pourra pas créer d'objet du type associé.

Sources (sans la classe Marchandise) :

```
abstract class Transport {  
    protected Marchandise m ;  
    public Transport (Marchandise m) {  
        this.m = m ;  
    }  
    public int tarif() {  
        return prixUnitaire() * encombrement() ;  
    }  
    abstract public int prixUnitaire() ;  
    abstract public int encombrement() ;  
}
```

Classe Aerien

```
class Aerien extends Transport{
    public Aerien (Marchandise m) {
        super(m) ;
    }
    protected int prixUnitaire(){
        return 10 ;
    }
    protected int encombrement(){
        return m.poids() ;
    }
}
```

Classe Urgent

```
class Urgent extends Aerien{  
    public Urgent (Marchandise m) {  
        super(m) ;  
    }  
    protected int prixUnitaire(){  
        return 2 * super.prixUnitaire() ;  
    }  
}
```

Classe Routier

```
class Routier extends Transport{
    public Routier (Marchandise m){
        super(m) ;
    }
    protected int prixUnitaire(){
        return 2 ;
    }
    protected int encombrement(){
        return m.volume() ;
    }
}
```

Exemple : GenerateurSuite

- public void **setTermelInitial** (double v){
 }
 ...
}
- public String **getNomSuite**(){
 }
 ...
}
- protected double **recurrence**(double terme, int i){
 }
 ...
}

GenerateurSuiteAbstraite

- **public abstract void setTermelInitial (double v);**
...
- **public abstract String getNomSuite();**
...
- **protected abstract double recurrence(double terme, int i);**
...

Résumé

- **Règle 1 :** classe automatiquement abstraite si une de ses méthodes est abstraite.
- **Règle 2 :** On peut déclarer qu'une classe est abstraite par le mot clef abstract.
- **Règle 3 :** Une classe abstraite n'est pas instanciable (on ne peut pas utiliser les constructeurs d'une classe abstraite et donc on ne peut pas créer d'objet de cette classe.).
- **Règle 4 :** Une classe qui hérite d'une classe abstraite ne devient concrète que si elle implémente toutes les méthodes abstraites de la classe dont elle hérite.
- **Règle 5 :** Une méthode abstraite ne contient pas de corps, mais doit être implementée dans les sous-classes non abstraites:
`abstract nomDeMéthode (<arguments>);` n'est qu'une signature

Résumé (fin)

- **Règle 6** : une classe est abstraite peut contenir des méthodes non abstraites et des déclarations de variables ordinaires.
- **Règle 7** : Une classe abstraite peut être dérivée en une sous-classe :
 - abstraite si une ou plusieurs méthodes ne sont pas implémentées par la classe fille,
 - non abstraite si toutes les méthodes abstraites sont implémentées dans la classe fille.
- **Le Rôle des classes abstraites :**
- dénomination commune de classes que l'on ne veut pas voir exister en tant que tel.
- outil de spécification

INTERFACES

Utilité

- Équipes travaillant sur un même projet
- Chacune doit pouvoir connaître
 - méthodes qui seront implémentées par les autres
 - spécification de ces méthodes pour pouvoir
 - Inclure appels dans leurs propres méthodes.

Déclaration

- Interface ne comporte que des **constantes** et des **méthodes abstraites**. Elle n'est pas classe abstraite? **POURQUOI?**
- Classe peut implémenter **1** ou **+** interfaces → définir le corps de toutes les méthodes abstraites. C'est **l'héritage multiple de Java**.
- [**<modificateurs d'accès>**] interface **<nomInterface>** [**implements <interfaces>**] {
 <constantes> <méthodes abstraites> }
 - **Constantes** explicitement ou implicitement **static** et **final** :
 - [**<modificateurs>**] **static final** **<type> <nomVariable> = <valeur>** ;
 - [**<modificateurs>**] **<type> <nomVariable> = <valeur>** ;
 - **Méthodes** explicitement ou implicitement abstraites :
 - [**<modificateur>**] **<type> <nomMéthode> ([<paramètres formels>]);**
 - [**<modificateur>**] **abstract <type> <nomMéthode> ([<paramètres formels>]);**

Interface (1)

- Ensemble de prototypes de méthodes ou de propriétés qui forme un contrat.
- Classe **100% abstraite**, vous n'avez qu'à y mettre des méthodes abstraites sans le mot clé **abstract**.
- Implémentation une ou plusieurs interfaces → **OBLIGATOIREMENT redéfinir leurs méthodes !**
- Les **sous-classes** d'une superclasse qui implémente une interface, héritent (et peuvent redéfinir) des méthodes implémentées

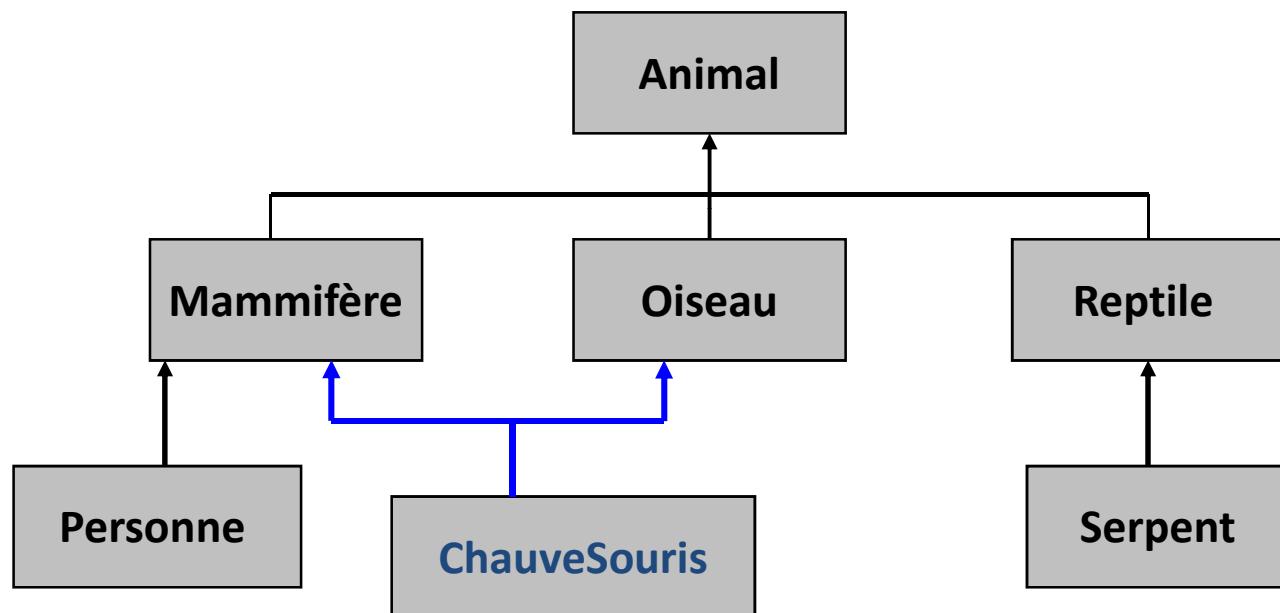
Interface (2)

- Implémentation interface → s'engager à fournir une implémentation de toutes les méthodes de l'interface.
- Implémentation vérifiée par le compilateur.

Héritage multiple en Java

- On ne peut hériter qu'une classe
 - Mot clé (*extends*)
- On peut hériter de plusieurs interfaces
 - Mot clé (*implements*)
- Les interfaces permettent de modéliser cet aspect de la POO

Exemple d'héritage multiple



Exemple

```
public interface Animal
{
    // tous les animaux doivent implémenter les méthodes suivantes
    void manger(String nourriture);
    void seDeplacer();
    void respirer();
}
```

La classe ChauveSouris

- public class ChauveSouris **extends** Oiseau **implements** Animal {
 //données membres
 public void **seDeplacer** (){
 System.out.println("Je vole car suis oiseau");
 }
 public void **respirer** (){
 System.out.println("Je vole car suis oiseau");
 }

 public void **manger** (String nourriture){
 System.out.println("Je suis omnivore");
 }
}

Héritage multiple d'interfaces

- Héritage d'interfaces peut être multiple, c.à.d. qu'on peut écrire
- ```
public class classeDérivée extends classeDeBase
 implements i1, i2, ..., in {
```

...

```
}
```
- où les  $i_j$ ,  $i=1..n$  sont des interfaces.

# Exemple

```
public interface InterfaceA {
 public void methodA();
}

public interface InterfaceB {
 public void methodB();
}

public interface InterfaceAB extends InterfaceA,
InterfaceB {
 public void otherMethod();
}
```

# Exemple

```
public class ClassAB implements InterfaceAB{
 public void methodA(){
 System.out.println("A");
 }

 public void methodB(){
 System.out.println("B");
 }
 public void otherMethod(){
 System.out.println("blablah");
 }

 public static void main(String[] args) {
 ClassAB classAb = new ClassAB();
 classAb.methodA();
 classAb.methodB();
 classAb.otherMethod();
 }
}
```

# TYPE GÉNÉRIQUE

# Définition

- Les génériques (*generics*) = classes typés au moment de la compilation.
- Les classes utilisent des typages en paramètres.
- Exemple : liste chainée peut contenir
  - entiers, chaînes, autres
- Typée en liste de chaîne ou liste d'entier

# Exemple : Java < 5

```
public class MaListe
{
 private LinkedList liste;
 public setMembre(String s)
 {
 liste.add(s);
 }
 public int getMembre(int i)
 {
 return (Int)liste.get(i);
 }
}
```

# Problèmes

- Programmeur → systématiquement des transtypages
- Transtypepage obligatoire car *LinkedList* manipule des objets *Object*
- Compilateur ne peut détecter de problème
- Problème qui ne surviendra qu'à l'exécution (*RunTimeError*).
- Méthode dangereuse car compilateur vérifie cohérence données.

# A partir de Java 5

- Introduction de types génériques
- Templates (modèles) du C++
- Code produit est unique pour tous les objets obtenus à partir de la même classe générique

# Exemple

```
public class Famille < MaClasse >
{
 private LinkedList < MaClasse > liste;

 public setMembre (MaClasse m)
 {
 liste.add(m);
 }

 public MaClasse getMembre (int i)
 {
 return liste.get(i);
 }

 public Int getInt (int i) //première erreur
 {
 return liste.get(i);
 }
}
```

# Utilisation de la classe

```
Famille<String> famille = new Famille<String>();
```

```
famille.setMembre ("essai");
```

```
famille.setMembre (210); //seconde erreur
```

## Détails ...

- Déclaration de la classe le paramètre placé entre les caractères < et > = classe
- Détermination lors de la déclaration de la création de l'objet.
- Erreur de typage à la compilation si les types utilisés par les méthodes **#** types attendus.
- **Exemple** : erreur signalée sur le second ajout.

## Détails ...

- Déclaration de la classe : *liste membre* est déclarée ne pouvant contenir que des objets de classe *MaClasse*.
- Identifiant *MaClasse* : pas une classe existante dans le packages
  - préférable qu'il ne le soit pas
  - pour qu'aucune confusion ne soit faite,
- Déclaration de l'objet *Famille* que l'identifiant *MaClasse* sera résolu.

## Détails ...

- Possibilité d'utiliser objet d'une **classe héritant** de celle en **paramètre** du type générique.
- Assurance de compatibilité ascendante avec les versions antérieures de Java.
- Non indication de classe de paramétrage la classe par défaut est *java.lang.Object*.

# Plus d'un paramètre

- De la même façon que pour les classes basées sur les List, les déclarations de vos classes peuvent utiliser ces génériques.
- Cela permet de rendre le code plus souple et surtout réutilisable dans des contextes très différents.
- Plusieurs paramètres, séparés par des virgules, peuvent être utilisés entre les caractères < et >.

# Exemple

```
public class ListeDeTruc<Truc, Bidule>
{
 private LinkedList < Truc > liste;
 private ArrayList <Bidule> tableau;
 public void accumule(Truc m)
 {
 liste.add(m);
 }
 public Bidule recherche(int i)
 {
 return tableau.get(i);
 }
}
```

# Utilisation de la classe

- `ListeDeTruc<String,Integer> liste1 = new  
ListeDeTruc<String,Integer>();`
- `ListeDeTruc<Thread,Date> liste2 = new  
ListeDeTruc<Thread,Date>();`

# Héritage

- Dans certains cas de spécifications précises, il est possible d'écrire
- **public class ListeDeTruc<Truc extends Bidule,  
MaList<String>> implements  
Moninterface<Chose>**

# Remarques

- Créer classe qui hérite de ces objets est délicat.
- Chose et Bidule sont des classes existantes
- Truc sera résolu au moment de la déclaration de l'objet ListeDeTruc
- L'utilisation du mot clef super autorisé dans une classe héritant d'une classe générique.

# Conventions BP

- Possible d'utiliser n'importequel identifiant suivant la convention de nommage des classes
- Recommandation d'utiliser identifiant composé d'une seule lettre selon la convention :
  - **<E>** « Element », type éléments d'une collection ;
  - **<K>** et **<V>** « Key », « Value », type clés et valeurs d'une Map ;
  - **<N>** « Number » ;
  - **<T>** « Type » ;
  - **<S>, <U>, <V> etc.** second, troisième, i<sup>ème</sup> type.

# Remarques

- Syntaxe similaire avec templates du C++ mais sont différents
- Moyen d'éviter conversion entre `java.lang.Object` et le **type spécifié** de manière implicite.
- Impossible de
  - Implémenter plusieurs fois la même interface avec des paramètres différents
  - Créer deux versions surchargées d'une méthode : une utilisant la classe `Object` et l'autre un **type générique**.

# Plan

1. Rappels POO et Java
2. Classe abstraite, Interface et Type générique
- 3. Exceptions Java et Javadoc**
4. Swing: Création d'interfaces graphiques
5. Collections
6. Entrées/Sorties (Fichiers Textes et binaires)
7. JDBC
8. Compléments

# Exception (1)

- **Définition** : signal indiquant que quelque chose d'exceptionnel (erreur) s'est produit
- **Utilités**
  - Gestion erreur indispensable : si mauvaise catastrophe (Ariane 5)
  - Mécanisme simple et lisible
    - Séparation code erreur et code algorithme
    - Propagation dans la pile des appels de méthodes (possibilité de récupérer erreur à plusieurs niveaux d'une application)

# Exception (2)

- **Vocabulaire :**

- Lancer ou déclencher (**throw**) exception consiste à signaler les erreurs
- Capturer ou attraper (**catch**) exception permet de traiter les erreurs

# Exception (3)

- Lancer et capturer une exception

```
public class Point {
 ...//Déclaration des attributs
 ...//Autres méthodes et constructeurs
 public Point (int a, int b) throws ErrConst {
 if ((a < 0) || (b < 0)) throw new ErrConst();
 this.x = a; this.y = b;
 }
 public void affiche (){
 System.out.println("Coordonnées : " + x + " " + y);
 }
}
```

La classe ErrConst n'est pas encore définie

# Exception (3)

- Lancer et capturer une exception

```
public class TestPoint {
 public void static main (String args[]) {
 try {
 Point a = new Point(1, 4);
 a.affiche();
 a = new Point (-2, 4);
 a.affiche();
 }catch(ErrConst e){
 System.out.println("Erreur de construction");
 System.exit(-1);
 }
 }
}
```

La classe **ErrConst** n'est pas encore définie

Coordonnées : 1 4  
Erreur de construction

# Lancer ou déclencher (1)

- Méthode déclare qu'elle peut lancer exception

```
public Point (int a, int b) throws ErrConst {

}
```

Permet au constructeur Point de lancer ErrConst

- Méthode lance exception en créant un objet

```
public Point (int a, int b) throws ErrConst {
 if ((a < 0) || (b < 0)) throw new ErrConst();
 this.x = a; this.y = b;
}
```

Permet au constructeur Point de lancer ErrConst

## Lancer ou déclencher (2)

- Méthode lance exception par intermédiaire d'un code

```
public Point (int a, int b) throws ErrorConst {
 checkXYValue(a, b);
 this.x = a; this.y = b;
}
```



```
Private void checkXYValue(int a, int b) throws ErrorConst {
 if ((a < 0) || (b < 0)) throw new ErrConst();
}
```

# Capturer ou attraper (1)

- Gestionnaire exception= traiter avec actions situations exceptionnelles
- Délimiter les instructions à problèmes par bloc  
**try { ... }**

```
try {
```

```
 Point a = new Point(1, 4);
```

```
 a.affiche();
```

```
 a = new Point (-2, 4);
```

```
 a.affiche();
```

```
}
```

Permet au constructeur Point de lancer **ErrConst**

## Capturer ou attraper (2)

- Gestionnaire risques avec  
**catch (TypeException e ) { ...}**

```
}catch(ErrConst e){
 System.out.println("Erreur de construction");
 System.exit(-1);
}
```

- Capturer exception et exécuter les tâches adéquates

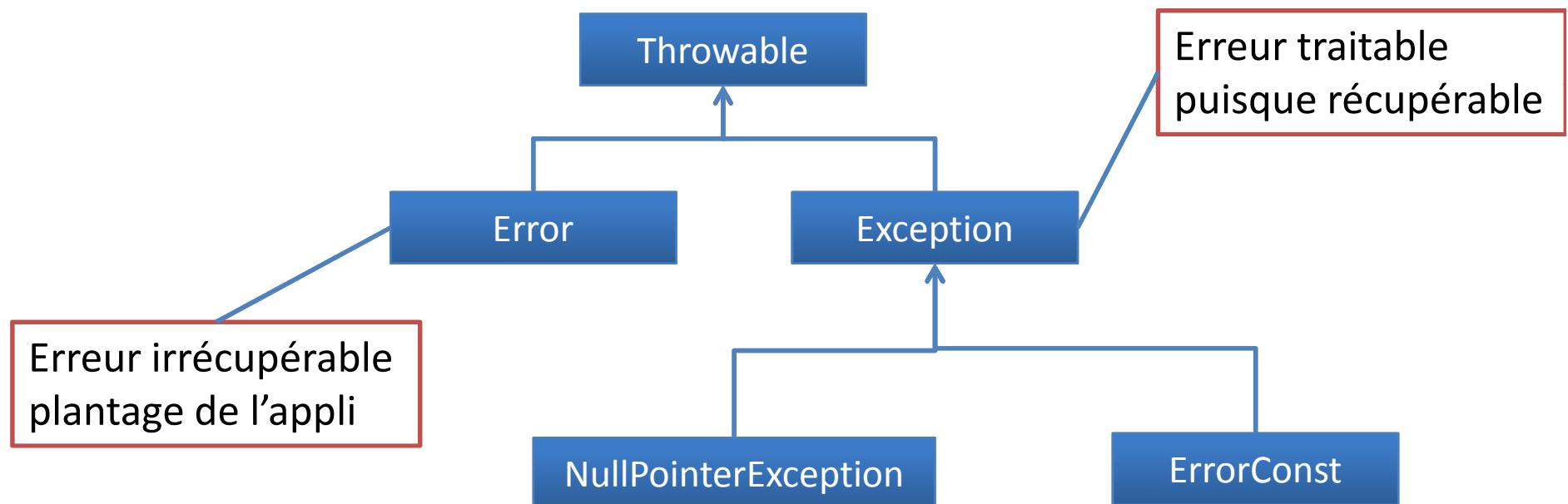
# Capturer ou attraper (3)

```
public class TestPoint {
 public void static main (String args[]) {
 try {
 Point a = new Point(1, 4);
 a.affiche();
 a = new Point (-2, 4);
 ! a.affiche();
 }catch(ErrConst e){
 System.out.println("Erreur de construction");
 System.exit(-1);
 }
 ...
 }
}
```

Poursuite de l'exécution  
du programme.  
**Remarque :** si exit(...) le  
programme s'arrête

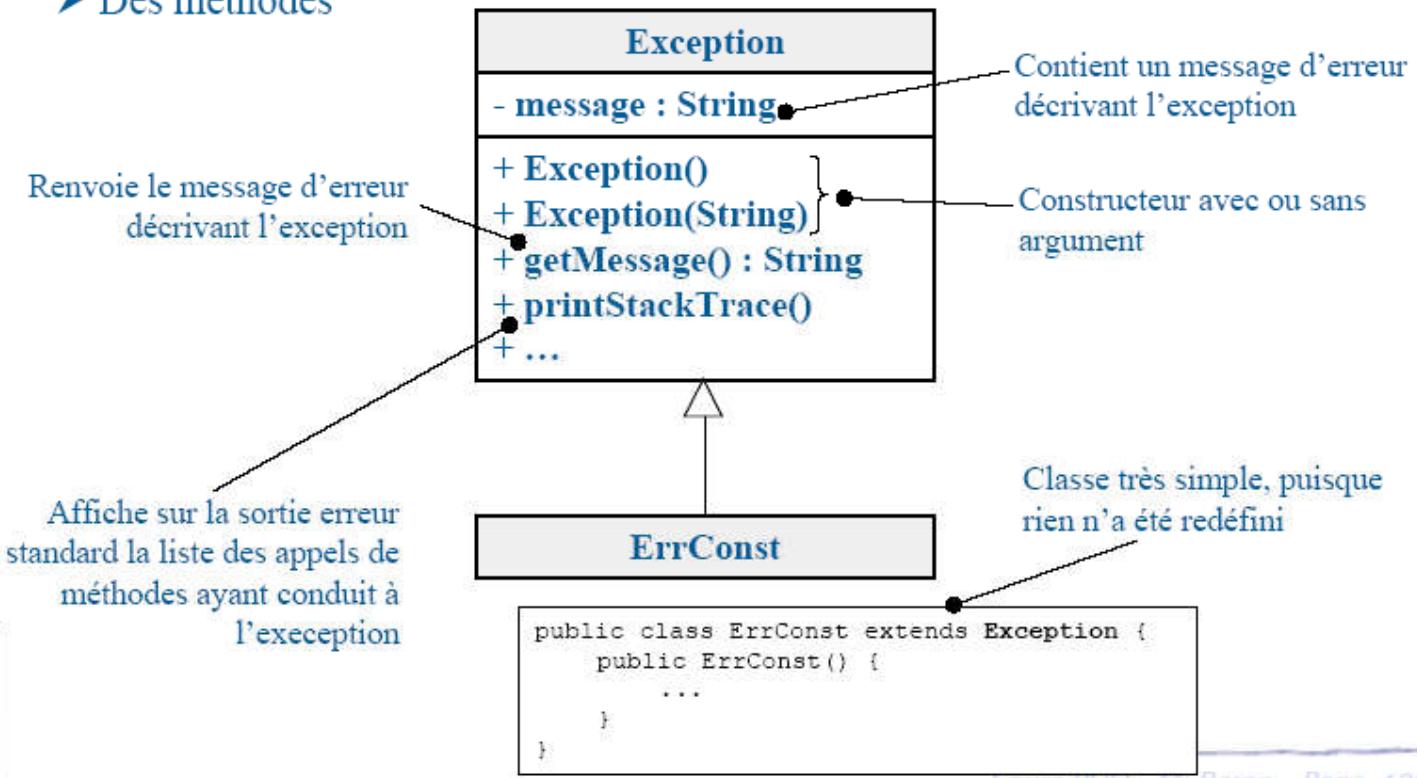
# Hiérarchie des exceptions

- Exception en java = objet
- Exception = sous-classe de *java.lang.Throwable*



# Exception : modélisation (2)

- Les exceptions sont des objets nous pouvons donc définir
  - Des attributs particuliers
  - Des méthodes



# Capturer ou attraper (3)

```
public class TestPoint {
 public static void main (String args[]) {
 try {
 ...
 }catch(ErrConst e){
 System.out.println("Erreur de construction");
 System.out.println(e.getMessage());
 e.printStackTrace();
 System.exit(-1);
 }
 ...
 }
}
```

Coordonnées : 1 4  
Erreur de construction  
ErrorConst  
at Point.<init>[Point.java:23]  
at Test.main[Test.java:32]

# Exception : finally (1)

- Bloc **finally**
  - Optionnel
  - Nettoyage
  - Indépendant du bloc **try**
- Code dont exécution garantie quoi qu'il arrive
- Intérêt double
  - Regrouper code à dupliquer au cas échéant
  - Effectuer traitement après **try** même si exception levée et non attrapée

# Exception : finally (2)

```
public class TestPoint {
 public static void main (String args[]) {
 try {
 ...
 } catch(ErrConst e){
 System.out.println("Erreur de construction");
 System.out.println("Fin du programme");
 System.exit(-1);
 } catch(ErrDepl e){
 System.out.println("Erreur de déplacement");
 System.out.println("Fin du programme");
 System.exit(-1);
 }
 }
}
```

Éviter de répéter du code

```
public class TestPoint {
 public static void main (String args[]) {
 try {
 ...
 } catch(ErrConst e){
 System.out.println("Erreur de construction");
 } catch(ErrDepl e){
 System.out.println("Erreur de déplacement");
 } finally{
 System.out.println("Fin du programme");
 System.exit(-1);
 }
 }
}
```

# Exception : pour ou contre?

```
ErreurtType lireFichier () {
 int codeErr = 0;
 //ouvrir le fichier
 if (isFileIsOpen()) {
 //déterminer la longueur du fichier
 if (getfileSize()){
 //vérification de l'alloction mémoire
 if (getEnoughMemory()){
 //lire le fihcier en mémoire
 if (readFailed()){
 codeErr = -1;
 }else{
 codeErr = -2;
 }
 }else {
 codeErr = -3;
 }
 }
 //fermer le fichier
 if (closeTheFileFailed()){
 codeErr = -4;
 }
 }else {
 codeErr = -5
 }
}
```

02/07/2018

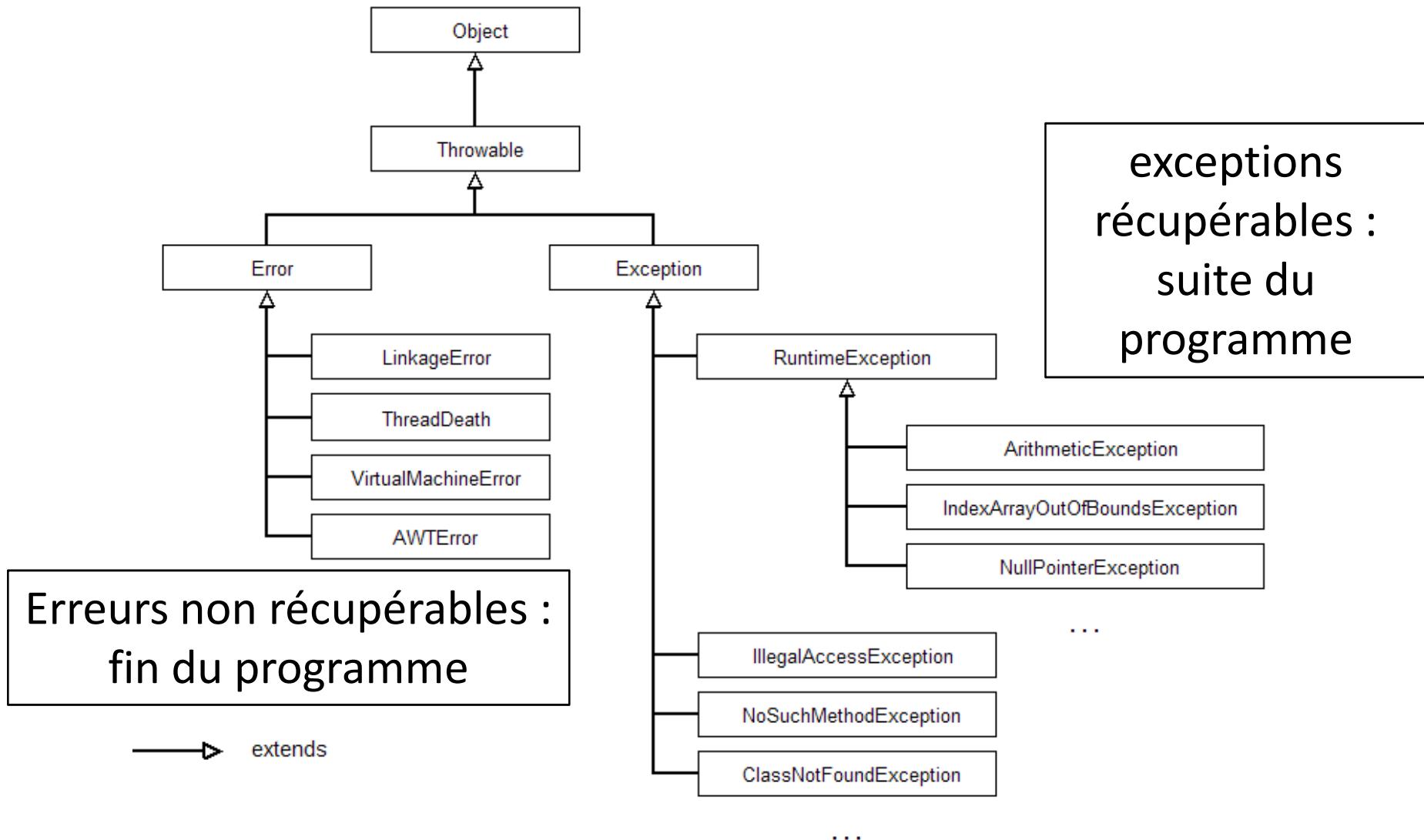
- Gestion des erreurs difficile
- Difficile de gérer les retours de fonction
- Le code devient de plus en plus conséquent

# Exception : pour ou contre?

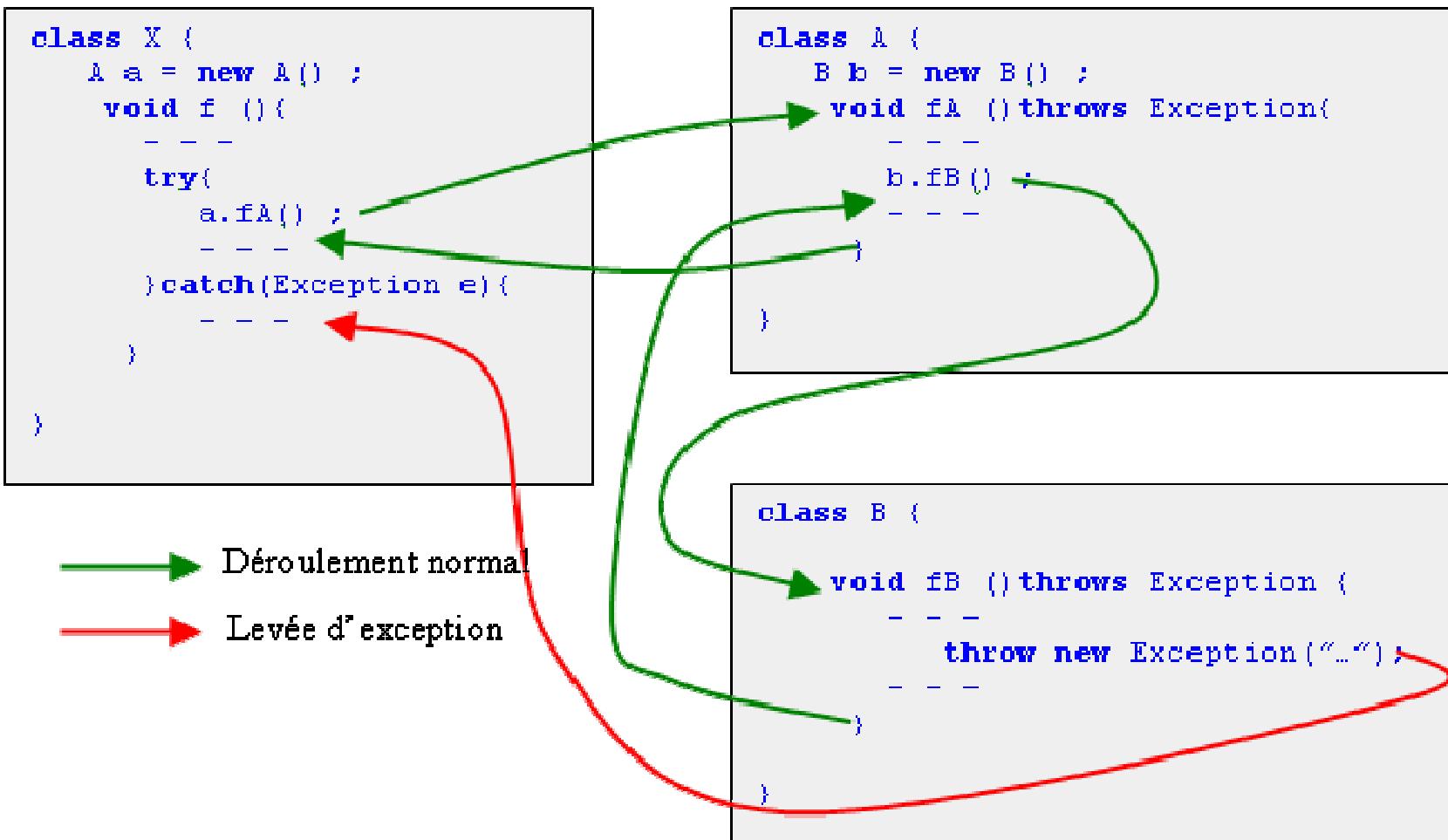
- Cossion
- Lisibilité
- Programmation propre et professionnelle

```
ErreurType lireFichier () {
 try {
 //ouvrir le fichier
 //déterminer la longueur du fichier
 //vérification de l'alloction mémoire
 //lire le fichier en mémoire
 //fermer le fichier
 }catch(FileOpenFailed){
 ...
 }
 catch(FileSizeFailed){
 ...
 } catch(FileOpenFailed){
 ...
 } catch(MemoryAllocFailed){
 ...
 } catch(FileReadFailed){
 ...
 } catch(FileCloseFailed){
 ...
 }
}
```

# Hiérarchie des erreurs et exceptions



# Déroulement d'une exécution normale et d'une exécution avec levée d'exception



# JAVADOC

02/07/2018

Pr. Mouhamadou THIAM Maître de  
conférences en informatique

350

# Objectifs

- Javadoc = un programme qui
  - Lit programme source Java
  - Produit documentation en format HTML
- Javadoc, par défaut → structure du programme
- javadoc lit les “javadoc comments” et les inclue dans la documentation

# API = Quoi ?

- Modèle de contract pour les méthodes
  - Pre-conditions
  - Post-conditions
- JavaDoc
  - Standard industriel pour la documentation des APIs
  - Covre plus que les contrats
- Comment passer d'une méthode à une autre?

# API Java

- API (Application Programmer's Interface) Java de Sun = resource importante
- A disposer à portée de main quand on programme en Java
- <http://docs.oracle.com/javase/8/docs/api/index.html>
- Diapositive suivante → tête

# En ligne...

The screenshot shows a web browser window with the URL [docs.oracle.com](http://docs.oracle.com) in the address bar. The page title is "Java™ Platform, Standard Edition 7 API Specification". The left sidebar contains navigation links for "All Classes" and "Packages", listing various Java packages like java.awt, java.awt.event, etc. The main content area displays a table of packages with their descriptions.

| Package                   | Description                                                                                                                                                                                                                      |
|---------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| java.applet               | Provides the classes necessary to create an applet and the classes an applet uses to communicate with its applet context.                                                                                                        |
| java.awt                  | Contains all of the classes for creating user interfaces and for painting graphics and images.                                                                                                                                   |
| java.awt.color            | Provides classes for color spaces.                                                                                                                                                                                               |
| java.awt.datatransfer     | Provides interfaces and classes for transferring data between and within applications.                                                                                                                                           |
| java.awt.dnd              | Drag and Drop is a direct manipulation gesture found in many Graphical User Interface systems that provides a mechanism to transfer information between two entities logically associated with presentation elements in the GUI. |
| java.awt.event            | Provides interfaces and classes for dealing with different types of events fired by AWT components.                                                                                                                              |
| java.awt.font             | Provides classes and interface relating to fonts.                                                                                                                                                                                |
| java.awt.geom             | Provides the Java 2D classes for defining and performing operations on objects related to two-dimensional geometry.                                                                                                              |
| java.awt.im               | Provides classes and interfaces for the input method framework.                                                                                                                                                                  |
| java.awt.im.spi           | Provides interfaces that enable the development of input methods that can be used with any Java runtime environment.                                                                                                             |
| java.awt.image            | Provides classes for creating and modifying images.                                                                                                                                                                              |
| java.awt.image.renderable | Provides classes and interfaces for producing rendering-independent images.                                                                                                                                                      |
| java.awt.print            | Provides classes and interfaces for a general printing API.                                                                                                                                                                      |
| java.beans                | Contains classes related to developing beans -- components based on the JavaBeans™ architecture.                                                                                                                                 |
| java.beans.beancontext    | Provides classes and interfaces relating to bean context.                                                                                                                                                                        |

# Pourquoi

- Connaissance nécessaire pour implementation propre.
- Pas un guide de programmation qui est un
  - Document très important, mais très différent
- Définition **\*contrat\*** entre fonction appelante et implémentation
- Doit être **\*indépendent\*** de l'implementation
  - Exceptions sont très indésirables
  - Mais souvent nécessaires
- Doit contenir assez d'instructions pour le tester

# Comment ?

- Commentaires ordinaire : `/* any text */`
- javadoc commentairess: `/** any text */`
- `/**` commentaires sur une ligne `*/`
- `/**  
 * Multi-line comments are usually  
 * written like this; the stars at the  
 * front of lines are ignored.  
 */`

# Éléments à commenter

- javadoc commentaires pour
  - Classes
  - Interfaces
  - Champs (variables)
  - Méthodes
  - Constructeurs
  - Interfaces
- javadoc ignore commentaires internes  
(devant expressions, blocs, etc.)

# Où placer les commentaires

- Commentaires javadoc *juste avant* classe, champ, méthode, constructeur, ou interface
- Rien sauf espace blanc entre javadoc commentaire l'objet à commenter
- Mal placé commentaires javadoc ignorés

# Exemple

```
/** This is a comment for variable max */
double max;
double min;
/** This comment is for avg */
double avg;
/** This comment is ignored. */
//
class Something { ... }
```

## **Field Detail**

**max**

**private double max**

This is a comment for variable max

**min**

**private double min**

**avg**

# Exemple 2

```
/**
 * Returns an Image object that can then be painted on the screen.
 * The url argument must specify an absolute {@link URL}. The name
 * argument is a specifier that is relative to the url argument.
 * <p>
 * This method always returns immediately, whether or not the
 * image exists. When this applet attempts to draw the image on
 * the screen, the data will be loaded. The graphics primitives
 * that draw the image will incrementally paint on the screen.
 *
 * @param url an absolute URL giving the base location of the image
 * @param name the location of the image, relative to the url argument
 * @return the image at the specified URL
 * @see Image
 */
public Image getImage(URL url, String name) {
 try { return getImage(new URL(url, name));
 } catch (MalformedURLException e) { return null; }
}
```

## **Method Detail**

### **getImage**

```
public java.awt.Image getImage(java.net.URL url,
 java.lang.String name)
```

Returns an Image object that can then be painted on the screen. The url argument must specify an absolute `toString()`. The name argument is a specifier that is relative to the url argument.

This method always returns immediately, whether or not the image exists. When this applet attempts to draw the image on the screen, the data will be loaded. The graphics primitives that draw the image will incrementally paint on the screen.

**Parameters:**

`url` - an absolute URL giving the base location of the image

`name` - the location of the image, relative to the `url` argument

**Returns:**

the image at the specified URL

**See Also:**

[Image](#)

# Sortie formatée

## **getImage**

public [Image](#) getImage([URL](#) url, [String](#) name)

Returns an [Image](#) object that can then be painted on the screen. The url argument must specify an absolute [URL](#). The name argument is a specifier that is relative to the url argument. This method always returns immediately, whether or not the image exists. When this applet attempts to draw the image on the screen, the data will be loaded. The graphics primitives that draw the image will incrementally paint on the screen.

