



Joseph NDONG

Département de Mathématiques et Informatique UCAD.

Que donne ce cours?

Ce cours donne les bases fondamentales de la programmation orientée objet appliquée au langage JAVA.

Quelles compétences acquérir après le cours ?

Après ce cours, le lecteur pourra réaliser des applications autonomes fonctionnant sur un seul ordinateur. Le lecteur pourra aussi réaliser des pages web via les applets.

Vers où aller après le cours?

Après le cours, le lecteur pourra attaquer les notions très avancées en Java, notamment la programmation d'applications entreprises avec la technologie JEE. Il pourra aussi s'attaquer à la programmation réseaux avec les sockets ou RMI.



Apprendre à optimiser !!!

Compiler et ça marche ne veut pas dire que le programme est BON.

Il faut OPTIMISER ...



Plan du Cours (1/2)

- Module 1 : Introduction à Java
- Module 2 : Techniques de base du langage
- Module 3 : Les types primitifs
- Module 4 : Les structures de contrôle
- Module 5 : Programmation Orientée Objet
- Module 6 : Les tableaux
- Module 7: Les chaînes de caractères



Plan du cours (2/2)

- Module 8 : L'héritage
- Module 9 : Gestion des exceptions
- Module 10 : Threads
- Module 11: Les Swing GUI
- Module 12 : Entrées Sorties
- Module 13: Collections et algorithmes
- Module 14 : Applets
- Module 15 : JDBC



Introduction à Java

Ce cours enseigne les aspects suivants:

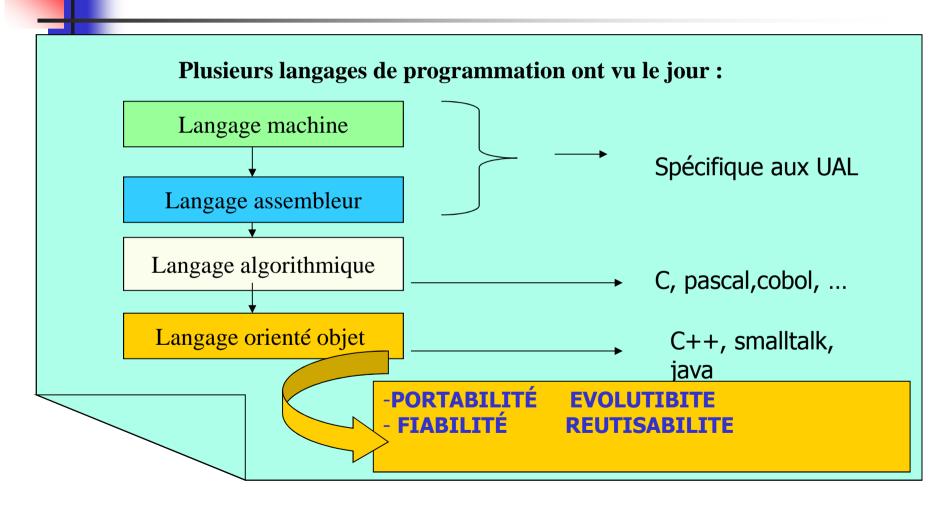
- des origines de Java;
- de la position du langage par rapport aux autres catégories de langages;
- des notions liées au langage (classe, objet);
- de l'encapsulation;
- de la Machine Virtuelle (JVM);
- du JDK/JRE;
- des éditions de Java;
- de l'organisation de l'API en paquetages
- des types de programmes que Java offre

Naissance du langage Java

Historique :Origines de Java

- 1990
- Internet très peu connu, World Wide Web inexistant.
- boom des PCs (puissance)
- Projet Oak de SUN Microsystems
 - Langage pour la communication des appareils électroniques .
- 1993
- mars : le NCSA lance MOSAIC, le premier navigateur internet (protocole http, langage html), le web décolle...
- été : Oak change d'orientation et s'adapte à la technologie internet
- 1995
- mai 1995 : projet *HotJava*,, navigateur WEB, écrit par SUN en Java .

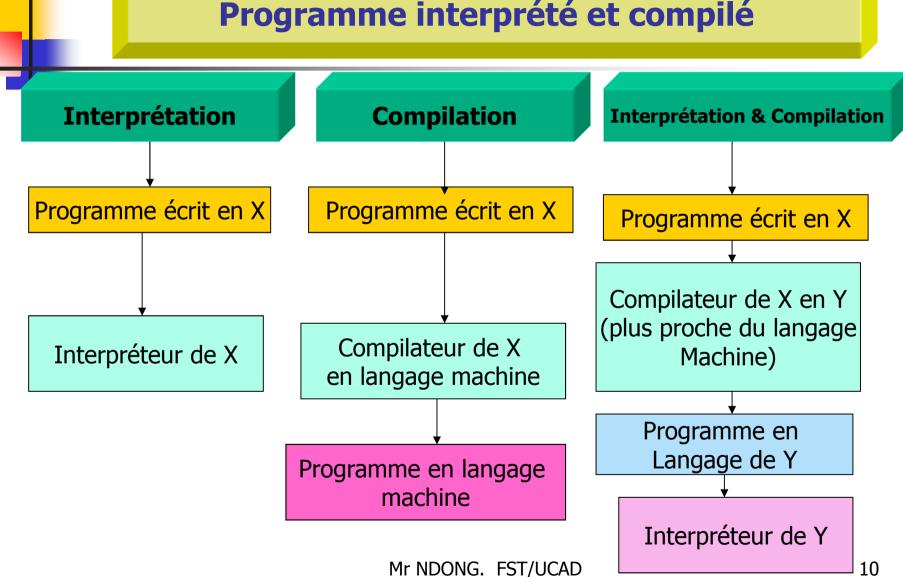
Les générations des langages de programmation

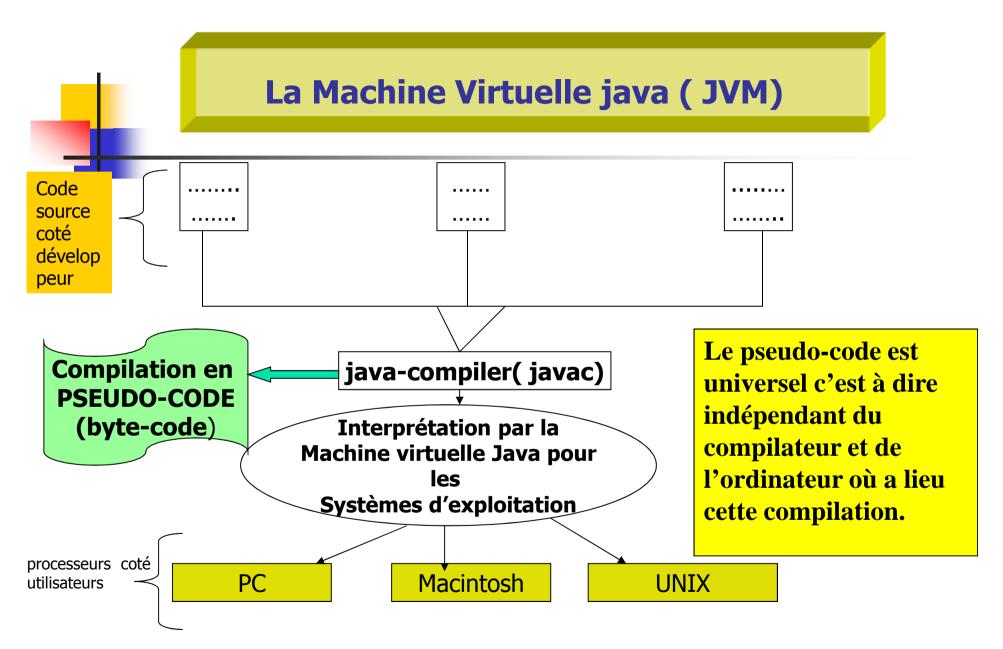


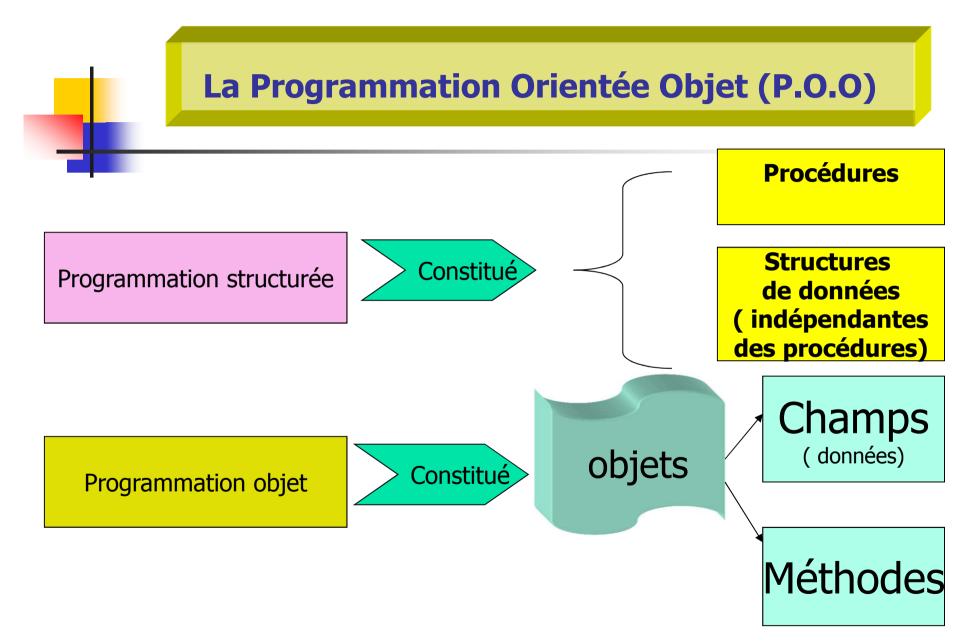


- Imprégné du C++ mais améliorant ses insuffisances
 - -> gestion automatique de la mémoire (Garbage Collector)
 - -> facilité de stockage des fichiers sur disque (sérialisation)
- Une gigantesque API (Application Programming Interface)
 - une librairie de classes très importante (interface graphique, réseau, web, base de données, ...)
 - portabilité sans mesure
 - langage de plus en plus utilisé et évoluant rapidement.

Programme interprété et compilé







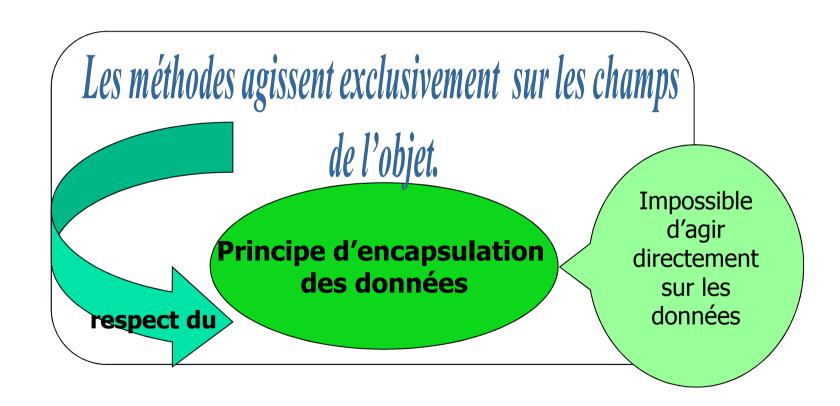
Concept de classe

Le concept de classe correspond simplement à la généralisation de type que l'on rencontre dans les langages classiques. En effet, une classe n'est rien d'autre que la description d'un ensemble d'objets ayant une structure de données commune et disposant des mêmes méthodes.

Les objets apparaissent alors comme des variables d'un tel type classe (en P.O.O, on dit aussi qu'un objet est une *instance* de sa classe). Bien entendu, seule la structure est commune , les valeurs des champs étant propres à chaque objet. En revanche, les méthodes sont communes à l'ensemble des objets d'une même classe.

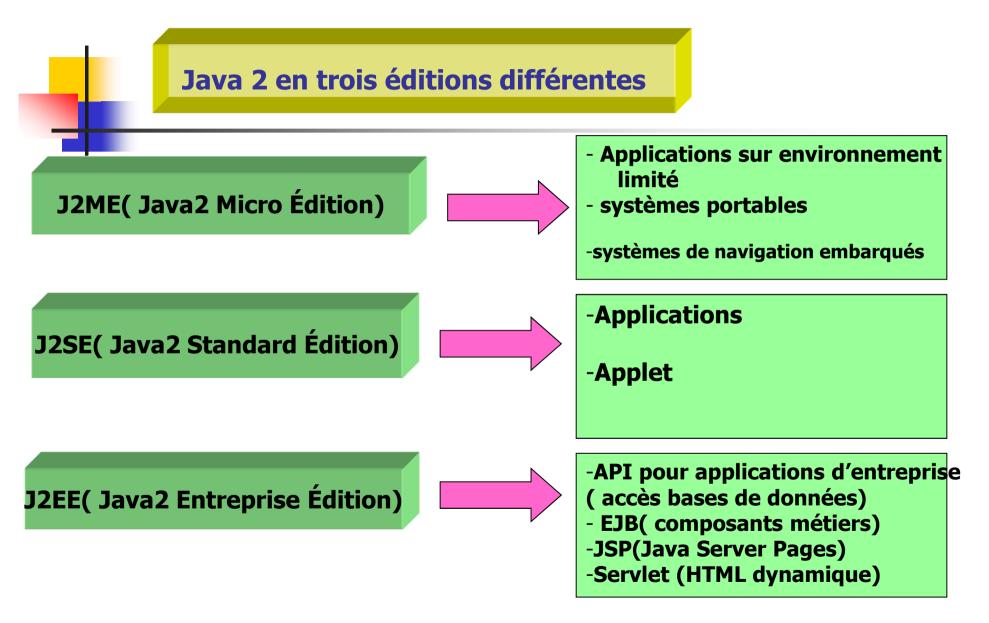


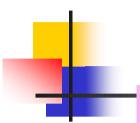
La P.O.O: l'encapsulation



JRE (Java Runtime Environnment)

- **Le JRE** contient uniquement *l'environnement d'exécution* de programmes Java. Le JDK contient lui même le JRE. Le
- **JRE** seul doit être installé sur les machines ou des applications java doivent être exécutées.
- Depuis sa version 1.2, Java a été renommé Java 2. Les numéros de versions 1.2 et 2 désignent donc la même version.
- Le JDK a été renommé J2SDK (Java 2 Software Development Kit) mais la dénomination JDK reste encore largement
- utilisée.
- Le JRE a été renommé J2RE (Java 2 Runtime Édition).
- Trois éditions de Java existent :
- J2ME : Java 2 Micro Édition ·
- J2SE: Java 2 Standard Édition •
- J2EE : Java 2 Entreprise Édition ·
- Sun fourni le JDK, à partir de la version 1.2, sous les plate-formes Windows, Solaris et Linux.