Parameters:

- url - an absolute URL giving the base location of the image
- name - the location of the image, relative to the url argument

Returns:

[the image at the specified URL](#)

See Also:

[Image](#)

# Html alors

- Spécialiste du HTML, ajouter des commandes HTML dans commentaires javadoc
- Possibilité d'utiliser bold, italic, paragraph, différents types de listes, liens hypertexte, images, etc.
- Commandes de structuration de document, comme `<head>`, `<h2>`, or `<hr>`

# Tags (étiquettes) spéciaux

- Commencent avec @ et placé début de ligne
- Tags décrivent : paramètres, types retour, methods connexes, etc.
- Toujours utiliser @author tag pour
  - classes et interfaces seulement, required
  - Example:
    - @author Ndeye Rong Ming

# Tags (étiquettes) spéciaux

@param (méthodes et constructeurs seulement)

@return (méthodes seulement)

@exception (@throws son synonyme depuis Javadoc 1.2)

@see (références additionnelles)

@since (depuis version/quand disponible?)

@serial (or @serialField or @serialData)

@deprecated (pourquoi il est obsolète, depuis quand, alternatif à utiliser)

# Exécuter CMD javadoc

- Syntaxe de javadoc :  
`-author -private *.java`
- Le paramètre
  - `-author` : ne pas ignorer commentaires `@author`
  - `-private` : documenter les éléments `private`, `package`, and `protected` comme ceux `public`

# Paramètres ...

- **javadoc**
  - dispose de plein d'options et très flexible
  - génère plusieurs fichiers HTML, non un seul
- Cependant après avoir lancé **javadoc**, vous devez regarder les sorties pour vous assurer qu'elles correspondent à vos attentes

# Paramètres ...

- Options basiques de **javadoc** :
  - author* - generated documentation will include a author section
  - classpath [path]* - specifies path to search for referenced .class files.
  - classpathlist [path];[path];...;[path]* - specifies a list locations (separated by ";") to search for referenced .class files.
  - d [path]* - specifies where generated documentation will be saved.
  - private* - generated documentation will include private fields and methods (only public and protected ones are included by default).
  - sourcepath [path]* - specifies path to search for .java files to generate documentation form.
  - sourcepathlist [path];[path];...;[path]* - specifies a list locations (separated by ";") to search for .java files to generate documentation form.
  - version* - generated documentation will include a version section

# Plus...

- Outil javadoc ne documente pas
  - ✓ Directement les classes anonymes
  - ✓ Déclarations et commentaires ignorés.
- <http://www.oracle.com/technetwork/java/javase/documentation/index-137868.html>

# Exo d'application

Voici une interface définissant un type abstrait "Pile de <A>" avec les fonctionnalités classiques :

```
public interface IPile<A> {
 boolean estVide();
 void empile(A a);
 A depile(); // retourne l'élément en sommet
de pile et dépile
 int nbElements();
 A sommet(); // retourne le sommet de pile
mais ne le dépile pas
}
```

## A faire 1

1. Ecrivez une classe générique **CPile** qui implémente l'interface **IPile**. Vous stockerez les éléments de la pile dans une liste chaînée (instance de **java.util.LinkedList**, voir en *annexe* quelques méthodes publiques de cette classe).

## A faire 2

2. Ecrivez un petit programme qui crée et manipule des piles en instanciant la classe générique de différentes façons (par exemple pile de String, pile de Integer, d'animaux, ...).

## Exo d'application 2

- La classe suivante **Tableau** encapsule un tableau d'entiers et comporte deux méthodes permettant de trier le tableau et d'afficher son contenu. La méthode main de cette classe fournit un exemple d'utilisation. On voudrait disposer d'une méthode de tri *pour n'importe quel type de tableau*. Proposez une solution basée sur la généricité.

# Plan

1. Rappels POO et Java
2. Classe abstraite, Interface et Type générique
3. Exceptions Java et Javadoc
- 4. Swing: Création d'interfaces graphiques**
5. Collections
6. Entrées/Sorties (Fichiers Textes et binaires)
7. JDBC
8. Compléments

# AWT

## The Abstract Windowing Toolkit

# Objectifs

- Notions de graphisme
- Éléments de AWT
- Création d'interface avec AWT
- Interception des actions de l'utilisateur
- Création d'interface avec Swing
- **Approfondissement : SWT et JFace (IBM)**

# Introduction

- Dessiner avec AWT
- API AWT pour créer des interfaces graphiques
- API Swing.
- 2 API à combiner pour développer des
  - applications ou
  - applets

# Dessiner avec Java

# Classe **Graphics**

- Contient outils nécessaires pour dessiner.
- Est **abstraite** et
- Ne possède pas de constructeur public
  - Impossible de construire des instances de **Graphics** nous même.
  - SE fournit instances nécessaires
  - SE instancie grâce à la JVM une sous-classe de **Graphics** dépendante de la plate-forme utilisée.

# Dessiner avec l'AWT (1)

- Rendu déclenché par le *système*
  - Le système demande à un composant de se **peindre** quand :
    - Composant rendu visible la première fois sur l'écran
    - Composant redimensionné
    - Rendu du composant endommagé et doit être réparé (par exemple, quelque chose qui cachait une partie du composant a été déplacé. Cette partie à nouveau visible doit être repeinte)

## Dessiner avec l'AWT (2)

- Rendu déclenché par l'application
  - Le composant décide de se repeindre pour refléter un changement dans son état interne

## Dessiner avec l'AWT (3)

- La demande de repeinte du composant
  - demandée par l'intermédiaire de la méthode public void paint (Graphics g)
    - qu'elle soit déclenchée par le système ou par l'application
- Donc le code indiquant ce qu'il faut peindre dans un composant doit être placée dans cette méthode (en la redéfinissant)

## Dessiner avec l'AWT (4)

- Quand AWT appelle la méthode `paint`
  - objet `Graphic` est pré-configuré avec un état approprié pour dessiner sur ce composant particulier
  - objet `color` de cet objet `Graphics` prend pour valeur celle du `foreground` du composant

## Dessiner avec l'AWT (4)

- Quand AWT appelle la méthode `paint`
  - objet font celle de la propriété font du composant
  - objet translation est fixé de sorte que les coordonnées (0, 0) représente le coin supérieur gauche du composant
  - objet clip rectangle est fixé à la zone qui a besoin d'être repeinte

## Dessiner avec l'AWT (5)

- Généralement éviter de placer du code servant pour le dessin/redessin des composants en dehors de la méthode paint()
  - pour éviter que ce genre de code soit appelé à un moment inapproprié (par exemple avant que le composant ne soit visible ou ait accès à un objet Graphics valide)

# Dessiner avec l'AWT (6)

- Il faut également éviter dans un programme de faire un appel direct à la méthode paint()
  - L'AWT fournit un certain nombre de méthodes pour appeler la méthode paint() indirectement
    - **public void repaint ()**
    - **public void repaint (long tm)**
    - **public void repaint (int x, int y, int width, int height)**
    - **public void repaint (long tm, int x, int y, int width, int height)**

## Dessiner avec l'AWT (7)

- Opérations réalisées dans un rendu déclenché par le système
  - l'AWT détermine si le re-dessin concerne tout ou partie du composant
  - l'AWT appelle la méthode `paint()` sur le composant

# Dessiner avec l'AWT (8)

- Opérations réalisées dans un rendu déclenché par une application
  - le programme détermine si le re-dessin concerne tout ou partie du composant en réponse à un changement d'état interne
  - le programme appelle repaint() sur le composant qui a besoin d'être repeint.
  - l'AWT provoque l'appel de la méthode update() sur le composant
  - si ce composant ne redéfinit pas update(), l'implémentation par défaut d'**update()** nettoie l'arrière plan du composant et appelle paint() .

# Résumé (I)

- Généralement code pour le rendu graphique du composant placé dans la méthode paint().
- Les programmes peuvent faire déclencher un futur appel à paint() en appelant repaint() mais ne doit généralement pas appeler paint() directement.
- Sur des composants avec un rendu complexe, la méthode repaint() devrait être invoquée avec des arguments pour définir un rectangle de clipping

## Résumé (II)

- Appel de repaint() provoque appel à update() suivi d'un appel à paint() par défaut, les composants lourds peuvent redéfinir update() pour réaliser du « dessin incrémental » (le dessin incrémental n'est pas possible avec les composant légers)
- Classes dérivées de java.awt.Container qui redéfinissent paint() devrait toujours invoquer dans le corps de cette méthode super.paint() pour s'assurer que tous ses composants sont redessinés.
- Composants avec un rendu graphique complexe devraient utilisées le **rectangle de clipping** pour restreindre la zone à redessiner

# **OPERATIONS SUR LE CONTEXTE GRAPHIQUE**

# Tracé formes géométriques (1)

- Carré ou rectangle
  - `drawRect(x, y, largeur, hauteur)`
  - `fillRect(x, y, largeur, hauteur)`
- Carré ou rectangle aux angles arrondis
  - `drawRoundRect(x, y, largeur, hauteur, hor_arr, ver_arr)`
  - `fillRoundRect(x, y, largeur, hauteur, hor_arr, ver_arr)`

# Tracé formes géométriques (2)

- Ligne
  - `drawLine(x1, y1, x2, y2)`
- Cercle ou ellipse spécifiant le rectangle dans lequel ils s'inscrivent
  - `drawOval(x, y, largeur, hauteur)`
  - `fillOval(x, y, largeur, hauteur)`
- polygone ouvert ou fermé
  - `drawPolygon(int[], int[], int)`
  - `fillPolygon(int[], int[], int)`

# Tracé formes géométriques (3)

- Arc d'ellipse inscrit dans rectangle ou carré
  - `drawArc(x, y, largeur, hauteur, angle_deb, angle_bal)`
  - `fillArc(x, y, largeur, hauteur, angle_deb, angle_bal);`
- Tracé de texte
  - `g.drawString(texte, x, y );`

# Utilisation des fontes

- Police de caractères particulière
  - Font fonte = new Font("TimesRoman", Font.BOLD, 30);
  - nom de la police, le style (**BOLD**, *ITALIC*, PLAIN ou 0,1,2) et la taille des caractères en points
- Plusieurs styles, il suffit de les additionner
  - Font.BOLD + Font.ITALIC
  - getName() de **Font** → le nom de la fonte.
  - setFont() de **Graphics** → changer la police
- Polices : Dialog, Helvetica, TimesRoman, Courier, ZapfDingBats

# Couleur

- `setColor()` fixer, à posteriori, la couleur
  - `g.setColor(Color.black);`
- Couleurs :
  - green, blue, red, white, black, ...

# Méthodes complémentaires

- Chevauchement de figures
  - `setXORMode`(couleur alternative) : Dans ce cas, la couleur de l'intersection prend une autre couleur.
- Effacement d'une aire
  - `clearRect(x1, y1, x2, y2)` dessine un rectangle dans la couleur de fond courante.
- Copie d'une aire rectangulaire
  - `CopyArea(x1, y1, x2, y2, dx, dy)`. `dx` et `dy` spécifient un décalage en pixels de la copie par rapport à l'originale.

# **AWT – COMPOSANTS LOURDS**

# Introduction

- Classes de AWT (Abstract Windows Toolkit)  
interfaces graphiques indépendantes du OS.
- Librairie utilise le système graphique de la plate-forme (Windows, MacOS, X-Window)
- Toolkit contient classes décrivant les composants graphiques, les polices, les couleurs et les images.

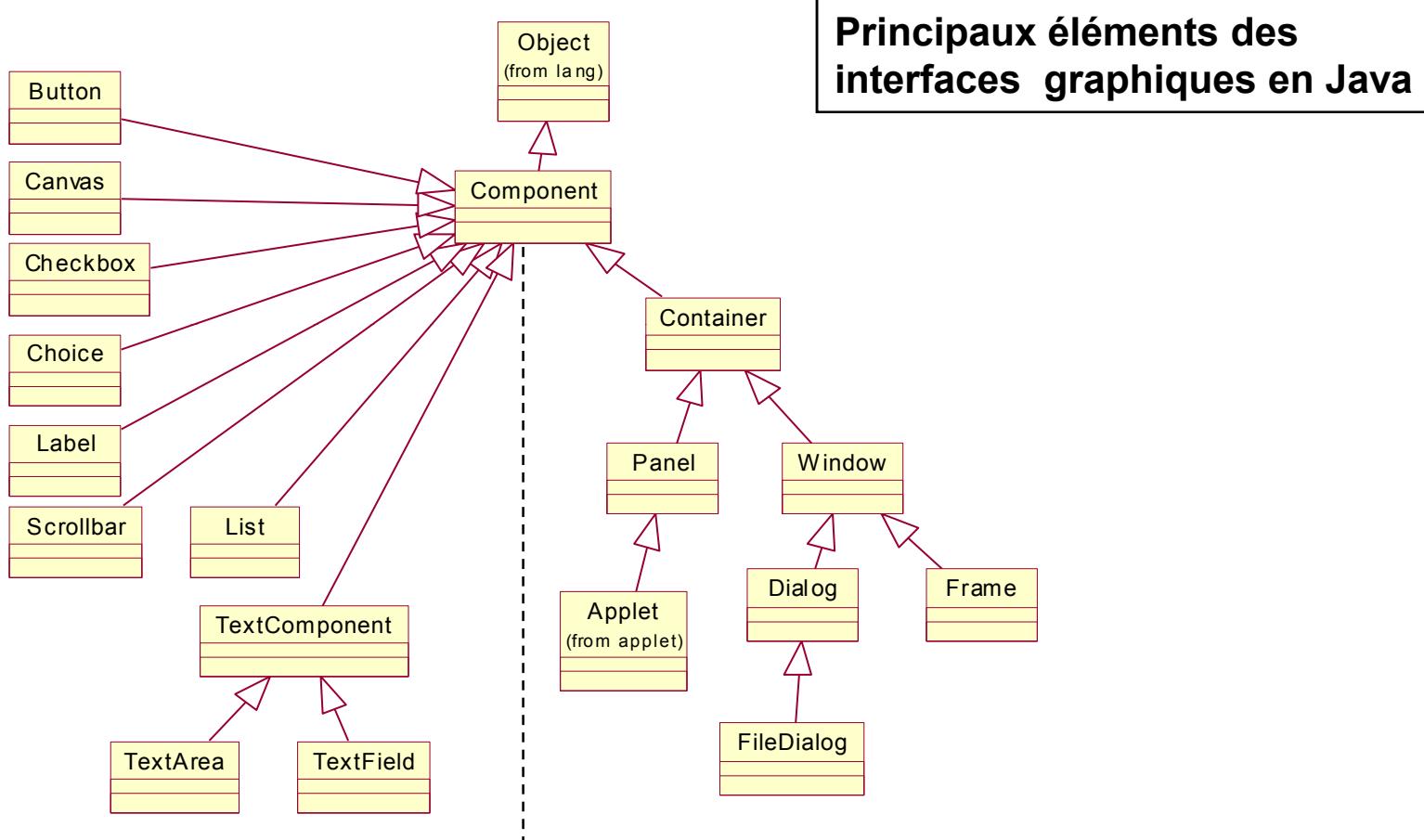
# Composants

- Une interface graphique en Java est un assemblage
  - conteneurs (*Container*) et de
  - composants (*Component*).
- **Un composant** est une partie "visible" de l'interface utilisateur Java.
  - C'est une sous-classes de la classe abstraite `java.awt.Component`.
  - Exemple : les boutons, les zones de textes ou de dessin, etc.

# Conteneurs

- **Un conteneur** est un espace dans lequel on peut positionner plusieurs composants.
  - Sous-classe de la classe `java.awt.Container`
  - La classe Container est elle-même une sous-classe de la classe Component
  - Par exemple les fenêtres, les applets, etc.

# Hiérarchie d'héritage



# Conteneurs

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

# Ajout de composant

- Ajout composant dans un conteneur : méthode add()

*Panel p = new Panel();*

*Button b = new Button();*

*p.add(b);*

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

*p.remove(b);*

# Dimensions d'un composant

- Un composant a (notamment) une taille :
  - préférée : `getPreferredSize()`
  - minimum : `getMinimunSize()`
  - maximum : `getMaximumSize()`

# Exemple

```
import java.awt.*;

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



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

Création du bouton ayant pour label « coucou »

Ajout du bouton dans la fenêtre

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

# Gestionnaire de présentation (1)

- A chaque conteneur est associé un gestionnaire de présentation (*layout manager*)
- Le gestionnaire de présentation gère
  - positionnement et
  - (re)dimensionnement des composants d'un conteneur.

# Gestionnaire de présentation (2)

- Réagencement des composants dans un conteneur a lieu lors de :
  - la modification de sa taille,
  - le changement de la taille ou le déplacement d'un des composants.
  - l'ajout, l'affichage, la suppression ou le masquage d'un composant.
- Principaux gestionnaires de présentation de l'AWT sont :
  - `FlowLayout`, `BorderLayout`, `GridLayout`,  
`CardLayout`, `GridBagLayout`

# Gestionnaire de présentation (3)

- Conteneur possède un gestionnaire de présentation par défaut.
  - Tout instance de *Container* → instance de *LayoutManager*.
  - Changer de gestionnaire → méthode **setLayout()**.
- Gestionnaires de présentation par défaut :
  - BorderLayout pour Window et ses descendants (Frame, Dialog, ...)
  - Le FlowLayout pour Panel et ses descendants (Applet, etc.)
- Gestionnaire de présentation fonctionne "tout seul" en interagissant avec le conteneur.

# **FLOWLAYOUT**

# FlowLayout (1)

- Plus simple des managers de l'AWT
- Gestionnaire de présentation utilisé par défaut dans les Panel si aucun LayoutManager spécifié.
- Un FlowLayout peut spécifier :
  - justification à gauche, à droite ou centrée,
  - espacement horizontal ou vertical entre 2 composants.
  - Par défaut, les composants sont centrés à l'intérieur de la zone qui leur est allouée.

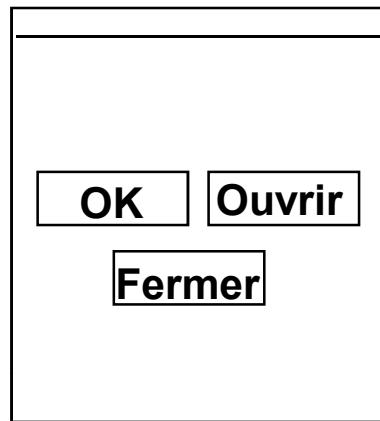
# Constructeurs

- `FlowLayout( );`
- `FlowLayout( int align);`
- `FlowLayout( int align, int hgap, int vgap);`

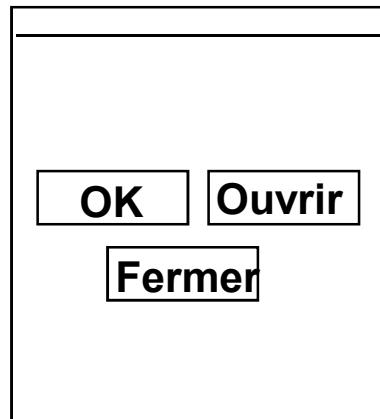
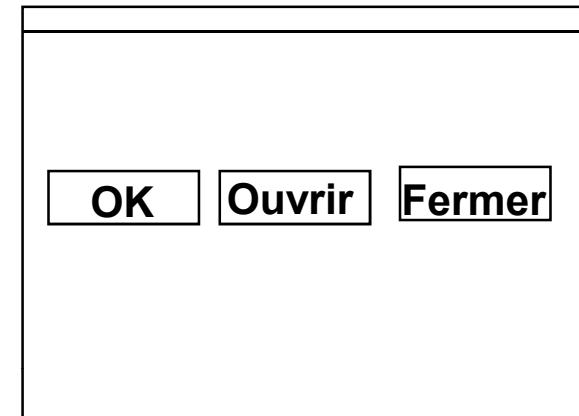
# Stratégie FlowLayout

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

# Redimensionnement FlowLayout



Redimensionnement



Redimensionnement



plus visible

# Redimensionnement ...



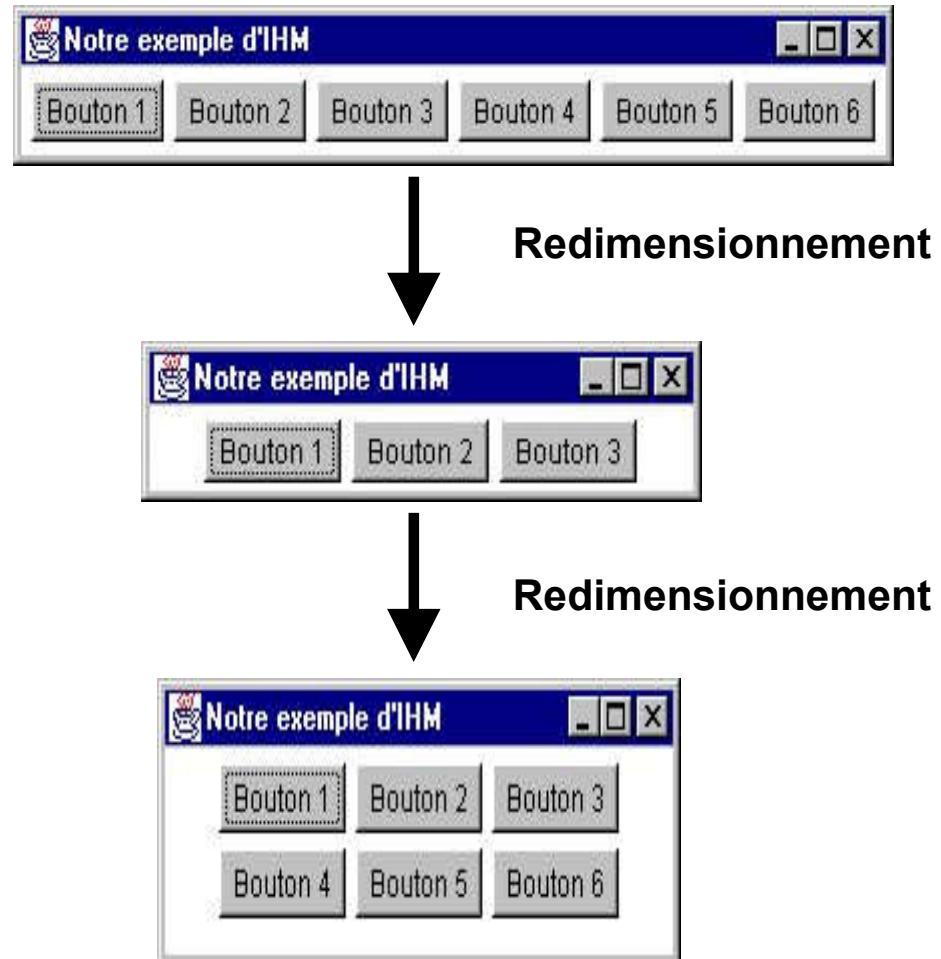
**Redimensionnement**



**Redimensionnement**



# Redimensionnement ...



# Remarques

- **FlowLayout**
  - Cacher réellement et effectivement les composants qui ne rentrent pas dans le cadre.
  - Utiliser quand il y a peu de composants.
  - Positionner les composants ligne par ligne : chaque fois qu'une ligne est remplie, une nouvelle ligne est commencée.
- Équivalent vertical du FlowLayout n'existe pas
- N'impose pas de taille aux composants mais leur permet d'avoir la taille qu'ils préfèrent.

# **BORDERLAYOUT**

# BorderLayout (1)

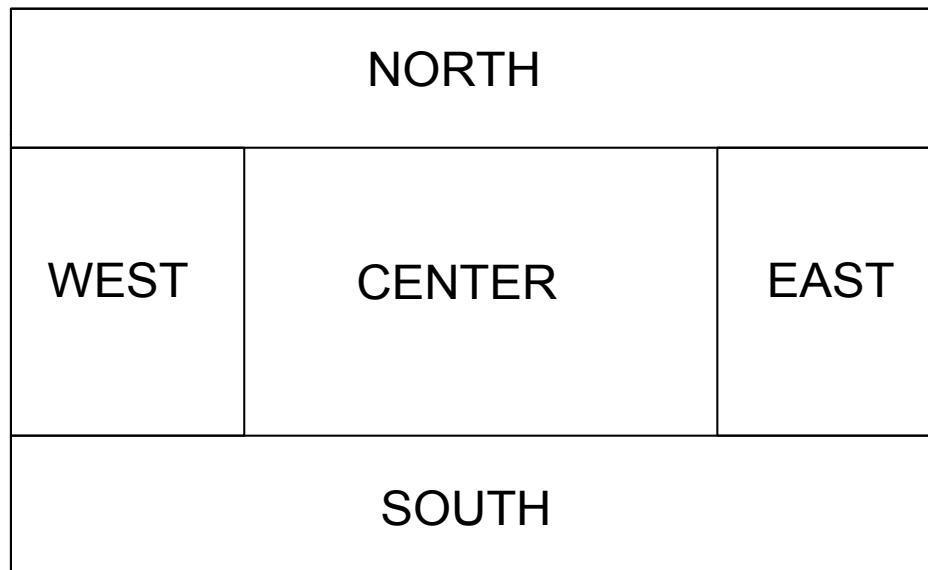
- Divise son espace de travail en cinq zones géographiques : *North*, *South*, *East*, *West* et *Center*.
- Les composants sont ajoutés par nom à ces zones (un seul composant par zone).
  - **Exemple**

```
add("North", new Button("Le bouton nord !));
```

- Si une des zones de bordure ne contient rien, sa taille est 0.

# BorderLayout (2)

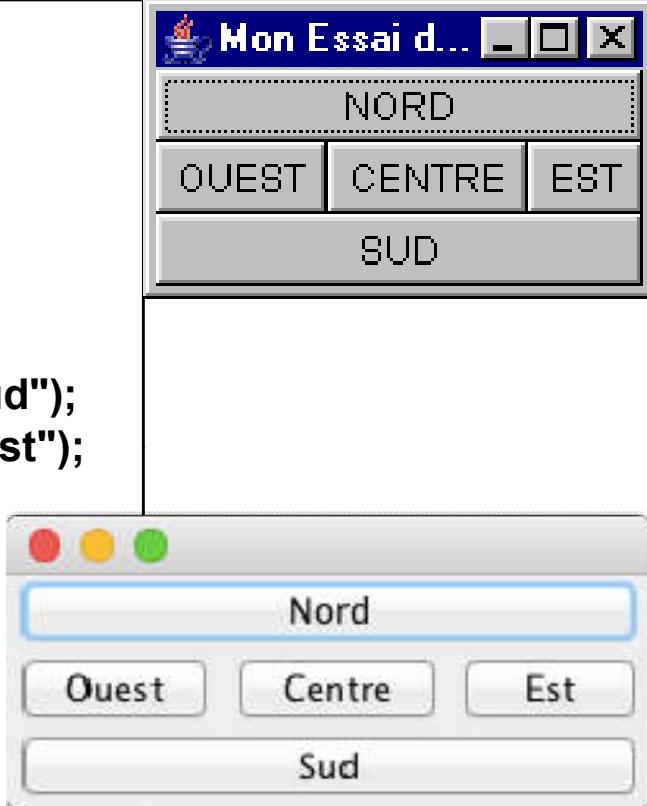
- Division de l'espace avec le **BorderLayout**



# Exemple

```
import java.awt.*;

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



# Stratégie de disposition (1)

- Si composant dans la partie **NORTH**,
  - récupère sa *taille préférée*,
  - respecte sa *hauteur préférée* et
  - fixe (si possible) sa largeur à la totalité de la largeur disponible de l'objet Container.
- Si composant dans la partie **SOUTH**,
  - fait pareil que dans le cas de la partie **NORTH**.

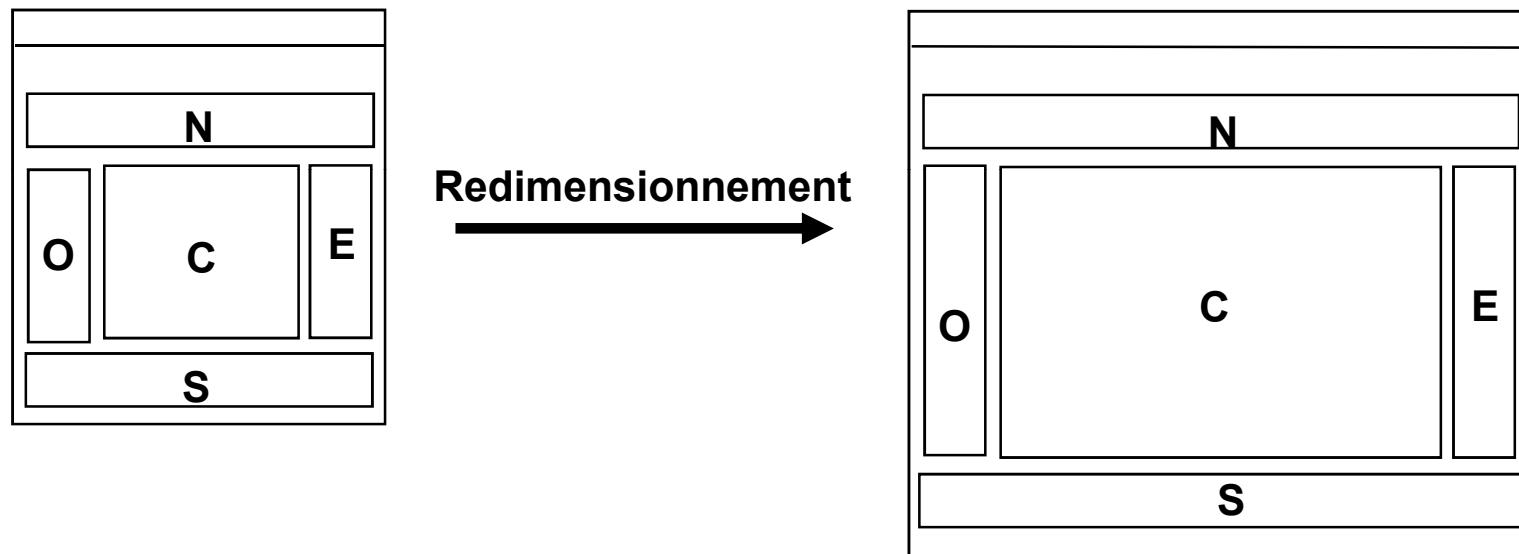
# Stratégie de disposition (2)

- Si composant dans la partie **EAST**,
  - récupère sa **taille préférée**,
  - respecte sa ***largeur préférée*** et
  - fixe (si possible) sa hauteur à la totalité de la hauteur encore disponible.
- Si composant dans la partie **WEST**,
  - fait pareil que dans le cas de la partie **EAST**.
- Si composant dans la partie **CENTER**, il lui donne la place qui reste, s'il en reste encore.

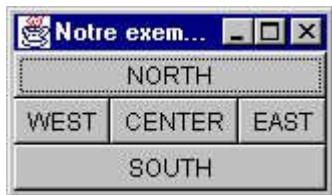
# Redimensionnement BorderLayout

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

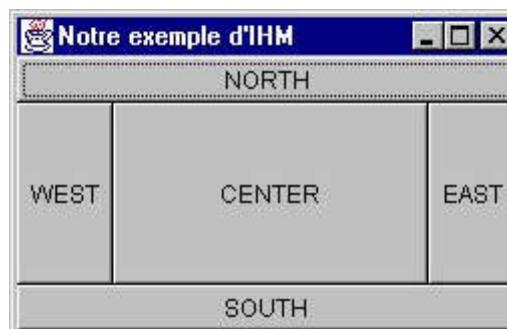
# Redimensionnement ...



# Redimensionnement ...



**Redimensionnement**



**Redimensionnement**



# GRIDLAYOUT

# GridLayout

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

# Exemple

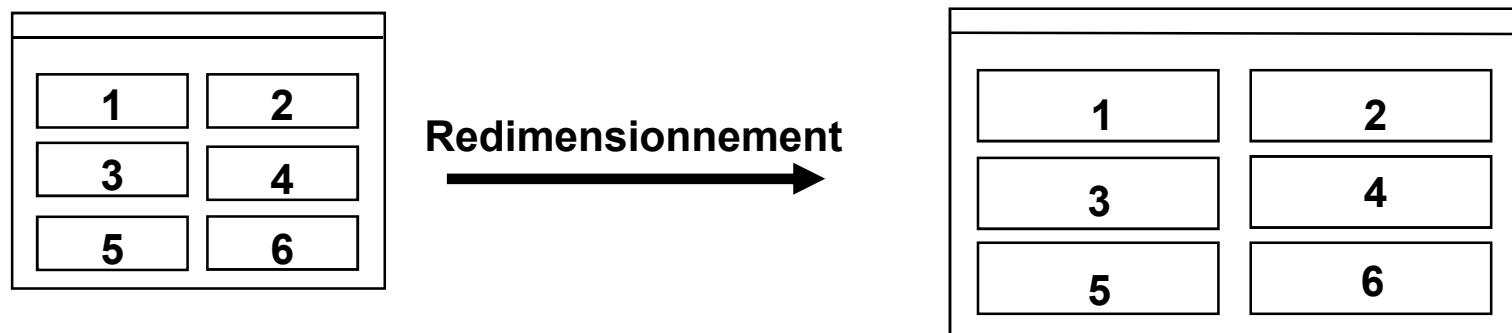
```
import java.awt.*;
public class AppliGridLayout extends Frame
{
 public AppliGridLayout()
 {
 super("AppliGridLayout");
 this.setLayout(new GridLayout(3,2));
 for (int i = 1; i < 7; i++)
 add(new Button(Integer.toString(i)));
 this.pack();
 this.show();
 }

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



# Redimensionnement ...

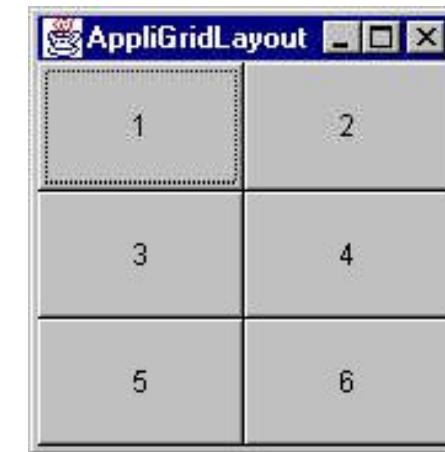
- Redimensionnement → composants changent tous de taille mais leurs positions relatives ne changent pas.



# Redimensionnement ...



Redimensionnement



# CARDLAYOUT

02/07/2018

Pr. Mouhamadou THIAM Maître de  
conférences en informatique

433

# CardLayout

- Affiche un seul composant à la fois :
  - Composants considérées comme empilées (tas de cartes).
- Permet à plusieurs composants de partager le même espace d'affichage (seul un d'entre eux visible à la fois)
- Utilise *add(String cle, Component monComposant)* ajouter un composant à un conteneur utilisant un CardLayout
- Permet de passer de l'affichage d'un composant à un autre en appelant les méthodes **first**, **last**, **next**, **previous** ou **show**

# **GRIBAGLAYOUT**

# GridLayout (1)

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

# GridBagLayout (2)

- Il est associé à un objet `GridBagConstraints`
  - l'objet **GridBagConstraints** définit des contraintes de
    - positionnement,
    - alignements,
    - taille, etc.
  - d'un composant dans un conteneur géré par un **GridBagLayout**.

# GridLayout (3)

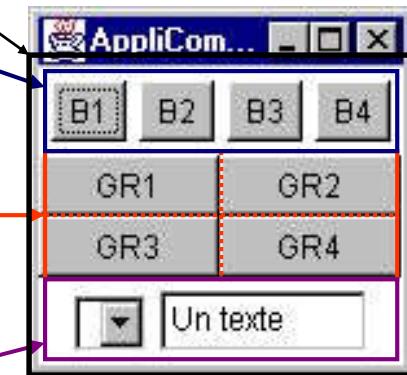
- Il est associé à un objet **GridBagConstraints**
  - On associe chaque composant que l'on place dans le **GridLayout** avec un objet **GridBagConstraints**
    - Un même objet **GridBagConstraints** peut-être associé à plusieurs composants.
    - Définir les objets **GridBagConstraints** en spécifiant les différents paramètres est assez fastidieux...
    - Voir la doc

# Mise en forme complexe

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

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

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



# D'autres gestionnaires?

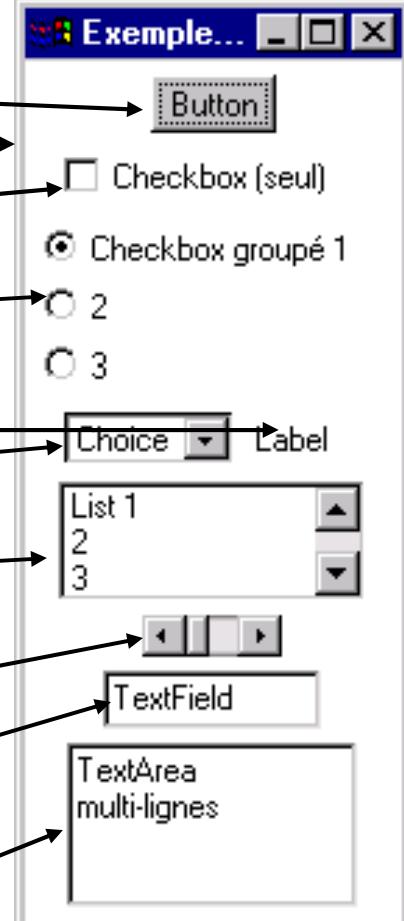
- On peut imposer à un objet « container » de n'avoir pas de gestionnaire en fixant son LayoutManager à la valeur `null`
  - `Frame f = new Frame(); f.setLayout(null);`
  - A la charge alors du programmeur de positionner chacun des composants « manuellement » en indiquant leur position absolue dans le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

# Récapitulatif

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

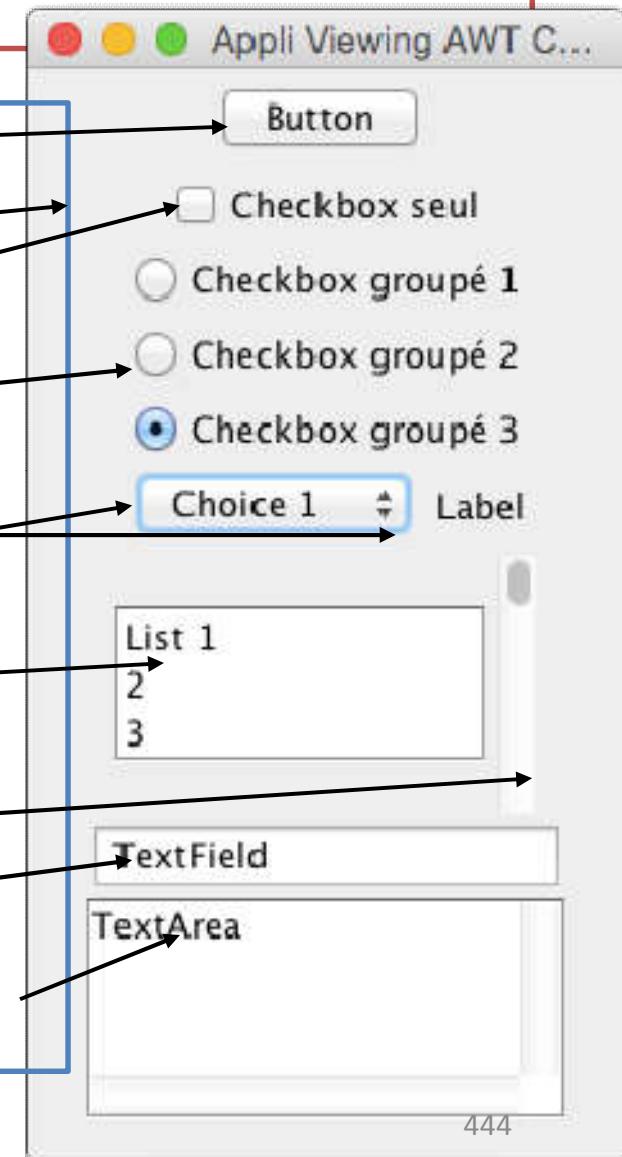
# **COMPOSANTS GRAPHIQUES**

# Les composants graphiques (1)

- Button
  - Canvas (zone de dessin)
  - Checkbox (case à cocher)
  - CheckboxGroup
  - Label
  - Choice (Sélecteur)
  - List
  - Scrollbar (barre de défilement)
  - TextField (zone de saisie d'1 ligne)
  - TextArea (zone de saisie multilignes)
- 
- The screenshot shows a window titled "Exemple..." containing the following components:
- A "Button" component.
  - A "Checkbox (seul)" (unselected).
  - A "Checkbox groupé 1" with three radio buttons: "2" (selected) and "3".
  - A "Choice" dropdown menu with items "List 1", "2", and "3".
  - A "Label" component next to the choice dropdown.
  - A "List" component containing items "List 1", "2", and "3", with scroll bars on the right.
  - A "Scrollbar" component with arrows at the bottom.
  - A "TextField" component with three small buttons to its left.
  - A "TextArea" component labeled "TextArea multi-lignes".

# Les composants graphiques (1)

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



# Button (1)

- Composant d'interface utilisateur de base de type "appuyer pour activer".
- Étiquette (un label) précisant son rôle à l'utilisateur.
- Objet de la classe **Button** est une source d'**ActionEvent**

# Button (2)

- Écouteurs associés aux objets de Button doivent implémenter interface **ActionListener**
- Interface **ActionListener** une seule méthode (*public void actionPerformed(ActionEvent e)*).  
**Button b = new Button ("Sample") ;**  
**add (b) ;**  
**b.addActionListener (...);**

# CheckBox (1)

- La case à cocher fournit un dispositif d'entrée "actif / inactif" accompagné d'une étiquette de texte.
- La sélection ou la désélection est notifiée par un *ItemEvent* à un écouteur implémentant l'interface *ItemListener*.

## CheckBox (2)

- méthode *getStateChange()* de **ItemEvent** retourne une constante :  
    ItemEvent.DESELECTED  
    ItemEvent.SELECTED.
- méthode *getItem()* de **ItemEvent** retourne la chaîne contenant l'étiquette de la case à cocher considérée.  
**Checkbox one = new Checkbox("One", false);  
        add(one);  
one.addItemListener(...);**

# CheckboxGroup

- regrouper des cases à cocher dans un **CheckboxGroup** → comportement de type boutons radio
  - Sélectionner une seule case du groupe en même temps.
  - Sélectionner une case → toute autre case déjà cochée sera désélectionnée

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

# Choice (1)

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

# Choice (2)

- Cet objet est source de **ItemEvent**, l'écouteur correspondant étant un **ItemListener**

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

# Label

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

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

# List (1)

- Objet de classe **List** permet de présenter plusieurs options de texte parmi lesquelles sélectionner
  - un ou
  - plusieurs éléments.
- Source de **ActionEvent** et d'**ItemEvent**

# List (2)

- méthode
  - *String getSelectedItem()* récupérer l’item sélectionné.
  - *String[] getSelectedItems()* récupérer des items.

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

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

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

# TextField

- dispositif d'entrée texte avec une seule ligne
  - source de **ActionEvent**
  - Possibilité d'être défini comme éditables ou non.
  - Méthodes
    - ***void setText(String text)*** : mettre du texte du **TextField**
    - ***String getText()*** : récupérer le texte du **TextField**

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

Texte par défaut mis  
dans le **TextField**

Nombre de caractères visibles  
dans le **TextField**

# TextArea

- dispositif d'entrée de texte
  - Multi-lignes, multi-colonnes
  - Présence éventuelle ou non de «scrollbars» horizontal et/ou vertical.
  - Possibilité d'être ou non éditable.
  - Méthode `setText()`, `getText()` et `append()` (pour ajouter du texte à la fin d'un texte existant déjà dans le TextArea)

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

Texte par défaut mis  
dans le TextArea

Nombre de lignes

Nombre de colonnes  
(en nbre de caractères)

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

# Menu

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

# Exemple: Menu

```
public class MaFrameWithMenu extends
Frame {

 public MaFrameWithMenu() {
 super();
 this.setTitle(" Titre de la Fenetre ");
 this.setSize(3000, 1500);

 MenuBar mb = new MenuBar();
 this.setMenuBar(mb);

 Menu m = new Menu(" un menu ");
 mb.add(m);

 m.add(new MenuItem(" 1er element "));
 m.add(new MenuItem(" 2eme element
"));
 }
}
```

```
 Menu m2 = new Menu(" sous menu ");
 CheckboxMenuItem cbm1 = new
 CheckboxMenuItem(" menu item 1.3.1 ");
 m2.add(cbm1);
 cbm1.setState(true);
 CheckboxMenuItem cbm2 = new
 CheckboxMenuItem(" menu item 1.3.2 ");
 m2.add(cbm2);
 m.add(m2);
 pack();
 show(); // affiche la fenetre
}

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

# PopupMenu

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

# Exemple : PopupMenu

```
public class MaFrameWithPopMenu extends
Frame {

 public MaFrameWithPopMenu() {
 super();
 this.setTitle(" Titre de la Fenetre ");
 this.setSize(3000, 1500);

 MenuBar mb = new MenuBar();
 this.setMenuBar(mb);

 PopupMenu m = new
 PopMenu("PMENU");
 mb.add(m);

 m.add(new MenuItem(" 1er element "));
 m.add(new MenuItem("2e element "));
```

```
 Menu m2 = new Menu(" sous menu ");
 CheckboxMenuItem cbm1 = new
 CheckboxMenuItem(" menu item 1.3.1 ");
 m2.add(cbm1);
 cbm1.setState(true);
 CheckboxMenuItem cbm2 = new
 CheckboxMenuItem(" menu item 1.3.2 ");
 m2.add(cbm2);
 m.add(m2);
 pack();
 show(); // affiche la fenetre
 }

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

# Canvas

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

# Exemple de canvas (1)

```
class Ctrait extends Canvas implements MouseListener
{
 Point pt;
 public Ctrait() {
 addMouseListener(this);
 }
 public void paint(Graphics g){
 g.drawLine(0,0,pt.x,pt.y);
 g.setColor(Color.red);
 g.drawString((""+pt.x+";"+pt.y+""),pt.x,pt.y+5);
 }
 public Dimension getMinimumSize(){
 return new Dimension(200,100);
 }
 // suite ...
}
```

# Exemple de canvas (2)

```
class Ctrait extends Canvas implements MouseListener
{
 Point pt;
 ...
 public Dimension getPreferredSize(){
 return getMinimumSize();
 }
 public void mouseClicked(MouseEvent e){}
 public void mousePressed(MouseEvent e){}
 public void mouseReleased(MouseEvent e){
 pt=e.getPoint();repaint();
 }
 public void mouseEntered(MouseEvent e){}
 public void mouseExited(MouseEvent e){}
}
```

# Exemple de canvas (3)

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

**Utilisation de la classe précédente (Ctrait) par une autre classe.  
Voir le fichier Dessin.java pour voir la manière exacte dont les 2 classes sont définies et utilisées.**



# Couleurs d'un composant

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

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

# Polices de caractères

- Contrôle des polices de caractères
  - police utilisée pour afficher du texte dans un composant
  - définie avec *setFont(...)* avec comme argument une instance de `java.awt.Font`.
    - **Font f = new Font ("TimesRoman", Font.PLAIN, 14) ;**

# Polices de caractères

- Contrôle des polices de caractères
  - constantes de style de police sont en réalité des valeurs entières, parmi celles citées ci-après :
    - **Font.BOLD**
    - **Font.ITALIC**
    - **Font.PLAIN**
    - **Font.BOLD + Font.ITALIC**
  - Les tailles en points doivent être définies avec une valeur entière.

# **GESTION DES ACTIONS SUR LES COMPOSANTS**

# Les événements graphiques (1)

- L'utilisateur effectue
  - action au niveau de l'interface utilisateur (clic souris, sélection d'un item, etc.)
  - alors un **événement graphique** est émis.
- Lorsqu'un événement se produit
  - il est reçu par le composant avec lequel l'utilisateur interagit, par exemple un
    - bouton,
    - curseur,
    - champ de texte, etc.

# Les événements graphiques (2)

- Lorsqu'un événement se produit
  - Ce composant transmet cet événement à un autre objet,
  - un écouteur qui possède une méthode pour traiter l'événement
    - cette méthode reçoit l'objet événement généré de façon à traiter l'interaction de l'utilisateur.

# Les événements graphiques (3)

- Gestion événements → utilisation d'objets "*écouteurs d'événements*" (*Listener*) et d'objets sources d'événements.
  - objet écouteur est instance d'une classe implémentant l'interface **EventListener** (ou une interface "fille").
  - source d'événements est objet pouvant recenser des objets écouteurs et leur envoyer des objets événements.

# Les événements graphiques (4)

- Lorsqu'un événement se produit,
  - source d'événements envoie un objet événement correspondant à tous ses écouteurs.
  - objets écouteurs utilisent alors l'information contenue dans l'objet événement pour déterminer leur réponse.

# Exemple

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

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

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

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

# Écouteur événements (1)

- Les écouteurs sont des **interfaces**
- Donc une même classe peut implémenter plusieurs interfaces écouteur.
  - Par exemple une classe héritant de Frame implémentera les interfaces
    - MouseMotionListener (pour les déplacements souris)
    - MouseListener (pour les clics souris).

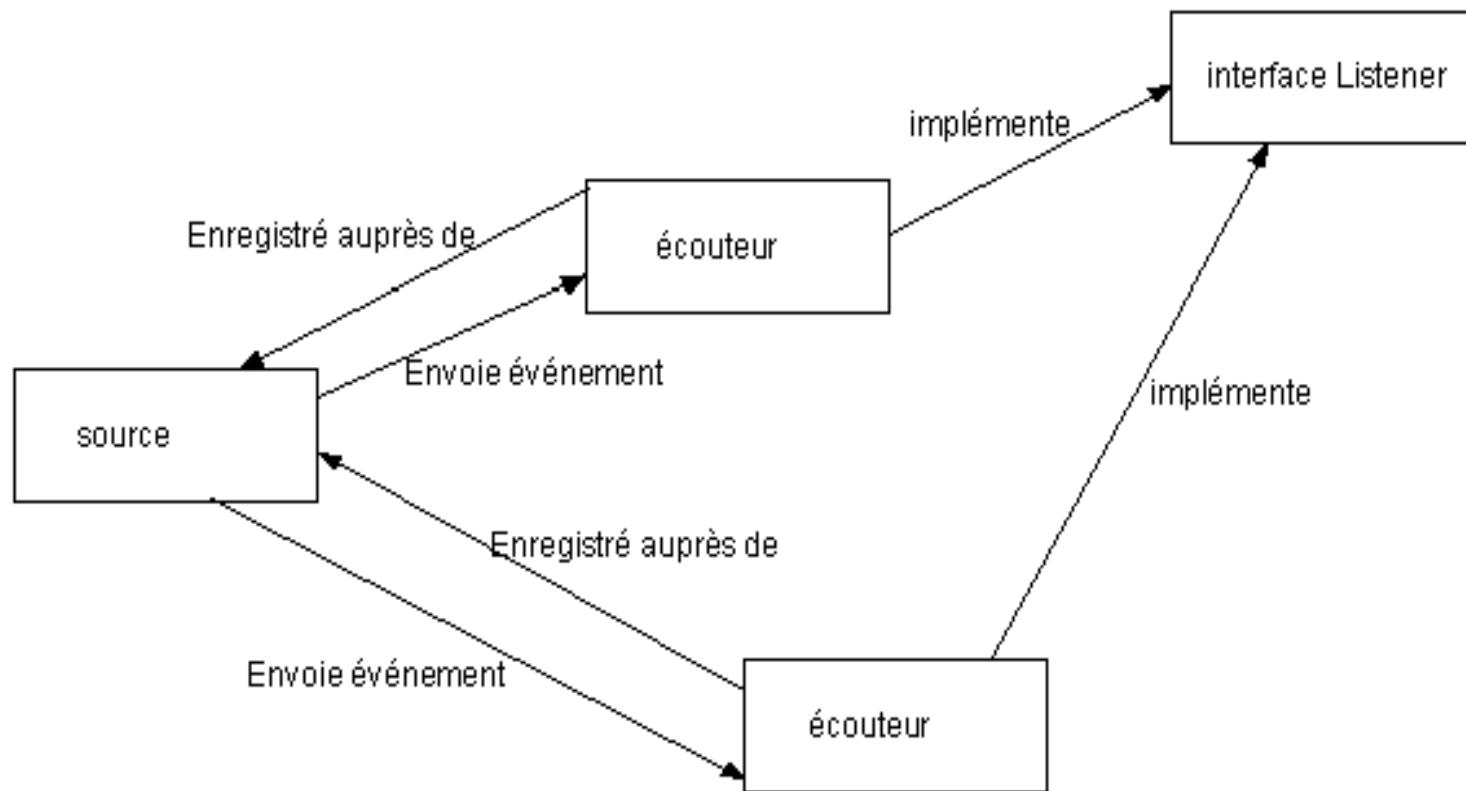
## Écouteur événements (2)

- Chaque composant de l'AWT est conçu pour être la source d'un ou plusieurs types d'événements particuliers.
  - Cela se voit notamment grâce à la présence dans la classe de composant d'une méthode nommée addXXXListener().

# Écouteur événements (3)

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

# Résumé en image



# **CATÉGORIES D'ÉVÉNEMENTS GRAPHIQUES**

# Catégorisation des événements

- Plusieurs types d'événements sont définis dans le package `java.awt.event`.
- Événements classé en catégorie
- Catégorie → interface qui doit être définie par
- Classe souhaitant recevoir cette catégorie doit implémenter cette interface
  - Cette interface exige aussi qu'une ou plusieurs méthodes soient définies.
  - Ces méthodes sont appelées lorsque des événements particuliers surviennent.

# Détail Catégories d'événements

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

# Détail Catégories d'événements (fin)

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

# Résumé : Catégories d'événements

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

# Résumé : Catégories d'événements (suite)

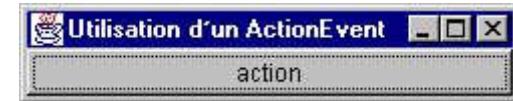
- **MouseListener**
  - Clic, enfoncement/relâchement des boutons de la souris, etc.
- **MouseMotionListener**
  - Déplacement de la souris, drag&drop avec la souris, etc.
- **AdjustmentListener**
  - Déplacement d'une échelle
- **ComponentListener**
  - Savoir si un composant a été caché, affiché ...

# Résumé : Catégories d'événements (fin)

- **ContainerListener**
  - Ajout d'un composant dans un Container
- **FocusListener**
  - Pour savoir si un élément a le "focus"
- **KeyListener**
  - Pour la gestion des événements clavier

# Exemples

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



Implémentation de l'interface ActionListener

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

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



# Exemples (suite)

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



Les 2 boutons ont le même écouteur (la fenêtre)

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



# Exemples (fin)

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



Implémenter cette interface impose l'implémentation de bcp de méthodes

La fenêtre est son propre écouteur

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

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

# Les adaptateurs

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

# Adapteur pour WindowListener

## Solution en implémentant l'interface

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

## Solution en utilisant un WindowAdapter

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

# Liste des adapteurs

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

# Adaptateur et classe anonyme

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

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

# CONTENEURS PARTICULIERS

# Dialog

- Ressemble à *Frame* mais ne sert qu'à afficher des messages devant être lus par l'utilisateur.
  - pas de boutons permettant de le fermer ou de l'iconiser.
  - associe habituellement un bouton de validation.
  - Il est réutilisable pour afficher tous les messages au cours de l'exécution d'un programme.
- Dépend d'un objet *Frame* (ou héritant de Frame)
  - ce Frame est passé comme 1<sup>er</sup> argument au constructeur.
- N'est pas visible lors de sa création. Utiliser la méthode
  - *show()* pour la rendre visible
  - *hide()* pour la cacher).



# Exemple

```
public class Apropos extends Dialog {
 public Apropos(Frame parent) {
 super(parent, "A propos ", true);
 addWindowListener(new
AProposListener(this));
 setSize(500, 1300);
 setResizable(true);
 pack();
 show();
 }
 public static void main(String[] args) {
 new Apropos(new Frame("Zin Zin"));
 }
}
```

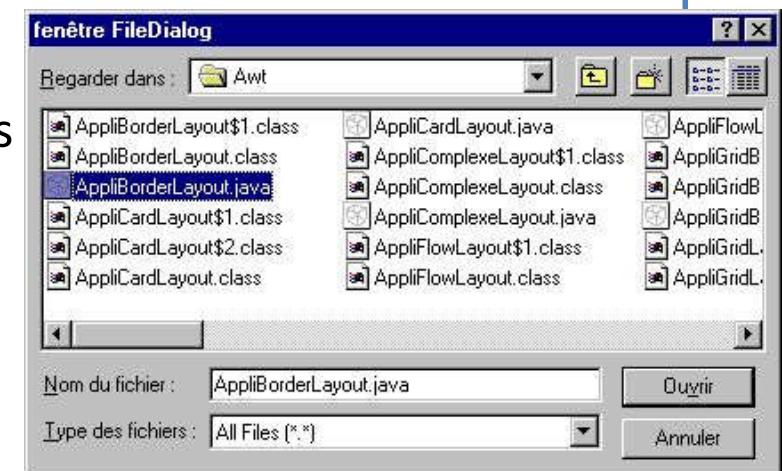
```
class AProposListener extends
WindowAdapter {
 Dialog dialogue;
 public AProposListener(Dialog
dialogue) {
 this.dialogue = dialogue;
 }
 public void
windowClosing(WindowEvent e) {
 dialogue.dispose();
 System.exit(0);
 }
}
```

# FileDialog

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

# FileDialog

- Sélection de fichier
- Méthode *open()*
  - Ouvre la boite
  - Retourne nom fichier sélectionné ou null
- Méthode *setFilterExtensions()*
  - Précise sous la forme d'un tableau de chaînes la liste des extensions de fichiers acceptées par la sélection.



# Exemple

```
....
private void showFileDialogDemo(){
 headerLabel.setText("Control in action: FileDialog");

 final FileDialog fileDialog = new FileDialog(mainFrame,"Select file");
 Button showFileDialogButton = new Button("Open File");
 showFileDialogButton.addActionListener(new ActionListener() {
 @Override
 public void actionPerformed(ActionEvent e) {
 fileDialog.setVisible(true);
 statusLabel.setText("File Selected :" + fileDialog.getDirectory() + fileDialog.getFile());
 }
 });
 controlPanel.add(showFileDialogButton);
 mainFrame.setVisible(true);
}
....
```

# ScrollPane

- Conteneur général de type **Panel**
  - Pas utilisable de façon indépendante (comme **Panel**)
  - Fournit barres de défilement (scrollbars) verticales et/ou horizontales
  - Ne peut contenir qu'un seul composant
  - Pas très intéressant mais est cité pour information.
- Avec **Swing** à voir dans la suite, de nouveaux très intéressants composants font leur apparition...

# **SWING – COMPOSANTS LÉGERS**

# Introduction

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

# Présentation

- **Swing** fait partie de la bibliothèque *Java Foundation Classes* (JFC)
- **But** similaire à celui de l'**API AWT** mais dont modes de fonctionnement et d'utilisation sont complètement différents
- **JFC** contient en plus de **Swing**
  - Accessibility API :
  - 2D API: support du graphisme en 2D
  - API pour l'impression et le cliquer/glisser

# Composants graphiques lourds

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

# Remarques

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

# Composants légers de Swing

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

# Atouts de Swing

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

# Look & Feel (1)

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

## Look & Feel (2)

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

# Exemple de Look & Feel

| Platform                                          | Look and Feel |
|---------------------------------------------------|---------------|
| Solaris, Linux with GTK+ 2.2 or later             | GTK+          |
| Other Solaris, Linux                              | Motif         |
| IBM UNIX                                          | IBM*          |
| HP UX                                             | HP*           |
| Classic Windows                                   | Windows       |
| Windows XP                                        | Windows XP    |
| Windows Vista                                     | Windows Vista |
| Macintosh                                         | Macintosh*    |
| * L&F : propriétaires à commander avec la machine |               |

# Conventions de nommage

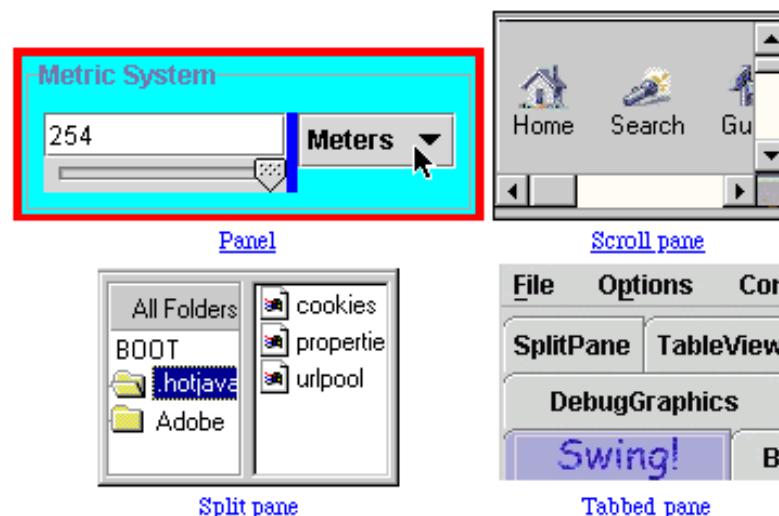
- Composants Swing → paquetage **javax.swing** et ses sous paquetages.
  - Un certain nombre de paquetages d'extensions du langage java 1.1 (par opposition aux paquetages standards java) sont regroupés sous **javax**.
    - Convention permet de télécharger ces paquetages dans un environnement navigateur avec une machine virtuelle **java 1.1**.
    - Navigateurs, en effet, pas autorisés à télécharger des paquetages dont le nom commence par **java**.
- Noms similaires aux correspondants de AWT précédés d'un J.
  - JFrame, JPanel, JTextField, JButton, JCheckBox, JLabel, etc.

# Aperçu des composants Swing (1)

Top-Level Containers



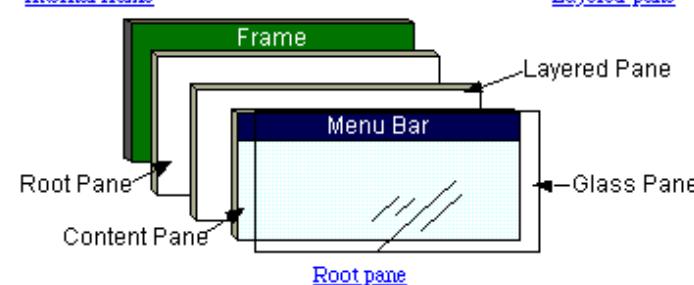
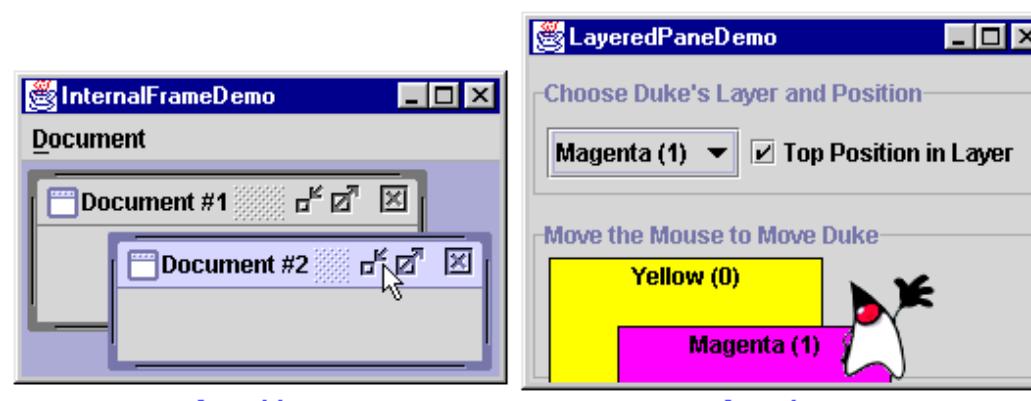
General-Purpose Containers



# Aperçu des composants Swing (2)



## Special-Purpose Containers



# Aperçu des composants Swing (3)

## Basic Controls



[Buttons](#)



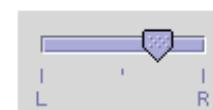
[Combo box](#)



[List](#)

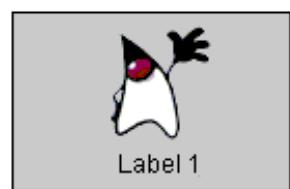


[Menu](#)



[Text fields](#)

## Uneditable Information Displays



[Label](#)



[Progress bar](#)



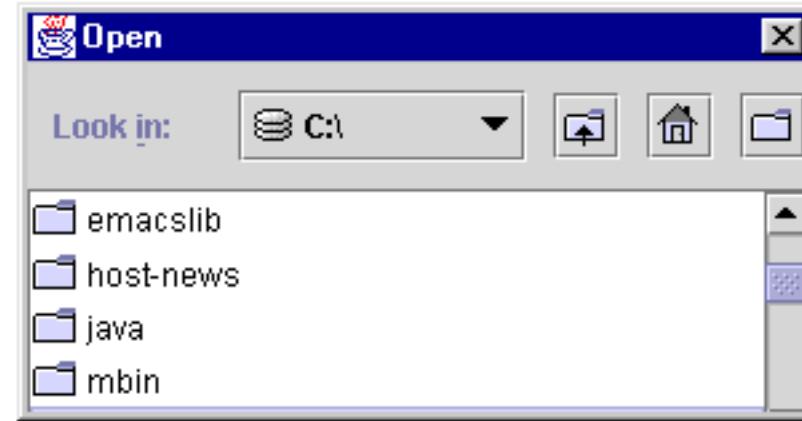
[Tool tip](#)

# Aperçu des composants Swing (4)

## Editable Displays of Formatted Information



[Color chooser](#)



[File chooser](#)

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

[Table](#)

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

[Text](#)



[Tree](#)

# Principaux paquetages Swing (1)

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

# Principaux paquetages Swing (2)

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

# JFrame

- Ancêtre commun : classe **JComponent**

```
java.lang.Object
|
+--java.awt.Component
|
+--java.awt.Container
|
+--javax.swing.JComponent
```

- **Le JFrame**

- Un objet **JFrame** a un comportement par défaut associé à une tentative de fermeture de la fenêtre.
  - Contrairement à la classe **Frame**, qui ne réagissait pas par défaut, l'action de fermeture sur un **JFrame** rend par défaut la fenêtre invisible.
  - Ce comportement par défaut peut être modifié par **setDefaultCloseOperation()**.

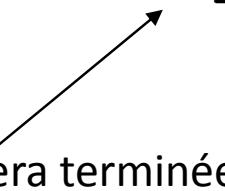
# JFrame

- 4 comportements sont possibles lors de la fermeture de la fenêtre
- **DO NOTHING ON CLOSE**
- **HIDE ON CLOSE**
- **DISPOSE ON CLOSE**
- **EXIT ON CLOSE**

```
import javax.swing.*;
public class Simple {
 public static void main(String[] args) {
 JFrame cadre = new JFrame("Ma fenêtre");

 cadre.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
 cadre.setSize(300, 200);
 cadre.setVisible(true);
 }
}
```

détermine que l'application sera terminée  
Lorsqu'on fermera la fenêtre



# JFrame

- Le conteneur d'un **JFrame** n'est plus confondu avec le cadre lui-même.
  - Il possède une couche de contenu, dans laquelle on peut ajouter les composants graphiques et dont on peut changer le gestionnaire de présentation.
  - Cette couche est obtenue par la méthode **getContentPane()**;

## Avec AWT

```
Frame monFrame = new Frame ("Mon Frame");
Button monBouton = new Button ("Mon Bouton");
monFrame.add(monBouton);
```

## Avec Swing

```
JFrame monFrame = new JFrame ("Mon Frame");
JButton monBouton = new JButton ("Mon Bouton");
Container panneauContenu = monFrame.getContentPane();
panneauContenu.add(monBouton);
```

# Composants Swing (1)

- La classe **ImageIcon** et l'interface **Icon**
  - Les objets de cette classe permettent de définir des icônes à partir d'images (gif, jpg, etc.) qui peuvent être ajoutés au classique texte d'un JLabel, d'un JButton, etc.

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

// Un JLabel
JLabel monLabel = new JLabel("Mon Label");
monLabel.setIcon(monIcone);
monLabel.setHorizontalTextPosition(JLabel.RIGHT);

// Un JButton
JButton monBouton = new JButton("Mon bouton", monIcone);
```

# Composants Swing (2)

- **JTextPane**
  - éditeur de texte qui permet la gestion de texte formaté, le retour à la ligne automatique (word wrap), l'affichage d'images.
- **JPasswordField**
  - champ de saisie de mots de passe : la saisie est invisible et l'affichage de chaque caractère tapé est remplacé par un caractère d'écho (\* par défaut).
- **JEditorPane**
  - un *JTextComponent* pour afficher et éditer du code HTML 3.2 et des formats tels que RTF.

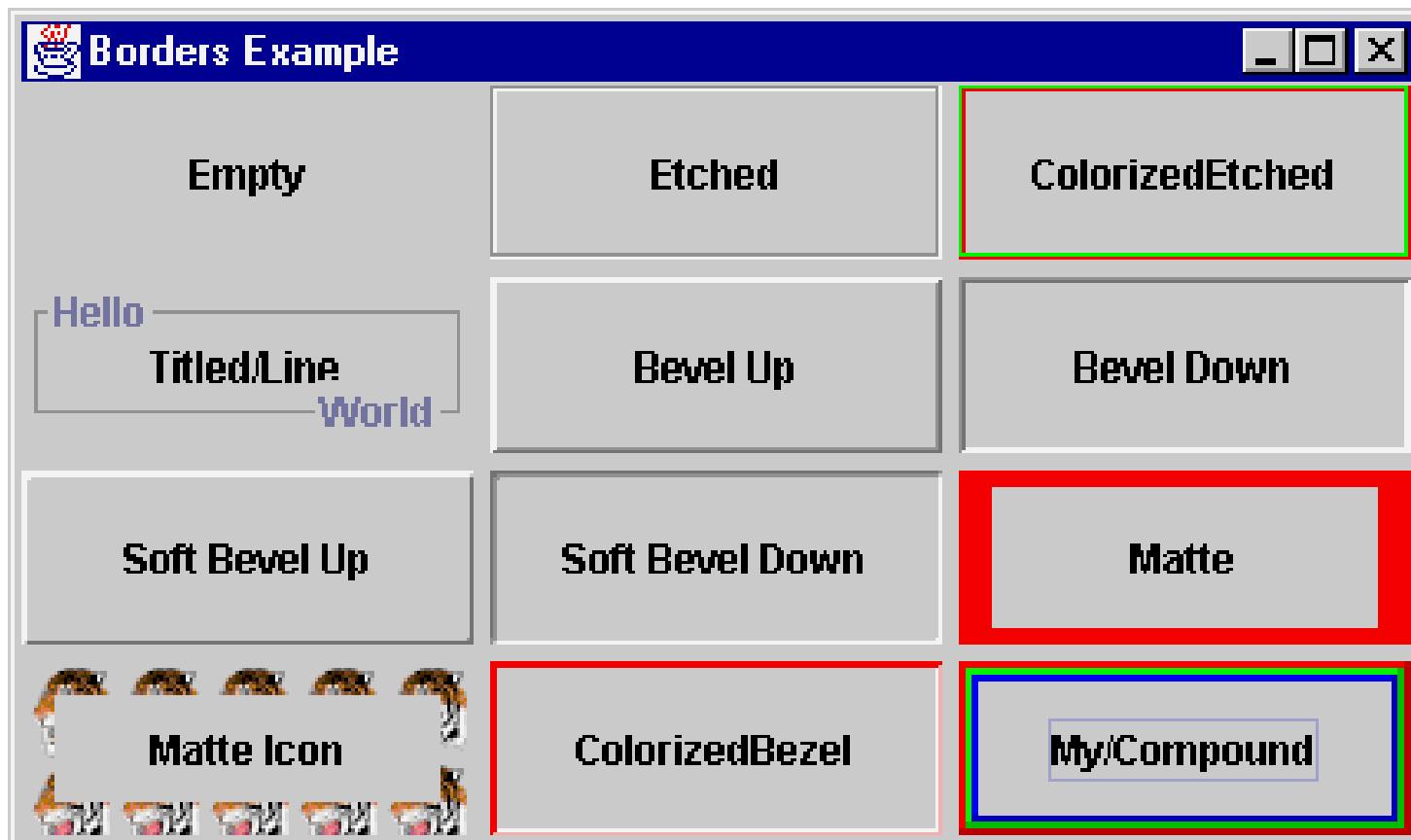
# Exemple : JEditorPane

```
import javax.swing.*;
import java.awt.*;
public class EditeurWEB {
 public static void main(String[] args)
 {
JFrame monFrame = new JFrame ("Mon Frame");
monFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
try {
 JEditorPane monEditeur = new
 JEditorPane("http://www.google.fr");
 Container panneauContenu = monFrame.getContentPane();
 panneauContenu.setLayout(new FlowLayout());
 panneauContenu.add(monEditeur);
 monFrame.pack();
 monFrame.show();
} catch (Exception e) {System.out.println(e.getMessage());}
 }
}
```

# Composants Swing (3)

- Des bordures peuvent être dessinées autour de tous composants graphiques. Swing en définit 9 types :
  - **AbstractBorder** : ne fait rien
  - **BevelBorder** : une bordure 3D en surépaisseur ou en creux
  - **CompoundBorder** : permet de composer des plusieurs bordures
  - **EmptyBorder**
  - **EtchedBorder**
  - **LineBorder** : bordures d'une seule couleur)
  - **MatteBorder**
  - **SoftBevelBorder** : une bordure 3D aux coins arrondis
  - **TitledBorder** : une bordure permettant l'inclusion d'une chaîne de caractères

# Composants Swing (4)



# Composants Swing (5)

- **JLayeredPane**
  - un conteneur qui permet de ranger ses composants en couches (ou transparents).
  - Cela permet de dessiner les composants, selon un certain ordre: premier plan, plan médian, arrière plan, etc.
  - Pour ajouter un composant, il faut spécifier la couche sur laquelle il doit être dessiné **monJLayeredPane.add (monComposant, new Integer(5));**
    - des « layer » sont définis en natif et des constantes pour y accéder existent dans la classe.

# Composants Swing (6)

- **ToolTipText**
  - Créer aides en lignes qui apparaissent lorsque la souris passe sur un composant.

```
 JButton monBouton = new JButton ("Un bouton");
monBouton.setToolTipText ("Aide de mon bouton");
```
- Représenter des listes : classe **JList**
- Représenter des arbres : classe **JTree**
- Représenter des tables (Excel) : classe **JTable**

# Composants Swing : conclusion

- Ces quelques nouveautés ne sont qu'un aperçu de ce que propose Swing.
  - Il y a beaucoup de composants, de nouveaux
    - gestionnaires de présentation,
    - événements graphiques
  - Qu'il est difficile de
    - présenter et
    - détailler.

# Dessin avec Swing (1)

- Swing utilise comme l'AWT, les méthodes *paint()* et *repaint()*.
- Swing supporte de plus quelques propriétés additionnelles
  - le double buffering
  - la transparence
  - le chevauchement des composants

## Dessin avec Swing (2)

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

# Conclusions

- Nous avons vu quelles sont les capacités de l'**API Swing** et les concepts sous-jacents.
- Ce cours est uniquement une généralisation des concepts de Swing.
- Pour maîtriser l'API, une certaine pratique sera nécessaire.
- **Donc à vos claviers**

# **ENTRÉES SORTIES**

# Flux de caractères (1)

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

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

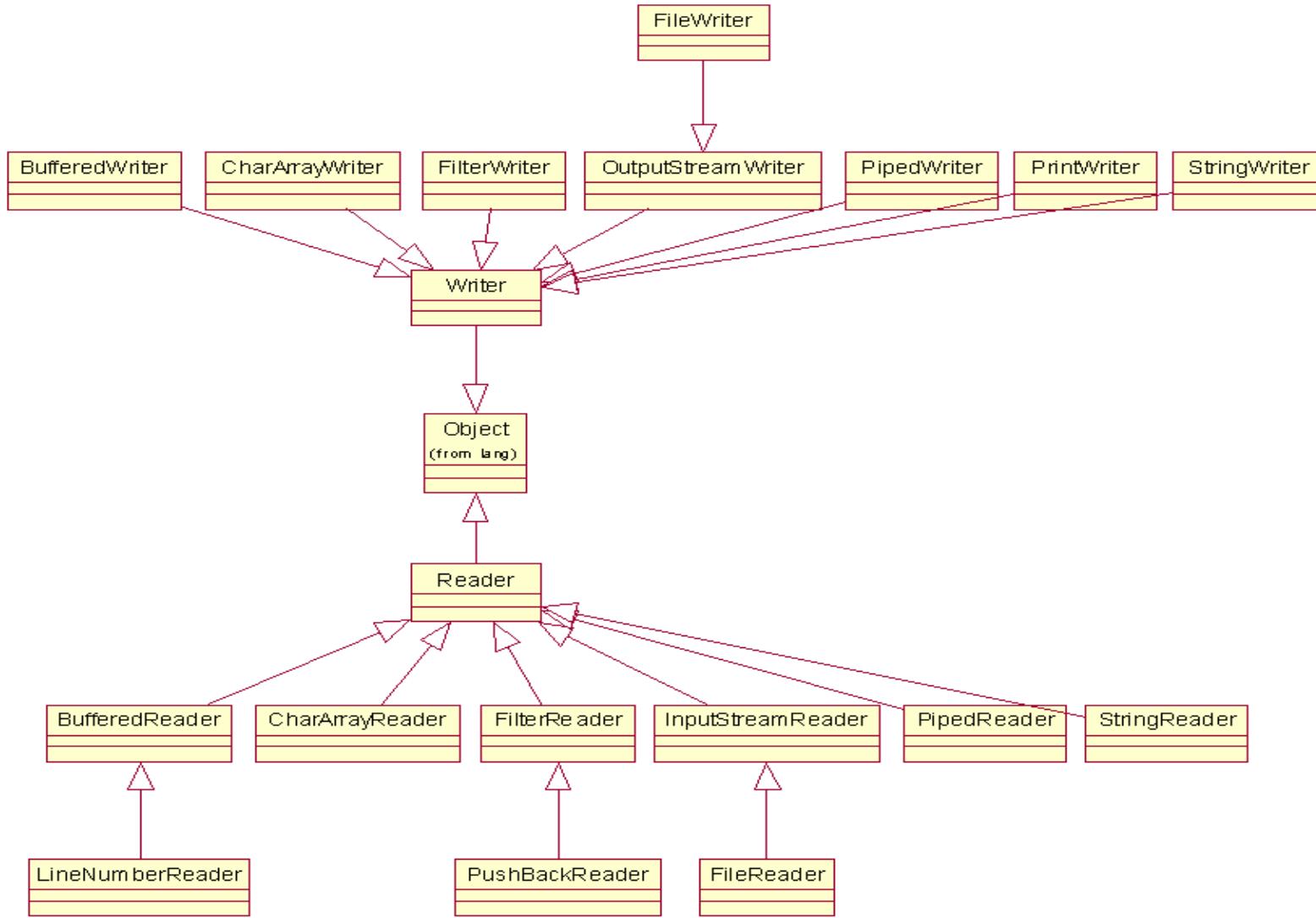
Conversion des caractères d'un fichier  
avec un codage explicitement indiqué

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

# Flux de caractères (2)

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

# Flux de caractères : hiérarchie



# Flux de données prédéfinis (1)

- Il existe 3 flux prédéfinis :
  - entrée standard `System.in` (instance de `InputStream`)
  - sortie standard `System.out` (instance de `PrintStream`)
  - sortie standard erreur `System.err` (instance de `PrintStream`)

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

# Flux de données prédéfinis (2)

- Classe `InputStream` ne propose que des méthodes élémentaires.
- Classe `BufferedReader` permet de récupérer des chaînes de caractères.

```
try {
 Reader reader = new InputStreamReader(System.in);
 BufferedReader keyboard = new BufferedReader(reader);
 System.out.print("Entrez une ligne de texte : ");
 String line = keyboard.readLine();
 System.out.println("Vous avez saisi : " + line);
} catch(IOException e) {
 System.out.print(e);
}
```

# **GESTION DES FICHIERS**

# Les entrées / sorties

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

# Introduction

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

# La classe File

- **Constructeurs et méthodes** de la classe File :
  - **File (String name)**
  - **File (String path, String name)**
  - **File (File dir, String name)**
  - boolean **isFile( )** / boolean **isDirectory( )**
  - boolean **mkdir( )**
  - boolean **exists( )**
  - boolean **delete( )**
  - boolean **canWrite( )** / boolean **canRead( )**
  - File **getParentFile( )**
  - long **lastModified( )**

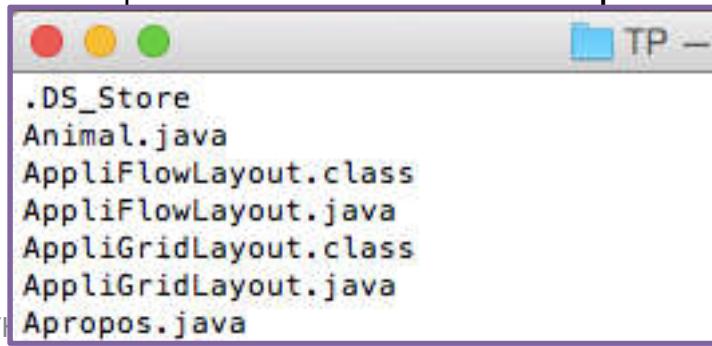
# Exemple 1

```
import java.io.*; ←
public class Listeur
{
 public static void main(String[] args)
 {
 litrep(new File("."));
 }
 public static void litrep(File rep)
 {
 if (rep.isDirectory())
 { //liste les fichier du répertoire
 String t[]={};rep.list();
 for (int i=0;i<t.length;i++)
 System.out.println(t[i]);
 }
 }
}
```

Les objets et classes relatifs à la gestion des fichiers se trouvent dans le package java.io

A partir du chemin d'un dossier ou d'un fichier, on peut créer un objet File : ici on va lister le répertoire courant (« . »)

Les méthodes isFile() et isDirectory() permettent de déterminer si mon objet File est une fichier ou un répertoire



# Exemple 2

```
import java.io.*;
public class Listeur
{
 public static void main(String[] args)
 { litrep(new File("/home/mac"));

 public static void litrep(File rep)
 {
 File r2;
 if (rep.isDirectory())
 {String t[]=rep.list();
 for (int i=0;i<t.length;i++)
 {
 r2=new File(rep.getAbsolutePath()+"/"+t[i]);
 if (r2.isDirectory()) litrep(r2);
 else System.out.println(r2.getAbsolutePath());
 }
 }
}
```

Le nom complet du fichier est rep\fichier

Pour chaque fichier, on regarde s'il est un répertoire.