L'API

L'A.P.I. (Application Programming Interface) est un ensemble de classes et interfaces prédéfinies utilisables par le programmeur .Elles sont présentées dans des packages (*java.lang*, *java.math*, *java.util*, etc..

Un programme JAVA est constitué d'un certain nombre de classes :

- des classes prédéfinies de l'API (environ un millier de classes) (Application Programming Interface).
- des classes définies par l'utilisateur

Un programme utilise au maximum les fonctionnalités présentes dans l'A.P.I.

PROGRAMME

classes définies pour ce programme

API

classe : type de données + fonctions pour les manipuler



Application standalone et Application WEB

Les applications rentrent généralement dans le cadre des applications standalone ou des applications web destinées à l'

internet.

Application Standalone

Les applications autonomes sont des programmes exécutés sous le contrôle du système d'exploitation. Les programmes C, C++ ... sont des applications de ce type.

Application WEB

Les applications web sont exécutées lors de la visualisation par un navigateur WEB d'une page HTML. En général elles sont hébergées par des serveurs web comme Apache Tomcat. Elles sont très répandues avec la technologie JEE.

Applications : la console vs G.U.I

Programme à interface console

Programme à interface graphique : GUI (Graphical User Interface) L'utilisateur fournit des infos au clavier sous forme de lignes de texte. Le programme décide du séquencement des opérations. L'opérateur est sollicité au moment voulu.

L'interaction programme-opérateur se fait essentiellement via des *composants* graphiques. C'est la **programmation évènementielle** :le programme réagit à des évènements provoqués par l'utilisateur.



Techniques de base du langage

Ce cours enseigne les aspects suivants:

- élément constitutifs (structure) d'un programme autonome Java exécutable;
- compilation et exécution d'un programme Java en mode ligne de commande;
- de certaines commandes en ligne;
- des règles d'écritures;
- de la documentation automatique

Premier programma Java.

```
package hello.java.essai;

// un premier programme

/* la version JAVA du classique
Hello World

*/
public class HelloWorld {

    public static void main(String [ ] args) {

        System.out.println("Hello World !");
    }
}

Hello World !
```

Notes importantes



Ce cours a été initialement écrit avec le JDK 1.4.

Il faut savoir que le JDK 1.5 et supérieur a amené beaucoup d'améliorations et par conséquent beaucoup de mises à jour sont nécessaires. Je tente au mieux d'en apporter dans ce cours, mais le lecteur est vivement invité à consulter la documentation en ligne de Java pour mieux appréhender les améliorations.

Mises à jour JDK 5



Vous constatez que l'affichage sur la console se fait avec l'instruction System.out.println (..).

A partir du JDK 5, il est possible de faire des importations statiques; On peut donc désormais écrire simplement:

out.println (...)

A condition de faire l'importation:

import static java.lang.System.out;



Structure du programme (1/2)

```
package hello.java.essai;
import static java.lang.System.Out;

public class HelloWorld

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



Structure du programme (2/2)

- Le mot clé *static* précise que la méthode *main* n'est pas liée à une instance (objet) particulière de la classe.
- Le paramètre *String[] args* est un tableau de chaînes de caractères qui permet de récupérer des arguments transmis au programme au moment de son lancement. Ce paramètre est *OBLIGATOIRE* en Java.
- Le mot clé *public* dans *public class* sert à définir les **droits d'accès** des autres Classes (en fait de leurs méthodes) à la classe . [A voir].
- Le mot clé *public* dans *public static void main* est *obligatoire* pour que votre programme s'exécute. Il s'agit d'une convention qui permet à la machine virtuelle d'accéder à la méthode *main*.



Paquetage de la classe

La notion de paquetage se définit comme étant un regroupement (ensemble) de classes en une structure commune.

La classe définit ici (HelloWorld) appartient au paquetage nommé : hello.java.essai

La classe a un nom <u>simple</u>: HelloWorld Le nom <u>complet</u> de la classe est : hello.java.essai.HelloWorld

On précisera qu'une classe appartient à un paquetage en plaçant <u>en début de fichier</u> l'instruction <u>package nom_du_paquet</u>;

Pour utiliser des classes de l'API, il faut préciser leur package. On pourra utiliser l'instruction import nom_package.nom_de_la_classe.



Contenu du programme

Le programme est constitué d'une seule instruction :

System.out.println ("Hello World!");

System.out.println (argument)

Affiche la valeur de *argument* **puis passe** à la ligne

Classe
System du
package
java.lang

Variable de classe (référence un objet de type *PrintStream*)

Méthode d'instance de la classe PrintStream du package *java.io*

System.out.print (argument)

Affiche la valeur de *argument* sans passer à la ligne

Exécution du programme (1/2)

La sauvegarde du programme se fait impérativement dans un fichier qui porte le nom HelloWorld.jaya

→ extension → nom de la classe

Le code source d'une classe *publique* doit toujours se trouver dans un fichier portant <u>le même nom</u> et possédant l'extension *.java*.

La classe contenant la méthode main est appelée la classe principale

du programme. C'est cette classe qu'il faut exécuter. EN FAIT ON EXECUTE QUE LES INSTRUCTIONS DE LA METHODE main.



On procède à la COMPILATION de ce programme pour la génération du byte code.

Si elle se passe bien(sans erreurs) on obtient un fichier d'extension •class. Ici, il s'agit de HelloWorld.class.



java HelloWorld

aucune extension ne doit figurer

Commande *java* pour exécuter le byte code

<u>ATTENTION</u>: en pratique, on tiendra toujours compte des variables d'environnement PATH et CLASSPATH (cf. diapos suivantes).

Exemple pratique de compilation et d'exécution d'une classe (1/2)

(ici dans le paquet essai, il y a une seule classe : LectureClavier)

On dispose de la classe essai.LectureClavier se trouvant dans le répertoire C:\allndong\src\essai\LectureClavier.

```
SYNTAXE de la commande javac (compilation):
javac -d <répertoire où doivent être mis les fichiers générés>
-classpath <adresse des classes déjà compilées nécessaires à la compilation>
<adr. du(des) fichiers à compiler>
l'option -d permet de préciser le répertoire de base des fichiers .class générés par la compilation.
```

Pour ce premier exemple, aucune classe n'est nécessaire lors de la compilation, l'option classpath est donc absente.

```
C:\> javac -d D:\allndong\mescompils C:\ndong\src\essai\LectureClavier.java
```

```
SYNTAXE de la commande java(exécution) :
java -classpath <adresse des classes utilisées lors de l'exécution>
<nom complet de la classe principale>
```

Exemple pratique de compilation et d'exécution d'une classe (2/2)

(ici, dans le paquet essai, il y a maintenant la classe Lecture Clavier et la classe Utilise Lecture Clavier qui utilise les méthodes de cette dernière)

On veut maintenant compiler la classe essai. Utilise Lecture Clavier se trouvant dans le répertoire C:\ndong.

C:\> javac

-d D:\allndong\mescompils

-classpath C:\ndong\classes

C:\ndong\src\essai\UtiliseLectureClavier.java

C:\> java

-classpath D:\allndong\mescompils essai.UtiliseLectureClavier

Lorsqu'il y a plusieurs adresses à préciser pour une même option, séparer les adresses

par des ; (sous Windows) ou : (sous Linux))

compilation simultanée de plusieurs fichiers

Compilation de deux fichiers: essai.HelloWorld et essai.Compte.

javac -d D:\allndong\mescompils

-classpath D:\allndong\alldev.jar

C:\ndong\src\essai\HelloWorld.java

C:\ndong\src\essai\Compte.java

Compilation de tous les fichiers d'un répertoire (on compile toutes les classes du package essai):

javac -d D:\allndong\mescompils

-classpath D:\allndong\alldev.jar

C:\ndong\src\essai*.java

La variable d'environnement classpath

l'option classpath permet de préciser où trouver les classes

- utilisées lors de l'exécution du programme pour la commande java
- nécessaires à la compilation du programme pour la commande javac

A cette option peut correspondre une ou plusieurs valeurs, chacune d'elle pouvant être :

- l'adresse (relative ou absolue) d'un fichier jar.
- l'adresse (relative ou absolue) d'un répertoire de base de classes

Remarque : les classes de l'A.P.I. ne sont pas concernées par cette option

Si plusieurs valeurs sont associées à une option classpath, elles doivent être séparées par des ; (sous Windows) ou des : (sous linux).

La valeur par défaut de cette option est le répertoire courant (désigné par un ".")

<u>ATTENTION</u>: le répertoire de base d'une classe est le répertoire contenant le répertoire racine du paquetage.

Exemple: le répertoire de base de la classe essai.test.ndong.hello.HelloWorld est le répertoire contenant le dossier essai (qui lui même contient le dossier test etc...)

Création et utilisation de fichiers jar

(lors de la compilation d'une classe, celle-ci peut nécessiter l'utilisation <u>d'autres classes rassemblées au sein d'une archive jar</u>)

On peut créer un fichier jar correspondant au répertoire C:\ndong\classes.

Pour compresser le répertoire courant

- 1. Placez vous dans le répertoire de base des classes
- 2. Exécutez la commande

jar cf <adr. du fichier jar à créer> . (n'oubliez pas le point)

Exemple:

cd C:\ndong\classes

jar cf D:\allndong\mescompils\mesprogs.jar .

javac -d D:\allndong\mesclasses

-classpath D:\allndong\mescompils\mesprogs.jar

C:\ndong\src\essai\UtiliseLectureClavier.java

java -classpath D:\allndong\mesclasses essai. UtiliseLectureClavier

jar exécutable

On peut exécuter une classe à partir d'un fichier jar, il suffit simplement d'éditer le fichier MANIFEST.MF contenu dans le répertoire META-INF de l'archive. Avec n'import quel éditeur de texte, ouvrez le fichier Manifest (il ne contient que deux lignes) et éditer le de la façon suivante:

conserver la ligne (c'est la première):

Manifest-Version: 1.0

et remplacer la deuxième ligne c à d:

Created-By: 1.4.1_05 (Sun Microsystems Inc.)

par:

Main-Class: nom_complet_de_la_classe_principale.

Par exemple, pour la classe LectureClavier, le fichier manifest sera:

Manifest-Version: 1.0

Main-Class: essai.LectureClavier.

// pour exécuter le programme à partir du jar

C:\> java -jar D:\allndong\mescompils\mesprogs.jar

Les commandes de base du langage

Java fournit un ensemble de commandes de base très utiles pour diverses opérations que vous pouvez réaliser en ligne de commande. Voici quelques commandes:

javac : pour la compilation (générer le .class).

java: pour l'exécution (du main).

appletviewer: pour exécuter une applet.

javadoc: pour générer une documentation

automatique.

Catégories d'instructions