Si le fichier est un répertoire  
**litrep** s'appelle récursivement elle-même

# Notion de flux (1)

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

## Notion de flux (2)

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

## Notion de flux (3)

- L'intérêt de la notion de flux est qu'elle permet une gestion homogène indépendant :
  - De la ressource associée au flux de données,
  - Du flux (entrée ou sortie).
- Certains flux peuvent être associés à des filtres
  - Combinés à des flux d'entrée ou de sortie, ils permettent de traduire les données.

# Notion de flux (4)

- Les flux sont regroupés dans le paquetage **java.io**
- De nombreuses classes représentant les flux
  - pas toujours aisément repérables.
- Certains types de flux agissent sur la façon dont sont traitées les données qui transitent par leur intermédiaire :
  - E/S bufferisées, traduction de données, ...
- Gestion des E/S peut nécessiter de combiner différents types de flux.

# Flux d'octets et flux de caractères

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

# Lecture de fichier

```
import java.io.*;
public class LireLigne
{
 public static void main(String[] args)
 {
 try
 {
 FileReader fr=new FileReader("./TP/IPile.java");
 BufferedReader br= new BufferedReader(fr);
 while (br.ready())
 System.out.println(br.readLine());
 br.close();
 }
 catch (Exception e)
 {
 System.out.println("Erreur "+e);
 }
 }
}
```

A partir du chemin d'un dossier ou d'un fichier, on peut créer un objet FileReader puis à partir de celui-ci, on crée un BufferedReader

Dans l'objet BufferedReader on dispose d'une méthode readLine()

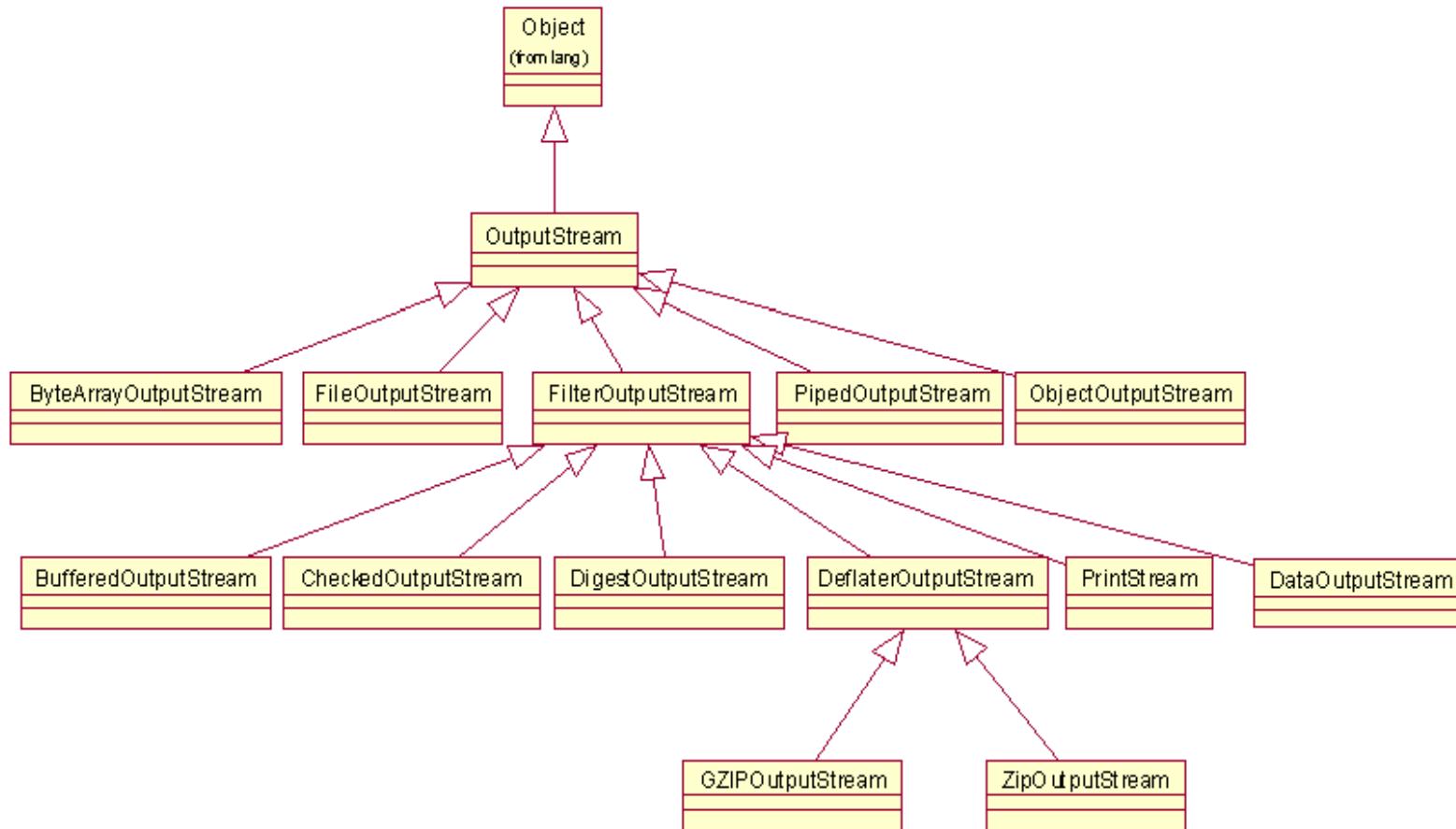
# Ecriture dans un fichier

```
import java.io.*;
public class Ecrire
{
 public static void main(String[] args)
 {
 try
 {
 FileWriter fw=new FileWriter("./TP/IPile.java");
 BufferedWriter bw= new BufferedWriter(fw);
 bw.write("Ceci est mon fichier");
 bw.newLine();
 bw.write("Il est à moi...");
 bw.close();
 }
 catch (Exception e)
 {
 System.out.println("Erreur "+e);
 }
 }
}
```

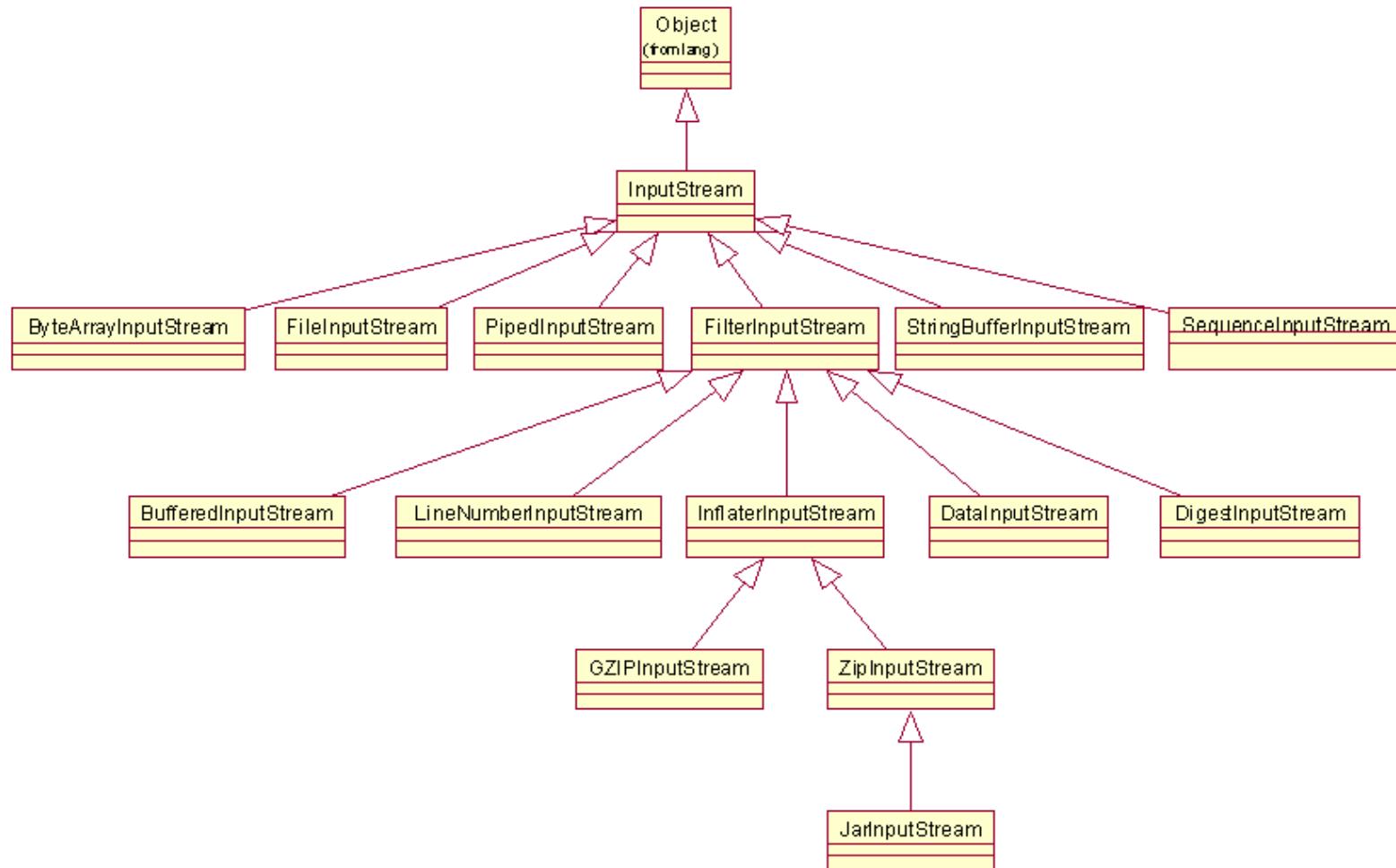
A partir du chemin d'un dossier ou d'un fichier, on peut créer un objet FileWriter puis à partir de celui-ci, on crée un BufferedWriter

Attention, lorsque l'on a écrit, il ne faut pas oublier de fermer le fichier

# La hiérarchie flux d'octets en sortie



# La hiérarchie flux d'octets en entrée



# **LES FLUX D'OCTETS**

# La classe InputStream

- Un **InputStream** est un flux de lecture d'octets.
- **InputStream** est une classe abstraite.
  - Ses sous-classes concrètes permettent une mise en œuvre pratique.
  - Par exemple, **FileInputStream** permet la lecture d'octets dans un fichier.

# Méthodes principales (1)

- **public abstract int read () throws IOException** qui retourne
  - l'octet lu ou
  - -1 si la fin de la source de données est atteinte.
- Méthode à définir dans les sous-classes concrètes et qui est utilisée par les autres méthodes définies dans la classe **InputStream**.

## Méthodes principales (2)

- **int read (byte[] b)** qui remplit un tableau d'octets et retourne le nombre d'octets lus
- **int read (byte [] b, int off, int len)** qui remplit un tableau d'octets à partir d'une position donnée et sur une longueur donnée
- **void close ()** qui permet de fermer un flux,
  - Il faut fermer les flux dès qu'on a fini de les utiliser. En effet, un flux ouvert consomme des ressources du système d'exploitation qui sont en nombre limité.

# Méthodes principales (3)

- **int available ()** qui retourne le nombre d'octets prêts à être lus dans le flux,
  - Attention : Cette fonction permet d'être sûr qu'on ne fait pas une tentative de lecture bloquante. Au moment de la lecture effective, il se peut qu'il y ait plus d'octets de disponibles.
- **long skip (long n)** qui permet d'ignorer un certain nombre d'octets en provenance du flot. Cette fonction renvoie le nombre d'octets effectivement ignorés.

# La classe OutputStream (1)

- Un **OutputStream** est un flot d'écriture d'octets.
- La classe **OutputStream** est abstraite.
- Les méthodes principales qui peuvent être utilisées sur un **OutputStream** sont :
  - **public abstract voidwrite (int) throws IOException** qui écrit l'octet passé en paramètre,
  - **voidwrite (byte[] b)** qui écrit les octets lus depuis un tableau d'octets,
  - **voidwrite (byte [] b, int off, intlen)** qui écrit les octets lus depuis un tableau d'octets à partir d'une position donnée et sur une longueur donnée.

# La classe OutputStream (2)

- Les méthodes principales qui peuvent être utilisées sur un **OutputStream** sont (suite) :
  - **void close ()** qui permet de fermer le flux après avoir éventuellement vidé le tampon de sortie,
  - **flush ()** qui permet de purger le tampon en cas d'écritures bufferisées.

# Les flux d'octets

- Classe **DataInputStream**
  - sous classes de **InputStream** permet de lire tous les types de base de Java.
- Classe **DataOutputStream**
  - sous classes de **OutputStream** permet d'écrire tous les types de base de Java.
- Classes **ZipOutputStream** et **ZipInputStream**
  - permettent de lire et d'écrire des fichiers dans le format de compression zip.

# Empilement de flux filtrés (1)

- En Java, chaque type de flux est destiné à réaliser une tâche.
- Lorsque le programmeur souhaite un flux qui ait un comportement plus complexe
  - "empile", à la façon des poupées russes, plusieurs flux ayant des comportements plus élémentaires.
    - On parle de flux filtrés.
  - Concrètement, il s'agit de passer, dans le constructeur d'un flux, un autre flux déjà existant pour combiner leurs caractéristiques.

# Empilement de flux filtrés (2)

- **FileInputStream**
  - permet de lire depuis un fichier mais ne sait lire que des octets.
- **DataInputStream**
  - permet de combiner les octets pour fournir des méthodes de lecture de plus haut niveau (pour lire un double par exemple), mais ne sait pas lire depuis un fichier.
- Une combinaison des deux permet de combiner leurs caractéristiques :

```
FileInputStream fic = new FileInputStream ("fichier");
DataInputStream din = new DataInputStream (fic);
double d = din.readDouble ();
```

# Empilement de flux filtrés (3)

Lecture bufferisée de nombres depuis un fichier

```
DataInputStreamdin = new DataInputStream(new BufferedInputStream(
 new FileInputStream ("monfichier")));
```

Lecture de nombre dans un fichier au format zip

```
ZipInputStreamzin = new ZipInputStream (
 new FileInputStream ("monfichier.zip"));
DataInputStreamdin = new DataInputStream (zin);
```

# Flux de fichiers à accès direct (1)

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

## Flux de fichiers à accès direct (2)

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

# Plan

1. Rappels POO et Java
2. Classe abstraite, Interface et Type générique
3. Exceptions Java et Javadoc
4. Swing: Création d'interfaces graphiques
5. Entrées/Sorties (Fichiers Textes et binaires)
- 6. Collections**
7. JDBC
8. Compléments

# TABLEAUX

# Tableaux : Arrays

- Classe contient des méthodes statiques permettant de manipuler de tableaux :
  - Recherche, recherche dichotomique
  - Tris
  - Remplissage
  - Egalité
  - *toString()*

# Méthode *toString*

- Méthode *static String toString(X[] a)*
  - retourne une chaîne de caractères contenant les éléments du tableau
  - convertis en chaîne de caractères
  - séparés par des virgules et entre crochets.

# Méthode *equals*

- Méthode *static boolean equals(X[] a, X[] a2)* où *X* peut être un type primitif ou un *Object* retourne
  - *true* si les 2 tableaux sont égaux et
  - *false* sinon.
- Deux tableaux sont égaux si
  - Ils contiennent le même nombre d'éléments,
  - les éléments de même rang sont égaux.

# Méthode *deepEquals*

- Si des éléments du tableau peuvent être des tableaux,
  - il faut utiliser la méthode *deepEquals(X[] a, X[] a2)*.

# Méthode fill

- La méthode *static void fill(X[] a, X val)*
  - affecte la valeur *val* à tous les éléments du tableau *a*.

# Méthodes : *binarySearch*

- Méthode *static int binarySearch(X[] a, X cle)* effectue une recherche de *cle* dans le tableau trié *a*.
- La méthode retourne
  - l'indice de l'élément s'il existe,
  - Et *- pointDInsertion-1* si *cle* ne se trouve pas dans le tableau.
  - La valeur *pointDInsertion* est le rang où la clé serait ajoutée, si on l'ajoutait au tableau. C'est le rang du premier élément plus grand que l'élément cherché, ou la taille du tableau.

# Code de la méthode

```
public static int binarySearch(int[] a, int cle) {
 int debut = 0;
 int fin = a.length-1;
 while (debut <= fin) {
 int milieu =(debut + fin)/2;
 if (a[milieu]<cle) debut = milieu + 1;
 else if (a[milieu]>cle) fin = milieu - 1;
 else return milieu; // trouvé
 }
 return-(debut + 1); // pas trouvé.
}
```

# Méthode *sort*

- La méthode *static void sort(Object[] a)* est programmée en utilisant :
  - Un **tri rapide** si *X* est un type **primitif** [Jon L. Bentley and M. Douglas McIlroy's "[Engineering a Sort Function](#)", *Software-Practice and Experience*, Vol. 23(11) P. 1249-1265 (Novembre 1993)].
  - Un **tri par fusion** si *X* est un *Object*.
- Les objets du tableau à trier doivent être **Comparable**

# Collections

- but est de stocker de multiples objets
- permettent de gérer des ensembles d'objets
- plusieurs caractéristiques :
  - doublons,
  - ordre tri

# Familles de Collections

- 4 grandes familles de collections
  - **List** : éléments ordonnés avec doublons
  - **Set** : éléments non ordonnés sans doublons
  - **Map** : sous forme association de paires *clé/valeur*
  - **Queue** et **Deque** : stocke des éléments dans un certain ordre avant d'être consommés pour être traités

# Framework collection

- Tableaux peuvent pas répondre à tous les besoins de stockage d'un ensemble d'objets
- Manque de certaines fonctionnalités.
- Diversité des implémentations proposées par l'API **Collections** répond à la plupart des besoins.

# Interfaces de Collections

- Deux grandes familles
  - **java.util.Collection** : gérer un groupe d'objets
  - **java.util.Map** : gérer éléments de type de paires clé/valeur
- Stocke groupe d'éléments avec fonctionnalités selon l'implémentation :
  - Ajouter
  - Supprimer
  - Obtenir éléments
  - Parcourir éléments.

# Accès concurrentiel

- Interfaces et classes de l'API Collections sans gestion accès concurrents → package *java.util*.
- Java 5 propose plusieurs collections dans la package *java.util.concurrent* permettant d'être modifiées durant leur parcours
  - CopyOnWriteArrayList,                              ConcurrentHashMap,  
CopyOnWriteArraySet.
- Fonctionnalités définies dans 5 interfaces de base : *Collection, List, Set, Map, Queue*.

# Interfaces spéciales

- Avec fonctionnalités particulières :
  - SortedSet
  - NavigableSet
  - SortedMap
  - NavigableMap
  - ConcurrentMap
  - ConcurrentNavigableMap
  - BlockingQueue
  - Deque
  - BlockingDeque

# Classes abstraites

- Classes mères de plusieurs implémentations :
  - AbstractCollection,
  - AbstractSet,
  - AbstractList,
  - AbstractSequentialList,
  - AbstractQueue,
  - AbstractMap.

# Implémentations usage généraliste

- Propose plusieurs classes implémentant les interfaces
  - **HashSet** : Hashtable implémente **Set**
  - **TreeSet**: arbre implémente l'interface **SortedSet**
  - **LinkedHashSet**,
  - **ArrayList** : tableau dynamique implémente l'interface **List**
  - **ArrayDeque**,
  - **LinkedList** : liste doublement chaînée implémente **List**
  - **PriorityQueue**,
  - **HashMap** : Hashtable qui implémente l'interface **Map**
  - **TreeMap** : arbre qui implémente l'interface **SortedMap**
  - **LinkedHashMap**.

# Propriétés des interfaces

| Interface | Use générale                              | Use spécifique                            | Accès concurrent                                                                                                                                     |
|-----------|-------------------------------------------|-------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------|
| List      | ArrayList<br>LinkedList                   | CopyOnWriteArrayList                      | Vector<br>Stack<br>CopyOnWriteArrayList                                                                                                              |
| Set       | HashSet<br>TreeSet<br>LinkedHashSet       | CopyOnWriteArraySet<br>EnumSet            | CopyOnWriteArraySet<br>ConcurrentSkipListSet                                                                                                         |
| Map       | HashMap<br>TreeMap<br>LinkedHashMap       | WeakHashMap<br>IdentityHashMap<br>EnumMap | Hashtable<br>ConcurrentHashMap<br>ConcurrentSkipListMap                                                                                              |
| Queue     | LinkedList<br>ArrayDeque<br>PriorityQueue |                                           | ConcurrentLinkedQueue<br>LinkedBlockingQueue<br>ArrayBlockingQueue<br>PriorityBlockingQueue<br>DelayQueue<br>SynchronousQueue<br>LinkedBlockingDeque |

# Des outils aussi

- 2 interfaces → parcours certaines collections
  - Iterator
  - ListIterator.
- 1 interface + 1 classe → tri certaines collections :
  - Comparable
  - Comparator
- Classes utilitaires :
  - Arrays
  - Collections

# Résumé

| Collection           | Ordonné | Accès direct | Clé / valeur | Doublons | Null | Thread Safe |
|----------------------|---------|--------------|--------------|----------|------|-------------|
| ArrayList            | Oui     | Oui          | Non          | Oui      | Oui  | Non         |
| LinkedList           | Oui     | Non          | Non          | Oui      | Oui  | Non         |
| HashSet              | Non     | Non          | Non          | Non      | Oui  | Non         |
| TreeSet              | Oui     | Non          | Non          | Non      | Non  | Non         |
| HashMap              | Non     | Oui          | Oui          | Non      | Oui  | Non         |
| TreeMap              | Oui     | Oui          | Oui          | Non      | Non  | Non         |
| Vector               | Oui     | Oui          | Non          | Oui      | Oui  | Oui         |
| Hashtable            | Non     | Oui          | Oui          | Non      | Non  | Oui         |
| Properties           | Non     | Oui          | Oui          | Non      | Non  | Oui         |
| Stack                | Oui     | Non          | Non          | Oui      | Oui  | Oui         |
| CopyOnWriteArrayList | Oui     | Oui          | Non          | Oui      | Oui  | Oui         |
| ConcurrentHashMap    | Non     | Oui          | Oui          | Non      | Oui  | Oui         |
| CopyOnWriteArraySet  | Non     | Non          | Non          | Non      | Oui  | Oui         |

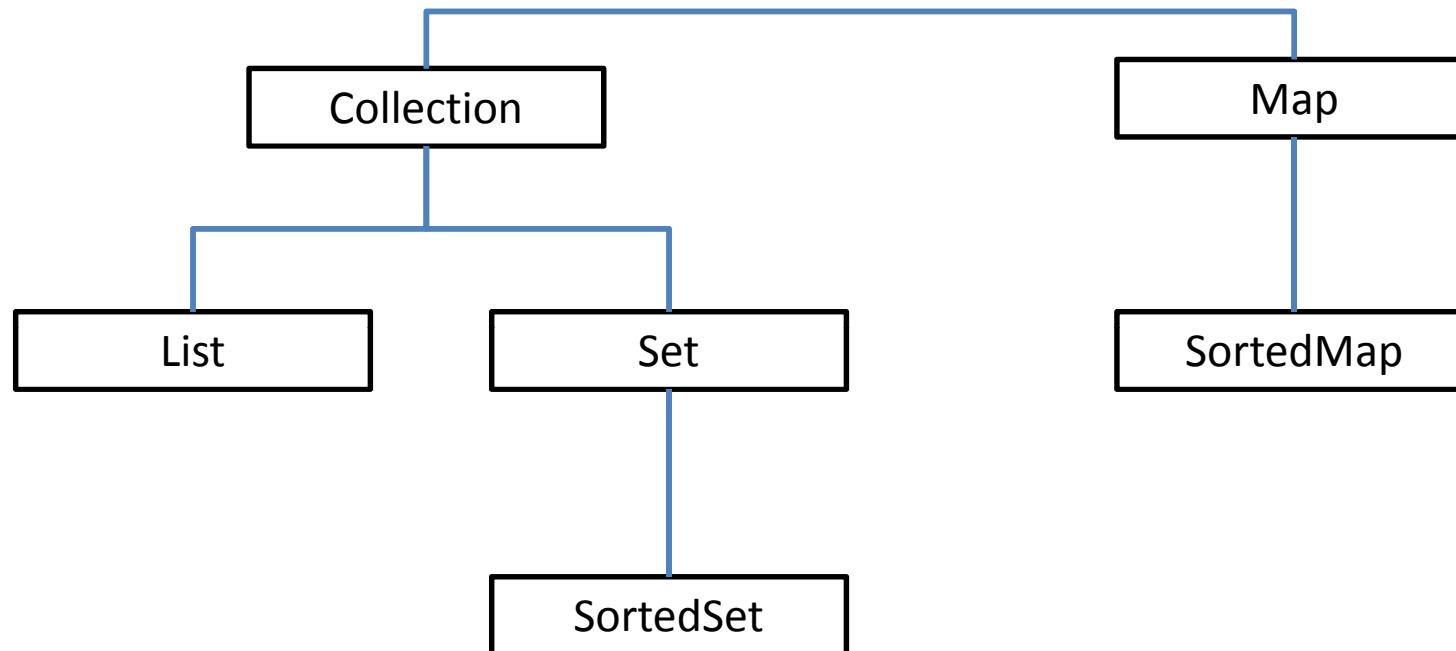
# Parcours et tri de collections

- Interfaces
  - **Iterator** : parcours des collections
  - **ListIterator** : parcours des listes dans les 2 sens avec modification d'éléments lors du parcours
  - **Comparable** : définir ordre de tri naturel pour un objet
  - **Comparator** : définir un ordre de tri quelconque

# Et enfin la classe ...

- **Collections** contient nombreuses méthodes statiques pour réaliser
  - Certaines opérations sur une collection.
  - méthodes (ou **XXX** = interface d'une collection)
    - ***unmodifiableXXX()*** rendre une collection non modifiable.
    - ***synchronizedXXX()*** obtenir une version synchronisée d'une collection pouvant ainsi être manipulée de façon sûre par plusieurs threads.

# Interfaces des collections



# Interfaces de base : Set, List et Map

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

# INTERFACE COLLECTION

# Interface Collection (1)

- Cette interface représente un minimum commun pour les objets qui gèrent des collections :
  - ajout d'éléments,
  - suppression d'éléments,
  - vérifier la présence d'un objet dans la collection,
  - parcours de la collection
  - quelques opérations diverses sur la totalité de la collection.

# Interface Collection (2)

- 2 constructeurs au moins
  - sans argument (par défaut)
  - prend en paramètre un objet de type collection
- plusieurs méthodes
  - Cf. diapositives suivantes

# Méthodes (1)

| Méthode                                   | Rôle                                                                                                        |
|-------------------------------------------|-------------------------------------------------------------------------------------------------------------|
| boolean add(E e)                          | Ajouter un élément à la collection (optionnelle)                                                            |
| boolean addAll(Collection<? extends E> c) | Ajouter tous les éléments de la collection fournie en paramètre dans la collection (optionnelle)            |
| void clear()                              | Supprimer tous les éléments de la collection (optionnelle)                                                  |
| boolean contains(Object o)                | Retourner un booléen qui précise si l'élément est présent dans la collection                                |
| boolean containsAll(Collection<?> c)      | Retourner un booléen qui précise si tous les éléments fournis en paramètre sont présents dans la collection |
| boolean equals(Object o)                  | Vérifier l'égalité avec la collection fournie en paramètre                                                  |
| int hashCode()                            | Retourner la valeur de hachage de la collection                                                             |
| boolean isEmpty()                         | Retourner un booléen qui précise si la collection est vide                                                  |

# Méthodes (2)

| Méthode                            | Rôle                                                                                                                                                                                                     |
|------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Iterator<E> iterator()             | Retourner un Iterator qui permet le parcours des éléments de la collection                                                                                                                               |
| boolean remove(Object o)           | Supprimer un élément de la collection s'il est présent (optionnelle)                                                                                                                                     |
| boolean removeAll(Collection<?> c) | Supprimer tous les éléments fournis en paramètre de la collection s'ils sont présents (optionnelle)                                                                                                      |
| boolean retainAll(Collection<?> c) | Ne laisser dans la collection que les éléments fournis en paramètre : les autres éléments sont supprimés (optionnelle). Elle renvoie un booléen qui précise si le contenu de la collection a été modifié |
| int size()                         | Retourner le nombre d'éléments contenus dans la collection                                                                                                                                               |
| Object[] toArray()                 | Retourner un tableau contenant tous les éléments de la collection                                                                                                                                        |
| <T> T[] toArray(T[] a)             | Retourner un tableau typé de tous les éléments de la collection                                                                                                                                          |

# Remarques

- Certaines méthodes de cette interface peuvent lever une exception de type **UnsupportedOperationException** car leur implémentation est optionnelle :
  - add(),
  - addAll(),
  - remove(),
  - removeAll(),
  - retainAll()
  - clear()
- Cette exception peut aussi être levée si l'opération n'a aucune influence sur l'état de la collection.

# INTERFACE ITERATOR

# Interface Iterator (1)

- Définit méthodes pour des objets capables de parcourir les données d'une collection

| Méthode           | Rôle                                                                   |
|-------------------|------------------------------------------------------------------------|
| boolean hasNext() | Indiquer s'il reste au moins un élément à parcourir dans la collection |
| Object next()     | Renvoyer le prochain élément dans la collection                        |
| void remove()     | Supprimer le dernier élément parcouru                                  |

# Exemple d'usage

- Affichage de tous les éléments

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

- supprimer suppression du premier élément

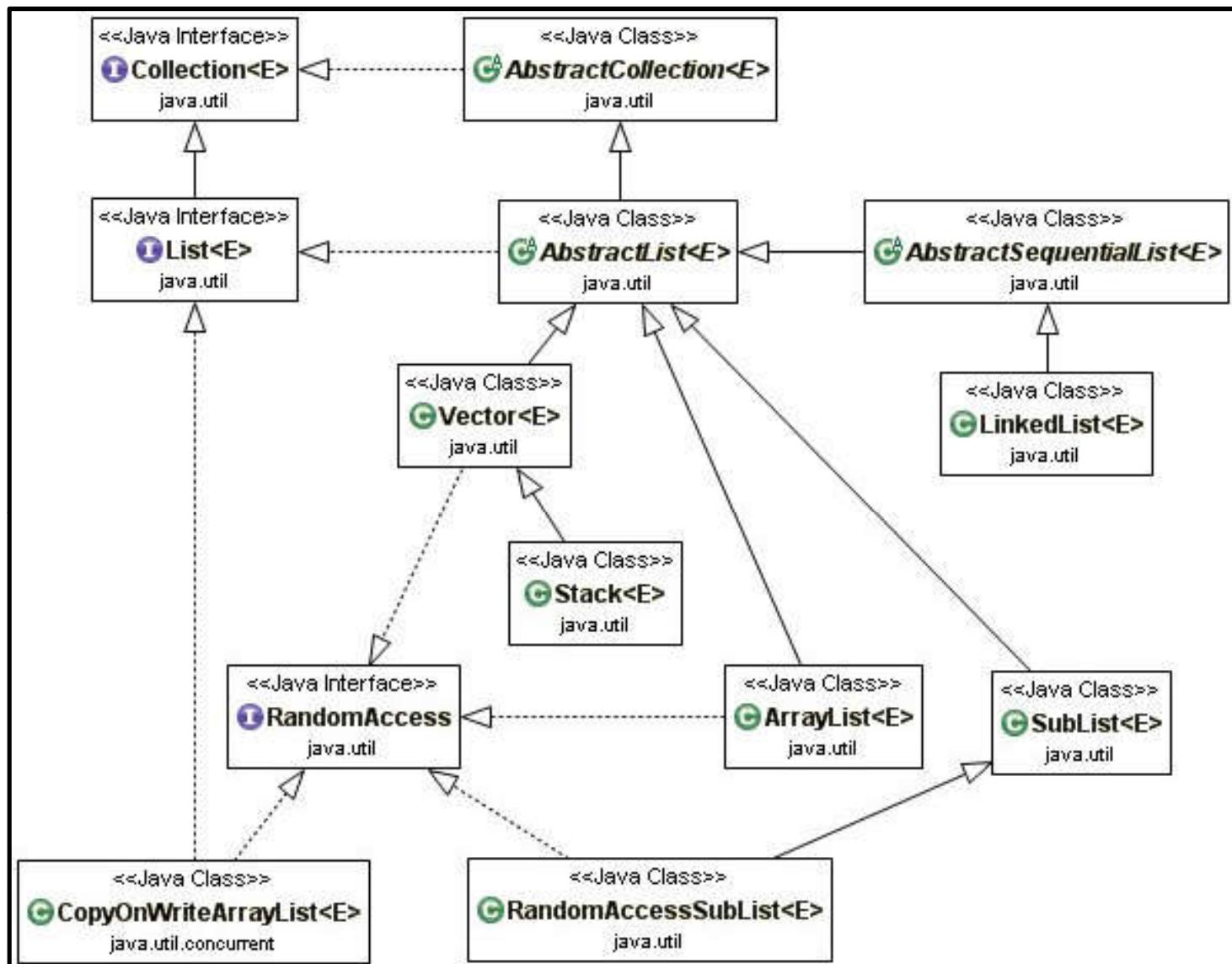
```
if (iterator.hasNext()) {
 iterator.next();
 iterator.remove();
}
```

# INTERFACE LIST

# Collections de type List (1)

- Une collection de type List est une collection
  - simple et
  - ordonnée d'éléments qui
  - autorise les doublons.
- La liste étant ordonnée, un élément peut être accédé à partir de son index.

## Hiérarchie collections de type List



# Collections de type List (2)

| Implémentation                               | Rôle                                                                                                                                                                                                                 |
|----------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| java.util.Vector<E>                          | Une implémentation thread-safe fournie depuis Java 1.0                                                                                                                                                               |
| java.util.Stack<E>                           | Une implémentation d'une pile : elle hérite de la classe Vector et fournit des opérations pour un comportement de type LIFO (Last In First Out)                                                                      |
| java.util.ArrayList<E>                       | Une implémentation qui n'est pas synchronized, donc à n'utiliser que dans un contexte monothread                                                                                                                     |
| java.util.LinkedList<E>                      | Une implémentation qui n'est pas synchronized d'une liste doublement chaînée. Les insertions de nouveaux éléments sont très rapides                                                                                  |
| java.util.concurrent.CopyOnWriteArrayList<E> | Une variante thread-safe de la classe ArrayList dans laquelle toutes les opérations de modification du contenu de la liste recréent une nouvelle copie du tableau utilisé pour stocker les éléments de la collection |

# Interface List

- Cette interface, ajoutée à Java 1.2, étend l'interface Collection.
- Une collection de type List permet de :
  - contenir des doublons
  - interagir avec un élément de la collection en utilisant sa position
  - insérer des éléments null

# Méthodes de List (1)

- **List** définit méthodes qui permettent de
  - Accéder aux éléments de liste à partir d'un index,
  - Gérer les éléments,
  - Rechercher la position d'un élément,
  - Obtenir une liste partielle (sublist)
  - Obtenir des **Iterator**

# Méthodes de List (2)

| Méthode                                              | Rôle                                                                                                                                                |
|------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------|
| void add(int index, E e)                             | Ajouter un élément à la position fournie en paramètre                                                                                               |
| boolean addAll(int index, Collection<? extends E> c) | Ajouter des éléments à la position fournie en paramètre                                                                                             |
| E get(int index)                                     | Retourner l'élément à la position fournie en paramètre                                                                                              |
| int indexOf(Object o)                                | Retourner la première position dans la liste du premier élément fourni en paramètre. Elle renvoie -1 si l'élément n'est pas trouvé                  |
| int lastIndexOf(Object o)                            | Retourner la dernière position dans la liste du premier élément fourni en paramètre. Elle renvoie -1 si l'élément n'est pas trouvé                  |
| ListIterator<E> listIterator()                       | Renvoyer un Iterator positionné sur le premier élément de la liste                                                                                  |
| ListIterator<E> listIterator(int idx)                | Renvoyer un Iterator positionné sur l'élément dont l'index est fourni en paramètre                                                                  |
| E remove(int index)                                  | Supprimer l'élément à la position fournie en paramètre                                                                                              |
| E set(int index, E e)                                | Remplacer l'élément à la position fournie en paramètre                                                                                              |
| List<E> subList(int fromIndex, int toIndex)          | Obtenir une liste partielle de la collection contenant les éléments compris entre les index fromIndex inclus et toIndex exclus fournis en paramètre |

# Remarques importantes

- Interface ***ListIterator*** définie pour le parcours dans les deux sens de la liste et réaliser des mises à jour
- Collection de type List retournée par la méthode ***subList()*** est liée à la collection qui a permis sa création.
- Modification de la sous liste est reportée dans la liste originelle.
- Par contre, si un élément est ajouté ou supprimé dans la liste originelle alors une exception de type ***ConcurrentModificationException*** est levée lors d'une utilisation de la sous liste

# Exemple et classes

- Cf . ***TestSubList.java***
  - Regarder le résultat de son exécution
- Classes implémentant l'interface List :
  - Vector,
  - ArrayList,
  - LinkedList
  - CopyOnWriteArrayList.

# VECTOR

02/07/2018

Pr. Mouhamadou THIAM Maître de  
conférences en informatique

607

# Classe Vector (1)

- Présente depuis Java 1.0, est un tableau dont la taille peut varier selon son nombre d'éléments
- A la création d'une instance possible de préciser capacité initiale et taille d'incrémentation en utilisant la surcharge correspondante du constructeur.
- Toutes ses méthodes sont synchronized : donc moins performante que la classe **ArrayList** car elle est thread-safe.

## Classe Vector (2)

- Antérieure à l'API Collections : mise à jour ultérieurement pour implémenter l'interface Liste.
- Des méthodes redondantes : add() et addElement().
- Fréquemment utilisée avant l'API Collections, préférable d'utiliser une des implémentations de l'API Collections.
- Éléments stockés dans l'ordre d'ajout dans la collection.
- Un élément peut être ajouté ou supprimé à n'importe qu'elle position dans la collection.

# ARRAYLIST

# Classe ArrayList (1)

- Tableaux font partis du langage Java et sont facile à utiliser mais leur taille ne peut pas varier.
- ***ArrayList***, ajoutée à Java 1.2 = tableau d'objets dont la taille est dynamique
- Adaptation couteuse car nécessite instantiation nouveau tableau et copie des éléments
- Hérite de classe ***AbstractList*** donc implémente l'interface ***List***.

# Classe ArrayList (2)

- Fonctionnement similaire à la classe Vector.
- Différence avec Vector = Vector multithread.
- Pour une utilisation dans un thread unique,
  - Synchronisation des méthodes inutile et coûteuse.
  - Donc préférable d'utiliser objet de classe ArrayList.

# Classe ArrayList (3)

- Plus simple implémentation de l'interface List.
  - n'est pas thread-safe
  - utilise un tableau pour stocker ses éléments : 1<sup>er</sup> élément de la collection possède l'index 0
  - accès à un élément se fait grâce à son index
  - implémente toutes méthodes de l'interface List
  - autorise l'ajout d'éléments null

# Constructeurs de ArrayList

| Constructeur                         | Rôle                                                                                                                           |
|--------------------------------------|--------------------------------------------------------------------------------------------------------------------------------|
| ArrayList()                          | Créer une instance vide de la collection avec une capacité initiale de 10                                                      |
| ArrayList(Collection<? extends E> c) | Créer une instance contenant les éléments de la collection fournie en paramètres dans l'ordre obtenu en utilisant son iterator |
| ArrayList(int initialCapacity)       | Créer une instance vide de la collection avec la capacité initiale fournie en paramètre                                        |

# Principales méthodes (1)

| Méthode                         | Rôle                                                                                                                |
|---------------------------------|---------------------------------------------------------------------------------------------------------------------|
| boolean add(Object)             | Ajouter un élément à la fin du tableau                                                                              |
| boolean addAll(Collection)      | Ajouter tous les éléments de la collection fournie en paramètre à la fin du tableau                                 |
| boolean addAll(int, Collection) | Ajouter tous les éléments de la collection fournie en paramètre dans la collection à partir de la position précisée |
| void clear()                    | Supprimer tous les éléments du tableau                                                                              |
| void ensureCapacity(int)        | Augmenter la capacité du tableau pour s'assurer qu'il puisse contenir le nombre d'éléments passé en paramètre       |
| Object get(index)               | Renvoyer l'élément du tableau dont la position est précisée                                                         |
| int indexOf(Object)             | Renvoyer la position de la première occurrence de l'élément fourni en paramètre                                     |

# Principales méthodes (2)

| Méthode                    | Rôle                                                                                                                       |
|----------------------------|----------------------------------------------------------------------------------------------------------------------------|
| boolean isEmpty()          | Indiquer si le tableau est vide                                                                                            |
| int lastIndexOf(Object)    | Renvoyer la position de la dernière occurrence de l'élément fourni en paramètre                                            |
| Object remove(int)         | Supprimer dans le tableau l'élément fourni en paramètre                                                                    |
| void removeRange(int, int) | Supprimer tous les éléments du tableau de la première position fournie incluse jusqu'à la dernière position fournie exclue |
| Object set(int, Object)    | Remplacer l'élément à la position indiquée par celui fourni en paramètre                                                   |
| int size()                 | Renvoyer le nombre d'éléments du tableau                                                                                   |
| void trimToSize()          | Ajuster la capacité du tableau sur sa taille actuelle                                                                      |

# Remarques (1)

- Si le tableau stockage trop petit lors de l'ajout alors un nouveau, plus grand est créé, pour contenir les éléments courant plus le nouvel élément donc
  - temps d'ajout d'un élément pas constant.
  - Temps d'insertion ou suppression (élément à une position) variable puisque cela peut nécessiter l'adaptation de la position d'autres éléments.

## Remarques (2)

- Lors de l'ajout ou le retrait d'un élément, la collection doit réindexer ces éléments.
- Si taille collection est importante + nombreux ajouts et suppression alors préférable d'utiliser ***LinkedList***.
- Méthodes non synchronized par défaut : si plusieurs threads modifient contenu de la collection, utiliser une instance retournée par la méthode ***synchronizedList()*** de la classe ***Collections***.
  - *List liste = Collections.synchronizedList(new ArrayList());*

# Remarques (3)

- **Iterator** et **ListIterator** de la classe `ArrayList` sont de type fail-fast : ils peuvent lever une exception de type **ConcurrentModificationException** si
  - modification du nombre d'éléments durant parcours sans utiliser leurs méthodes `add()` ou `remove()`.
- L'API Collections propose deux solutions pour convertir un tableau en `ArrayList` : méthode
  - `asList()` de la classe **Arrays**
  - `addAll()` de la classe **Collections** : c'est la meilleure solution car elle copie les éléments. Les deux objets peuvent alors être modifiés de manière indépendante

## Remarques (4)

- Méthode ***Arrays.asList()*** facile à utiliser mais les éléments du tableau et de la liste sont liés.
- Modifications faites aux éléments du tableau propagées dans la liste.
- Tentative modification liste lève exception de type ***UnsupportedOperationException***.
- *Solution : appel méthode **Collections.addAll()** car elle copie les éléments et les deux objets peuvent alors être modifiés de manière indépendante.*

# Exemple et

- Cf . *ArrayToArrayList.java*
  - Résultat de son exécution
  - Contenu du tableau
    - ABC D
  - Contenu de la liste
    - ABC D
  - Contenu de la liste
    - AA B C D
  - Exception in thread "main" java.lang.UnsupportedOperationException at  
java.util.AbstractList.add(AbstractList.java:148) at  
java.util.AbstractList.add(AbstractList.java:108) at  
com.jmdoudoux.test.collections.ArrayToArrayList.main(ArrayToArrayList.java:35)

# **CLASSE LINKEDLIST**

# Classe LinkedList (1)

- implémentation liste doublement chaînée
- ajoutée à Java 1.2
- éléments reliés par des pointeurs.
- suppression ou ajout fait en modifiant des pointeurs.
- hérite de classe ***AbstractSequentialList***
- implémente toutes les méthodes, même optionnelles, de ***List***.
- implémente l'interface ***Deque*** à partir de Java 6.

# Classe LinkedList (2)

- Elle possède plusieurs caractéristiques :
  - pas besoin d'être redimensionnée
  - permet l'ajout d'un élément null.
- Elle possède plusieurs constructeurs :

| Constructeur                          | Rôle                                                                                                                                |
|---------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------|
| LinkedList()                          | Créer une nouvelle instance vide                                                                                                    |
| LinkedList(Collection<? extends E> c) | Créer une nouvelle instance contenant les éléments de la collection fournie en paramètre triés dans l'ordre obtenu par son Iterator |

# Exemple

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

# Méthodes

| Méthode               | Rôle                                                                                    |
|-----------------------|-----------------------------------------------------------------------------------------|
| void addFirst(Object) | Insérer l'objet au début de la liste                                                    |
| void addLast(Object)  | Insérer l'objet à la fin de la liste                                                    |
| Object getFirst()     | Renvoyer le premier élément de la liste                                                 |
| Object getLast()      | Renvoyer le dernier élément de la liste                                                 |
| Object removeFirst()  | Supprimer le premier élément de la liste et renvoie l'élément qui est devenu le premier |
| Object removeLast()   | Supprimer le dernier élément de la liste et renvoie l'élément qui est devenu le dernier |
| String toString()     | renvoyer une chaîne qui contient tous les éléments de la liste                          |

# Remarques (1)

- Gérer une collection de façon ordonnée
- Ajouter au début ou à la fin de la collection
- Utiliser LinkedList plus avantageuse que ArrayList lorsque ajout ou suppression en dehors de début ou de fin.
- temps d'exécution opérations toujours constant car = simplement manipulation de pointeurs

## Remarques (2)

- pas de moyen d'accès direct
- Pourtant
  - méthode contains() permet de savoir si un élément est contenu dans la liste et la méthode
  - get() permet d'obtenir l'élément à la position fournie
- ces méthodes parcourrent la liste jusqu'à obtention du résultat (gourmand en de temps de réponse si get() est appelée dans une boucle).
- Donc ne pas utiliser la get() pour parcourir liste.

## Remarques (3)

- Méthodes non synchronized.
  - Si accès multi-threads avec modification de la structure de la liste (ajout ou suppression)
  - Alors créer une instance de type **List** en invoquant la méthode **synchronizedList()** de la classe **Collections** avec l'instance de type **List**.
  - **List liste = Collections.synchronizedList(new LinkedList());**

# Remarques (4)

- **Iterator** et **ListIterator** de la classe `LinkedList` sont de type fail-fast : ils peuvent lever une exception de type **ConcurrentModificationException** si
  - Modification de la structure durant parcours
- Ajout élément après n'importe quel autre lié à la position courante lors d'un parcours
- Pour ce besoin, l'interface **ListIterator** utilisée pour le parcours est une sous-classe de l'interface **Iterator**

# Comparaison avec ArrayList

|                       | LinkedList                      | ArrayList                                   |
|-----------------------|---------------------------------|---------------------------------------------|
| Stockage              | liste doublement<br>chainée     | interne dans un<br>tableau à taille<br>fixe |
| Accès                 | Parcours de la<br>liste         | Direct                                      |
| Cout de variation     | Faible                          | Très couteux                                |
| Ajout début ou<br>fin | Performant et<br>temps constant | Non performant<br>et temps variable         |

# INTERFACE LISTITERATOR

# ListIterator

- définit fonctionnalités Iterator permet aussi
  - parcours en sens inverse de la collection
  - ajout d'un élément ou la modification du courant.

# Méthodes

| Méthode               | Rôle                                                                                  |
|-----------------------|---------------------------------------------------------------------------------------|
| void add(E e)         | Ajouter un élément dans la collection                                                 |
| boolean hasPrevious() | Retourner true si l'élément courant possède un élément précédent                      |
| int nextIndex()       | Retourner l'index de l'élément qui serait retourné en invoquant la méthode next()     |
| E previous()          | Retourner l'élément précédent dans la liste                                           |
| int previousIndex()   | Retourner l'index de l'élément qui serait retourné en invoquant la méthode previous() |
| void set(E e)         | Remplacer l'élément courant par celui fourni en paramètre                             |

# Performances fonctionnalités de base

|                      | get  | add  | contains | next | remove(0) | iterator.remove |
|----------------------|------|------|----------|------|-----------|-----------------|
| ArrayList            | O(1) | O(1) | O(n)     | O(1) | O(n)      | O(n)            |
| LinkedList           | O(n) | O(1) | O(n)     | O(1) | O(1)      | O(1)            |
| CopyOnWriteArrayList | O(1) | O(n) | O(n)     | O(1) | O(n)      | O(n)            |

# Plan

1. Rappels POO et Java
2. Classe abstraite, Interface et Type générique
3. Exceptions Java et Javadoc
4. Swing: Création d'interfaces graphiques
5. Entrées/Sorties (Fichiers Textes et binaires)
6. Collections
7. JDBC
8. Compléments

# **JDBC**

02/07/2018

Pr. Mouhamadou THIAM Maître de  
conférences en informatique

637

# JDBC

- JDBC est une marque déposée de Sun
  - Considérée comme standard pour Java Database Connectivity
- Java très standardisé, mais beaucoup de versions de SQL
- JDBC = méthodes d'accès aux BD SQL à partir de Java
  - JDBC est API standard pour les programmes Java
  - JDBC dispose de spécification de pilotes d'accès de tiers à des versions spécifiques de SQL

# Driver types

- 4 types de pilotes :
  - **JDBC Type 1 Driver** -- JDBC/ODBC Bridge drivers
    - ODBC (Open DataBase Connectivity) est une API standard indépendante des langages de programmation
    - Sun fournit une implémentation de JDBC/ODBC
  - **JDBC Type 2 Driver** – utilise une API spécifique à la plateforme pour l'accès aux données
  - **JDBC Type 3 Driver** -- 100% Java, utilise un protocole réseau pour accéder à un écouteur distant et traduit les appels en *vendor-specific calls*
  - **JDBC Type 4 Driver** -- 100% Java
    - Plus efficace que tous les autres types

# Connector/J

- Connector/J est un JDBC Type 4 Driver pour connecter Java à MySQL
- Installation très simple:
  - Télécharger le “Production Release” ZIP file from <http://dev.mysql.com/downloads/connector/j/3.1.html>
  - Dézipper le
  - Mettre le fichier JAR où Java peut le trouver
    - Ajouter le fichier JAR dans votre CLASSPATH, ou
    - Dans Eclipse: Project --> Properties --> Java Build Path --> Libraries --> Add External Jars...

# Connection au serveur

- D'abord assurez vous que le serveur MySQL est en marche
  - Dans votre programme,
    - `import java.sql.Connection; // non com.mysql.jdbc.Connection  
import java.sql.DriverManager;  
import java.sql.SQLException;`
    - Enregistrer le pilote JDBC,  
`Class.forName("com.mysql.jdbc.Driver").newInstance();`
    - Appeler la méthode `getConnection()`,  
`Connection con =  
DriverManager.getConnection("jdbc:mysql:///myDB",  
myUserName,  
myPassword);`
    - Ou `getConnection("jdbc:mysql:///myDB?user=dave&password=xxx")`

# Un programme complet

- ```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;

public class JdbcExample1 {

    public static void main(String args[]) {
        Connection con = null;
        try {
            Class.forName("com.mysql.jdbc.Driver").newInstance();
            con = DriverManager.getConnection("jdbc:mysql:// /test", "root", "rootpswd");
            if (!con.isClosed())
                System.out.println("Successfully connected to MySQL server...");
        } catch(Exception e) {
            System.err.println("Exception: " + e.getMessage());
        } finally {
            try {
                if (con != null)
                    con.close();
            } catch(SQLException e) {}
        }
    }
}
```

Utilisation de l'objet Connection

- `public Statement createStatement()
throws SQLException`
 - Crée un objet **Statement** pour l'envoie de requêtes SQL à la database. Requêtes SQL sans paramètres sont normalement exécutées avec les objets **Statement**.
 - L'objet **Statement** peut être réutilisés pour plusieurs instructions

Utilisation de l'objet Connection

- `public PreparedStatement prepareStatement(String sql)
throws SQLException`
 - Crée un objet **PreparedStatement** pour l'envoie de requêtes SQL paramétrées à la database.
 - Une requête SQL avec ou sans paramètres **IN** peut être précompilée et stockée dans un objet **PreparedStatement**. Cet objet peut être utilisé pour exécuter efficacement cette instruction plusieurs fois.

Lancer des requêtes

- Les méthodes suivantes sont de l'objet **Statement** :
 - `int executeUpdate()` – pour l'émission de requêtes qui modifient la database et ne retournent pas de résultats
 - Utiliser pour DROP TABLE, CREATE TABLE, and INSERT
 - Retourne le nombre de lignes dans la table résultante
 - `ResultSet executeQuery()` – pour des requêtes devant retourner un ensemble de résultat.
 - Retourne des résultats sous forme d'objet **ResultSet**

Creation d'une table

- Cet exemple vient de <http://www.kitebird.com/articles/jdbc.html>
- ```
CREATE TABLE animal (
 id INT UNSIGNED NOT NULL AUTO_INCREMENT,
 PRIMARY KEY (id),
 name CHAR(40),
 category CHAR(40)
)
```
- ```
Statement s = conn.createStatement ();
s.executeUpdate ("DROP TABLE IF EXISTS animal");
s.executeUpdate (
    "CREATE TABLE animal (
        + "id INT UNSIGNED NOT NULL AUTO_INCREMENT,"
        + "PRIMARY KEY (id),"
        + "name CHAR(40), category CHAR(40))");
```

Insertion dans la table

- ```
int count;
count = s.executeUpdate (
 "INSERT INTO animal (name,
category)"
 + " VALUES"
 + "('snake', 'reptile'),"
 + "('frog', 'amphibian'),"
 + "('tuna', 'fish'),"
 + "('raccoon', 'mammal'));

s.close ();
System.out.println (count +
 " rows were inserted ");
```

# L'objet ResultSet

- `executeQuery()` retourne un **ResultSet**
  - **ResultSet** dispose d'un grand nombre de méthodes `getXXX`, parmi lesquelles
    - `public String getString(String columnName)`
    - `public String getString(int columnIndex)`
  - Résultats sont retournés à partir de la ligne courante
  - Il est possible d'itérer sur les lignes en utilisant :
    - `public boolean next()`
- Les objets **ResultSet**, comme ceux **Statement**, doivent être fermés quant ils ne sont plus utilisés
  - `public void close()`

# Exemple, suite

- Statement s = conn.createStatement ();  
s.executeQuery ("SELECT id, name, category " +  
"FROM animal");  
ResultSet rs = s.getResultSet ();  
int count = 0;  
  
// Loop (next slide) goes here  
  
rs.close ();  
s.close ();  
System.out.println (count + " rows were retrieved");

# Example, suite

- ```
while (rs.next ()) {  
    int idVal = rs.getInt ("id");  
    String nameVal = rs.getString ("name");  
    String catVal = rs.getString ("category");  
    System.out.println (  
        "id = " + idVal  
        + ", name = " + nameVal  
        + ", category = " + catVal);  
    ++count;  
}
```

Instructions préfabriquées

- Instructions préfabriquées sont précompilées, donc beaucoup plus efficientes

```
– PreparedStatement s;  
s = conn.prepareStatement (  
    "INSERT INTO animal (name, category) VALUES(?,?)");  
s.setString (1, nameVal);  
s.setString (2, catVal);  
int count = s.executeUpdate ();  
s.close ();  
System.out.println (count + " rows were inserted");
```

Gestion des erreurs

```
• try {  
    Statement s = conn.createStatement ();  
    s.executeQuery ("XYZ"); // issue invalid query  
    s.close ();  
}  
catch (SQLException e) {  
    System.err.println ("Error message: "  
                      + e.getMessage ());  
    System.err.println ("Error number: "  
                      + e.getErrorCode ());  
}
```

Plan

1. Rappels POO et Java
2. Classe abstraite, Interface et Type générique
3. Exceptions Java et Javadoc
4. Swing: Création d'interfaces graphiques
5. Entrées/Sorties (Fichiers Textes et binaires)
6. Collections
7. JDBC
8. Compléments

XML ET JAVA

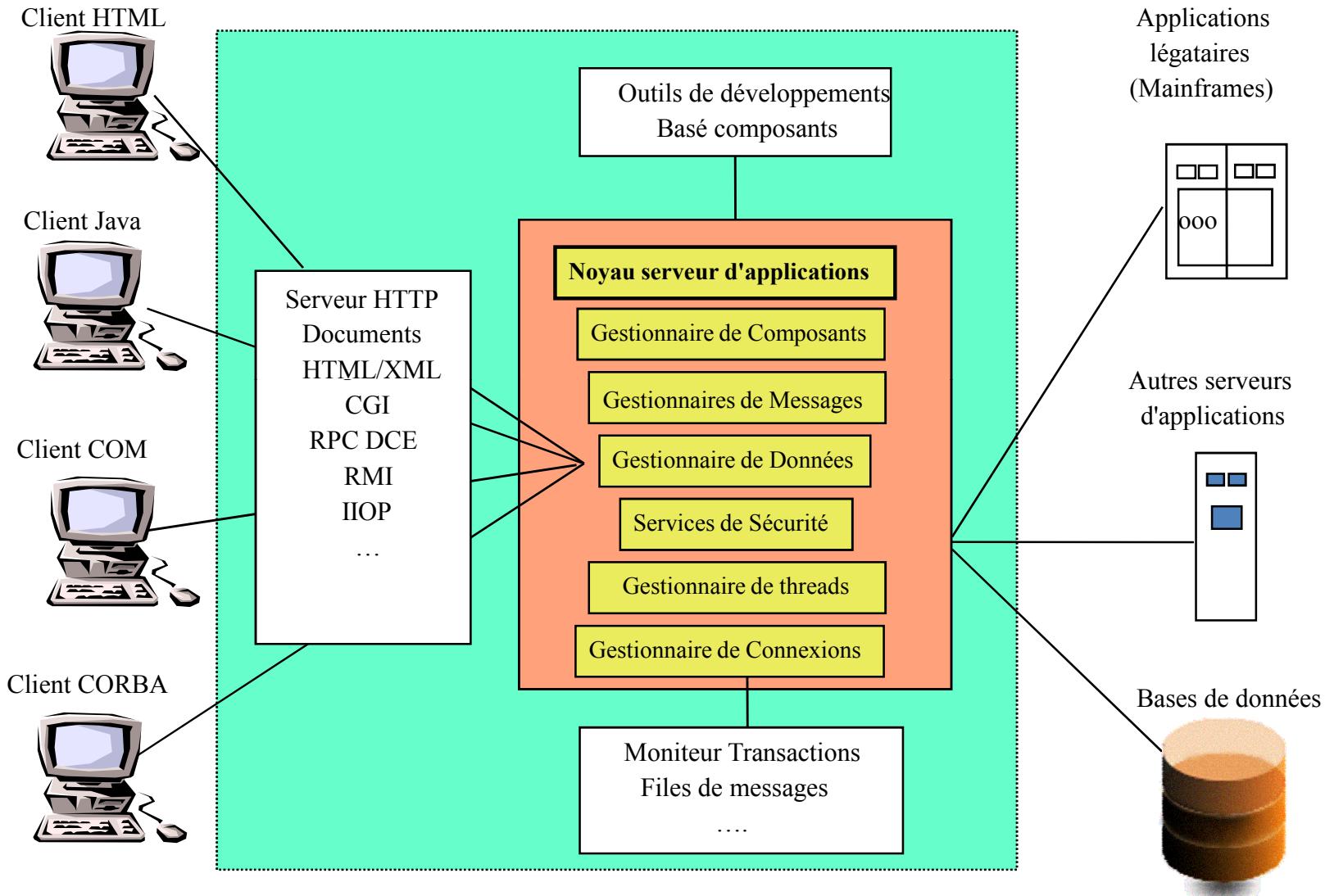
Programmation : XML et Java

1. Introduction
2. Interface DOM
3. Interface SAX
4. Mapping objet - XML
5. Conclusion

Introduction

- Java, un langage portable
- Mais aussi une plate-forme distribuée (J2EE)
 - XML est déjà le standard de description des composants
 - XML devient le standard de communication
- .Net la plate-forme concurrente de Microsoft
- Avantages de XML:
 - échange sécurisé sur HTTP
 - un message porte plusieurs objets
 - interopérabilité des Services Web

Architecture d'un serveur d'appli



Principaux serveurs d'appli

- EJB commerciaux
 - WebLogic de BEA
 - WebSphere Application Server d'IBM
 - iPlanet Application Server de Sun
 - Domino Application Server de Lotus -IBM
 - Oracle Application Server
 - Sybase Enterprise Application
- EJB Open Source
 - Tomcat+Jboss (Open Source)
 - Tomcat+Jonas (Open Source)
- Microsoft .NET, Vista & Indigo

EJB

XML et la POO

- Les services sont souvent programmés en langage objet
 - Java, C++, C#, VB, PHP ...
- Java
 - langage pur objet portable et sûr, semi-interprété
 - le compilateur produit un fichier de byte code par classe
 - possibilité de chargement dynamique de classe
 - support du standard de composants JB et EJB
- XML
 - les messages XML doivent être traduits en objets
 - différents niveau d'interface :
 - Flux d'événements (SAX)
 - Traduction en objet génériques (DOM)
 - Mapping sur objets métiers (Data Binding)

XML et autres langages

- VB, C#
 - Possibilités similaires à Java (parseurs MS)
- C++
 - Possibilités similaires à Java (parseurs libres)
 - Chargement dynamique de classe impossible
- PHP
 - Il existe des parseurs libres

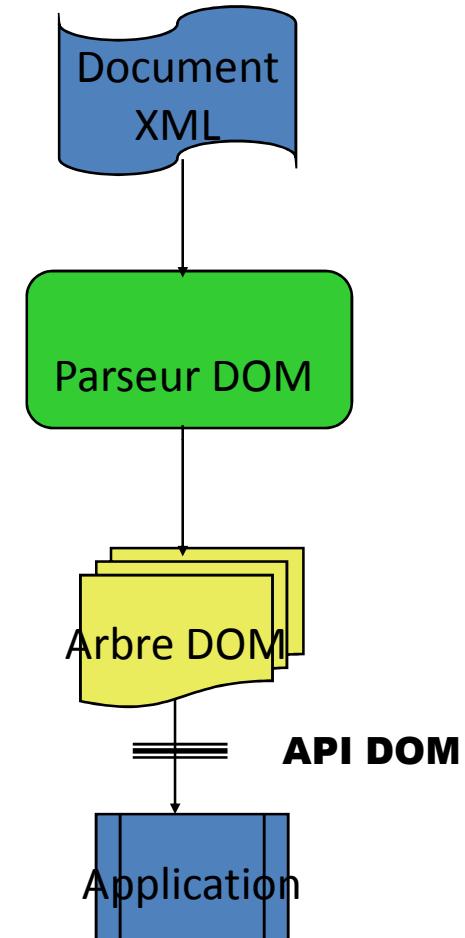
INTERFACE DOM

Présentation

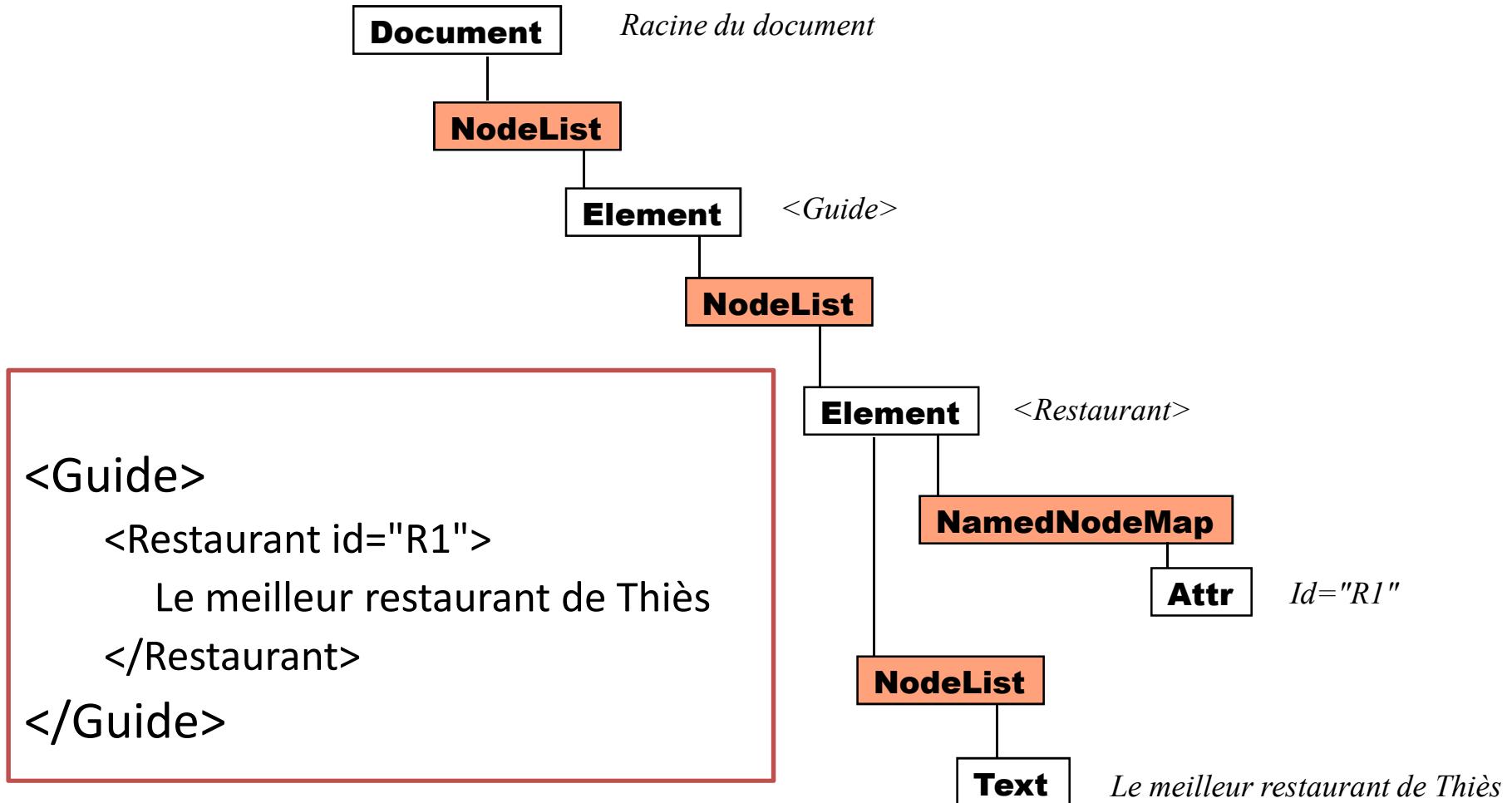
- Standard **W3C** fait pour **HTML** et **XML**
- Structure d'objets pour représenter un document
 - Résultat d'un "**parser**"
 - Arbre d'objets reliés entre eux
- Interface objet pour naviguer dans un document
 - Orientée objet
 - Peut être utilisée en:
 - Java, C++, C#, VB, Python, PHP

Principaux parseurs

Xerces	Apache (Java, C++)
MSXML	Microsoft
SDK Oracle	Oracle
JAXP J2EE	Sun, ...

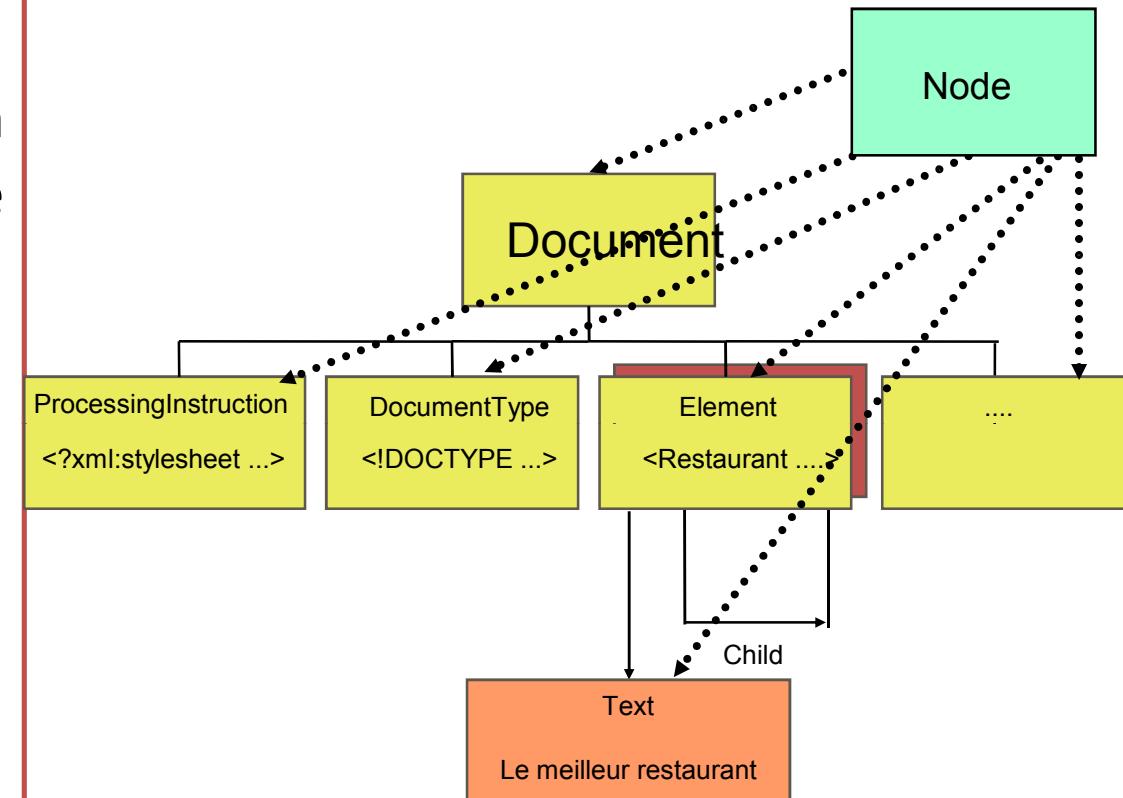


Exemple d'arbre DOM



Gestion de l'arbre DOM

- Navigation via un arbre générique de nœuds
 - Node
 - NodeList (Parent/Child)
 - NamedNodeMap
- Tout nœud hérite de Node



Interface Document

- **createElement** (*Nom_Element*):
 - créer un élément avec le nom spécifié en paramètre.
- **createComment** (*commentaire*):
 - créer une ligne de commentaires dans le document avec le texte *commentaire*.
- **createAttribute** (*Nom_Attribut*):
 - créer un attribut avec le nom pris en paramètre.
- **getElementsByName** (*nom_Tag*):
 - retourne tous les descendants des éléments correspondants au *Nom_Tag*.