```
package info.fst.jnd.pro;
public class TypesInstructions
{ public static void main(String args [] )
                                                 Instruction simple terminée
   { System.out.print("instruction simple");
                                                     par un point-virgule
      int i = 0;
      if (i > 1000)
          System.out.print("instruction de structuration");
                                                Instruction de structuration
                                                          ( if, for,....)
     static { int tab[] = new int[12];
              for (i = 0;i < tab.length; i++)
                                                     Instruction en bloc
                   tab[i] = i + 10;
                                                      (présence des { } )
```



Les différentes entités utilisées dans un programme (méthodes, variables, classes, objets,) sont manipulées au travers *d'identificateurs*.

Un *identificateur* est formé de *lettres* ou de *chiffres*, <u>le premier caractère</u> étant obligatoirement une <u>lettre</u>. Les lettres comprennent les majuscules A-Z et les minuscules a-z, ainsi que le caractère « souligné »(_) et le caractère \$

Exemple:

ligne Clavier valeur_5 _total _56 \$total 2nombre

Remarque très importante :

Java est très sensible à la casse : ligne

Ligne



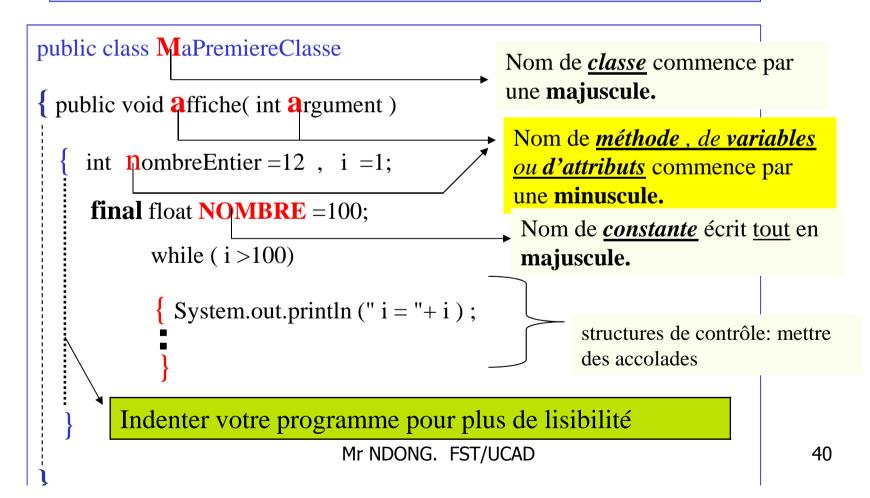
Règles d'écriture en Java (2/3)

Un identificateur ne peut pas appartenir à la liste des mots réservés du langage Java :

abstract	assert	boolean	break	byte
case	catch	char	class	const
continue	default	do	double	else
extends	false	final	finally	float
for	if	implement	ts import	instanceof
int	interface	long	native	new
null	package	private	protected	public
return	short	static	super	switch
synchronized this		throw	throws	transient
true	try	void	volatile	

Règles d'écriture en Java (3/3)

Voici quelques conventions de codage en java





Pour pouvoir utiliser une classe, il faut et il suffit de connaître son interface (la déclaration commentée de tous les membres publics) consultable grâce à la documentation en ligne

```
// extrait de la documentation en ligne de Terminal
/** crée un Terminal de titre 'titre' et de taille en pixels 'w' * 'h' */
public MyFrame (String titre, int w, int h){...}
/** Affiche un 'message' et rend l'entier lu à partir de la console*/
public int readInt (String message)
/** affiche l'entier 'i' dans la console */
public void println (int i) {...}
```

```
MyFrame fen = new MyFrame ("Tri fusion",300,300);
int taille = fen.readInt ("Donner un entier");
fen.println (taille*taille);
```



La documentation des programmes

La documentation en ligne peut être générée à partir des fichiers sources par l'utilitaire javadoc.

Cette documentation est organisée et générée de la même manière pour toutes les classes que ce soit les classes de l'API ou les classes que nous définissons nous même.

Lors du développement d'applications, il NE faut PAS négliger la documentation de vos classes (bien documenter les fonctionnalités).



A propos des commentaires

Commenter toujours les entêtes de fonctions

Un bon commentaire permet de pouvoir utiliser la fonction sans consulter le code.

- il indique à l'aide d'une phrase le rôle de la fonction en faisant intervenir le nom de tous les paramètres
 - il précise le rôle de chaque paramètre
- il indique la signification du résultat retourné
- il indique les restrictions sur la valeur des paramètres

Commenter *si nécessaire* des fragments de codes difficiles (un bon programme en contient généralement peu)

Éviter les commentaires inutiles

A = 5; /* a prend la valeur 5 */

Types de commentaires en Java

```
package hello.java.essai;
/**
                              /* ceci est un commentaire de documentation
* @param autor Joseph
                                        automatique javadoc */
* @since 1.0
*/
                                // Ceci est un commentaire sur une seule ligne
// un premier programme
/* la version JAVA du classique
                                   /* ceci est un commentaire pouvant encadrer
  Hello World
                                       un nombre quelconques de caractères
*/
                                       sur un nombre quelconque de lignes */
public class HelloWorld {
         public static void main(String [ ] args) {
                  out.println("Hello World !");
```



Les commentaires JAVADOC

Les commentaires javadoc /** ... */ sont des commentaires spéciaux permettant la production automatique de documentation au format html.

Ils sont placés juste avant ce qu'ils commentent.

balises de commentaires **JAVADOC** de classe

@see <une autre classe> @author <nom de l'auteur> @version $< n^{\circ} de version >$

balises de commentaires **JAVADOC** de fonction

```
@param <nom paramètre> <description>
      @return <description>
      @exception < nom exception > < description >
      @since < n^{\circ} de \ version >
      @deprecated
Mr NDONG. FST/UCAD
                                                45
```



Contenu d'une documentation javadoc

Description générale de la classe

Description des attributs (FIELD)

Description des constructeurs (CONSTRUCTOR)

Description des méthodes (METHOD)

La description des <u>attributs</u>, des <u>constructeurs</u> et des <u>méthodes</u> publics est donnée

- brièvement en début de document
- en détail dans la suite du document.



La commande javadoc

GENERATION DE LA DOCUMENTATION

javadoc -d <répertoire où doivent être mis les fichiers générés> -sourcepath <répertoire(s) de base des fichiers sources> <nom du paquetage>

- sourcepath

le(s) répertoire(s) de base des sources (s'il y en a plusieurs, séparer par des ; (Windows) ou : (linux))

La documentation est ensuite consultable à partir du fichier index.html du répertoire mentionné avec l'option -d .



La commande javadoc: exemple

Pour générer la documentation des classes du paquetage essai

```
javadoc -d D:\allndong\javaprog\doc -sourcepath C:\ndong\src essai
```

Pour générer la documentation des classes des paquetages essai (dans C:\ndong) et exemple.test (dans D:\allndong)

```
javadoc -d D:\allndong\javaprog\doc -sourcepath C:\ndong\src;D:\allndong\src essai exemple.test
```



Environnement de développement: les IDEs

Vous pouvez développer de simples programmes Java à l'image de notre premier exemple, le célèbre HelloWorld avec un éditeur simple « non intelligent » comme le **BLOC-NOTES**. Dans ce cas, vous serez obligé de réaliser la compilation (**javac.exe**) et l'interprétation (**java.exe**) du programme via la ligne de commande.

Cependant, pour être plus à l'aise et réaliser des programmes sophistiqués, vous devez utiliser un environnement de développement intégré ou IDE (Integrated Development Environment). Un IDE est intelligent et présente plusieurs avantages parmi lesquels:

- la possibilité de créer complètement la hiérarchie du projet;
- la disponibilité d'un audit de code qui offre les fonctionnalités du JDK;
- la facilité d'utiliser les commandes de base de Java (javac, java, javadoc, jar,...);
- l'intégration à chaud d'un projet dans un autre projet à l'aide de fichiers jars;
- l'ajout de librairies tierces dans un projet ;
- etc

IDEs populaires

Il existe plusieurs IDEs utilisables dans le cas de la P.O.O en Java.

Parmi les plus célèbres on trouve:

- Eclipse;
- NetBeans

Dans ce cours, je propose l'utilisation Eclipse qui propose plusieurs avantages, notamment, sa souplesse, sa facilité d'installation, sa facilité d'extension à l'aide de plugins, sa prise en main assez facile.

Le lecteur peux aussi tester l'environnement NetBeans qui est également excellent.

Tutoriel sur Eclipse

Eclipse est proposé en plusieurs versions selon le type de programmation que vous souhaitez réaliser. La page http://www.eclipse.org/downloads/ donne les différentes versions.

Pour apprendre les fondamentaux de base de Java, la version nommée <u>Eclipse IDE for Java Developers</u> suffit. Ceux qui souhaitent apprendre les notions avancées en JEE ont besoin de la version nommée <u>Eclipse IDE for Java EE Developers</u>.

C'est ce dernier que j'utilise pour préparer les étudiants au cours de JEE.

Installation de Eclipse



L'installation d'un IDE pour Java nécessite au préalable l'installation du JDK standard. Il faut choisir une version du JDK compatible avec la version de l'IDE choisie.

Ici, j'ai choisi la version Eclipse JEE JUNO version 1.5.2 qui est compatible avec le JDK1.5 et supérieur.

L'installation de Eclipse es t facile:

- 1) Télécharger Eclipse sous format .ZIP;
- 2) Décompressez le dans un répertoire de votre choix (ex. C:\eclipse)

A cet instant vous pouvez créer un raccourci de l'exécutable d'Eclipse dans votre bureau. Lancer ce raccourci pour commencer à travailler avec Eclipse.

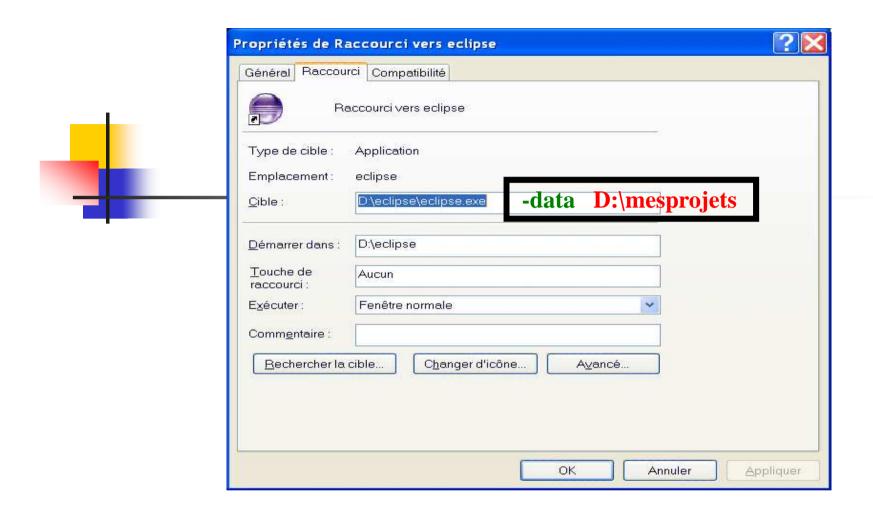
Utilisation de Eclipse



Pour bien travailler de façon générale avec un IDE, il faut commencer par créer un espace de travail (workspace). Il s'agit d'un répertoire où vos futurs projets Java résideront.

Vous pouvez laisser Eclipse vous demander de choisir votre workspace chaque fois que vous le démarrer. Cette option est intéressant si vous disposez de plusieurs workspaces. Dans le cas où vous avez un seul workspace, il faut mieux le configurer une seule fois. Il suffit de faire un clic droit sur le raccourci de votre bureau + clic sur propriétés.

Il apparait alors la fenêtre suivante:



Au niveau de **Cible**, placez un espace et mettez votre workspace. Par exemple, si mon workspace est le répertoire **D:\mesprojets**, alors on a comme cible:

Cible: D:\eclipse\eclipse.exe -data D:\mesprojets

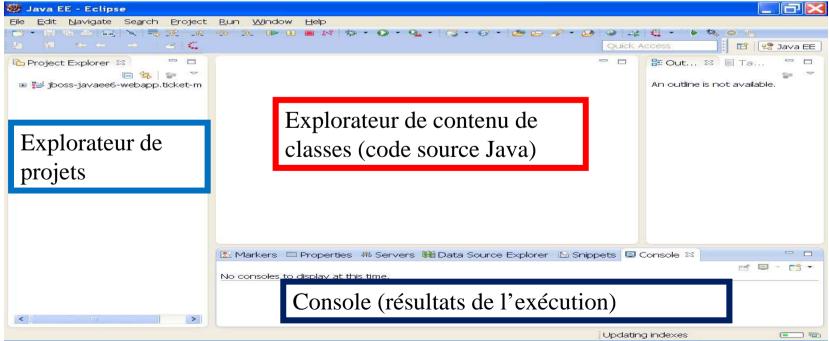
Utilisation de Eclipse



Une fois votre espace de travail mis en place, vous pouvez commencer à y développer des projets. Programmer en Java doit toujours faire l'objet d'un fichier projet. Il s'agit d'un répertoire se trouvant dans votre workspace et devant contenir vos classe Java.

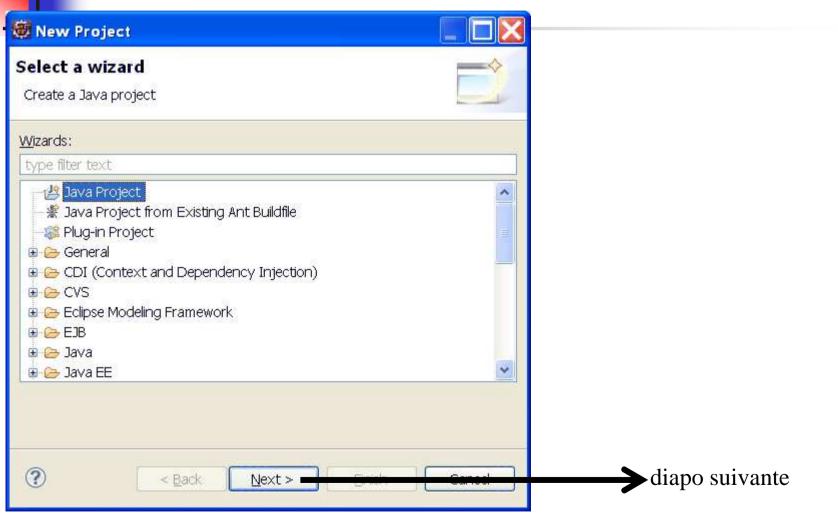
Je montre dans ce qui suit, à l'aide de captures d'écran les différentes phases allant de la création d'un nouveau projet jusqu'à la phase d'exécution du programme.

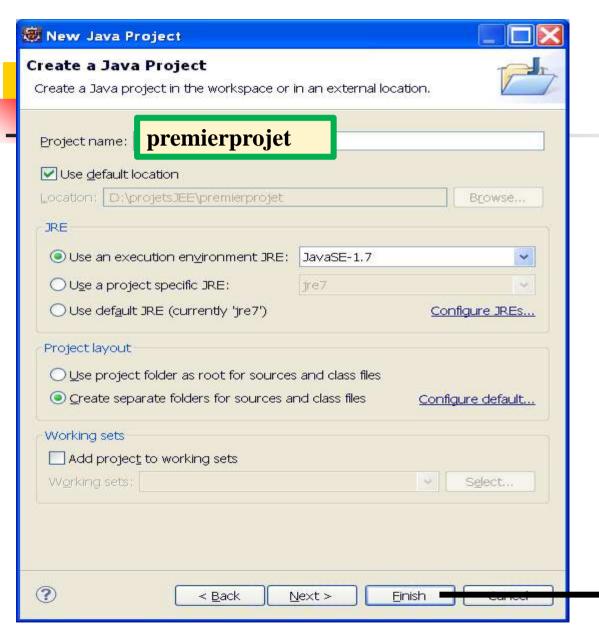




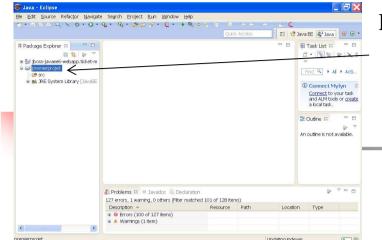
Pour créer un Nouveau projet Java, faire :

menu File → New → Java Project ou bien menu File → New → Other → Java → Java Project





Le projet vient d'être créé.

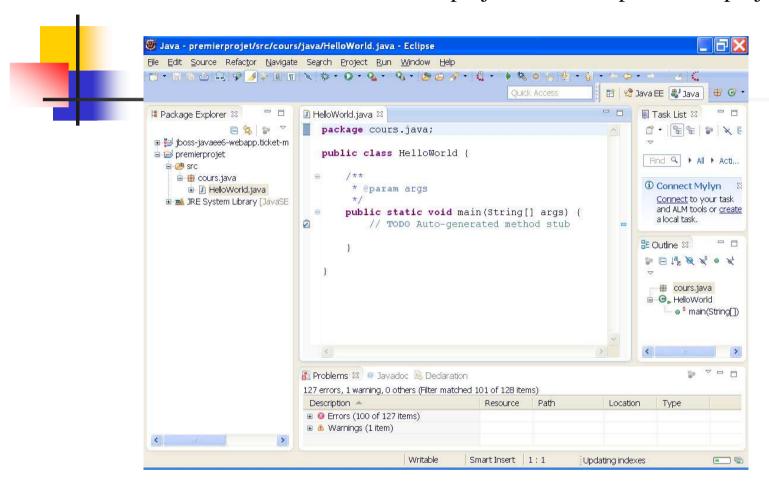


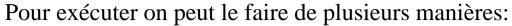
Pour ajouter une nouvelle classe dans ce projet, faire un clic droit sur le nom du projet + new classe.

- 1) Saisir le package ou Browse pour le choisir
- 2) Saisir le Nom de la nouvelle classe
- 3) EN OPTIONS: cochez la méthode main si la classe doit en contenir. Vous pouvez aussi modifier les droits de visibilité.

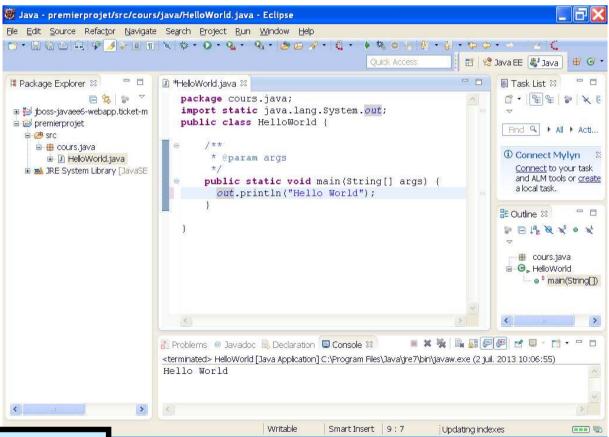


Vous obtenez la structure de projet ci-dessous pour votre projet.





- 1) cliquer sur le bouton en forme de flèche entourée en vert sur la barre des tâches ou
- 2) Clic droit sur le nom de la classe + Run As + Java Application
- 3) Clic droit sur le code source+ Run As + Java Application Le résultat ou les erreurs s'affichent alors sur la console.





Module 3

Les Types Primitifs

Ce cours enseigne les aspects suivants:

- les types de base;
- l'initialisation de variables;
- les opérateurs de base;
- les conversions implicites légales
- les conversions forcées (transtypage ou cast)
- les priorités sur les opérateurs

Les différentes familles de types primitifs

Java dispose d'un certain nombre de types de base dits *primitifs*, permettant de manipuler des entiers, des flottants, des caractères et des booléens.

Ce sont les seuls types du langage qui ne sont pas des classes.

Les types primitifs(au nombre de 8) se répartissent en quatre grandes catégories selon la nature des informations qu'ils permettent de manipuler:

- nombres entiers,
- nombres flottants,
- caractères
- booléens

Le type entier

Ils servent à représenter les nombres entiers relatifs, ils sont au nombre de 4.

BYTE (1 octet) SHORT (2 octets) INT (4 octets) LONG (8 octets)

byte 8 bits : -128 à 127

short 16 bits: −32768 à 32767

int 32 bits: -2147483648 à 2147483647

long 64 bits : −9223372036854775808 à 9223372036854775807

Les types élémentaires ont une taille identique quelque soit la plate-forme d'exécution.

Le type flottant (réel)

Ils permettent de représenter, de manière approchée, une partie des nombres réels. Java prévoit deux types de flottants correspondant chacun à des emplacements mémoire de tailles différentes: *float et double*.

double: 64 bits **float**: 32 bits

Exemple:

```
float pi = 3.141f; // nécessaire de suffixer par la lettre f sinon erreur de compilation
```

```
double v = 3d; // suffixe d non nécessaire 3d = 3
```

float $f = +1.1\mathbf{f}$, $d = 1e10\mathbf{f}$; // déclaration de plusieurs variables de même type



Le type caractère

```
CHAR (2 octets)
```

char: caractère isolé

-codage unicode sur 16 bits non signé

-expression littéral char entre apostrophes

Une variable de type caractère se déclare:

char c1; char c1,c2; // c1 et c2 sont 2 variables caractère

Pour manipuler une séquence de caractères, on utilise le type <u>String</u>.

String s1 = "bonjour", s2 = new String();

ATTENTION: String est une classe du paquetage java.lang.

Le type booléen

Ce type sert à représenter une valeur logique du type *vrai=true/faux=false*

```
BOOLEAN

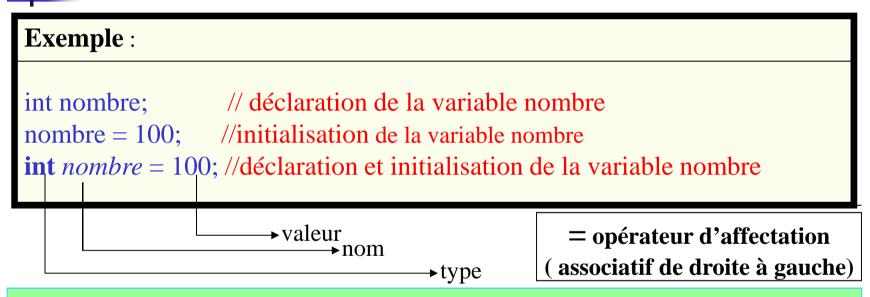
-Deux états: true / false
-Exemples:

boolean ok=false;

if (n < p) ... //n < p est expression booléenne valant vrai ou faux
boolean ordonne; // déclaration d'une variable de type booléenne

ordonne =n < p; //ordonne reçoit la valeur de l'expression n < p
```

Initialisation des variables (1/2)



Remarque:

Une variable manipulée dans une <u>méthode</u> (variable locale) ou un <u>bloc</u> devra <u>toujours</u> être initialisée avant toute utilisation.

Une variable GLOBALE (déclarée comme champ dans une classe) est toujours initialisée par défaut à une valeur « nulle » dépendant de son type.

La déclaration d'une variable réserve de la mémoire pour stocker sa valeur.

Initialisation des variables (2/2)

En java, toute variable appartenant à un objet (définie comme étant un attribut de l'objet) est initialisée avec une <u>valeur par défaut</u> en accord avec son <u>type</u> au moment de la création.

Cette initialisation ne s'applique pas aux <u>variables locales</u> des méthodes de la classe (cf :remarque précédente).

ТҮРЕ	VALEUR PAR DÉFAUT		
boolean	false		
byte, short, int, long	0		
float, double	0.0		
char	u\000		
classe (type objet)	null		

Utilisation des variables

```
package home.user.java.essai;
import static java.lang.System.out;
public class UtilVariable
      String chaine; // variable de type (class ) String, valeur par défaut null
      double solde; // valeur par défaut 0.0 assignée à solde
      public static void main(String [ ] args)
         out.println(" valeur de solde =" +solde);
                                                                    new pour
                                                                  créer un objet
    public void affiche( )
      chaine = new String(" bonjour" ); // objet de type String initialisée
      long nombre;
      out.println(" valeur de nombre= "+nombre); // erreur de compilation
                                                 // car nombre non initialisée
```

variables finales (le mot clé final)

Java permet de déclarer que la valeur d'une variable ne doit pas être modifiée pendant l'exécution du programme.

```
package home.user.java.essai;
public class VariableFinale
{ final long NOMBRE;

final double MAX = 100; // variable finale = constante
public static void main(String [] args)
{
    out.println(" utilisation de variable constante");
}
public void affiche()
{ NOMBRE = 1000; // initialisation différée de la variable NOMBRE
    out.println(" valeur de MAX= "+MAX);
}
```

Les Opérateurs (arithmétiques binaires)

```
package home.user.java.essai;
public class Op_Arith_Binaire
   { public static void main(String [] args)
       { int a = 1, b = 2;
          int addit = a + b,
             soustr = a - b,
             div = a/b.
             multi = a * b;
System.out.println(" addit =" +addit );
System.out.println(" soustr =" +soustr );
System.out.println(" div =" +div );
System.out.println(" multi =" +multi );
```

RETENEZ BIEN

+ - * / sont des
opérateurs arithmétiques
binaires qui portent sur deux
opérandes de <u>même type</u> et
renvoient un résultat du même
type que le type des opérandes ...

```
addit = 3
soustr = -1
div = 0
multi = 2
```



Les Opérateurs (unaire et modulo)

```
package home.user.java.essai;
                                                             Opérateur modulo:
public class Op_Unaire_et_Modulo
                                                             fournit le reste de la
   { public static void main(String [] args)
                                                             division de deux
       { int a = 100, b = 50;
                                                             opérandes.
          int modulo = 100 \% 50;
System.out.println(" le reste de la division de " +a+" par" +b+" est" +_modulo );
           float x = +10.2f;
           double d= 50.2;
System.out.println (" x et d sont des opérateurs unaires portant sur un seul opérande"):
                    le reste de la division de 100 par 50 est 0
                    x et d sont des opérateurs unaires portant sur un seul opérande
```



Les Opérateurs (priorité)

Lorsque plusieurs opérateurs apparaissent dans une même expression, il est nécessaire de savoir dans quel ordre ils sont mis en jeu

Les opérateurs unaires + et – ont la priorité la plus élevée. On trouve ensuite au même niveau, les opérateurs *, / et %. Au dernier niveau, se retrouvent les opérateurs binaires + et -.

En cas de priorité identique, les calculs s'effectuent de gauche à droite. On dit que l'on a une *associativité de gauche à droite*.

Exception sur la manipulation des opérateurs (1/2)

Un programme en cours d'exécution peut rencontrer des erreurs si la JMV ne parvient pas à exécuter des instructions. Ces erreurs sont correspondent soit à des erreurs système graves, soit à des erreurs dites « erreurs légères » ou exceptions. Si ces exceptions ne sont pas bien traitées, elles entraînent l'arrêt brutal du programme.

```
package home.user.java.essai;
/* test sur la division par zero de nombres entiers */
    public class Test {
        public static void main (String args[])
        {
            int valeur=10;
            double résultat = valeur / 0;
            out.println("index = " + résultat);
        }
        Exception in thread « main »
            java.lang.ArithmeticException:/
        by zéro at Test.main ( Test.java:9)
```



Exception sur la manipulation des opérateurs (2/2)

Pour les <u>entiers</u>, la division par zéro (par / ou %) conduit à une erreur d'exécution. Plus précisément, il y a déclenchement de ce que l'on nomme une *exception* de type *ArithmeticException*.

Nous verrons plus tard comment traiter convenablement les exceptions.

Si nous ne le faisons pas, nous aboutissons à l'arrêt de l'exécution du programme, avec un message de ce type:

Exception in thread « main » java.lang.ArithmeticException:/ by zéro at Test.main (Test.java:9)

REMARQUE:

Pour les *flottants*, aucune opération ne conduit à un arrêt de l'exécution (pas même une division par zéro) ; seulement on a une valeur non exploitable.

Conversions implicites (1/5)

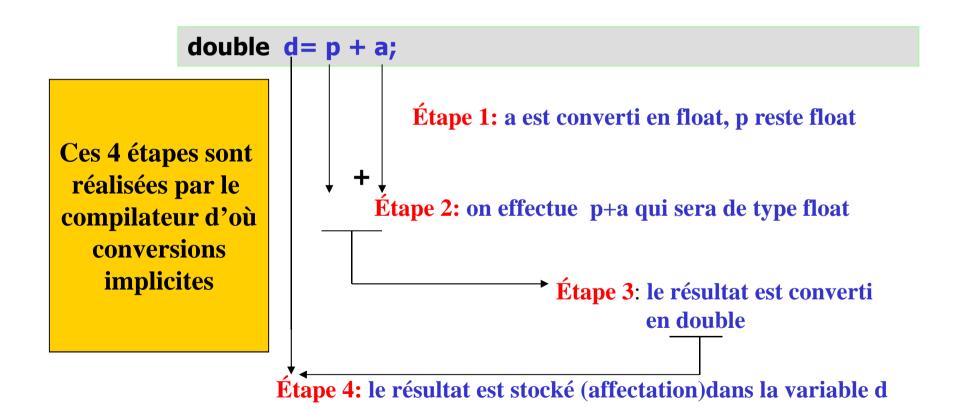


Les opérateurs arithmétiques ne sont définis que lorsque leurs deux opérandes sont de même type. Mais, vous pouvez écrire des *expressions mixtes* dans lesquelles interviennent des opérandes de types différents.