Interface Node

- **insertBefore** (*Nouveau_Noeud, Noeud_Reference*):
 - insère un nouveau nœud fils avant le " nœud référence" déjà existant.
- **replaceChild** (*Nouveau_Noeud, Ancien_Noeud*):
 - remplace le nœud "*Ancien_Noeud*" par le nœud "*Nouveau_Noeud*".
- **removeChild** (*Noeud*):
 - supprime le nœud entré en paramètre de la liste des nœuds.
- **appendChild** (*Nouveau_Noeud*):
 - Ajoute un nouveau nœud à la fin de la liste des nœuds.
- **hasChildNodes()**:
 - Retourne VRAI si le nœud possède un enfant et FAUX sinon

Interfaces fondamentales

- DOMImplementation
- Document
- Comment
- DocumentFragment
- Element
- Attr(ibute)
- NamedNodeMap
- CharacterData
 - Comment
 - Text

Autres étendues XML

- ProcessingInstruction
- DocumentType
- CDATASection
- Notation
- Entity
- EntityReference

Exemple simple

- ```
import javax.xml.parsers.*;
import org.w3c.dom.*;

public class DomExample {
 public static void main(String args[]) {
 try {
 ...programme principal est inséré ici...
 } catch (Exception e) {
 e.printStackTrace(System.out);
 }
 }
}
```

# Création du parseur

- Appel de “**DocumentBuilder**” pour créer un parseur DOM
- Parseur non construit par constructeur mais avec une *méthode statique*
  - Bonne technique en programmation avancée en Java
  - Facilite le changement de parseur

`DocumentBuilderFactory factory =`

`DocumentBuilderFactory.newInstance();`

`DocumentBuilder builder =`

`factory.newDocumentBuilder();`

# Chargement du document XML

- Ajouter la ligne suivante au programme  
Document document =  
  
    builder.parse("hello.xml");
- Notes:
  - **document** contient la totalité du fichier XML (sous forme d'arbre); c'est le Document Object Model
  - En ligne de commande le fichier XML dans le même répertoire
  - Si **java.io.FileNotFoundException**, c'est lié à sa visibilité

# Lecture du texte du document XML

- Element root =  
document.getDocumentElement();
- Node textNode = root.getFirstChild();
- System.out.println(textNode.getNodeValue());

# Programme entier

```
import javax.xml.parsers.*;
import org.w3c.dom.*;

public class SecondDom {
 public static void main(String args[]) {
 try {
 DocumentBuilderFactory factory =
 DocumentBuilderFactory.newInstance();

 DocumentBuilder builder =
 factory.newDocumentBuilder();

 Document document = builder.parse("hello.xml");

 Element root = document.getDocumentElement();

 Node textNode = root.getFirstChild();

 System.out.println(textNode.getNodeValue());

 } catch (Exception e) {
 e.printStackTrace(System.out);
 }
 }
}
```

# Illustration

```
import javax.xml.parsers.*;
import org.w3c.dom.*;
import java.io.IOException;

public class ExempleDOM {
 public static void main(String argc[]) throws IOException, DOMException {
 DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
 DocumentBuilder builder;
 try {
 builder = factory.newDocumentBuilder();

 Document xmlDoc = builder.newDocument();
 // creation des noeuds
 Element nom = (Element) xmlDoc.createElement("nom");
 Element prenom = (Element) xmlDoc.createElement("prenom");
 Element nomfam = (Element) xmlDoc.createElement("nomfam");
 // creation de l'arbre
 xmlDoc.appendChild(nom);
 nom.appendChild(prenom);
 prenom.appendChild(xmlDoc.createTextNode("Jean"));
 nom.appendChild(nomfam);
 nomfam.appendChild(xmlDoc.createTextNode("Dupont"));
 // positionnement d'un attribut
 nom.setAttribute("ville", "Paris");
 } catch (ParserConfigurationException e) {
 // TODO Auto-generated catch block
 e.printStackTrace();
 }
 // sortie
 System.exit(0);
 }
}
```

Document:

```
<nom ville ="Paris">
 <prenom> Jean </prenom>
 <nomfam> Dupont </nomfam>
</nom>
```

# Résumé (1)

- Une interface objet standard
  - Navigation dans l'arbre XML
  - Traitements spécifiques
- Performance limitée
  - Place mémoire importante
  - Traitement à la fin de l'analyse
- DOM 2.0
  - Accède dynamiquement au contenu et à la structure du document

# Résumé (2)

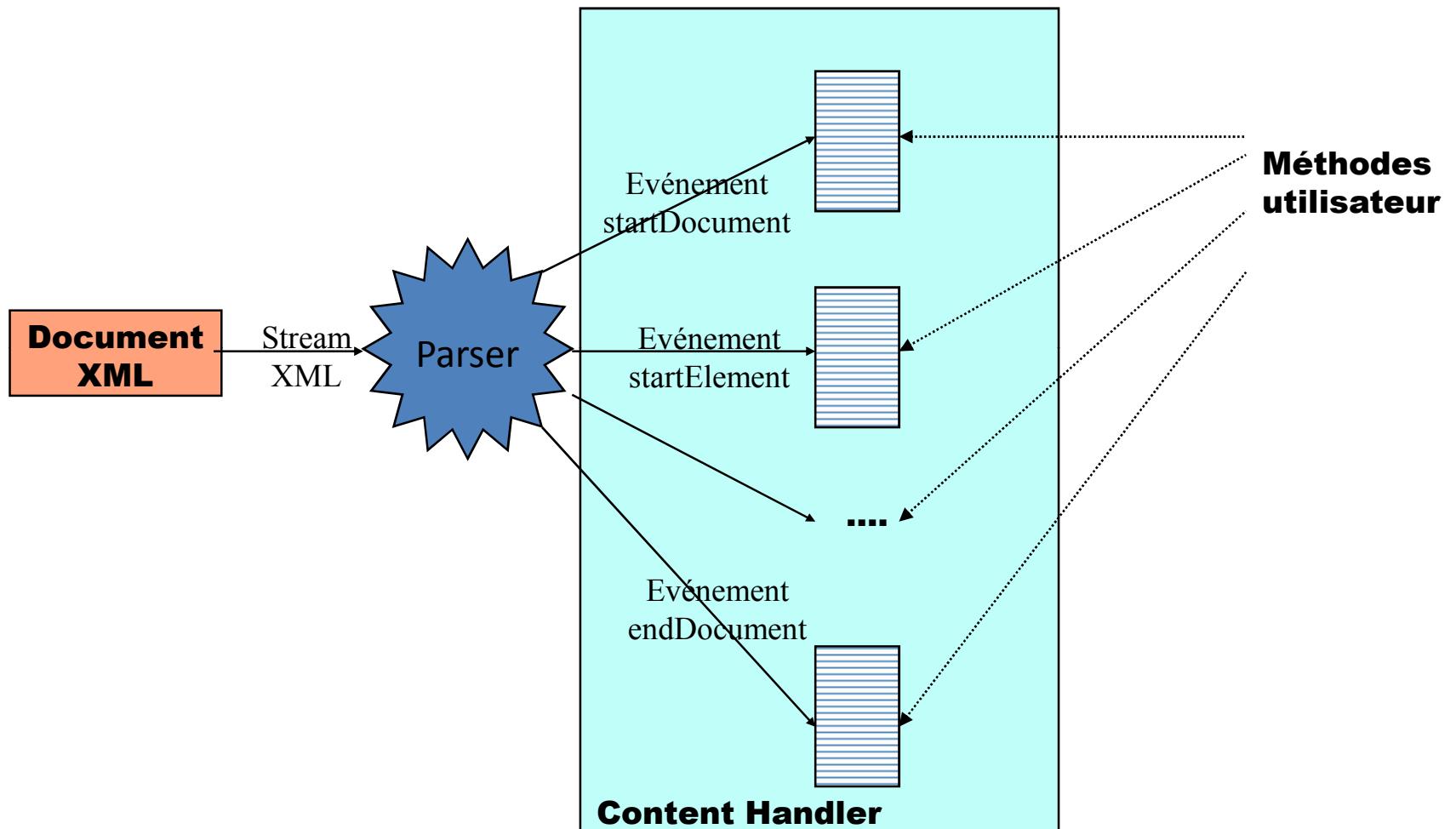
- Extensions en cours :
  - DOM Level 3 : **XPath**
    - Support direct de XPath
    - Définition d'un XPath Evaluator
    - Devrait être intégré aux parsers
  - DOM Level 3 : **Events**
    - Modèle d'événements
    - Associés à un nœud
    - Propagés dans l'arbre DOM
  - DOM Level 3 : **Style**
    - Accès aux styles
    - Mapping complet

# **INTERFACE SAX**

# L'interface SAX

- SAX (Simple API for XML)
  - modèle simplifié d'événements
  - développé par un groupe indépendant.
- Types d'événement :
  - début et fin de document ;
  - début et fin d'éléments ;
  - attributs, texte, ... .
- Nombreux parseurs publics
  - XP de James Clark, Alfred, Saxon
  - MSXML3 de Microsoft
  - Xerces de Apache
  - JAXP de SUN

# Principe de fonctionnement

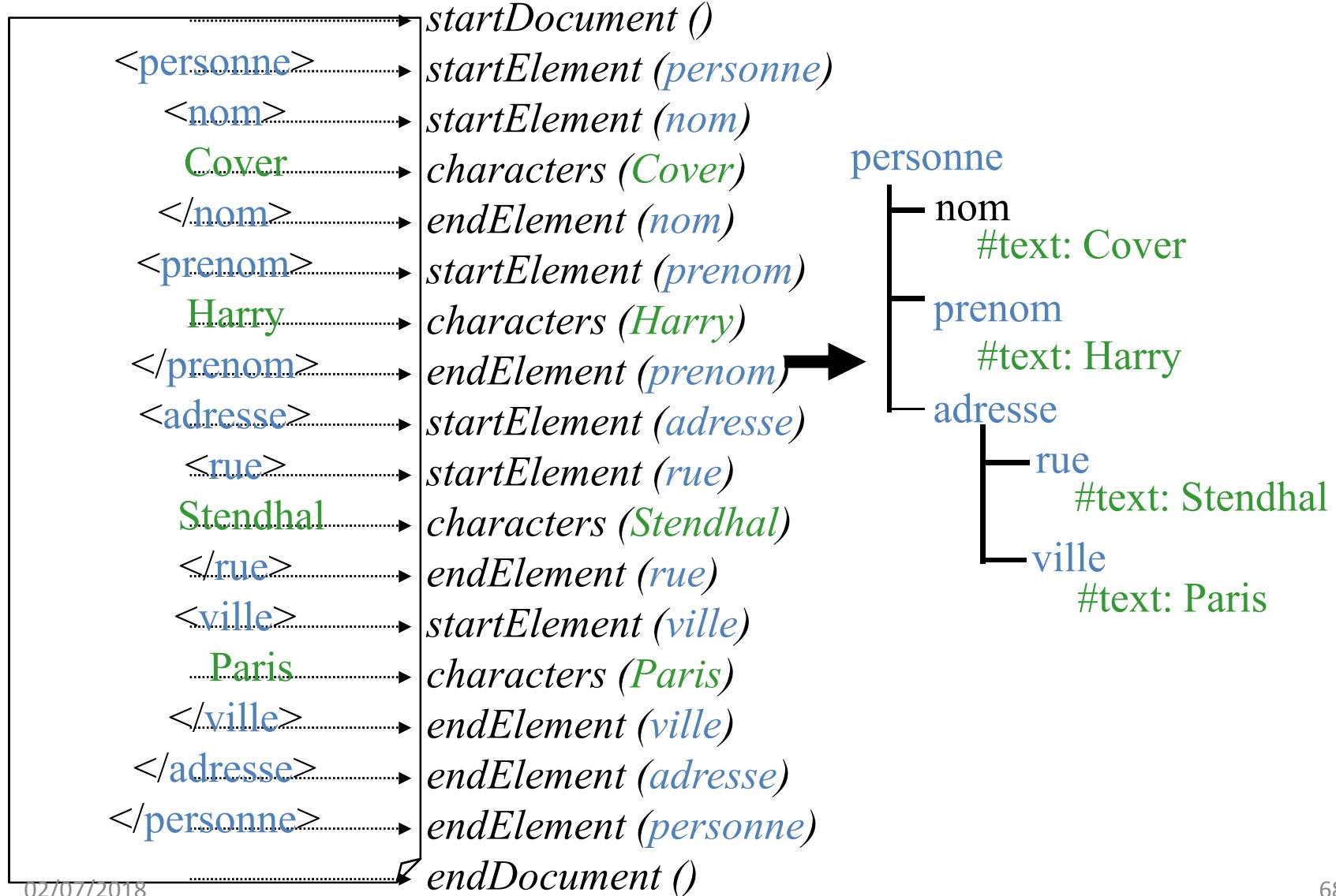


# Les méthodes essentielles

- **XMLReader**
  - setContentHandler
  - setErrorHandler
  - parse
- **ContentHandler**
  - startDocument
  - endDocument
  - startElement
  - endElement
  - characters

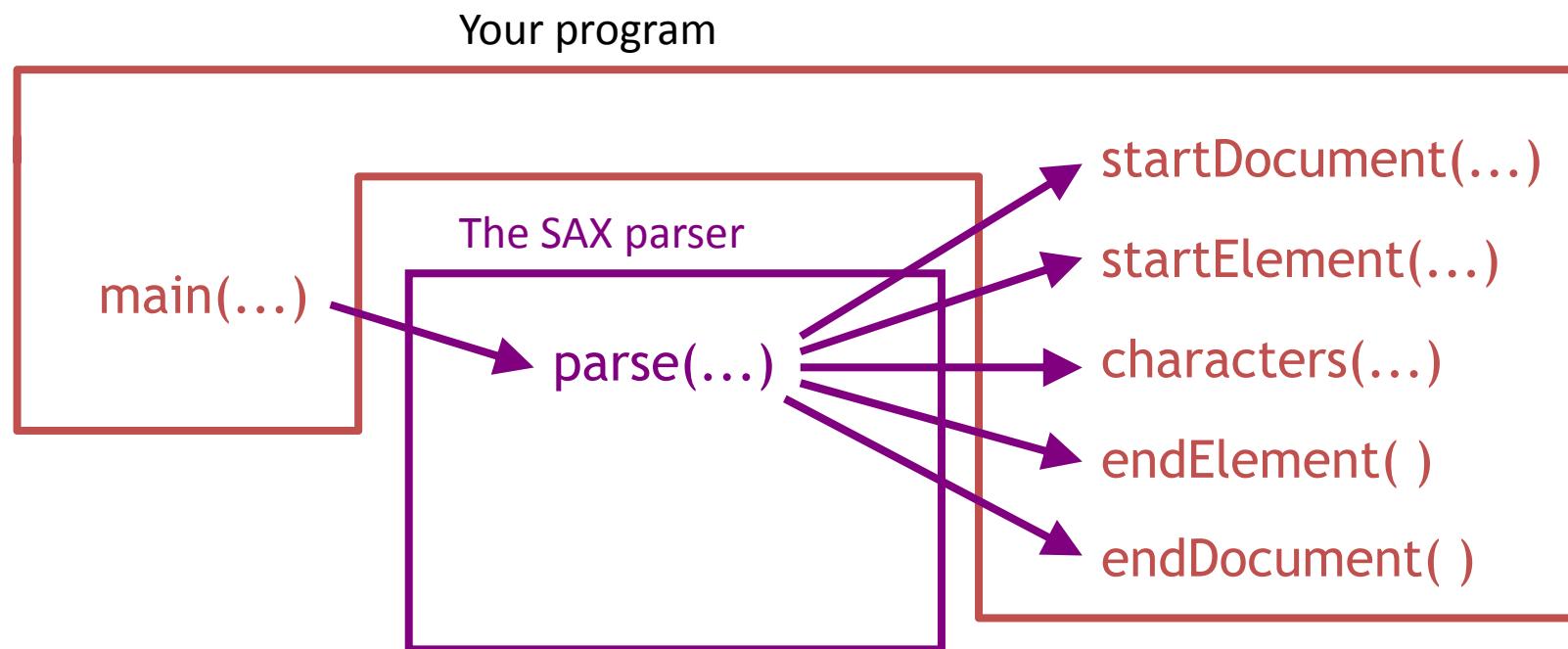
- **ErrorHandler**
  - fatalError
  - error
  - Warning

# Exemple SAX vs DOM



# Fonctionnement

- SAX fonctionne sur la base d'appels : appel au parseur qui appelle les méthodes nécessaires



# Simple programme SAX

Le programme comporte 2 classes:

- **SimpleSAXExemple** – contient la méthode **main**
  - Crée le **factory** des parseurs
  - Récupère un **parseur** avec ce dernier
  - Crée un objet **Handler** qui gère les appels du parseur
  - Associe le handler
  - Lie et parse le flux XML
- **Handler** – contient les handlers pour 3 évènements:
  - **startElement** début de document
  - **endElement** fin élément
  - **characters** texte rencontré

# SimpleSAXExemple

- ```
import javax.xml.parsers.*; // for both SAX and DOM
import org.xml.sax.*;
import org.xml.sax.helpers.*;
```
- ```
// For simplicity, we let the operating system handle exceptions
// In "real life" this is poor programming practice
public class SimpleSAXExemple {
 public static void main(String args[]) throws Exception {
```
- ```
        // Create a parser factory
        SAXParserFactory factory = SAXParserFactory.newInstance();
```
- ```
 // Tell factory that the parser must understand namespaces
 factory.setNamespaceAware(true);
```
- ```
        // Make the parser
        SAXParser saxParser = factory.newSAXParser();
        XMLReader parser = saxParser.getXMLReader();
```

SimpleSAXExemple (suite)

- Création slide précédent du **parser** de type **XMLReader**
- ```
// Create a handler
Handler handler = new Handler();
// Tell the parser to use this handler
parser.setContentHandler(handler);
// Finally, read and parse the document
parser.parse("hello.xml");
} // end of Sample class
```

# Remarques

- Vous aurez besoin de `hello.xml` :
  - En ligne de commande le fichier XML dans le même répertoire
  - Si `java.io.FileNotFoundException`, c'est lié à sa visibilité

# La classe Handler

- `public class Handler extends DefaultHandler {`
  - `DefaultHandler` définit les méthodes de gestion
  - 3 méthodes pour gérer
  - Chaque méthodes peut provoquer une `SAXException`
- `// SAX calls this method when it encounters a start tag`  
`public void startElement(String namespaceURI,`  
`String localName,`  
`String qualifiedName,`  
`Attributes attributes)`  
`throws SAXException {`  
`System.out.println("startElement: " + qualifiedName);`  
`}`

# La classe Handler (suite)

- // SAX calls this method to pass in character data  

```
public void characters(char ch[], int start, int length)
 throws SAXException {
 System.out.println("characters: " +
 new String(ch, start, length) + "\n");
}
```
  - // SAX call this method when it encounters an end tag  

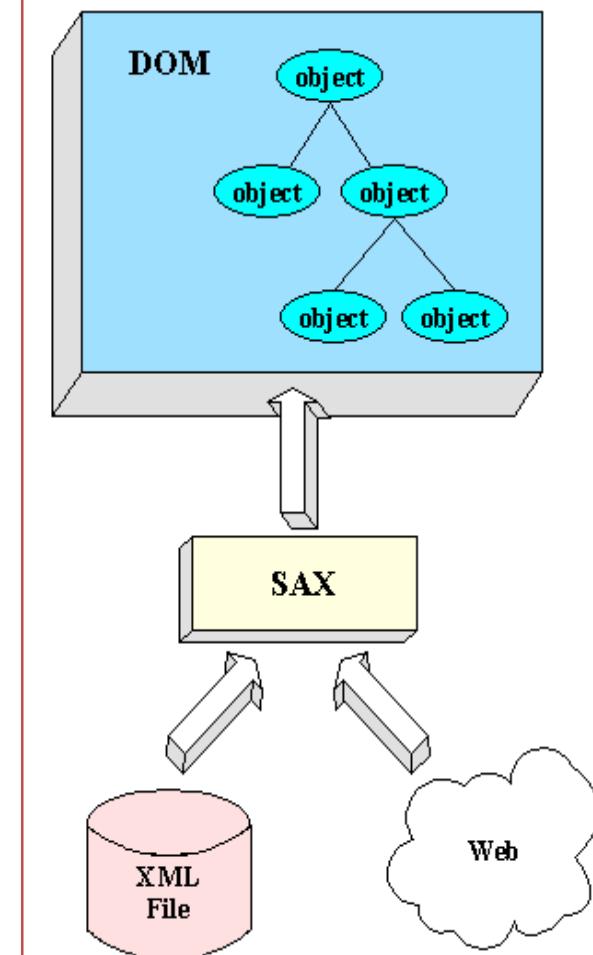
```
public void endElement(String namespaceURI,
 String localName,
 String qualifiedName)
 throws SAXException {
 System.out.println("Element: /" + qualifiedName);
}
```
- } // End of Handler class

# Resultats

- Si hello.xml contient:  
`<?xml version="1.0"?>`  
`<display>Hello World!</display>`
- Le résultat est :  
`startElement: display`  
`characters: "Hello World!"`  
`Element: /display`

# DOM versus SAX

- DOM utilise SAX pour la construction de l'arbre d'un document XML
- SAX est plus léger que DOM
- Au-dessus de DOM, il est possible d'implémenter un « method caller » ...



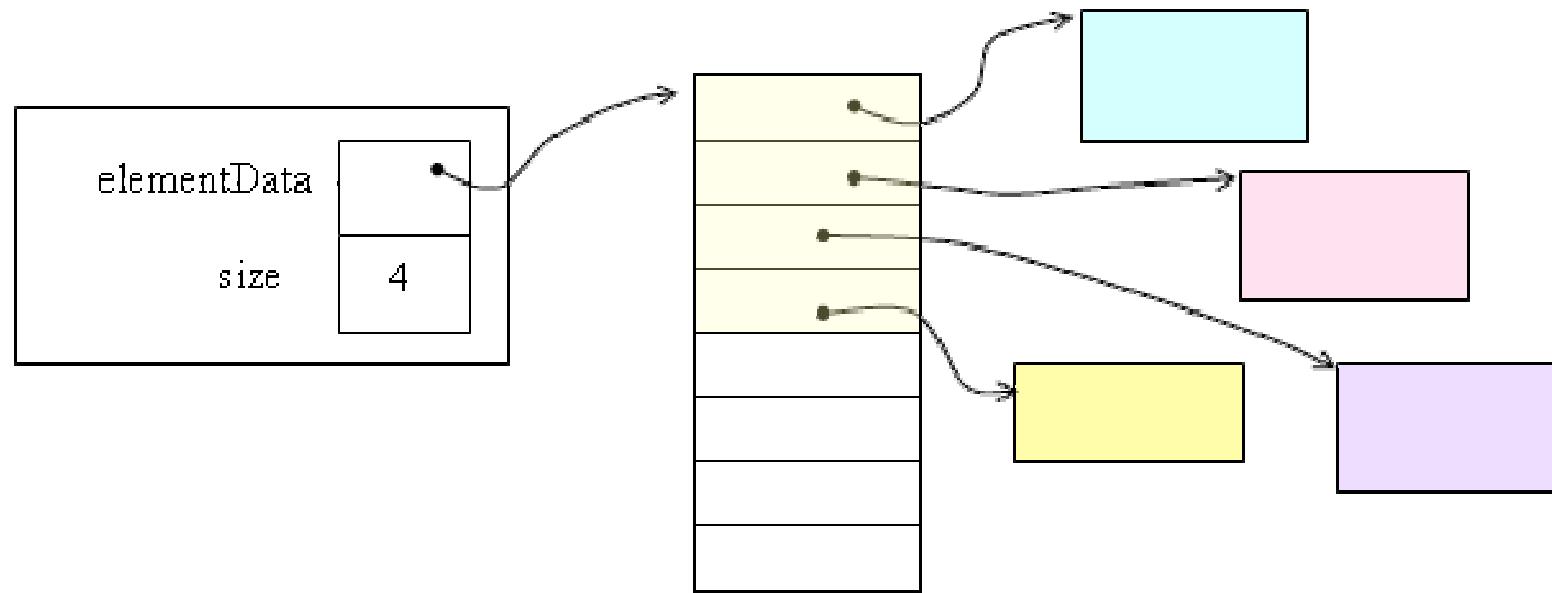
**FIN DU COURS –  
APPROFONDISSEMENT SI TEMPS**

# ARRAYLIST

# Classe ArrayList

- La classe *ArrayList* implémente un tableau d'objets qui peut grandir ou rétrécir à la demande
- débarrasse le programmeur de la gestion de la taille du tableau.
- Comme pour un tableau on peut accéder à un élément du *ArrayList*, par un indice.

# Classe ArrayList : Exemple



Lorsque le tableau est plein (`size == taille du tableau`), et qu'on veut ajouter un élément dans le tableau, le tableau est agrandi : la nouvelle taille est l'ancienne taille \*3/2 plus 1.

# Classe ArrayList : constructeurs

1. un *ListeTableau* avec sa capacité initiale :
  - **publicArrayList(intcapacityInitial)**
2. par défaut construit un *ListeTableau* de 10 emplacements.
  - **public ArrayList() { this(10); }**
3. un *ArrayList* à partir d'une collection d'objets
  - **publicArrayList(Collection**

# Classe ArrayList :méthodes (1)

- Interface *Collection*
  - *add*ajoute un élément en fin de tableau.
  - *clear*enlève tous les éléments du vecteur,
  - *size*retourne le nombre d'éléments du ArrayList
  - *isEmpty*→vraissi le ArrayListestvide.
  - *remove*enlève la première occurrence de *o*
  - *addAll*ajouteune collection

# Classe ArrayList : méthodes(2)

- Interface List
  - `add(index, objet)` ajoute `objet` à l'`index`, lève une exception si l'`indice` n'est pas « correct ». Il déplace tous les éléments d'indices supérieurs ou égaux à `index` pour faire la place avant l'insertion.
  - `get(index)` retourne l'élément se trouvant à un indice, ou lève une exception
  - `indexOf(Object)` et `lastIndexOf(Object)` retournent l'indice d'un objet
  - `remove(int)` enlève l'objet et l'indice et le retourne
  - `set(int, Object)` modifie l'objet se trouvant en position `index` dans le vecteur, en lui affectant `o`, et retourne l'objet qui occupait cette place avant.

# Classe ArrayList : méthodes (2)

- Méthodes propres à la classe
  - *trimToSize* ajuste la capacité du *ArrayList* à son nombre d'éléments.

# **LINKEDLIST**

02/07/2018

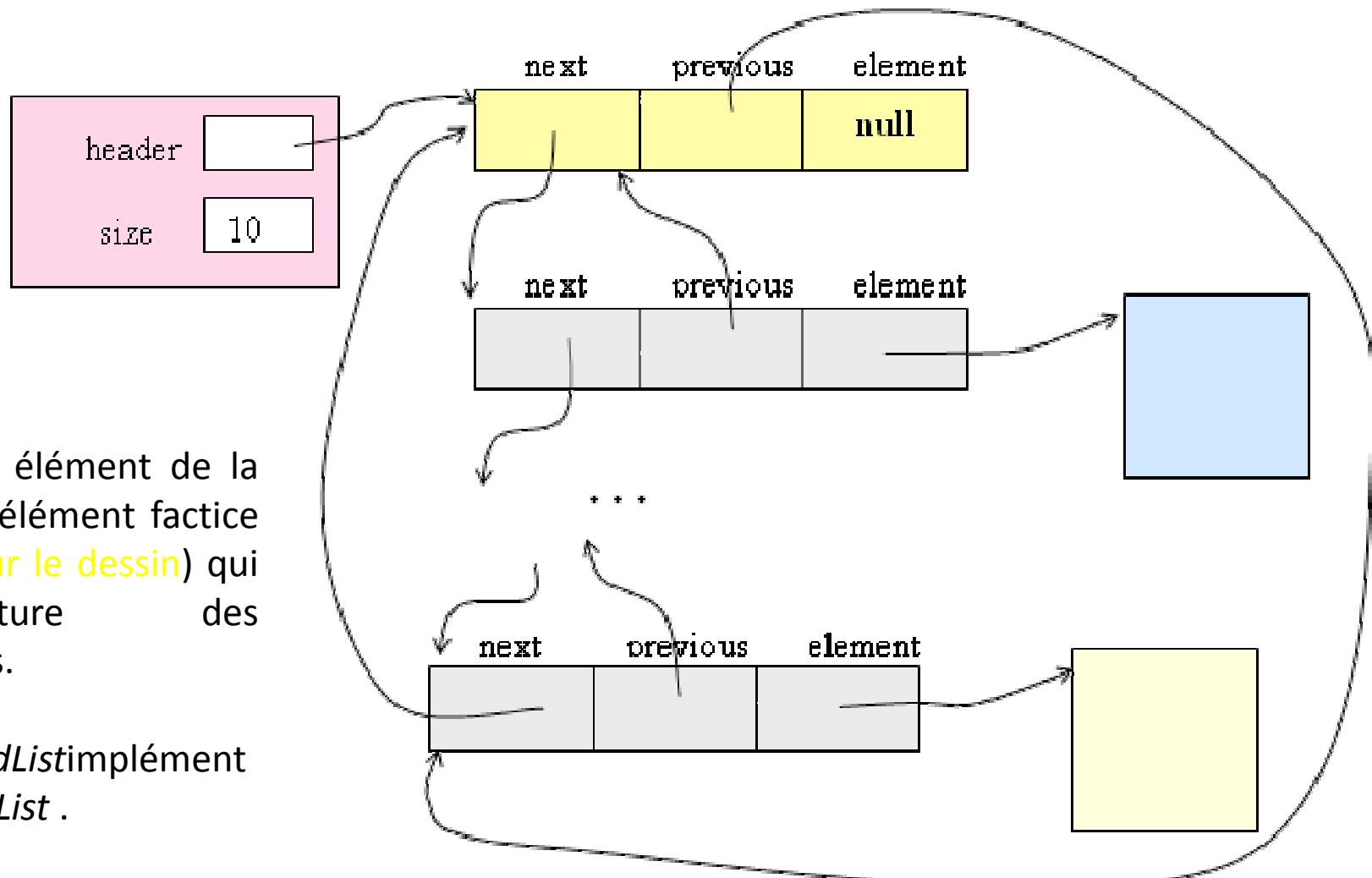
Pr. Mouhamadou THIAM Maître de  
conférences en informatique

701

# Représentation

Le premier élément de la liste est un élément factice (en jaune sur le dessin) qui facilite l'écriture des algorithmes.

La classe *LinkedList* implémente l'interface *List*.



# Constructeurs

- `LinkedList()`
- `LinkedList(Collection<? extends E>)`

# Méthodes (1)

- Outils de la classe
  - remove(e)
  - addBefore(o,e)
  - addAfter(o,e)
  - entry(**int** index)
- Interface Collection
  - add, addAll, clear, isEmpty, size, contains
- Interface List
  - Add, addAll, get, indexOf, lastIndexOf, remove, set

# Méthodes(2)

- Propres à la classe
  - `getFirst()`
  - `getLast()`
  - `removeFirst()`
  - `removeLast()`
  - `addFirst(Object o)`
  - `addLast(Object o)`

# **RAPPELONS QUE ....**

# Les flux de caractères (1)

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

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

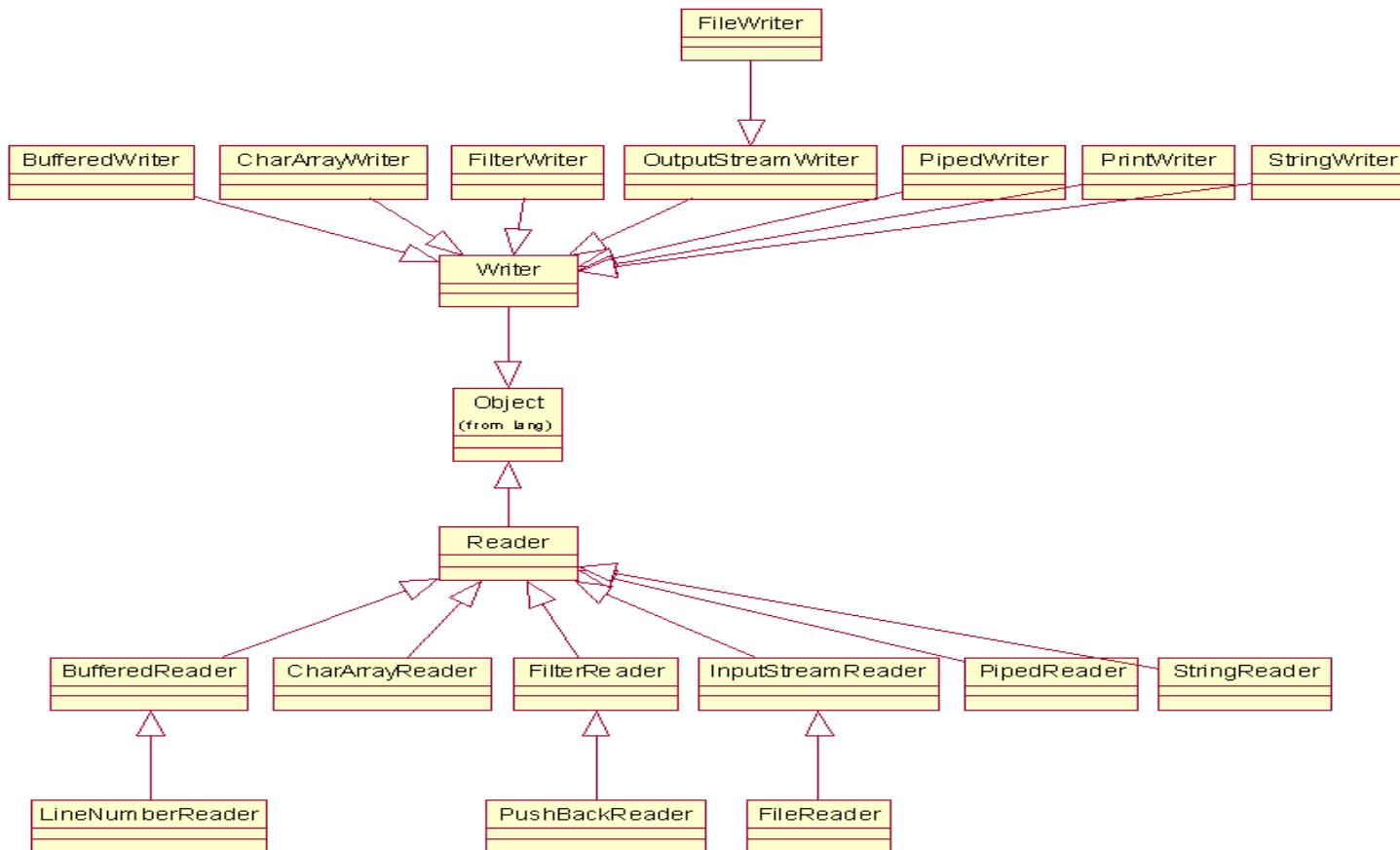
Conversion des caractères d'un fichier  
avec un codage explicitement indiqué

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

# Les flux de caractères (2)

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

# La hiérarchie des flux de caractères



# Les flux de données prédéfinis (1)

Il existe 3 flux prédéfinis :

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

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

# Les flux de données prédéfinis (2)

La classe `InputStream` ne propose que des méthodes élémentaires. Préférez la classe `BufferedReader` qui permet de récupérer des chaînes de caractères.

```
try {
 Reader reader = new InputStreamReader(System.in);
 BufferedReader keyboard = new BufferedReader(reader);

 System.out.print("Entrez une ligne de texte : ");
 String line = keyboard.readLine();
 System.out.println("Vous avez saisi : " + line);

} catch(IOException e) {
 System.out.print(e);}
```

# Les flux de données prédéfinis (3)

L'utilisation de flux "bufferisés" permet d'améliorer considérablement les performances

```
import java.io.*;

class TestVitesseFlux {
 public static void main(String[] args) {
 FileInputStream fis; BufferedInputStream bis;
 try {fis = new FileInputStream(new File("test.txt"));
 bis = new BufferedInputStream(new FileInputStream(new
File("test.txt")));
 byte[] buf = new byte[8];
 long startTime = System.currentTimeMillis();
 while(fis.read(buf) != -1);
 System.out.println("Temps de lecture avec FileInputStream : " +
 (System.currentTimeMillis() - startTime));
 startTime = System.currentTimeMillis();
 while(bis.read(buf) != -1);
 System.out.println("Temps de lecture avec BufferedInputStream : " +
 (System.currentTimeMillis() - startTime));
 fis.close(); bis.close(); }
 catch (FileNotFoundException e) { e.printStackTrace(); } catch
 (IOException e) { e.printStackTrace(); } }
02/07/2018 }
```

# La sérialisation (1)

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

# La sérialisation (2)

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

# Exemple de sérialisation

```
void sauvegarde(String s) {
 try {FileOutputStream f = new FileOutputStream(new File(s));
 ObjectOutputStream oos = new ObjectOutputStream(f);
 oos.writeObject(this);
 oos.close();}
 catch (Exception e)
 { System.out.println("Erreur "+e);}
}
```

```
static Object relecture(String s) {
 try {FileInputStream f = new FileInputStream(new File(s));
 ObjectInputStream oos = new ObjectInputStream(f);
 Object o=oos.readObject();
 oos.close();
 return o;}
 catch (Exception e)
 { System.out.println("Erreur "+e);
 return null;}
```

}  
02/07/2018

- ```
 /**
 * classe permettant de calculer
 */
public class Calculatrice{
    public static void main(String cals[]){
        int a=Integer.parseInt(cals[0]), b=Integer.parseInt(cals[2]);
        char op = cals[1].charAt(0);

        if (op == '+')
            System.out.println("Somme = " + (a+b));

        else if (op == '-')
            System.out.println("Différence = " + (a-b));

        else if (op == 'x')
            System.out.println("Produit = " + (a*b));

        else if (op == '/')
            System.out.println("Quotient = " + (a/b));

        else System.out.println(op + "n'est pas un opérateur");
    }
}
```

HERITAGE

02/07/2018

Pr. Mouhamadou THIAM Maître de
conférences en informatique

717

Introduction

- Pour raccourcir les
 - temps d'écriture et de
 - mise au point du code d'une application,
- il est intéressant de pouvoir réutiliser du code déjà écrit.
- Réutilisation ...

Notions importantes

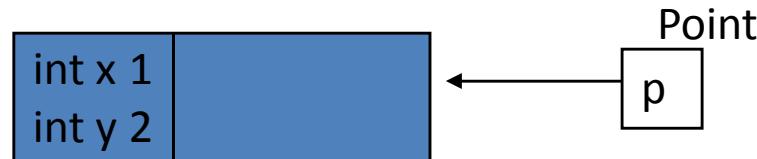
- Classe:
 - Une déclaration ~ un modèle à objet
 - Un type de donnée
 - (Aucun objet concret existe après la déclaration)
- Objet, Instance
 - Une instance d'une classe
 - (Une donnée concrète)

Comparaison

- Notion de **Cours** dans une université
 - Une vue abstraite
 - Une définition correspondant aux caractéristiques générales d'un 'cours'
 - On ne peut pas manipuler un **cours** abstrait
- **Instance**
 - Une instance concrète de cours (e.g. **POO**)
 - Contient des données concrètes (étudiants, prof, livre, ...)
 - Manipulable (inscrire un étudiant dans un cours, ...)
 - (En générale, on appelle une méthode à partir d'un objet/instance: **POO.inscrire(Etudiant)**)
- Relation: **POO** appartient à la classe **Cours**

Exemple

- public class Point{
 private int x, y;
 public Point(int x, int y) {this.x = x, this.y = y;}
 public int getX() {return x;}
 public int getY() {return y;}
 public void setX(int x) {this.x = x;}
 public void setY(int y) {this.y = y;}
}
- Point p = new Point(1,2);



Classe: rappel

- Déclaration d'une classe
 - Spécifier ce dont on a besoin pour une classe par ses comportements
 - Requêtes envoyées à un objet de cette classe pour obtenir des valeurs (accessors)
 - Commandes pour changer l'état de l'objet ou pour exécuter une tâche
 - (On ne spécifie pas les attributs à cette étape)
 - Implantation
 - Définir les attributs (variables, champs) à utiliser

Exemple

- public class Point
{
 private int x, y;

 public int getX() {return x;}
 public int getY() {return y;}
 public void setX(int x) {this.x = x;}
 public void setY(int y) {this.y = y;}
}

Réutilisation Par des classes clientes

- Soit une classe **A** dont on a le code compilé
- Une classe **C** veut réutiliser la classe **A**
- Elle peut créer des instances de **A** et leur demander des services
- On dit que la classe **C** est une classe cliente de la classe **A**

Avec modifications

- Souvent, cependant, on souhaite modifier en partie le comportement de **A** avant de le réutiliser
- Le comportement de **A** convient, sauf pour des détails qu'on aimeraient changer
- Ou alors, on aimeraient ajouter une nouvelle fonctionnalité à **A**

Avec modifications du code source

- On peut copier, puis modifier le code source de **A** dans des classes **A1, A2,...**
- **Problèmes:**
 - on n'a pas toujours le code source de **A**
 - les améliorations futures du code de **A** ne seront pas dans les classes **A1, A2,...**

Par l'héritage (1)

- L'héritage existe dans tous les langages objet à classes
- L'héritage permet d'écrire une classe **B**
 - qui se comporte dans les grandes lignes comme la classe **A**
 - mais avec quelques différences sans toucher au code source de **A**
 - On a seulement besoin du code compilé de **A**

Par l'héritage (2)

- Le code source de **B** ne comporte que ce qui a changé par rapport au code de **A**
- On peut par exemple
 - ajouter de nouvelles méthodes
 - modifier certaines méthodes

Vocabulaire

- 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**

Exemple Java – classe mère

```
public class Rectangle {  
    Private int x, y; // point en haut à gauche  
    Private int largeur, hauteur;  
    // La classe contient des constructeurs,  
    public int getX(){return x;} public void setX(int x){this.x=x;}  
    public int getY(){return y;} public void setY(int y){this.y=y;}  
    public int getHauteur() {return hauteur;} public void setHauteur(int h){hauteur = h;}  
    public int getLargeur() {return largeur;} public void setLargeur(int l){largeur = l;}  
    // contient(Point), intersecte(Rectangle)  
    // translateToi(Vecteur), toString(), ...  
    ...  
    public void dessineToi(Graphics g) {  
        g.drawRect(x, y, largeur, hauteur);  
    }  
}
```

Exemple Java – classe fille

```
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());  
    }  
}
```

Code des classes filles

- Quand on écrit la classe **RectangleColore**, on doit seulement
 - écrire le code (variables ou méthodes) lié aux nouvelles possibilités ; on ajoute ainsi une variable **couleur** et les méthodes qui y sont liées
 - redéfinir certaines méthodes ; on redéfinit la méthode **dessineToi()**

Redéfinition et surcharge

Ne pas confondre redéfinition et surcharge des méthodes : on

- **redéfinit** une méthode quand une nouvelle méthode a la même signature qu'une méthode héritée de la classe mère
- **surcharge** une méthode quand une nouvelle méthode a le même nom, mais pas la même signature, qu'une autre méthode de la même classe

Rappel: Signature d'une méthode (nom de la méthode + ensemble des types de ses paramètres)

Héritage : 2 façons de voir

- Particularisation-généralisation:
 - un rectangle coloré *est un* rectangle mais un rectangle particulier
 - la notion de figure géométrique est une généralisation de la notion de polygone
 - Une classe fille offre de nouveaux services ou enrichit les services rendus par une classe : la classe **RectangleColore** permet de dessiner avec des couleurs et pas seulement en « noir et blanc »

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 RectangleColoreextends Rectangle`
- Par défaut (pas de **extends** dans la définition d'une classe), une classe hérite de la classe **Object**

Exemples d'héritages

- class Voiture **extends** Vehicule
- class Velo **extends** Vehicule
- class VTT **extends** Velo
- class Employe **extends** Personne
- class ImageGIF **extends** Image
- class PointColore **extends** Point

Ce que peut faire une classe fille

La classe qui hérite peut

- ajouter des variables, des méthodes et des constructeurs
- redéfinir des méthodes (exactement les mêmes types de paramètres)
- surcharger des méthodes (même nom mais pas même signature) (**possible aussi à l'intérieur d'une classe**)

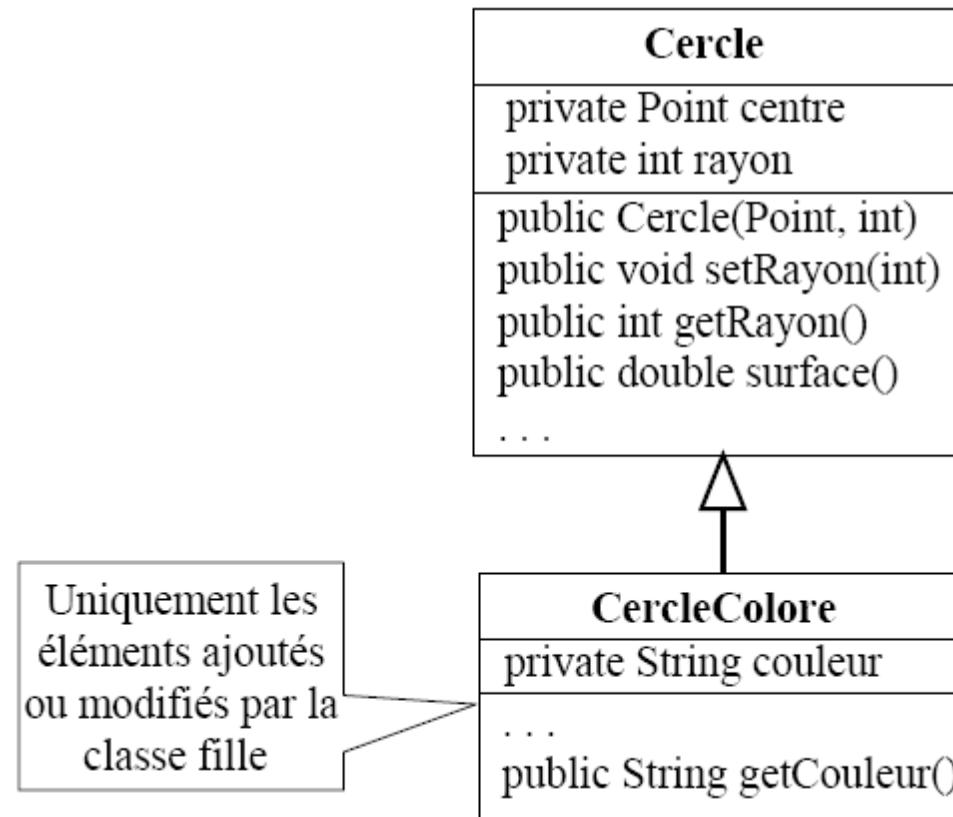
Principe important de la notion d'héritage

- Si « **B extends A** », le grand principe est que tout **B** est un **A**
- Par exemple,
 - un rectangle coloré *est un* rectangle ;
 - un poisson *est un* animal ;
 - une voiture *est un* véhicule
- En Java, on évitera d'utiliser l'héritage pour réutiliser du code dans d'autres conditions

Sous-type

- **B** est un sous-type de **A** si on peut ranger une expression de type **B** dans une variable de type **A**
- Les sous-classes d'une classe **A** sont des sous types de **A**
- En effet, si **B** hérite de **A**, tout **B** est un **A** donc on peut ranger un **B** dans une variable de type **A**
- Par exemple,
A a = new B(...); est autorisé

L'héritage en notation UML



Constructeurs : quoi de neuf?