```
public class ConversionLegale01 {
  public static void main(String args [])
  { int a = 10 ;
    float p = 14 ;
     double d= p + a;
     out.println("la valeur de l'expression mixte (p+a) est :"+d);
  }
}
```

La valeur de l'expression mixte (p+a) est : 24.0







Conversions implicites (3/5)

Conversions d'ajustement de type

Une conversion telle que int en float est appelée ajustement de type.

Elle ne peut se faire que suivant une hiérarchie qui permet de ne pas *dénaturer la valeur initiale*, à savoir :

int -> long ->float ->double

NB : une conversion de double en float (par exemple) n'est pas légale . pour l'exemple précèdent on ne peut pas faire :

int k = p + a; // erreur de compilation



Promotions numériques

les opérateurs numériques ne sont pas définis pour les types byte, char et short.

Toute opération qui utilise l'un de ces types nécessite une conversion préalable dans le type int

Cette conversion porte le nom de promotion numérique.

Conversions implicites (5/5)

```
public class ConversionLegale02 {
  public static void main(String args [])
  { char c = 'd'; // le code du caractère 'd' est converti en int
    Short s =0; // la variable s est convertie également en int
    int n = c + s; // le résultat de type int est affecté à n
    System.out.println("la valeur de l'expression mixte (c+s) est :"+n );
  }
}
```

La valeur de l'expression mixte (c+s) est : 100

Les opérateurs relationnels(1/2)

signification
inférieur à
inférieur ou égal à
supérieur à
supérieur ou égal à
égal à
différent de

Les quatre premiers (<, <=, >, >=) sont de même priorité. Les deux derniers (= = et !=) possèdent également la même priorité, mais celle-ci est inférieur à celle des précédents

Ces opérateurs sont <u>moins</u> prioritaires que les opérateurs arithmétiques. Ils soumettent eux aussi leurs opérandes aux promotions numériques et ajustement de type.



Les opérateurs relationnels(2/2)

Exemple:

```
public class Oper_Relat {
  public static void main(String args [])
  { int n = 10;
    short s = 10;
    float x = 100;
    double d = 200;
    out.println("Affichage 1 :"+(n = s));
    out.println("Affichage 2 :"+(n = s));
}

Affichage 1 : true
Affichage 2 : false
```



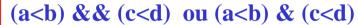
Les opérateurs logiques(1/3)

Java dispose d'opérateurs logiques classées par ordre de priorités décroissantes (il n'existe pas deux opérateurs de même priorité).

Le résultat est toujours un booléen.

Opérateur	Signification
!	négation
&	et
^	ou exclusif
	ou inclusif
&&	et(avec cout-circuit)
	Ou inclusif(avec court-circuit

Les opérateurs logiques (2/3)



prend la valeur true (vrai) si les deux expressions a
b et c<d sont toutes les deux vraies (true), la valeur false (faux) dans le cas contraire.

prend la valeur true si l'une **au moins** des deux conditions a
b et c<d est vraie, la valeur false dans le cas contraire.

prend la valeur true si **une et une seule** des deux conditions a
b et c<d est vraie, la valeur false dans le cas contraire.

prend la valeur true si la condition a<b est fausse, la valeur false dans le cas contraire. Cette expression possède la même valeur que a>=b.



Les opérateurs logiques (3/3)

Les opérateurs de cout-circuit && et || .

Ces deux opérateurs recèlent une propriété très importante: leur second opérande (figurant à droite de l'opérateur) n'est évalué qui si la connaissance de sa valeur est indispensable pour décider si l'expression correspondante est vraie ou fausse.

Exemple:

```
( 15 < 10 ) && ( 10 > 4 ) //on évalue 15 < 10 , le résultat // est faux donc on n' évalue pas // 10 > 4
```

Opérateurs d'incrémentation et de décrémentation (1/2)

incrémentation

```
int i = 10;
    post incrémentation

i++; // cette expression vaut 10
    //mais i vaut 11
int j = 10;

++j; // cette expression vaut 11
    //et j vaut aussi 11

    pré incrémentation
```

```
En fait en écrivant :
int n= i++ ;
on a :
n= i ;
i = i+1 ;
Et en écrivant :
int p= ++j ;
on a:
j = j+ 1 ;
p = j ;
```

Il existe un opérateur de décrémentation notée - -



Remarque

Il faut bien constater que le comportement des opérateurs de post ou de pré incrémentation ou décrémentation dépend de leur utilisation dans une <u>instruction</u> ou dans une <u>expression</u>.

Une **instruction** est **indivisible** tandis qu'une **expression** est un ensemble d'instructions. Dans l'exemple précédent la post et la pré incrémentation sont équivalentes et donnent le même résultat dans une instruction (i++ et ++i). Tandis dans une expression, la **post** et la **pré** incrémentation/décrémentation sont totalement différentes.



Opérateurs d'incrémentation et de décrémentation (2/2)

Les opérateurs d'incrémentation et de décrémentation permettent d'alléger l'écriture de certaines expressions et ils n'appliquent aucune conversion à leur opérande.

Opérateurs d'affectation élargie

Rôle de ces opérateurs:

- 1) condenser certaines expressions;
- 2) faire des opérations avec les types byte, char et short en conservant le type d'origine.

variable = variable opérateur expression ⇔ variable opérateur = expression

Liste complète des opérateurs d'affectation élargie :

Transtypage: opérateur de cast



(permet de forcer le type d'une expression dans un autre type)

```
On a vu qu'il n'était pas possible d'écrire :
byte b = 10;
b = b + 1; //erreur de compilation
             // incompatible type for =. Explicit cast
            //needed to convert int to byte.
Mais on peut faire ceci:
byte b = 10;
b = (byte)(b + 1); //OK \Leftrightarrow b += 1
                     On force le résultat dans le type byte
                 (double) 10/3 # (double)(10/3)
```

Opérateurs de manipulation de bits (1/3)

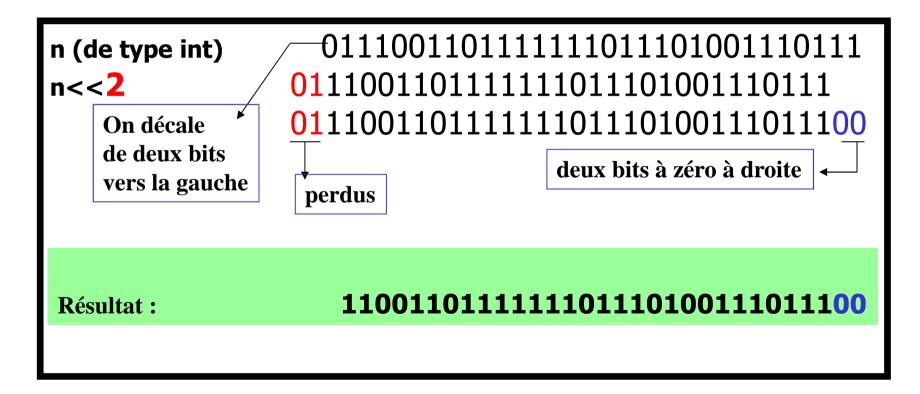
Ils travaillent sur le motif binaire d'un ENTIER, avec possibilité de conversions implicites.

Opérateur	Signification	Opérateur	s bit	à bit	-	
&	Et (bit à bit)	Opérande 1	0	0	1	1
1	Ou inclusif(bit à bit)	Opérande 2	0	1	0	1
^	Ou exclusif(bit à bit)	&	0	0	0	1
<<	Décalage à gauche					
>>	Décalage arithmétique à droite	1	0	1	1	1
>>>	Décalage logique à droite					_
~(unaire)	Complément à un (bit à bit)			•	1	U





Opérateur de décalage <<





Opérateurs de manipulation de bits (3/3)

Opérateur de décalage >> et >>>

Les bits de gauche sont des bits identiques au bit de signe. >>> identique à >> sauf que tous les bits de gauche sont à zéro.



Opérateur Conditionnel ?



temps =ciel==bleu ?"beau" : " mauvais"

Priorité des opérateurs (1/2) (ordre de priorité décroissante)

les parenthèses	()
les opérateurs d'incrémentation	++
les opérateurs de multiplication, division, et modulo	* / %
les opérateurs d'addition et soustraction	+ =
les opérateurs de décalage	<< >>
les opérateurs de comparaison	< > <= >=



les opérateurs d'égalité	== !=
l'opérateur OU exclusif	^
l'opérateur ET	&
l'opérateur OU	
l'opérateur ET logique	&&
l'opérateur OU logique	II
les opérateurs d'assignement	= += -=

Module 4

Les structures de contrôle

Ce cours enseigne les aspects suivants:

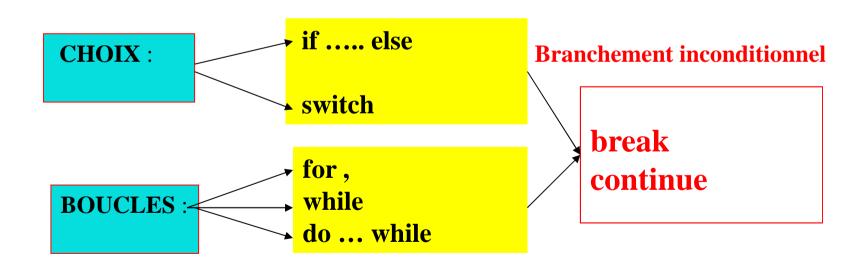
- les structures de choix;
- les structures de boucle;
- les boucles avec étiquette;
- les branchements inconditionnels: break et continu.

Types de structures



Les instructions d'un programme (main) sont à priori exécutées de façon séquentielle.

Les instructions de contrôle permettent de s'affranchir de cet ordre pour effectuer des **choix** et des **boucles**.



Choix: if ...else switch

```
package home.user.java.essai;
public class Exemple_If_Else{
int final MIN = 100;
int final Max = 1000;
int solde;
public static void main(String args [ ])
 { if ( solde < MIN)
   System.out.println("solde insuffisant");
  else
      if (solde = = MAX)
System.out.println("solde suffisant");
```

```
package home.user.java.essai;
public class Exemple_Switch{
int final MIN = 100;
int final Max = 1000;
int choix, solde;
public static void main(String args [ ])
 { switch(choix)
   { case 1: solde = MIN;
   System.out.println("solde insuffisant")
             break;
     case 2: solde = MAX;
    System.out.println("solde suffisant");
             break:
     default: break
```

Syntaxes: if ...else

```
switch
          if (condition)
                                                   Condition booléenne (true / false)
                         instruction_1
                                                   Expressions quelconques
              else
                  instruction_2
                                                Les crochets renferment des instructions
                                                facultatives.
switch (expression)
                                                          Expression de type byte, char,
                                                          short ou int.
       case constante_1 : [suite_d'instruction_1]
       case constante_2 : [suite_d'instruction_2 ]
                                                          Expression constante d'un type
       case constante_n : [suite_d'instruction_n ]
                                                          compatible par affectation avec
       [ default : suite_d'instructions ]
                                                          le type de expression
                                        Mr NDONG. FST/UCAD
```

L'instruction do while

```
package home.user.java.essai;
import java.util.Scanner; // importation de classe de l'API
public class Exemple_Do_While{
public static void main (String args [ ])
 { Scanner clavier = new Scanner (System.in);
do
     System.out.println ("saisir un entier strictement positif");
     n = clavier.nextInt(); // saisir à partir du clavier
     if (n < 0) System.out.println ("la saisie est invalidée: recommencez");
 while ((n < 0) || (n = = 0));
```

do instruction

Expression quelconque

Condition booléenne

while (condition);

L'instruction while

```
package joseph.cours.java;
public class Exemple_While{
public static void main(String args [ ])
 { while (n < = 0)
    { System.out.println ("saisir un entier strictement positif ");
     n = clavier.nextInt();  // saisir à partir du clavier
     if (n < 0) System.out.println ("la saisie est invalidée: recommencez");
while (condition);
                                          Condition booléenne
                                          Expression quelconque
   instruction
```

L'instruction for

```
for ([initialisation];[condition]; [incrémentation])
instruction
```





Ces instructions s'emploient principalement au sein des boucles.

break

Elle sert à interrompre le déroulement de la boucle, en passant à l'instruction suivant la boucle.

break avec imbrication de boucles

```
package home.user.java.essai;
public class Essai_Break_Imbr {
public static void main (String args [ ])
{ int tab [] = new int [10]; // tableau d'entiers de taille 100
 for( int i = 0; i < 10; i ++)
          for (int i = i; i < 10; i++)
               { if (j == 3 || j == 4 || j == 5) break;
                  tab [j] = j+1;
                } // ← le break branche ici
 for (int i = 0; i < 10; i ++)
System.out.println ("éléments du tableau:"+" "+tab [i]);
     éléments du tableau:
                               1 2 3
                                         0 0 0 7 8 9 10
```

En cas de boucles imbriquées, l'instruction break fait sortir uniquement de la boucle la plus interne.

break avec étiquette

```
package joseph.cours.java;
public class Essai_Break_Etiq {
public static void main (String args [ ])
{ int tab [] = new int [10]; // tableau d'entiers de taille 100
 repeat: for( int i = 0; i < 10; i ++)
          for (int j = i; j < 10; j++)
               { if (j == 3 || j == 4 || j == 5) break repeat;
                   tab [ i ] = i+1:
                                                            Étiquette : pour sorti
                                                            de deux boucles imbr quées
    } // ← cette fois le break branche ici
 for (int i = 0; i < 10; i ++)
System.out.println ("éléments du tableau:"+" "+tab [i]);
                    éléments du tableau:
                                              1 2 3
                                                         0 0 0 0 0 0 0
```



Continue ordinaire

continue

L'instruction continue permet de passer *prématurément* au tour de boucle suivant.

MI NOONG. 131/OCAD

Continue avec étiquette

```
package home.user.java.essai;
public class Essai_Continue_Etiq {
public static void main(String args [ ])
{ int tab [] = new int [10]; // tableau d'entiers de taille 100
 again: for(int i = 0; i < 10; i ++) // \leftarrow cette fois le continue branche ici
          for (int i = i; i < 10; i++)
               { if (j == 3 || j == 4 || j == 5) continue;
                   tab [ i ] = i+1:
                                          ATTENTION: on ne mentionne pas
                                          le nom de l'étiquette avec continue.
    } for( int i = 0; i < 10; i ++)
System.out.println(" éléments du tableau:" +" "+tab [ i] );
                  éléments du tableau:
                                            1 2 3
                                                      0 0 0 7 8 9 10
```

Module 5

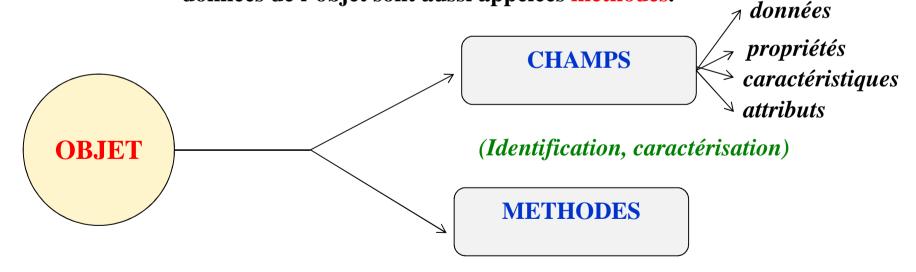
Programmation Orientée Objet

Ce cours enseigne les aspects suivants:

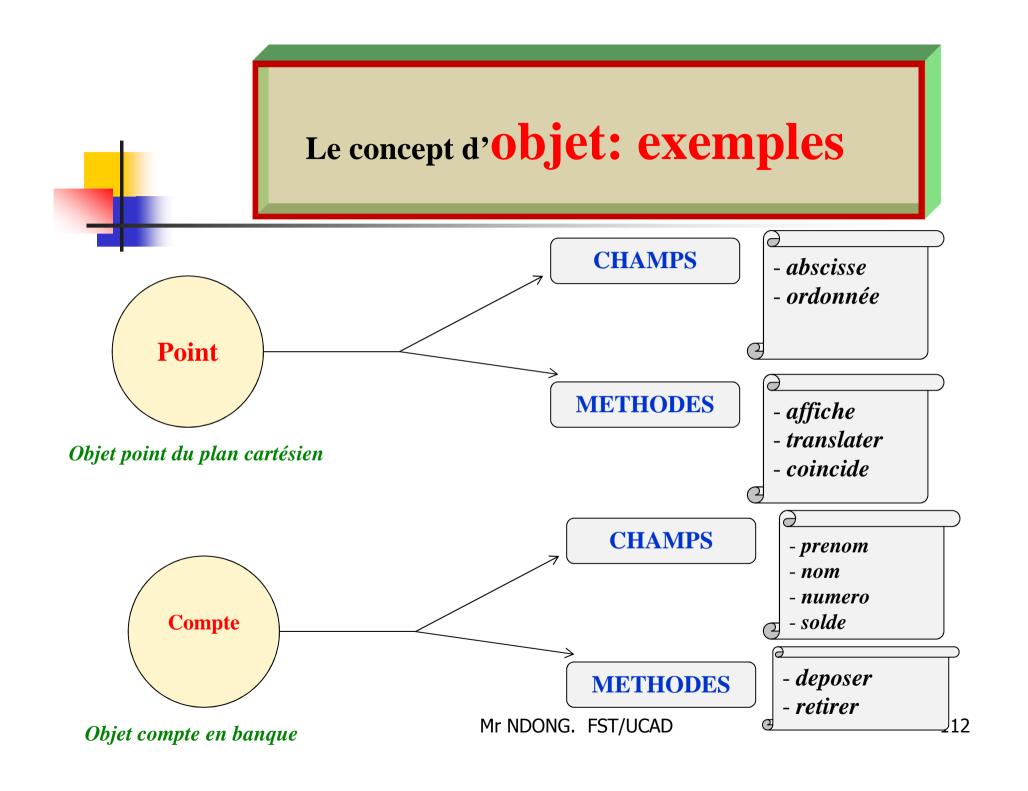
- objets, classes;
- construction des objets : notion de constructeur;
- champs d'instance, méthodes d'instance;
- autoréférence: this
- champs et méthodes de classe;
- affectation d'objets;
- transfert d'informations avec les méthodes;
- notion d'objet membre;
- référence null;
- comparaison d'objets;
- surdéfinition (surcharge) de fonctions;
- surdéfinition de constructeurs;
- appel de constructeur dans un autre constructeur de la même classe: this (...)
- classes internes;



<u>Un objet</u> peut être défini comme étant une entité concrète ou abstraite du monde réel. Les objets contiennent des attributs et des méthodes. Chaque objet sera caractérisé par son jeu de données(on parlera d' attributs ou aussi de champs). Les fonctions qui agissent sur les données de l'objet sont aussi appelées méthodes.



(Opérations, mouvements, manipulations, fonctions)



Objet = identité + état + comportement

Chaque objet possède:

- une identité qui lui est propre :

Même si deux personnes ont des noms identiques, elles désignent deux individus distincts. Elles ont une identité distincte.

- un état : les informations qui lui sont propres

Il est défini par les valeurs associées à chacun des champs de la classe. Pour une personne : **nom ,prénom, age, sexe, race**

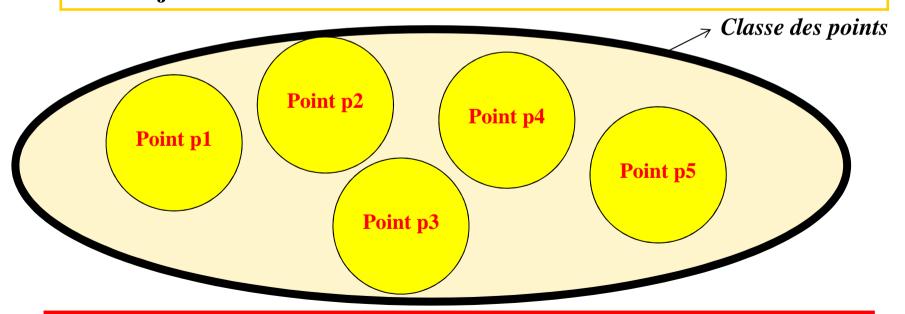
- un comportement : les méthodes applicables à l'objet

Pour une personne : respirer, marcher, mourir



<u>Une classe</u> est la description d'un ensemble de <u>données</u> et de <u>fonctions</u> regroupées au sein d'une même entité (appelée objet).

On peut définir une classe comme étant aussi une description abstraite d'un objet du monde réel.





Définition d'une classe

```
package test.point;
public class Point
{
    private int x ; // champ x d'un objet Point
    private int y ; // champ y d'un objet Point
    public Point (int abs, int ord)
    { x = abs ;
        y = ord ;
    }
} // fin de la classe
```

private: Les champs x et y ne sont visibles q'à l'intérieur de la classe et non à l'extérieur : principe de l'encapsulation des données. Les données ne seront accessibles que par l'intermédiaire de méthodes prévues à cet effet (accesseurs).

Permet d'attribuer des valeurs initiales aux champs de l'objet. Cette méthode est ce qu'on appelle un constructeur.



Remarques

Une méthode peut être déclarée private : dans ce cas elle n'est visible qu'à l'intérieur de la classe où elle est définie. Pour pouvoir l'utiliser dans un autre programme, il faut nécessairement passer par une méthode publique de sa classe ayant l'appelée.

Il est fortement déconseillé de déclarer des champs avec l'attribut public, cela nuit à l'encapsulation des données.

Créer un objet = instancier une classe

```
int a = 10; // réservation de l'emplacement mémoire pour une variable de type int
float x; // réservation de l'emplacement mémoire pour une variable de type float
Point a ; // cette déclaration ne réserve pas d'emplacement pour un objet de type Point
           // mais simplement une référence à un objet de type Point.
    La création d'un objet (on parle d'instanciation ) se fait avec l'opérateur new:
     on pourra écrire :
     a= new Point (1, 2); // crée un emplacement mémoire pour un objet de
                                 //type Point et on met sa référence dans la variable a
 référence
                                                                                objet
 a
                                                                                Point
```

Utilisation d'une classe



Lorsqu'une classe est définie, il ne reste qu'à l'utiliser, en instanciant des objets et en appelant les méthodes décrivant les fonctionnalités. Mais attention, l'accès DIRECT aux champs encapsulés est impossible en dehors de leur classe.

Lorsque vous définissez une classe, vous devez aussi y définir toutes les méthodes utiles. En particulier, songez à définir les ACCESSEURS ou MUTATEURS. Il s'agit de méthodes (que l'on nomme aussi getter et setter) servant à accéder et à modifier individuellement chaque champ de l'objet.

ACCESSEURS: getter et setter



Voici les accesseurs à insérer dans notre classe Point précédente.

```
public class Point
   //code manquant
public int getX()
   { return x;
public int getY()
   { return ;
public void setX (int abs )
   \{ x=abs; 
public void setY (int ord )
   { y=ord;
// fin de la classe Point
```

Remarque importante: Java a introduit une convention importante sur le nommage des getter et setter qui stipule que leur nom doit commencer par get (pour un getter) et set (pour un setter) suivi du nom du champ et que l'initiale du nom du champ doit commencer par une lettre MAJUSCULE.

Autres fonctionnalités

A part les accesseurs, les autres fonctionnalités de la classe doivent y être insérées sous forme de méthodes d'instances ou méthodes de classe. Voici par exemple, une méthode pour déplacer un point et une méthode pour affiche à la fois les coordonnées d'un point.

```
class Point
{
    // code manquant

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

public void affiche()
    { out.println("Point de coordonnées" + x + " et " +y);
    }
} //fin de la classe Point
```

Pour utiliser ces méthodes, il suffit de disposer d'une instance de la classe:

```
Point z = new Point (23,-12)

Et de faire: z.deplace (-3,-4); // on dit que l'instance z accède à SA méthode deplace z.affiche ();
```





En Java, la création d'objet se fait par allocation dynamique grâce à l'opérateur new qui appelle une méthode particulière : le constructeur.

Dans l'exemple précèdent, il n'y avait pas de constructeur mais plutôt c'était la méthode *initialise* qui se chargeait d'initialiser correctement les champs d'un objet de type Point. La démarche proposée suppose que l'utilisateur fera appel de cette méthode au moment opportun pour initialiser correctement un objet.

En fait un constructeur permet d'automatiser l'initialisation d'un objet.

Un constructeur est une *méthode* qui porte le même nom que le nom de la classe et qui est sans valeur de retour. Il peut disposer d'un nombre quelconque d'arguments.

Le constructeur par défaut: pseudo-constructeur

En Java, vous n'êtes pas obligé de créer effectivement un constructeur explicite lors de la définition d'une classe. Dans ce cas, Java vous fournit un constructeur par défaut appelé pseudo-constructeur.

Il s'agit d'un constructeur sans paramètre ne faisant aucun traitant. Il sert à créer des objets avec une initialisation par défaut des champs aux valeurs « nulles » par défaut.

Si on a:

```
class Point
{
}
```

Cela correspond à:

```
class Point
{ public Point(){ }
}
```

Et on peut toujours écrire:

Point a = new **Point** ();

Mais ATTENTION:

Si vous créez <u>explicitement</u> un constructeur dans votre classe, le pseudo-constructeur n'existe plus.



Quelques règles sur les constructeurs

Une classe peut disposer de plusieurs constructeurs: ils se différencieront par le nombre et le type de leurs arguments.

Une classe peut disposer d'un constructeur sans arguments qui est bien différent du pseudo-constructeur.

Un constructeur peut appeler un autre constructeur de la même classe (A VOIR).

Un constructeur peut être déclaré public ou privé.



Construction d'un objet

La construction et l'initialisation des champs d'un objet se font en 3 étapes :

- l'initialisation par défaut de tous les champs à une valeur "nulle",
- l'initialisation explicite lors de la déclaration du champ, —
- l'exécution des instructions du constructeur. —