- 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(...)

Constructeur de la classe mère

```
public class Rectangle {  
    private int x, y, largeur, hauteur;  
  
    public Rectangle(int x, int y, int largeur, int hauteur) {  
        this.x= x;  
        this.y= y;  
        this.largeur= largeur;  
        this.longueur= longueur;  
    }  
    . . .  
}
```

Constructeurs de la classe fille

```
public class RectangleColore extends Rectangle {  
    privateColor 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);  
    }  
    ...  
}
```

Appel implicite du constructeur de la classe mère (1)

- Si la première instruction d'un constructeur n'est ni **super(...)**, ni **this(...)**, le compilateur ajoute un appel implicite au constructeur sans paramètre de la classe mère (erreur s'il n'existe pas)

=> Un constructeur de la classe mère est toujours exécuté avant les autres instructions du constructeur

Appel implicite du constructeur de la classe mère (2)

- Mais la première instruction d'un constructeur de la classe mère est l'appel à un constructeur de la classe « grand-mère », et ainsi de suite...
- Donc la toute, première instruction qui est exécutée par un constructeur est le constructeur (sans paramètre) par défaut de la classe **Object** !
- (D'ailleurs c'est le seul qui sait comment créer un nouvel objet en mémoire)

La classe Cercle(1)

```
public class Cercle {  
    // Constante  
    public static final double PI = 3.14;  
    // Variables  
    private Point centre;  
    Private int rayon;  
    // Constructeur  
    public Cercle(Point c, int r) {  
        centre = c;  
        rayon = r;  
    }  
}
```

Ici pas de constructeur sans paramètre

Appel implicite du constructeur **Object()**

La classe Cercle(2)

// Méthodes

```
public double surface() {
    return PI * rayon * rayon;
}

public Point getCentre() {
    return centre;
}

public static void main(String[] args) {
    Point p = new Point(1, 2);
    Cercle c = new Cercle(p, 5);
    System.out.println("Surface du cercle: " + c.surface());
}
```

La classe CercleColore (1)

```
public class CercleColoreextendsCercle {  
    private String couleur;  
  
    public CercleColore(Point p, int r, String c) {  
        super(p, r);  
        couleur = c; Que se passe-t-il si on enlève  
cette instruction?  
    }  
  
    public voidsetCouleur(String c) {  
        couleur = c;  
    }  
  
    public String getCouleur() {  
        return couleur;  
    }  
}
```

Attention – débutant !

```
public class CercleColore extends Cercle {  
    Private Point centre;  
    Private int rayon;  
    Private String couleur;  
    public CercleColore(Point p, int r, String c) {  
        centre = p;  
        rayon = r;  
        couleur = c;  
    }  
    ...  
}
```

Que se passe-t-il ici?

Accès aux attributs (1)

```
public class Animal {  
    String nom; // remarquer que nom n'est pas private  
    public Animal() {  
    }  
    public Animal(String unNom) {  
        nom = unNom;  
    }  
    public void set Nom(String unNom) {  
        nom = unNom;  
    }  
    public String toString() {  
        return "Animal " + nom;  
    }  
}
```

Accès aux attributs (2)

```
public class Poisson extends Animal {  
    private int profondeurMax;  
  
    public Poisson(String nom, int uneProfondeur) {  
        this.nom= nom; // Et si nom est private ?  
        profondeurMax= uneProfondeur;  
    }  
    public void setProfondeurMax(int uneProfondeur) {  
        profondeurMax= uneProfondeur;  
    }  
    public String toString() {  
        return "Poisson " + nom + " ; plonge jusqu'à "+ profondeurMax + "  
               mètres";  
    }  
}
```

Résolution par encapsulation

```
public class Poisson extends Animal {  
    Private int profondeurMax;  
    public Poisson(String unNom, int uneProfondeur) {  
        super(unNom); // convient même si nom est private  
        profondeurMax= uneProfondeur;  
    }  
    public void setProfondeurMax(intuneProfondeur) {  
        profondeurMax= uneProfondeur;  
    }  
    public String toString() {  
        return "Poisson " + getNom()  
            + " plonge jusqu'à " + profondeurMax  
            + " mètres";  
    }  
}
```

De quoi hérite une classe ?

- Si une classe **B** hérite de **A** (**B extends A**), elle hérite automatiquement et implicitement de tous les membres de la classe **A** (mais pas des constructeurs)
- Cependant la classe **B** peut ne pas avoir accès à certains membres dont elle a implicitement hérité de **A** (par exemple, les membres **private**)
- Ces membres sont utilisés pour le bon fonctionnement de **B**, mais **B** ne peut pas les nommer ni les utiliser explicitement

Redéfinition des attributs

- Soit une classe **B** qui hérite d'une classe **A**
- Dans une méthode d'instance **m()** de **B**,
super. sert à désigner un membre de **A** :
 - en particulier, **super.m()** désigne la méthode de **A** qui est donc en train d'être redéfinie dans **B** :

```
intm(int i) {  
    return 500 + super.m(i);  
}
```

Classe Animal

```
public class Animal {  
    private String proprietaire;  
    private String nom;  
    private boolean vaccine = false;  
  
    public Animal(String p, String n){  
        proprietaire = p;  
        nom = n;  
    }  
  
    public void vacciner() {  
        vaccine = true;  
    }  
  
    public String toString() {  
        return "Proprietaire : " + proprietaire + "\nNom : " + n  
            "\nVaccine : " + vaccine;  
    }  
}
```

Rajouter une classe fille chat avec une méthode "miauler"

```
public class Chien extends Animal{
    private String race;

    public Chien(String r, String p, String n){
        super(p,n);
        race = r;
    }

    public void aboyer() {
        System.out.println("ouah ouah");
    }

    public String toString() {
        String s = super.toString();
        s = s + "\nRace : " + race;
        return s;
    }

    public static void main(String args[]){
        Chien c = new Chien("Caniche","Joe Blow", "Fifi");
        System.out.println(c);
        c.aboyer();
        c.vacciner();
        System.out.println(c);
    }
}
```

Méthode statique

- On ne peut utiliser **super.m()** dans une méthode **static m()** ;
 - une méthode **staticne** peut être redéfinie
 - **super.i**désigne la variable cachée **i** de la classe mère (ne devrait jamais arriver) ; dans ce cas, **super.i**est équivalent à **((A)this).i**

Méthode statique : exemple

```
public class C{
    int i;
}

public class D extends C{
    private int i;

    public void test() {
        i = 0; // D
        this.i = 0; // D
        super.i = 1; // C
        ((C)this).i = 1; // C
    }

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

Accès protected

- **Protected** joue sur l'accessibilité des membres (variables ou méthodes) par les classes filles
- Un membre **protected** de la classe A peut être manipulé par les classes filles de A sans que les autres classes non filles de A ne puisse les manipuler

Exemple

```
public class Animal {  
    protected String nom;  
    ...  
}
```

```
public class Poisson extends Animal {  
    private int profondeurMax;  
    public Poisson(String unNom, int uneProfondeur) {  
        nom = unNom; // utilisation de nom de la classe mère  
        profondeurMax = uneProfondeur;  
    }
```

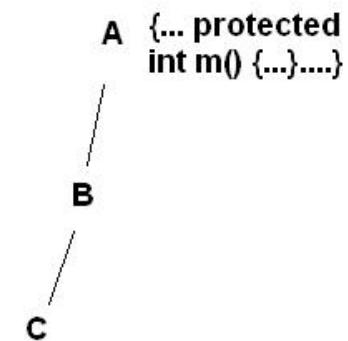
Autre règle

- **Protected** autorise également l'utilisation par les classes du même paquetage que la classe où est définie le membre ou le constructeur

Exemple

- Du code de **B** peut accéder à un membre **protected** de **A** (méthode **m()** ci-dessous) dans une instance de **B** ou d'une sous-classe **C** de **B** mais pas d'une instance d'une autre classe (par exemple, de la classe est **A** ou d'une autre classe fille **D** de **A**) ; voici du code de la classe **B** :

```
A a = new A(); // A classe mère de B
B b = new B();
C c = new C(); // C sous-classe de B
D d = new D(); // D autre classe de A
a.m(); // interdit
b.m(); // autorisé
c.m(); // autorisé
d.m(); // interdit
```



Encore des règles ...

Attention, **protected** joue donc sur

- l'accessibilité par **B** du membre **m** hérité (**B** comme sous-classe de **A**)
- mais pas sur l'accessibilité par **B** du membre **m** des instances de **A** (**B** comme cliente de **A**)

Polymorphisme (1)

- Contexte: soit **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(5);**
 - **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

a est un objet de la classe **B**
mais il est déclaré de la classe **A**

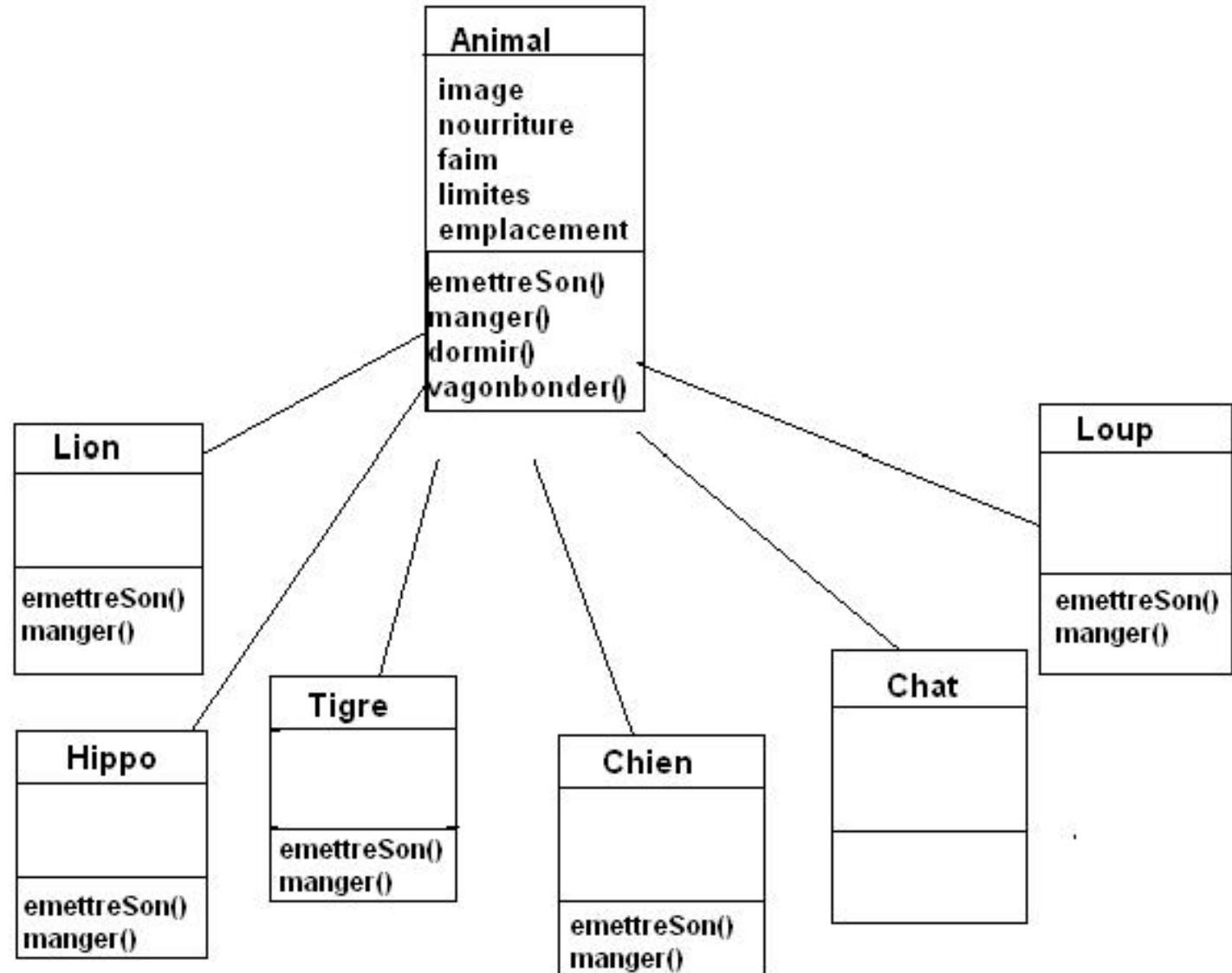
Polymorphisme (2)

- Le polymorphisme est le fait qu'une même écriture peut correspondre à différents appels de méthodes ; par exemple,
 - **A a = x.f();**
 - **a.m();**

f peut renvoyer une instance de A
ou de n'importe quelle sous-classe de A

appelle la méthode **m()** de **A** ou de n'importe quelle sous-classe de **A**
- Ce concept est une notion fondamentale de la programmation objet, indispensable pour une utilisation efficace de l'héritage

Exemple de la jungle



Exemple de la jungle (suite)

- Animal[] animaux=new Animal[5]
 - Animaux[0]= new Chien();
 - Animaux[1]= new Chat();
 - Animaux[2]= new Loup();
 - Animaux[3]= new Hippo();
 - Animaux[4]= new Lion();
- for (int i=0;i<animaux.length;i++) {
 Animaux[i].manger();
 Animaux[i].vagabonder();
}
- 
- Déclare un tableau de type animal
- On peut placer un objet de n'importe quelle sous-classe d'Animal dans le tableau

Utilisation

- Class **Veto** {
 Public void soigner(Animal a) {
 a.emettreSon();
 }
}
- Class **ProprietaireDAnimaux** {
 Public void start() {
 Veto v = new Veto();
 Chien c = new Chien();
 Hippo h = new Hippo();
 v.soigner(c);
 v.soigner(h);
 }
}

Remarques

- Le type de la référence peut être la superclasse du type de l'objet réel
- Les arguments et les types de retour peuvent également être polymorphes

Mécanisme du polymorphisme

- Le polymorphisme est obtenu grâce au « *latebinding* » (liaison retardée) : la méthode qui sera exécutée est déterminée
 - seulement à l'exécution, et pas dès la compilation
 - par le type réel de l'objet qui reçoit le message (et pas par son type déclaré)

Un peu de géométrie

```
public class Figure {  
    public void dessineToi() {}  
}
```

```
public class RectangleextendsFigure {  
    public void dessineToi() {}  
...  
}
```

```
public class CercleextendsFigure {  
    public void dessineToi() {}  
...  
}
```

Un peu de géométrie (suite)

```
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();
        ...
    }
}
```

Typage statique et polymorphisme

- En Java, le typage statique doit garantir dès la compilation l'existence de la méthode appelée :
 - la classe déclarée de l'objet qui reçoit le message doit posséder cette méthode
 - Ainsi, la classe **Figure** doit posséder une méthode **dessineToi()**, sinon, le compilateur refusera de compiler l'instruction « **figures[i].dessineToi()** »

Usage du polymorphisme (1)

- 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  
    }  
    elseif (figures[i] instanceof Cercle) {  
        ... // dessin d'un cercle  
    }  
}
```

Usage du polymorphisme (2)

- 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**, le code de **afficheToi()** sera toujours valable
- Sans polymorphisme, il aurait fallu modifier le code source de la classe **Dessin** pour ajouter un nouveau test :

```
if (figures[i] instanceof Losange) {  
    ... // dessin d'un losange  
}
```

Extensibilité

- Avec la programmation objet, une application est dite extensible si on peut étendre ses fonctionnalités **sans toucher au code source déjà écrit**
- C'est possible en utilisant en particulier le polymorphisme comme on vient de le voir

Mécanisme de la liaison retardée

- Soit **C** la classe réelle d'un objet **o** à qui on envoie un message « **o.m()** »
- Si le code de la classe **C** contient la définition (ou la redéfinition) d'une méthode **m()**, c'est cette méthode qui sera exécutée
- Sinon, la recherche de la méthode **m()** se poursuit dans la classe mère de **C**, puis dans la classe mère de cette classe mère, et ainsi de suite, jusqu'à trouver la définition d'une méthode **m()** qui est alors exécutée

Transtypage (*cast*)

- **Définitions**
 - Classe (ou type) réelle d'un objet : classe du constructeur qui a créé l'objet
 - Type déclaré d'un objet : type donné au moment de la déclaration
 - de la variable qui contient l'objet,
 - ou du type retour de la méthode qui a renvoyé l'objet

Cast: conversions de classes

- Le « *cast* » est le fait de forcer le compilateur à considérer un objet comme étant d'un type qui n'est pas le type déclaré ou réel de l'objet
- En Java, les seuls *casts* autorisés entre classes sont les *casts* entre classe mère et classes filles
- On parle de *upcast* et de *downcast* suivant *le fait que le type est forcé de la classe fille vers la classe mère ou inversement*

Syntaxe

- Pourcaster un objet en classe C :
(C) o;
- Exemple :

Velo v = new Velo();

Vehicule v2 = (Vehicule) v;

UpCast: classe fille → classe mère

- *Upcast*: un objet est considéré comme une instance d'une des classes ancêtres de sa classe réelle
- Il est toujours possible de faire un *upcast*: à cause de la relation *est-unde* l'héritage, tout objet peut être considéré comme une instance d'une classe ancêtre
- Le *upcast* est souvent implicite

Utilisation du *UpCast*

- Il est souvent utilisé pour profiter ensuite du polymorphisme :

```
Figure[] figures = new Figure[10];
```

```
figures[0] = new Cercle(p1, 15);
```

```
...
```

```
figures[i].dessineToi();
```

DownCast: classe mère → classe fille

- ***Downcast:*** un objet est considéré comme étant d'une classe fille de sa classe de déclaration
- Toujours accepté par le compilateur Mais peut provoquer une erreur à l'exécution ;
- A l'exécution il sera vérifié que l'objet est bien de la classe fille
- Un ***downcast*** doit toujours être explicite

Utilisation du *DownCast*

- Utilisé pour appeler une méthode de la classe fille qui n'existe pas dans une classe ancêtre
 - **Figure f1 = new Cercle(p, 10);**
 - ...
 - **Point p1 = ((Cercle)f1).getCentre();**

Downcast pour récupérer les éléments d'une liste

```
// Ajoute des figures dans un ArrayList.  
// Un ArrayList contient un nombre quelconque  
// d'instances de Object  
  
ArrayList figures = new ArrayList();  
figures.add(new Cercle(centre, rayon));  
figures.add(new Rectangle(p1, p2));  
...  
// get() renvoie un Object. Cast nécessaire pour appeler  
// dessineToi() qui n'est pas une méthode de Object  
  
((Figure)figures.get(i)).dessineToi();  
...
```

Classe final (et autres au final)

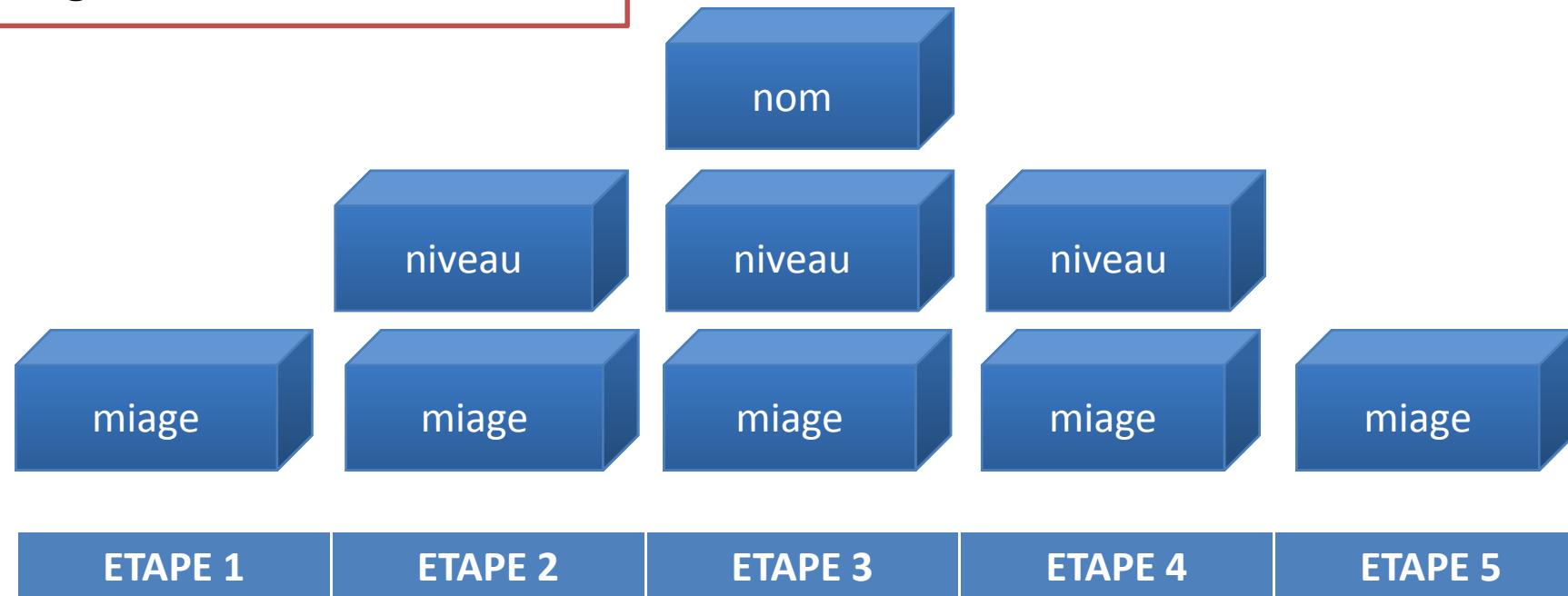
- Classe **final** : ne peut avoir de classes filles (**String** est **final**)
- Méthode **final** : ne peut être redéfinie
- Variable (locale ou d'état) **final** : la valeur ne pourra être modifiée après son initialisation
- Paramètre **final** (d'une méthode ou d'un **catch**) : la valeur (éventuellement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

Réalisation d'un parseur

- Un **parseur** est un **programme** qui **analyse syntaxiquement** un document écrit dans un langage donné (plus souvent de balises). Il permet de récupérer les informations contenues dans les balises d'un document **XML** ou **HTML** par exemple. Cet outil distinguera les informations en fonction de leur contenu et de leur situation dans le document : balise de **début**, balise de **fin**, **texte**, etc. Plus généralement, un parseur peut être assimilé à un **outil d'analyse syntaxique**. C'est d'ailleurs le sens premier du terme anglais **parser**.

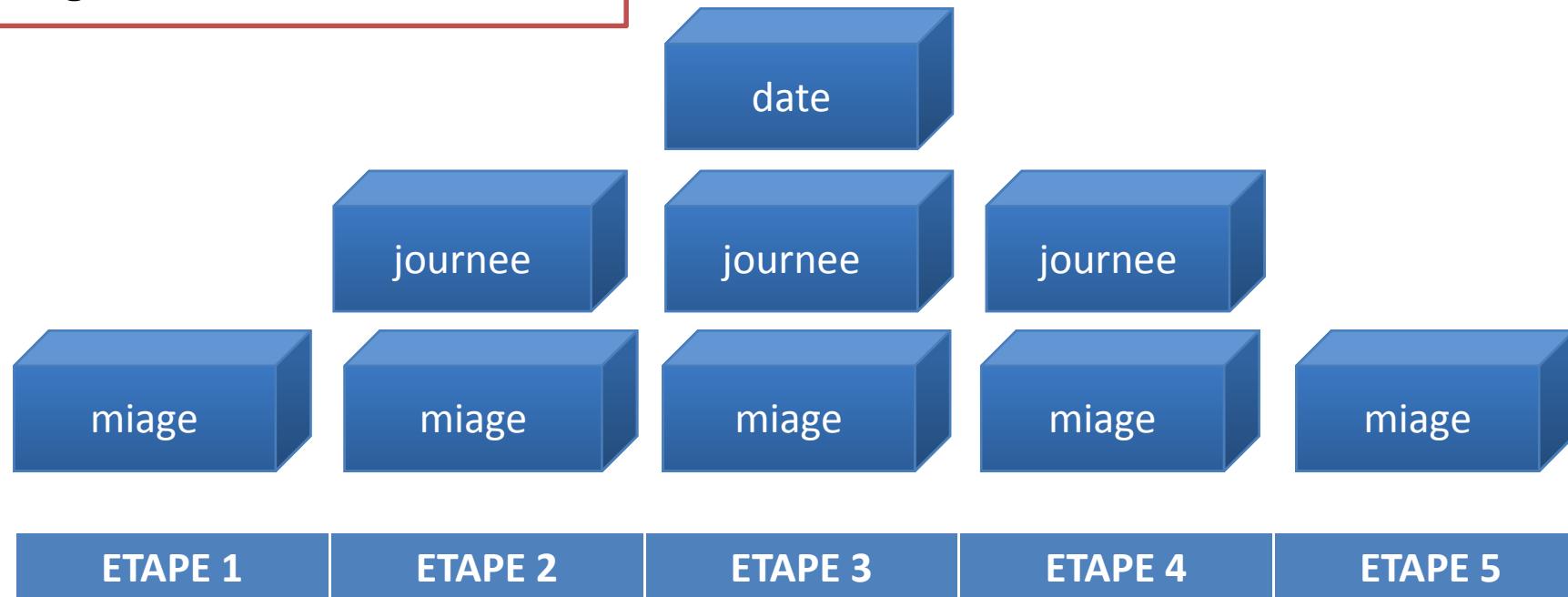
```
<miage>
  <niveau>
    <nom>
      Deuxième année de la
      formation à l'EPT de Thiès.
    </nom>
  </niveau>
  <journee>
    <date>
      Lundi 23 Février 2014 à l'EPT.
    </date>
  </journee>
</miage>
```

The parsing



```
<miage>
  <niveau>
    <nom>
      Deuxième année de la
      formation à l'EPT de Thiès.
    </nom>
  </niveau>
  <journee>
    <date>
      Lundi 23 Février 2014 à l'EPT.
    </date>
  </journee>
</miage>
```

The parsing



Définition de la classe Pile

- Hériter de *java.util.Stack*

```
public class Pile extends java.util.Stack {  
    ...  
}
```

- Définir sa propre classe permettant de gérer une pile de chaîne de caractères

```
public class Pile {  
    ...  
}
```

Méthodes

- ***openTag (String str)*** : teste si ***str*** ouvre une balise, si oui elle renvoie la chaîne sinon elle renvoie une chaîne vide.
- ***closeTag (String str)*** : teste si ***str*** ferme une balise, si oui elle renvoie la chaîne sinon elle renvoie une chaîne vide.
- ***subString (String str, int deb, int fin)*** : extrait de ***str*** la chaîne commençant à partir de ***deb*** et se terminant à ***fin***.
- ***taillePile ()*** : retourne la taille de la pile.
- ***pileVide()*** : retourne **vrai** si pile vide et **faux** sinon.

Travail demandé

- Le travail consiste à écrire une application permettant de lire un fichier et de tester s'il correspond à un document bien formé. En plus des fonctions suggérées, votre projet devrait comporter au moins les classes :
 - Pile
 - Parseur

Encore des méthodes ...

- de construire une pile,
- d'empiler une chaîne de caractère dans la pile,
- de dépiler une chaîne de caractère d'une pile,
- de tester si la pile est vide (condition à vérifier avant de dépiler),
- de lire un fichier et de retourner une ligne
- d'autres au besoin etc.

Bonus ...

- Non indispensable pour avoir 20/20
- Vous pouvez proposer une interface graphique permettant de voir l'enchaînement du processus
 - d'empilement et
 - de dépilement comme montré dans la figure ci-dessus.

Communication

- Génération des groupes
- 3 étudiants au plus
- Mail to mabigue@gmail.com
 - Objet : projet tc2 2014
 - Fichier avec noms des étudiants
- Deadline : 20/03/2014 – 23H 59 59
- Modèle : 03/03/2014
- Bonus : 31/03/2014

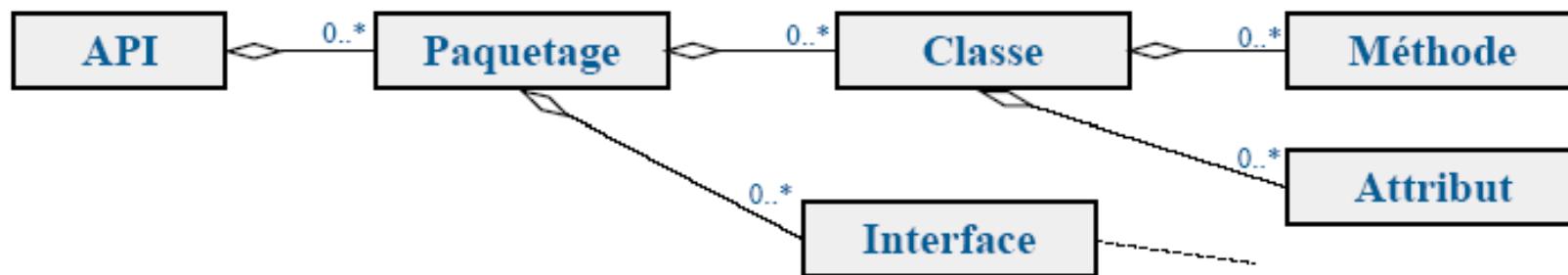
PACKAGES

Package : c'est quoi?

- Java offre la possibilité de regrouper les classes en ensembles appelés package. Ceci permet de ranger les classes de façon hiérarchique.

Les packages

- Le langage Java propose une définition très claire du mécanisme d'empaquetage qui permet de classer et de gérer les API externes
- Les API sont constituées :



- Un package est donc un groupe de classes associées à une fonctionnalité
- Exemples de packages
 - *java.lang* : rassemble les classes de base Java (*Object, String, System, ...*)
 - *java.util* : rassemble les classes utilitaires (*Collections, Date, ...*)
 - *java.io* : lecture et écriture

Utilisation des classes

- Lorsque, dans un programme, il y a une référence à une classe, le compilateur la recherche dans le package par défaut (*java.lang*)
- Pour les autres, il est nécessaire de fournir explicitement l'information pour savoir où se trouve la classe :
 - Utilisation d'**import** (classe ou paquetage)

```
import mesclasses.Point;  
import java.lang.String; // Ne sert à rien puisque par défaut  
import java.io.ObjectOutput;
```

ou

```
import mesclasses.*;  
import java.lang.*; // Ne sert à rien puisque par défaut  
import java.io.*;
```

- Nom du paquetage avec le nom de la classe

Ecriture très lourde
préférer la solution avec
le mot clé **import**

```
java.io.ObjectOutput toto = new java.io.ObjectOutput(...)
```

Jar

➤ Jar et intérêts

- L'archiveur **jar** est l'outil standard pour construire les archives qui ont le même objectif que les bibliothèques de programmes utilisées par certains langages de programmation (lib par exemple)

```
java -verbose HelloWorld
[Loaded java.nio.Buffer from C:\Program Files\Java\j2re1.4.1_01\lib\rt.jar]
[Loaded sun.misc.AtomicLong from C:\Program Files\Java\j2re1.4.1_01\lib\rt.jar]
[Loaded sun.misc.AtomicLongCSImpl from C:\Program Files\Java\j2re1.4.1_01\lib\rt.jar]
[Loaded java.lang.Boolean from C:\Program Files\Java\j2re1.4.1_01\lib\rt.jar]
[Loaded java.lang.Character from C:\Program Files\Java\j2re1.4.1_01\lib\rt.jar]
[Loaded java.lang.Number from C:\Program Files\Java\j2re1.4.1_01\lib\rt.jar]
[Loaded java.lang.Float from C:\Program Files\Java\j2re1.4.1_01\lib\rt.jar]
```

Montre les archives utilisées pour exécuter le programme HelloWorld

➤ Utilisation pour la création

- Utilisation de l'outil **jar**
- Pour créer un fichier *.jar* contenant tous les fichiers du répertoire courant

```
jar cvf hello.jar
```

Création Verbose Nom archive

Le . indique le répertoire courant

GESTION DES EXCEPTIONS

TABLEAUX (SUITE)

Tableaux : Arrays

- Cette classe contient des méthodes statiques permettant de manipuler de tableaux :
 - Recherche, recherche dichotomique
 - Tris
 - Remplissage
 - Égalité
 - *toString()*

Méthode `toString`

- La méthode `static String toString(X[] a)` retourne une chaîne de caractères contenant les éléments du tableau (convertis en chaîne de caractères), séparés par des virgules, et entre crochets.

Méthode equals

- La méthode *static boolean equals(X[] a, X[] a2)* où X peut être un type primitif ou un *Object* retourne *true* si les deux tableaux sont égaux et *false* sinon.
- Deux tableaux sont égaux si
 - ils contiennent le même nombre d'éléments,
 - et si les éléments de même rang sont égaux.
- Si des éléments du tableau peuvent être des tableaux, il faut utiliser la méthode *deepEquals(X[] a, X[] a2)*.

Méthode fill

- La méthode *static void fill(X[] a, X val)* affecte la valeur *val* à tous les éléments du tableau *a*.

Méthodes : binarySearch

- La méthode *static int binarySearch(X[] a, X cle)* effectue une recherche de *cle* dans le tableau trié *a*.
- La méthode retourne
 - l'indice de l'élément s'il existe,
 - Et – *pointDInsertion-1* si *cle* ne se trouve pas dans le tableau.
 - La valeur *pointDInsertion* est le rang où la clé serait ajoutée, si on l'ajoutait au tableau. C'est le rang du premier élément plus grand que l'élément cherché, ou la taille du tableau.

Code de la méthode

```
public static int binarySearch(int[] a, int cle) {  
    int debut = 0;  
    int fin = a.length - 1;  
    while(debut <= fin) {  
        int milieu = (debut + fin)/2;  
        if(a[milieu] < cle) debut = milieu + 1;  
        elseif (a[milieu] > cle) fin = milieu - 1;  
        else return milieu; // trouvé  
    }  
    return -(debut + 1); // pas trouvé.  
}
```

Méthode sort

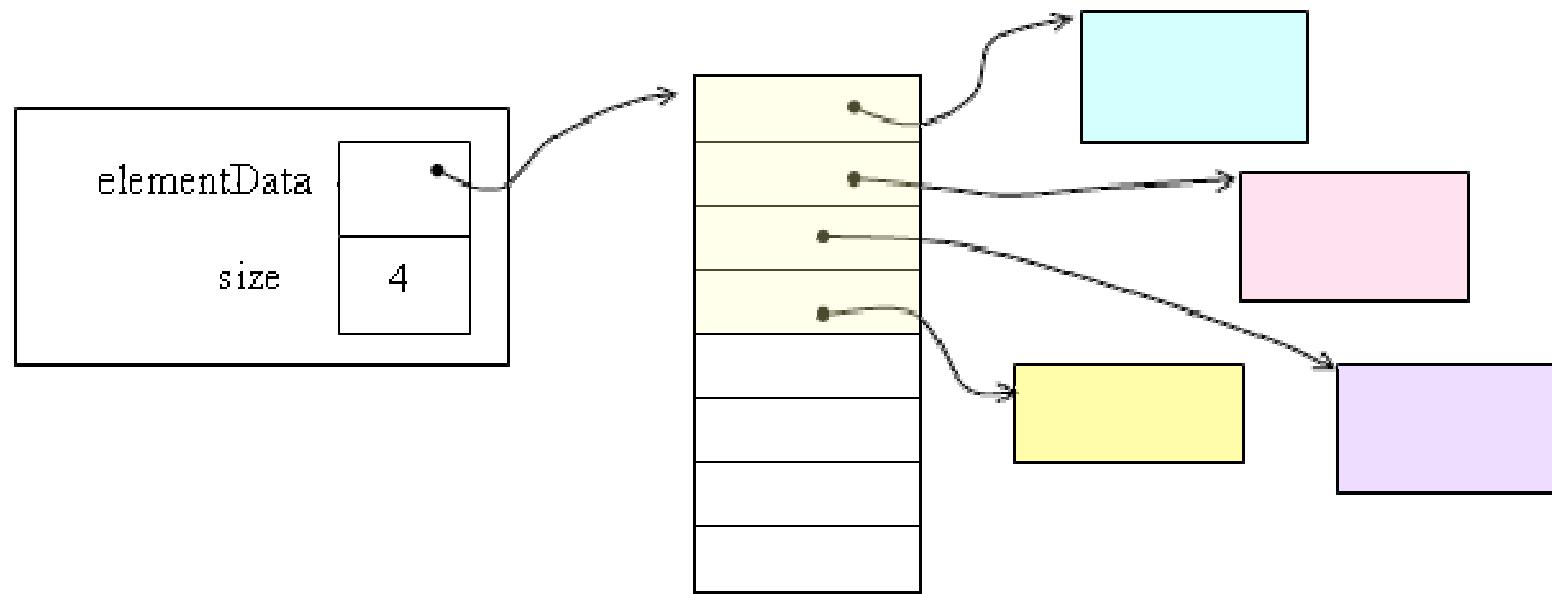
- La méthode *static void sort(Object[] a)* est programmée en utilisant :
 - Un tri rapide si *X* est un type **primitif**. adapté de Jon L. Bentley and M. Douglas McIlroy's "[Engineering a Sort Function](#)", Software-Practice and Experience, Vol. 23(11) P. 1249-1265 (Novembre 1993).
 - Un tri par fusion si *X* est un *Object*.
- Les objets du tableau à trier doivent être **Comparable**

ARRAYLIST

ClasseArrayList

- La classe *ArrayList* implémente un tableau d'objets qui peut grandir ou rétrécir à la demande
- débarrasse le programmeur de la gestion de la taille du tableau.
- Comme pour un tableau on peut accéder à un élément du *ArrayList*, par un indice.

ClasseArrayList : Exemple



Lorsque le tableau est plein (`size == taille du tableau`), et qu'on veut ajouter un élément dans le tableau, le tableau est agrandi : la nouvelle taille est l'ancienne taille ***3/2 plus 1**.