```
public class TestInit
{ private int nombre ;
    private int diviseur = 12 ;
    public TestInit() { nombre = 24 ;}
    public float diviser()
    { return (float) nombre / diviseur ;}
}
Résultat: 2.0
```





Type du champ	Valeur par défaut
booléen	false
char	caractère de code nul
entier(byte, short, int, long)	0
flottant(float, double)	0.f ou 0
objet	null

Initialisation par défaut des champs d'un objet

Autoréférence : this (1/3)

Le mot clé **this** joue deux rôles : 1) le rôle d'OBJET COURANT et 2) l'appel d'un constructeur dans un autre constructeur de la même classe. Regardons le premier rôle.

Lorsque vous exécutez un programme, la machine virtuelle a besoin de connaître effectivement l'objet sur lequel porte une opération particulière, par exemple l'appel d'une fonction. Ceci est nécessaire car généralement vous créer beaucoup d'objets qui résident en mémoire, et en un instant donné, un seul de ces objets est sollicité par la JVM.

L'objet sollicité par la JVM en instant donné est appelé objet courant et il est désigné par **this**. Lors de la définition de classe, ce this est souvent présent dans le programme de manière implicite devant le nom des champs et dans les méthodes d'instances.

Mais vous avez le libre choix de l'expliciter dans le programme.

Autoréférence : this (2/3)



L'utilisation de this est très pratique dans l'écriture des méthodes et surtout des constructeurs.

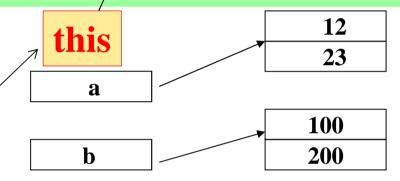
```
public class Point {
    //code manquant
public Point ( int abs, int ord ) {
    x = abs;
    y = ord;
    }
    public void affiche()
    { out.println("Point de coordonnées" + x + " et " +y);
    }
}
```



Autoréférence : this (3/3)



Si on a deux objets a et b de type Point: Point a = new Point (12, 23); Point b = new Point (100, 200);



Dans un appel de la forme a.affiche(), la méthode reçoit une information lui permettant d'identifier l'objet ayant fait l'appel. Si l'information est transmise, la méthode pourra agir sur les champs spécifiques de l'objet.

Cette transmission est gérée automatiquement par le compilateur.

Mais on peut vouloir agir sur l'objet globalement au lieu d'agir sur les champs.

Dans ce cas Java utilise le mot clé this.

Autoréférence : this : remarque

Le mot clé this peut être utilisé pour simplifier l'écriture du constructeur. En clair, on peut utiliser les noms des champs identiques aux noms des arguments.

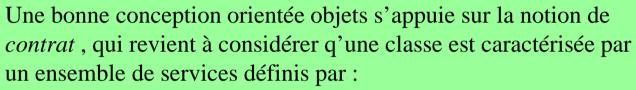
```
public class Point {
private int X ; // champ x d'un objet Point
private int Y ; // champ y d'un objet Point

public Point ( int X, int Y) {
    this.x = x;
    this.y = y;
}
Le this est obligatoire ici.
```

Champ déclaré avec l'attribut final

```
public class ChampFinal
{ private final int NOMBRE ;// initialisation différée
 private final float MAX; // initialisation différée
 private final int DIVISEUR = 12 ;// valeur fixée à la déclaration
 public ChampFinal( int nbre)
   { NOMBRE = nbre ;// la valeur de NOMBRE dépendra de celle de nbre
     MAX = 20;
                       // la valeur de MAX est fixée à 20 une seule fois.
public float diviser( )
    return (float) NOMBRE / DIVISEUR;
       <u>ATTENTION</u>: chaque objet possédera son propre champ NOMBRE,
       malgré que ce dernier est déclaré final.
```





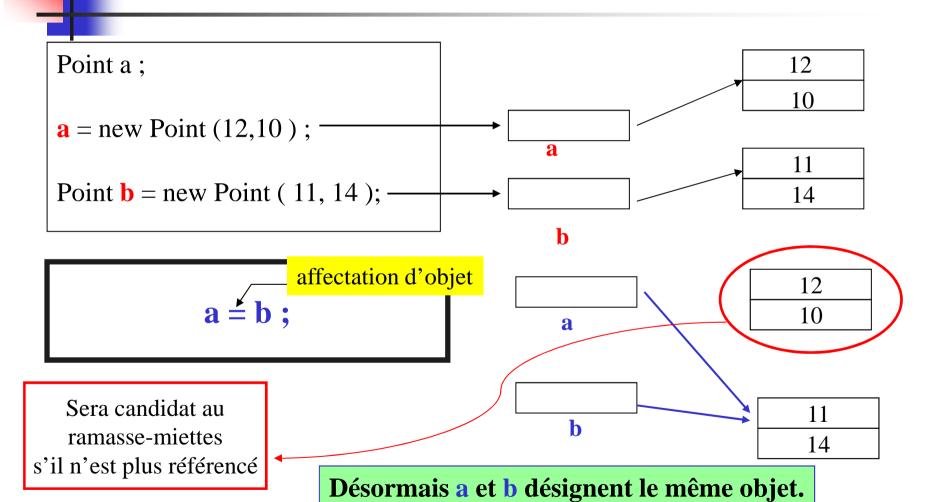
- -- les en-têtes de ses méthodes publiques,
- -- le comportement de ses méthodes.

Le reste, c'est-à-dire les champs et les méthodes privés ainsi que le corps des méthodes publiques, n'a pas à être connu de l'utilisateur. Il constitue ce que l'on appelle souvent l'implémentation de la classe.

En quelque sorte, le contrat définit ce que fait la classe tandis que l'implémentation précise comment elle le fait .

Le grand mérite de l'encapsulation des données est de permettre au concepteur d'une classe d'en modifier l'implémentation sans que l'utilisateur n'ait à modifier les programmes qui l'exploitent.

Affectation d'objets



Mr NDONG. FST/UCAD

Référence nulle: le mot clé null

```
class Point03
  private int x ; // champ x d'un objet Point
  private int y ; // champ y d'un objet Point
  public Point03 (int abs, int ord) // un constructeur à deux arguments
   \{ x = abs ; 
                                                 Les variables locales doivent
    y = ord;
                                                  toujours être initialisées avant toute
                                                  utilisation.
 public Point03 coincide (Point03 p )
   { Point03 t = null ; // t est locale donc il est nécessaire de l'initialiser
    if ((p.x = = this.x) & (p.y = = this.y))
                                                     t = this;
                                                     t = null;
    else
   return t;
 //fin de la classe Point03
```

Comparaison d'objets (1/2):

= = versus equals

```
public class Point03
{ private int x; // champ x d'un objet Point
  private int y; // champ y d'un objet Point
  public Point03 (int abs, int ord) // un constructeur à deux arguments
   \{ \mathbf{x} = \mathbf{abs} :
    \mathbf{v} = \mathbf{ord}:
public static void main(String args [ ] )
   Point03 a = new Point03 (1,1);
   Point03 b = new Point03 (1,1);
   System.out.println ("avec = =: + a == b);
   System.out.println ("avec equals :" + a.equals (b));
                                 Résultat
                                                         avec = = : false
} //fin de la classe Point03
                                                         avec equals : false
```



Comparaison d'objets (2/2)

= = teste s'il s'agit du même objet (pas d'une copie). equals teste l'égalité de contenu de deux objets .

ATTENTION:

dans l'exemple précédent la méthode equals dont il s'agit est celle de la classe Object (la super classe de toutes les classes en Java). Souvent, vous serez emmené à *redéfinir* cette méthode. Elle a pour en-tête:

public boolean equals (Object o)

L'opérateur != s'applique également à des références d'objet pour tester la différence.

Champs et méthodes de classe: le mot clé static.

Champs de classe

Les champs de classe ou champs statiques existent en un seul exemplaire pour toutes les instances de la classe. On les déclare avec le mot clé static.

```
public class ChampStatic
{
  int n ;
  static int k ;
}
```

```
ChampStatic a = new ChampStatic();
ChampStatic b = new ChampStatic();

a.n — b.n
a.k — b.k
```

a.k et b.k peuvent être remplacés par ChampStatic.k . Mais si k est <u>privé</u>, on ne peut faire ceci.

Exemple d'utilisation de champs de classe

```
public class ChampStatic {
 private static int nombreInstanceCrees; // champ static pour stocker le nombre
 public ChampStatic ( )
                                          // d'objets créés
      nombreInstanceCrees++;// on incrémente de 1 à chaque création d'un objet
 public void affiche()
      System.out.println ("nombre d'objets créés :" + nombreInstanceCrees );
 public static void main (String args [ ])
       ChampStatic a = new ChampStatic ();
       a.affiche();
       ChampStatic b = new ChampStatic ( );
        b.affiche();
                                       nombre d'objets créés : 1
                                       nombre d'objets créés : 2
```



Méthodes de classe

Une méthode d'une classe ayant un rôle indépendant de toute instance de la classe doit être déclarée avec le mot clé static et elle ne pourra être appliquée à aucun objet de cette classe, contrairement aux méthodes d'instances.

L'appel de la méthode ne nécessitera que le nom que de la classe.



Une méthode statique ne peut pas agir sur des champs usuels, c'est-à-dire non statiques.

Exemple d'utilisation de méthodes de classe

```
public class MethodeStatic {
 private long n;
 private static long nombreInstanceCrees; // champ static pour stocker le nombre
 public MethodeStatic( long k)
                                                 // d'objets créés
     nombreInstanceCrees++;
    n = k;
 public void affiche()
     System.out.println ("nombre d'objets créés : " + nombreObjet( ) );
 public static long nombreObjet()
                nombreInstanceCrees;
       return
```

Bloc d'initialisation statique



l'initialisation d'un champ statique se limite uniquement à :

- l'initialisation par défaut,
- l'initialisation explicite éventuelle.

Les blocs statiques sont souvent utilisés pour initialiser des variables complexes dont l'initialisation ne peut être faite par une simple instruction.

Les instructions n'ont accès qu'aux champs statiques de la classe.

Les instructions d'un bloc statique sont exécutées de façon automatique et une seule fois lorsque la classe est chargée.

Exemple d'utilisation de bloc statique

```
public class BlocStatic {
 private double solde;
 private static int [] tab;
 static { tab = new int[10]; // bloc d'initialisation délimité par des accolades
         for ( int i = 0; i < tab.length; i++)
         tab[i] = i + 1;
       } // fin du bloc static
public BlocStatic (double solde) {
                                     this.solde = solde;
 public static void main(String[] args) {
           BlocStatic a = new BlocStatic(12000); for (int i=0;i < tab.length; i++)
           System.out.print (tab[i]+" ");
 }// fin de la classe
                                              1 2 3 4 5 6 7 8 9 10
```



Surdéfinition de méthodes

La surdéfinition de méthodes signifie qu'un même nom de méthode peut être utilisé plusieurs fois dans <u>une même classe</u>. Dans ce cas, le nombre et/ou le type des arguments doit nécessairement changé.

On peut parler indifféremment de surdéfinition, surcharge ou overloading (en Anglais).

Exemple de surdéfinition de méthode

```
public class ExempleSurdefinition {
 private int x;
 private int y;
 public ExempleSurdefinition (int abs, int ord ) { x=abs; y=ord;
 public void deplace (int dx, int dy) { x += dx ; y += dy ;
 public void deplace (int dx) { x += dx;
 public void affiche(){ System.out.println(" Point de coordonnees :"+ x+ " "+y);}
 public static void main(String[] args) {
 ExempleSurdefinition ex = \frac{\text{new}}{\text{ExempleSurdefinition}} ExempleSurdefinition(10,10);
 ex.deplace (10);// appel de deplace (int)
 ex.affiche();
 ex.deplace(10, 10);// appel de deplace (int, int)
 ex.affiche();
                                 Point de coordonnes : 20 10
                                 Point de coordonnes : 30 20
```



Il peut y avoir des cas d'ambiguïté :

Surdéfinition de constructeurs

Les constructeurs peuvent être surdéfinis comme toute autre méthode.