ClasseArrayList : constructeurs

1. un *ListeTableau* avec sa capacité initiale :
 - **publicArrayList(intcapacityInitial)**
2. par défaut construit un *ListeTableau* de 10 emplacements.
 - **public ArrayList() { this(10); }**
3. un *ArrayList* à partir d'une collection d'objets
 - **publicArrayList(Collection**

ClasseArrayList : méthodes (1)

- Interface *Collection*
 - *add*ajoute un élément en fin de tableau.
 - *clear*enlève tous les éléments du vecteur,
 - *size*retourne le nombre d'éléments du ArrayList
 - *isEmpty*→vraissi le ArrayList est vide.
 - *remove*enlève la première occurrence de *o*
 - *addAll*ajoute une collection

ClasseArrayList : méthodes(2)

- Interface List
 - `add(index, objet)` ajoute `objet` à l'`index`, lève une exception si l'`indice` n'est pas « correct ». Il déplace tous les éléments d'indices supérieurs ou égaux à `index` pour faire la place avant l'insertion.
 - `get(index)` retourne l'élément se trouvant à un indice, ou lève une exception
 - `indexOf(Object)` et `lastIndexOf(Object)` retournent l'indice d'un objet
 - `remove(int)` enlève l'objet et l'indice et le retourne
 - `set(int, Object)` modifie l'objet se trouvant en position `index` dans le vecteur, en lui affectant `obj`, et retourne l'objet qui occupait cette place avant.

ClasseArrayList : méthodes (2)

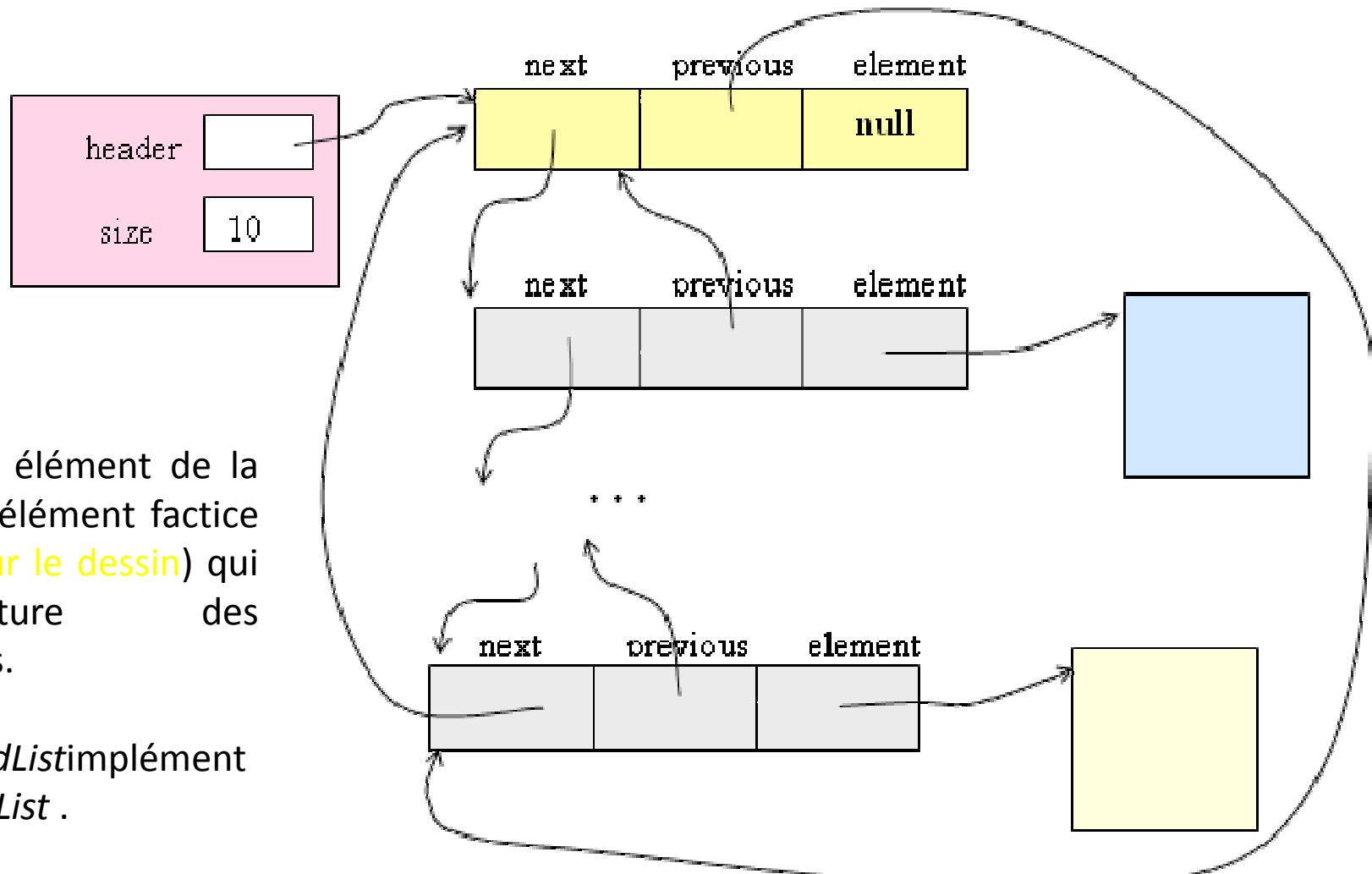
- Méthodes propres à la classe
 - *trimToSize* ajuste la capacité du *ArrayList* à son nombre d'éléments.

LINKEDLIST

Représentation

Le premier élément de la liste est un élément factice (en jaune sur le dessin) qui facilite l'écriture des algorithmes.

La classe *LinkedList* implémente l'interface *List*.



Constructeurs

- `LinkedList()`
- `LinkedList(Collection<? extends E>)`

Méthodes (1)

- Outils de la classe
 - remove(e)
 - addBefore(o,e)
 - addAfter(o,e)
 - entry(**int** index)
- Interface Collection
 - add, addAll, clear, isEmpty, size, contains
- Interface List
 - Add, addAll, get, indexOf, lastIndexOf, remove, set

Méthodes(2)

- Propres à la classe
 - `getFirst()`
 - `getLast()`
 - `removeFirst()`
 - `removeLast()`
 - `addFirst(Object o)`
 - `addLast(Object o)`

Héritage

- Concept naturel
 - Une classe peut être divisée en des sous-classes
 - E.g.
 - *Cours* en *CoursGradue* et *CoursSousGradue*
 - *Etudiant* en *EtudiantBac*, *EtudiantMaitrise* et *EtudiantDoctorat*
 - Certaines propriétés sont communes: définies dans la classe globale
 - Nom, Code permanent
 - Cours suivis
 - D'autres sont locales pour une sous-classe
 - Mémoire
 - Projet de recherche
- Héritage
 - Les sous-classes héritent les propriétés de la super-classe
 - Attributs et méthodes

Exemple

- class Etudiant
 - { public String nom;
 - public String codePermanent;
 - public String [] cours;
 - }
- class EtudiantGradue extends Etudiant
 - { public String projet;
 - public Prof superviseur;
 - }

Les attributs de EtudiantGradue:

public String nom;
public String codePermanent;
public String [] cours;
public String projet;
public Prof superviseur;

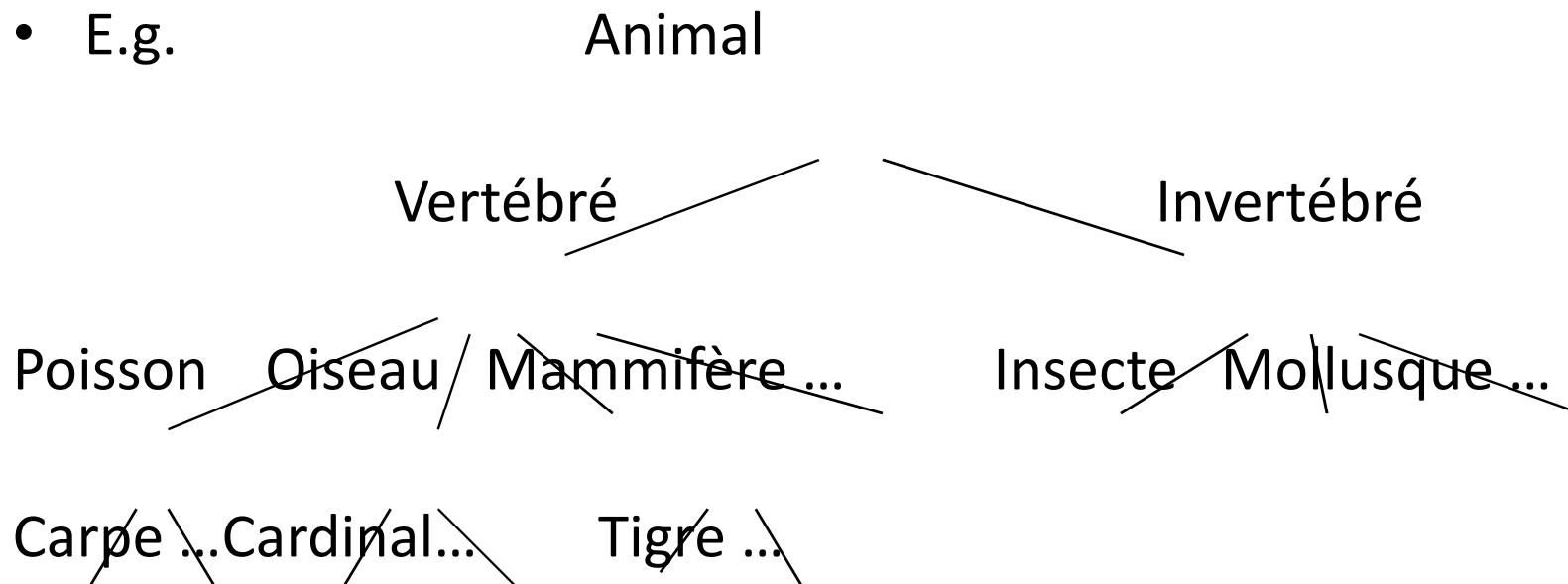
} Hérités

Avantages

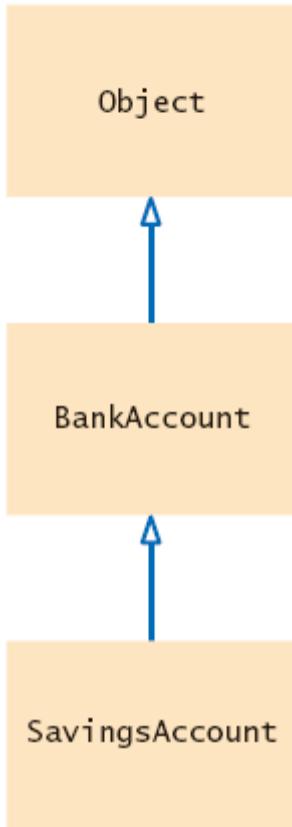
- partager les attributs et méthodes en commun dans la super-classe et les sous-classes
- Faire la différence entre les sous-classes
- Sous-classe: spécification supplémentaire
 - Attributs additionnels
 - Méthodes additionnelles

Hiérarchie des classes

- Développer des classes et des sous-classe selon la hiérarchie naturelle des concepts
- E.g.



En Java



```
public class SavingsAccount extends BankAccount
{
    public SavingsAccount (double rate)
        { interestRate = rate; }
    public void addInterest()
        { double interest = getBalance() *
            interestRate / 100;
            deposit(interest); }
    private double interestRate;
}
```

Hérités:

```
private double balance // attention
public double getBalance()
public void deposit(double amount)
public void withdraw(double amount)
```

Structure créée

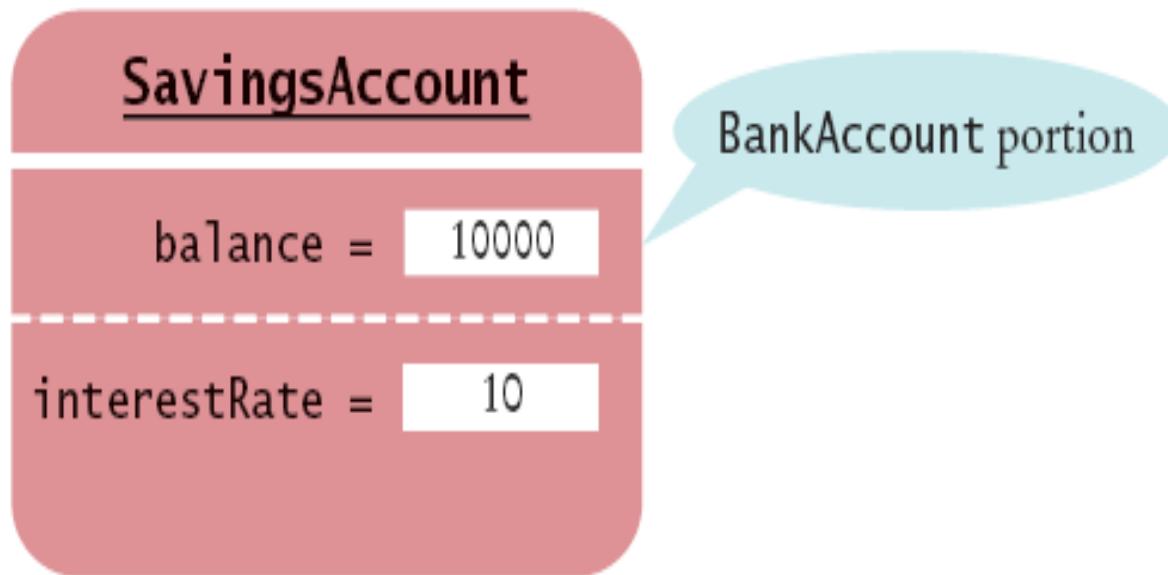


Figure 2 Layout of a Subclass Object

Créer une hiérarchie

- class SavingsAccount extends BankAccount {...}
- class CheckingAccount extends BankAccount {...}

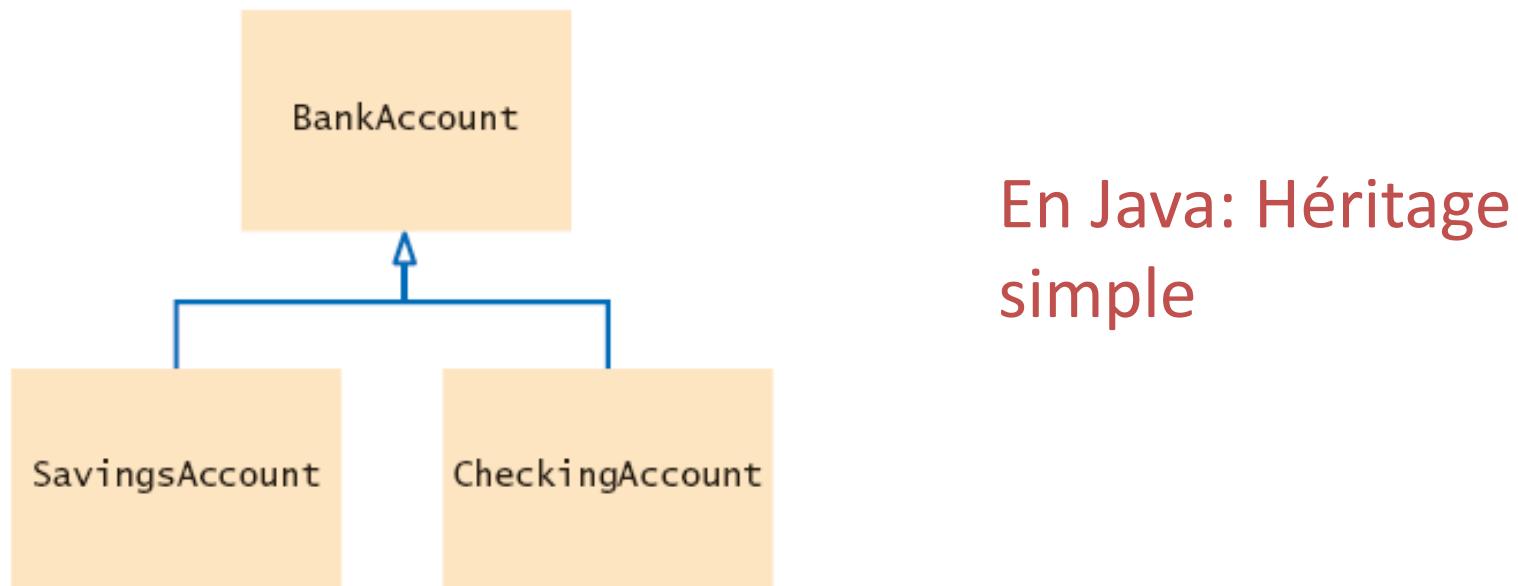


Figure 5 Inheritance Hierarchy for Bank Account Classes

Ce qu'on a dans une sous-classe

- Attributs:
 - Attributs supplémentaires
 - Ne peut pas enlever les attributs hérités de la super-classe
 - Tous les attributs de la super-classe sont hérités
 - On peut définir un attribut du même nom qu'un attribut hérité (déconseillé), mais les deux co-existent
 - E.g. class C1 {int a;}
 - class C2extends C1 {String a;
 void setA(String b) {a=b;} }

int	a
String	a

Pr Mouhamadou THIAM Maître de conférences en informatique

Ce qu'on a dans une sous-classe

- Méthodes supplémentaires définies dans la sous-classe
- Méthodes héritées
- Méthode réécrite (overriding) si même signature
- E.g. Overriding

```
class C1 {public void m1(int a) {System.out.println(a);} }  
class C2 extends C1 {  
    public void m1(String s) {System.out.println(a);} } // pas de overring  
class C3 extends C1 {  
    public void m1(int b) {System.out.println(a+1);} } // overriding
```

Exemple: CheckingAccount

```
public class CheckingAccount extends BankAccount
{
    public void deposit(double amount) { ... }
    public void withdraw(double amount) { ... }
    public void deductFees() { ... } // new method
    private int transactionCount; // new instance field
}
```

- Hérités:
 - Attribut: balance (hérité de BankAccount)
 - Méthodes: getBalance(),
~~deposit(double amount), withdraw(double amount)~~
- Ajoutés
 - Attribut: transactionCount
 - Méthode: deductFees()
- Overrid
 - Méthodes: deposit(double amount), withdraw(double amount)

```
public class BankAccount
{
    public BankAccount() {...}
    public BankAccount(double initialBalance) {...}
    public void deposit(double amount) {...}
    public void withdraw(double amount) {...}
    public double getBalance(){...}

    private double balance;
}
```

BankAccount

```
public class BankAccount
{
    public BankAccount()
    {
        balance = 0;
    }
    public BankAccount
        (double initialBalance)
    {
        balance = initialBalance;
    }
    public void deposit(double amount)
    {
        double newBalance =
            balance + amount;
        balance = newBalance;
    }
    public void withdraw(double amount)
    {
        double newBalance =
            balance - amount;
        balance = newBalance;
    }
    public double getBalance()
    {
        return balance;
    }
    private double balance;
}
```

Existence et Accessibilité

- class BankAccount
 - {
 - private double balance;
 - ...
 - }
- class CheckingAccount extends BankAccount
 - {
 - public void deposit(double amount)
 - { transactionCount++;
 - // now add amount to balance
 - balance = balance + amount// erreur
 - }
 - }
- balance existe, mais n'est pas accessible dans une sous-classe parce qu'elle est *private*
- Accès par méthode deposit de la super-classe: **super.deposit(amount);**
- Pour rendre balance accessible dans une sous-classe: protéger avec *protected*

Accessibilité

- Si un attribut/méthode est protégé avec
public protected private
- Alors il est accessible dans une sous-classe
oui oui non

super

- *super* évoque la classe supérieure
- Utilisation 1 (dans la définition d'une méthode de sous-classe)
 - Si une méthode (e.g. deposit) de la super-classe est redéfinie (overrid), dans la déclaration de la sous-classe, deposit(100) évoque la version de la sous-classe
 - **super.deposit(100)** évoque la version de la super-classe
 - (comparaison avec *this*)
- Utilisation 2 (dans un constructeur)
 - Utiliser *super(paramètres)* pour appeler le constructeur de la super-classe
 - Utile pour construire une version d'objet qui est ensuite enrichie ou complétée
 - Règles
 - Si un constructeur de sous-classe évoque **super(...)**, ce doit être la première instruction
 - Si un constructeur de la sous-classe n'évoque pas *super* explicitement, *super()* est implicitement évoquée comme la première instruction
 - Il faut que cette version de constructeur soit définie dans la super-classe

Exemple

```
public class CheckingAccount extends BankAccount
{
    public CheckingAccount(double initialBalance)
    {   // Construct superclass   super(initialBalance);
        // Initialize transaction count
        transactionCount = 0;
    }
    ...
}
```

Références

- Une référence de la super-classe peut référer à un objet (instance) de
 - Super-classe
 - Sous-classe
- Une référence d'une sous-classe peut référer à un objet de
 - Sous-classe
- E.g.

```
BankAccount a1;  
CheckingAccount a2;  
a1 = new CheckingAccount(0); //OK  
a2 = new BankAccount(0); //Erreur
```
- Comparaison
 - Appeler un étudiant en maîtrise « Eudiant »: OK
 - Appeler un étudiant « EtudiantMaitrise »: Non

Autre exemple (casting pour attribut)

```
class C1
{
    public int a;
    public C1 (int x) { a=x; }
}
Class C2 extends C1
{
    public int a; // ajout un autre attribut du même nom
    public int b;
    public C2 (int x, int y) { super(0); a=x, b=y;}
}

Utilisation
C1 ref1;
C2 ref2;
ref1 = new C1(1);           ref2 = new C2(2,3);

ref1 = new C2(2,3);         //OK
ref2 = new C1(1);          //Erreur

ref1 = new C2(2,3);
ref2 = ref1;                // Problème de compilation
ref2 = (C2) ref1;           // OK: dire au compilateur que ref1 réfère à une instance de classe C2.
System.out.println(ref1.a);   // Valeur 0
System.out.println( ( (C2) ref1).a ); // Valeur 2

ref2 = new C2(2,3);
System.out.println( ((C1) ref2).a ); // valeur 0
```

C1: a
|
C2: a; a, b

Problème si super(0) n'y est pas

Référence de super-classe

- Référer tous objets de la classe (super-classe) par la même référence (variable) peut simplifier certains traitements
- E.g. parcourir tous les objets d'une classe et de ses sous-classes avec la même référence
- Supposons: BankAccount oneAccount;
- Question:
 - `oneAccount.deposit(100)` = quelle version?

Règle

- La version de la méthode évoquée à partir d'une référence correspond à la version de l'instance référée
- `oneAccount.deposit(100)` = version de l'instance
- E.g.

```
BankAccount a1;  
a1 = new CheckingAccount(0);  
a1.deposit(100); //version de CheckingAccount
```

Condition

- Utiliser une référence de la super-classe pour évoquer une méthode:
 - Il faut que la méthode soit définie dans la super-classe
 - Sinon, une erreur de compilation: méthode non disponible
- E.g.

```
Object anObject = new BankAccount(); anObject.deposit(1000);  
//Wrong!
```

deposit(...) n'est pas une méthode disponible dans Object

Règle sur référence

- Si une méthode peut être appliquée sur toutes les instances d'une super-classe et ses sous-classe
 - Définir une version pour super-classe
 - Modifier (si nécessaire) dans les sous-classes
- Comparaison:
 - Pour tout Etudiant, on peut connaître sa moyenne, même si la façon de calculer la moyenne est différente pour des étudiants gradués et sous-gradués
 - Définir une méthode `getMoyenne()` pour tout Etudiant
 - Raffiner dans les sous-classes

Connaître la classe d'une instance référée (*instanceof*)

- On peut avoir besoin de connaître la classe de l'instance pour
 - déterminer si une méthode est applicable
 - choisir un traitement ou un autre
 - ...
- Test: *object instanceof TypeName*
 - Tester si *object* est de la classe *TypeName*

e.g. if (anObject instanceof BankAccount)

{

 BankAccount anAccount = (BankAccount) anObject;

...

}

Polymorphisme

- La capacité qu'une même évocation de méthode change de comportement selon l'instance avec laquelle la méthode est appelée.
- E.g.

```
BankAccount a1;  
for ( ...) {  
    a1.deposit(1000); // la version de l'instance  
}
```

Autre exemple

```
class C1
{
    public int a;
    public C1 (int x) { a=x; }
    public void print() { System.out.println("Version 1 " + a); }
}
Class C2 extends C1
{
    public int b;
    public C2 (int x, int y) { super(x); b=y; }
    public void print() { System.out.println("Version 2 " + b); }
}
```

Utilisation

```
C1 ref1;
C2 ref2;
ref1 = new C1(1);
ref2 = new C2(2,2);
ref1.print();           // Version 1 1
ref2.print();           // Version 2 2
ref1 = ref2;
ref1.print();           // Version 2 2
((C1)ref2).print();    // Version 2 2: Différence avec casting pour attribut
```

02/07/2018

C1: a
print()
|
C2: a; b
print()

Récapitulation

- Attribut
 - Détermine l'attribut accédé selon la classe de la référence
 - Possibilité de *cast* pour modifier la classe de référence temporairement
- Méthode
 - Détermine la version de la méthode selon la classe réelle de l'instance
 - Pas de *casting* possible

```
05: public class BankAccount
06: {
10:     public BankAccount()
11:     {
12:         balance = 0;
13:     }
19:     public BankAccount(double initialBalance)
20:     {
21:         balance = initialBalance;
22:     }
28:     public void deposit(double amount)
29:     {
30:         balance = balance + amount;
31:     }
37:     public void withdraw(double amount)
38:     {
39:         balance = balance - amount;
40:     }
46:     public double getBalance()
47:     {
48:         return balance;
49:     }
56:     public void transfer(double amount, BankAccount other)
57:     {
58:         withdraw(amount);
59:         other.deposit(amount);
60:     }
62:     private double balance;
63: }
```

```

04: public class CheckingAccount extends BankAccount
05: {
10:     public CheckingAccount(double initialBalance)
11:     {
13:         super(initialBalance);
16:         transactionCount = 0;
17:     }
19:     public void deposit(double amount)
20:     {
21:         transactionCount++;
23:         super.deposit(amount);
24:     }
26:     public void withdraw(double amount)
27:     {
28:         transactionCount++;
29:         super.withdraw(amount);
30:     }
31: }
37: public void deductFees()
38: {
39:     if (transactionCount > FREE_TRANSACTIONS)
40:     {
41:         double fees = TRANSACTION_FEE *
42:             (transactionCount - FREE_TRANSACTIONS);
43:         super.withdraw(fees);
44:     }
45:     transactionCount = 0;
46: }
48: private int transactionCount;
50: private static final int FREE_TRANSACTIONS = 3;
51: private static final double TRANSACTION_FEE = 2.0;
52: }

```

04: public class **SavingsAccount** extends BankAccount

05: {

10: public SavingsAccount(double rate)

11: {

12: interestRate = rate;

13: }

18: public void addInterest()

19: {

20: double interest = getBalance() * interestRate / 100;

21: deposit(interest);

22: }

24: private double interestRate;

25: }

```

05: public class AccountTester
06: {
07:   public static void main(String[] args)
08:   {
09:     SavingsAccount momsSavings
10:       = new SavingsAccount(0.5);      //1: Intérêt = 0.5%
12:     CheckingAccount harrysChecking
13:       = new CheckingAccount(100);    //2: balance = 100
15:     momsSavings.deposit(10000);     //1: balance = 10000
17:     momsSavings.transfer(2000, harrysChecking); //1: balance=10000-2000=8000
18:                               //2: balance=100+2000=2100
19:     harrysChecking.withdraw(1500); //2: balance=2100-1500=600
21:     harrysChecking.withdraw(80);   //2: balance=600-80=520
22:     momsSavings.transfer(1000, harrysChecking); //1: balance=8000-1000=7000
23:                               //2: balance=520+1000=1520
24:     harrysChecking.withdraw(400); //2: balance=1520-400=1120
25:     momsSavings.addInterest();  //1: balance=7000+7000*0.5%=7035
26:     harrysChecking.deductFees(); //2: balance=1120-(5-3)*2=1116
28:     System.out.println("Mom's savings balance = $"
29:                         + momsSavings.getBalance());
31:     System.out.println("Harry's checking balance = $"
32:                         + harrysChecking.getBalance());
33:   }
34: }

```

Sortie: Mom's savings balance = \$7035.0
 Harry's checking balance = \$1116.0

Méthodes cosmétiques

- Les méthodes qu'on utilise souvent, et qu'il est utile d'implanter
- E.g.
 - String `toString()`
 - boolean `equals(Object otherObject)`
 - Object `clone()`
- Déjà définies dans `Object`
- Il faut redéfinir (override) dans les sous-classes

Hiérarchie de classe en Java

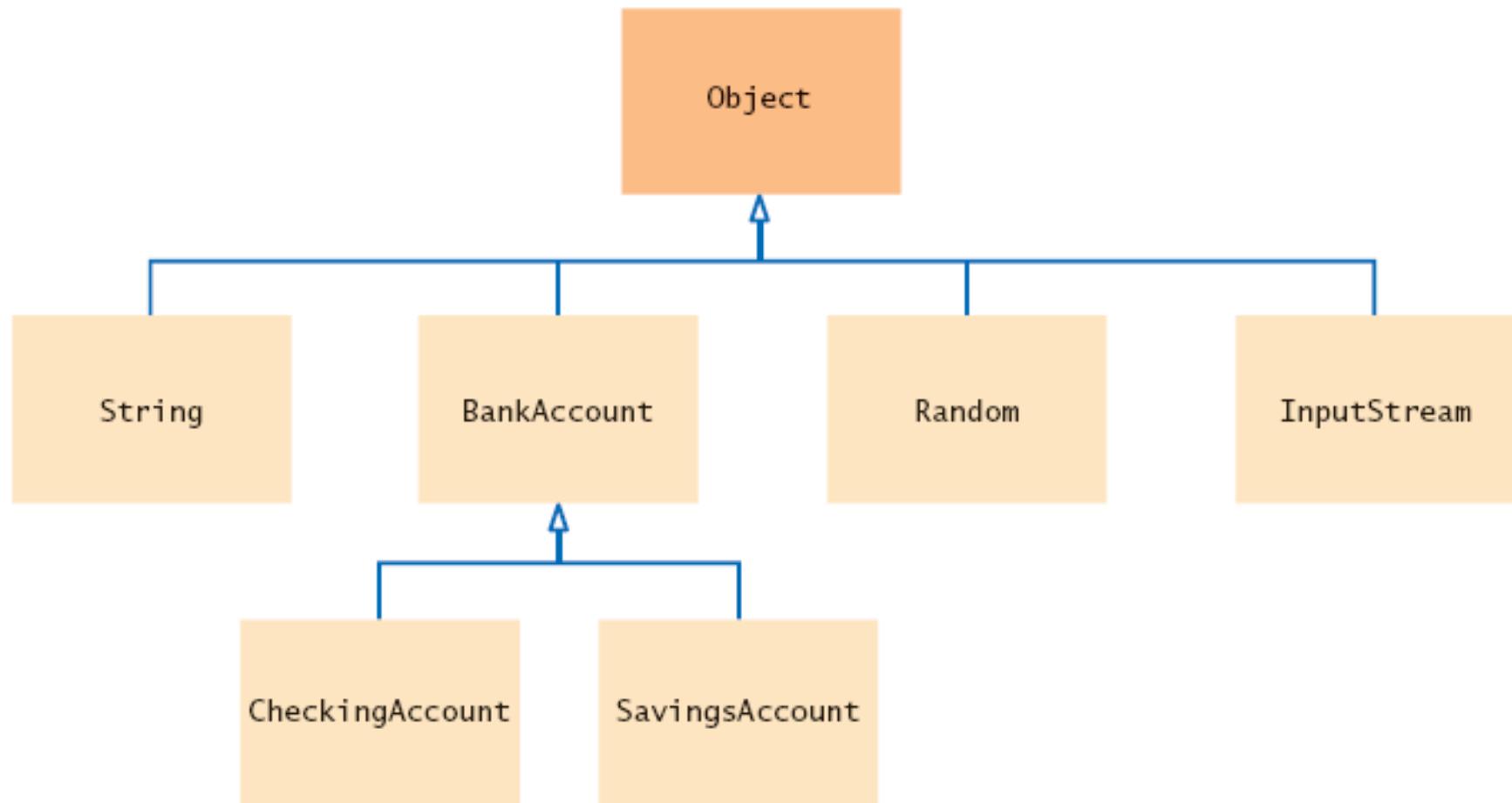


Figure 8 The `Object` Class Is the Superclass of Every Java Class

Implantation

- `toString`: retourne un texte décrivant l'instance
- E.g.

```
public String toString()  
{  
    return "BankAccount[balance=" + balance + "]";  
}
```

- Cette méthode redéfinit la méthode de `Object`

Implantation

- equals(Object obj): teste si deux instances contiennent le même contenu
- E.g.

```
public class BankAccount
{
    ...
    public boolean equals(Object otherObject)
    {
        BankAccount other = (BankAccount) otherObject;
        return balance = other.getBalance(); //comparer le contenu
    }
    ...
}
```

Implantation

- `clone()`: retourner une copie de l'instance
- Étapes:
 - Créer une instance
 - Copier tous les attributs (valeurs) dans la copie
 - Retourner l'instance

Pile abstraite en java

```
package pile;

abstract class Pile <T>{

    abstract public T empiler(T v) ;
    abstract public T dépiler() ;
    abstract public Boolean estVide() ;

}
```

Divers

- package: regroupement de diverses classes
- abstract : signifie qu'il n'y a pas d'implémentation
- public: accessible de l'extérieur
- La classe est paramétrée par un type (java 1.5)

Implémentations

- On va implémenter la pile:
 - avec un objet de classe `Vector` (classe définie dans `java.util.package`) en fait il s'agit d'un `ListArray`
 - Avec un objet de classe `LinkedList`
 - Avec `Integer` pour obtenir une pile de `Integer`

Uneimplémentation

```
package pile;
import java.util.EmptyStackException;
import java.util.Vector;
public class MaPile<T>extends Pile<T>{
privateVector<T> items;
    // Vector devrait être remplacé par ArrayList
    public MaPile() {
        items =new Vector<T>(10) ;
    }
    public BooleanestVide(){
        return items.size()==0;
    }
    public T empiler(T item){
items.addElement(item);
        return item;
    }
//...
```

Suite

```
//...
public synchronized T dépiler() {
    int len = items.size();
    T item = null;
    if (len == 0)
        throw new EmptyStackException();
    item = items.elementAt(len - 1);
    items.removeElementAt(len - 1);
    return item;
}
```

Autre implémentation avec listes

```
package pile;
import java.util.LinkedList;
public class SaPile<T>extends Pile<T> {
privateLinkedList<T> items;
    public SaPile() {
        items = new LinkedList<T>();
    }
    public Boolean estVide() {
        return items.isEmpty();
    }
    public T empiler(T item) {
items.addFirst(item);
        return item;
    }
    public T dépiler() {
        return items.removeFirst();
    }
}
```

02/07/2018

Pr. Mouhamadou THIAM Maître de
conférences en informatique

862

Une pile de Integer

```
public class PileInteger extends Pile<Integer>{  
    private Integer[] items;  
    private int top=0;  
    private int max=100;  
    public PileInteger(){  
        items = new Integer[max];  
    }  
    public Integer empiler(Integer item){  
        if (this.estPleine())  
            throw new EmptyStackException();  
        items[top++] = item;  
        return item;  
    }  
    //...
```

Suite...

```
public synchronizedInteger dépiler() {
    Integer item = null;
        if (this.estVide())
    throw new EmptyStackException();
        item = items[--top];
    return item;
}
public Boolean estVide() {
    return (top == 0);
}
public boolean estPleine() {
    return (top == max -1);
}
protected void finalize() throws Throwable {
    items = null; super.finalize();
}
}
```

Comment utiliser ces classes?

- Le but est de pouvoir écrire du code utilisant la classe **Pile abstraite**
- Au moment de l'exécution, bien sûr, ce code s'appliquera à un objet concret (qui a une implémentation)
- Mais ce code doit s'appliquer à toute implémentation de Pile

Un main

```
package pile;
public class Main {
    public static void vider(Pile p) {
        while (!p.estVide()) {
            System.out.println(p.dépiler());
        }
    }
    public static void main(String[] args) {
        MaPile<Integer> p1 = new MaPile<Integer>();
        for (int i=0;i<10;i++)
            p1.empiler(i);
        vider(p1);
        SaPile<String> p2 = new SaPile<String>();
        p2.empiler("un");
        p2.empiler("deux");
        p2.empiler("trois");
        vider(p2);
    }
}
```

Entrée-sortie

```
public static void main(String[] args) {  
    // sortie avec printf ou  
    double a = 5.6d ;  
    double b = 2d ;  
    String mul = "multiplié par" ;  
    String eq="égal";  
    System.out.printf(Locale.ENGLISH,  
                      "%3.2f x %3.2f = %6.4f \n", a ,b , a*b);  
    System.out.printf(Locale.FRENCH,  
                      "%3.2f %s %3.2f %s %6.4f \n", a, mul,b eq,a*b);  
    System.out.format(  
        "Aujourd'hui %1$tA, %1$te %1$tb," +  
        " il est: %1$th h %1$tm min %1$ts \n",  
        Calendar.getInstance());  
    // System.out.flush();
```

Sortie

$5.60 \times 2.00 = 11.2000$

5,60 multiplié par 2,00 égal 11,2000

Aujourd'hui mardi, 10 octobre, il est: 15 h 31
min 01

Scanner

```
Scanner sc = new Scanner(System.in);
for(boolean fait=false; fait==false;){
    try {
        System.out.println("Répondre o ou O:");
        String s1 =sc.next(Pattern.compile("[0o]"));
        fait=true;
    } catch(InputMismatchException e) {
        sc.next();
    }
}
if (sc.hasNextInt()){
    int i= sc.nextInt();
    System.out.println("entier lu "+i);
}
System.out.println("next token :"+sc.next());
sc.close();
```

Scanner

```
if (sc.hasNextInt()){
    int i= sc.nextInt();
    System.out.println("entier lu "+i);
}
System.out.println("next token :" +sc.next()); sc.close();
String input = "1 stop 2 stop éléphant gris stop rien";
Scanner s = new Scanner(input).useDelimiter("\s*stop\s*");
    System.out.println(s.nextInt());
    System.out.println(s.nextInt());
    System.out.println(s.next());
    System.out.println(s.next());
    s.close();
}
```

Sortie

- nexttoken :o
- 1
- 2
- éléphant gris
- rien

Les classes...

- System
 - System.out variable (static) de classe PrintStream
 - PrintStream contient print (et printf)
 - System.in variable (static) de classe InputStream
- Scanner

Chapitre II

Classes et objets (rappels)

(mais pas d'héritage)

Classes et objets

- I) Introduction
- II) Classe: membres et modificateurs
- III) Champs: modificateurs
- IV) Vie et mort des objets, Constructeurs
- V) Méthodes
- VI) Exemple

I) Introduction

- Classe
 - Regrouper des données et des méthodes
 - Variables de classe
 - Méthodes de classe
 - Classes<->type
- Objet (ou instance)
 - Résultat de la création d'un objet
 - Variables d'instance
 - Variables de classe
- Toute classe hérite de la classe Object

II) Classes

- Membres d 'une classe sont:
 - Champs = données
 - Méthodes = fonctions
 - Classes imbriquées

Modificateur de classe

- Précède la déclaration de la classe
 - Annotations (plus tard...)
 - `public` (par défaut package)
 - `abstract`(incomplète, pas d'instance)
 - `final`(pas d'extension)
 - `Strictfp` (technique...)

III) Champs

- Modificateurs
 - annotations
 - Contrôle d'accès
 - private
 - protected
 - public
 - package
 - static (variables de classe)
 - final (constantes)
 - transient
 - volatile
- Initialisations
- Création par opérateur new