```
public class Individu {
 private String nom;
 private String prenom;
private Compte c;
 /* constructeur à deux arguments*/
 public Individu ( String lenom, String leprenom ) {
  nom = lenom;
  prenom = leprenom;
 /* constructeur à trois arguments */
 public Individu (String lenom, String leprenom, Compte cp)
 nom = lenom;
 prenom = leprenom;
 c = cp;  }
                                   Mr NIDONG FST/LICAD
```

Attribut de type objet. Il doit exister obligatoirement une classe **Compte**.



Transfert d'informations avec les méthodes

En Java, la transmission d'informations se fait toujours par valeur.

La transmission, par exemple d'un argument à une méthode ou d'une valeur de retour se fait toujours par valeur.

A RETENIR

Cette transmission a des conséquences totalement différentes, selon que l'on manipule des types primitifs ou bien des types objet (c-à-d des références).

RAPPEL

Transmission par valeur: la méthode reçoit une copie de la valeur de *l'argument effectif* sur laquelle elle travaille. Pas de possibilité d'agir sur la valeur de l'argument effectif.

Transmission par référence (par adresse): la méthode reçoit l'adresse de l'argument effectif sur laquelle elle travaille directement. Ici, possibilité de modifier donc la valeur de l'argument effectif.

Illustration de la transmission de messages avec les méthodes (1/2)

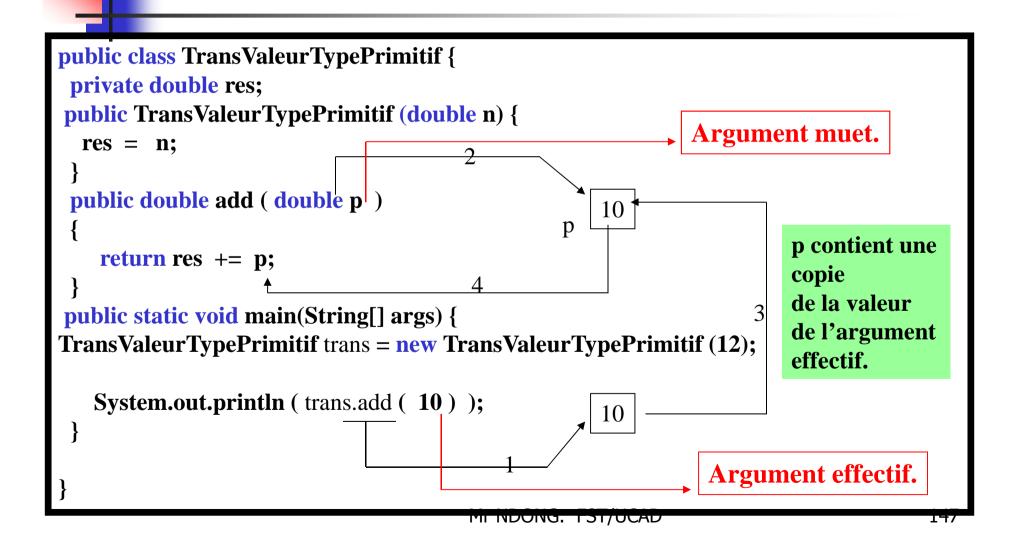
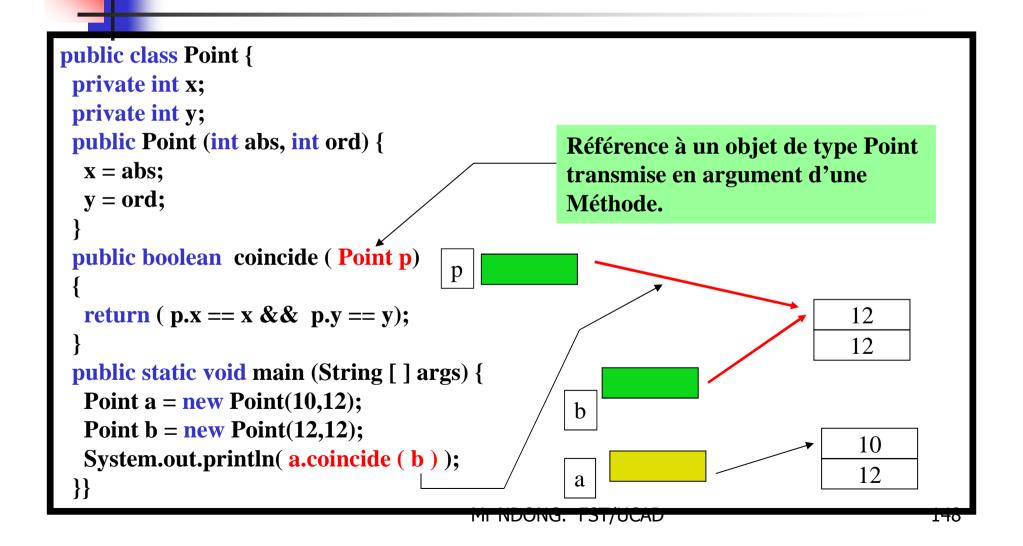


Illustration de la transmission de messages avec les méthodes (2/2)

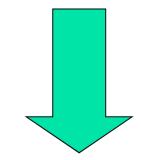


Appel d'un constructeur dans un autre constructeur. (1/2)

Un constructeur peut appeler un autre constructeur de la même classe en utilisant le mot clé this. L'objectif majeur est la simplification du code et aussi pour des problèmes de sécurité.

```
public class Individu {
 private String nom;
 private String prenom;
 private Compte c;
 public Individu ( String lenom, String leprenom ) {
  nom = lenom;
  prenom = leprenom;
 public Individu (String lenom, String leprenom, Compte c1)
 nom = lenom;
 prenom = leprenom;
 c = c1:
```

Cette classe peut être écrite de façon plus sophistiquée comme suit ...



Appel d'un constructeur dans un autre constructeur. (2/2)

```
public class Individu2 {
   private String nom;
   private Compte c;
   public Individu2 ( String nom, String prenom ) {
      this.nom = nom;
      this.prenom = prenom;
   }
   public Individu2 (String nom, String prenom, Compte c) {
      // appel du constructeurs a deux arguments
      this (nom, prenom);
      this. c = c;
}
```

L'appel this (...) doit nécessairement être la <u>première instruction</u> du constructeur appelant.



Objet membre

Objet membre = référence à un objet

```
public class Point {
                                                public class Cercle {
 private int x;
                                                 private double r; //rayon du cercle
 private int y;
                                                 private Point p;// objet membre
 public Point (int abs, int ord) {
                                                 public Cercle (double r, int x, int y) {
                                                   this.r = r;
  x = abs:
                                                  p = new Point (x, y);
  \mathbf{v} = \mathbf{ord};
 public void affiche()
                                                 public void affiche()
                                                 {System.out.println("Cercle de rayon:"+r);
System.out.println(" Point :" +x " " +y);
                                                   System.out.print(" et de centre:" );
                                                   p.affiche());
                                               DONG. FST/UCAD
                                                                                           151
```



Les classes internes

Une classe est dite interne lorsque que sa définition est située à l'intérieur de la

définition d'une autre classe. Les classes internes (inner classes) peuvent être situées à différent niveau d'une classe normale.

Il existe quatre types de classes imbriquées :

- les classes internes simples, définies au niveau des classes,
- les classes internes statiques, représentant une classe de sommet intérieure,
- les classes locales, définies au niveau des méthodes,
- les classes internes anonymes, définies au niveau d'une instance.

Les classes internes sont particulièrement utiles pour :

- permettre de définir une classe à l'endroit ou une seule autre en a besoin
- définir des classes de type adapter (essentiellement à partir du JDK 1.1 pour traiter des évenements émis par les interfaces graphiques)
- définir des méthodes de type callback d'une façon générale.

Classes internes simples (1/5)

```
package essai01;
public class ClasseParente {
 private int x = 10, static int y = 20;
  public int addition () { return (x + y);
  public class ClasseInterne
                            //DEBUT CLASSE INTERNE
  { static int p = 20; //erreur de compilation,
   static final int k = 12; //constante statique
   public int multiplier ( )
    { return x*y + addition (); }
      //FIN CLASSE INTERNE
 public static void main (String [ ] args) {
 ClasseParente ob out = new ClasseParente ();
 //ClasseInterne ob_in0 = new ClasseInterne (); IMPOSSIBLE
 ClasseInterne ob_in = ob_out.new ClasseInterne ();
 System.out.println (ob_in.multiplier ( ));
// System.out.println (ob_out.multiplier ( ));//ERREUR
// System.out.println (ob_in.addition ( )); IMPOSSIBLE
 } }
                                  Mr NDONG. FST/UCAD
```



Quelques remarques importantes:

Une classe interne peut être déclarée avec n'importe quel modificateur d'accès ((
(public, protected, par défaut ou private) et les modificateurs abstract, final, static.

Elles sont membres à part entière de la classe qui les englobe et peuvent accéder à tous les membres de cette dernière.

Les classes internes ne peuvent **pas être déclarées à l'intérieur d'initialiseurs statiques** (blocs statiques).

Les classes internes ne doivent pas déclarer de membres statiques, sauf s'ils comportent le modificateur *final*, dans le cas contraire, une erreur de compilation se produit. Toutefois, les membres statiques de la classe externe peuvent être hérités sans problème par la classe interne.

Les classes imbriquées sont **capables d'accéder à toutes les variables et méthodes de la classe parente**, y compris celles déclarées avec un modificateur *private*.

154



Classes internes simples (3/5)

On retient:

La notation particulière:

ClasseInterne ob_in = ob_out.new ClasseInterne();

spécifie que l'objet créé est une instance de la classe interne associée à l'objet résultant de l'instanciation d'une classe de plus haut niveau.

L'instanciation de la classe interne passe obligatoirement par une instance préalable de la classe d'inclusion.

La classe parente est d'abord instanciée, puis c'est au tour de la classe interne de l'être par l'intermédiaire de l'objet résultant de la première instance.

Classes internes simples (4/5)

Il est possible d'utiliser une méthode de la classe parente pour créer directement une instance de la classe interne. Toutefois, lors de l'appel de la méthode, il sera nécessaire de créer une instance de la classe d'inclusion.

```
package essai0;
public class ClasseParente02 {
 private int x = 10, int y = 20;
 public int addition ( )
  { ClasseInterne02 obj_in = new ClasseInterne02();
   return (x + y) + obj_i .multiplier ();
  public class ClasseInterne02
  { public int multiplier ( )
    { return x*y;
                                  Mr NDONG. FST/UCAD
```

Classes internes simples (5/5)

```
class ClasseParente03 {
 private int X = 10, int Y = 20;
 public int addition( )
  { ClasseInterne03 obj_in= new ClasseInterne03 (10,10);
   return (x + y)+ obj_in.multiplier();
  public class ClasseInterne03
  { private int X = 12; private int Y = 14;
  public ClasseInterne03 (int X, int Y)
   \{ this.X = X + ClasseParente03.this.X; \}
    this.y = y + ClasseParente03.this.y;
   public int multiplier( )
    { return x*y;
      }}
```

Parfois, il peut être nécessaire de distinguer les variables situées dans les classes interne et externe.

Classes internes statiques

Elles sont membres à part entière de la classe qui les englobent et peuvent accéder uniquement aux membres statiques de cette dernière.

```
public class Parente04 {
     private static int x = 1, y = 2;
     private int z = 3;
     public int addition () { return x + y + z;}
     public static int multiplier(){ return x*y;}
                class Interne04{
 public
         static
  private static int k = 1;
  private int p = 2;
 public void diviser ( )
 { System.out.println (new Parente04( ).addition ( )/p+x+y);}
 public static void imprimer(){
 System.out.println ( multiplier ( ) /x+y+k ); }
 public static void main(String [ ] args) {
 Parente04.Interne04().imprimer();
 new Parente04.Interne04( ).diviser ( )}
                                         Mr NDONG. FST/UCAD
```

Les classes internes statiques peuvent accéder à l'ensemble des membres statiques de leur classe parente, à l'instar des méthodes de classe.



Classes locales

Une classe locale est définie à l'intérieur d'une méthode ou un bloc, et agît librement et essentiellement au sein de cette dernière.

Elles peuvent être static ou non.

Il n'est possible de déclarer des classes locales, dont la portée est limitée au bloc, qu'avec les modificateurs *final* ou *abstract*. Les modificateurs suivants : *public*, *protected*, *private* et *static*, sont interdits.

Les données membres d'une classe externe peuvent être accédés par la classe locale.

Seules les <u>variables locales</u> et les <u>paramètres de la méthode d'inclusion</u>, déclarées avec le modificateur *final*, peuvent être exploitées par les classes internes locales, sinon une erreur se produit lors de la compilation. De plus, ces variables doivent être impérativement initialisées avant leur emploi dans la classe locale.



```
public class ClasseExterne {
private int x,y; private static int z;
public void addition( int p){
   final int k = 9;
   int u = 121;// inutilisable dans ClasseLocale
   class ClasseLocale {
    boolean verif ( )
     {if (x+y+k==z) return true;
                return false;
      else
   }// fin bloc de méthode
```

p et u ne peuvent pas
être utilisés dans ClassLocale.
Par contre , k est déclarée *final* donc on peut l'utiliser.

REMARQUE:

Lorsqu'une classe est déclarée dans une méthode statique, alors les variables d'instances de la classe externe ne sont plus accessibles pour la classe imbriquée.

L'utilisation d'une classe locale ne dépend pas de l'instanciation d'une classe externe.



Classes anonymes

Les classes anonymes (anonymous classes) sont déclarées immédiatement après l'expression d'instanciation d'une classe, permettant directement d'étendre ou d'implémenter respectivement la classe ou l'interface instanciée. Elles sont définies et instanciées à la volée sans *posséder de nom*.

```
new Classe ([Liste d'arguments]) { // Instructions de la classe anonyme... };
new Interface ( ) { // Instructions de la classe anonyme... };
```

Les classes anonymes **obéissent aux mêmes restrictions que les <u>classes locales</u>** et de plus, ne peuvent **ni être abstraites** (*abstract*) **ni être statiques** (*static*).

Par contre, elles portent toujours implicitement le modificateur final.

En fait, aucun modificateur n'est permis dans une déclaration de classe anonyme

On verra l'utilité des classes anonymes en programmation évènementielle.

Gros plan sur les packages

Un package regroupe un ensemble de classes sous un même espace de nomage.

Les noms des packages suivent le schéma : name.subname...

Une classe Watch appartenant au package time.clock doit se trouver

obligatoirement dans le fichier time/clock/Watch.class.

Les packages permettent au compilateur et à la JVM de localiser les fichiers contenant les classes à charger.

L'instruction package indique à quel paquetage appartient la ou les classe (s) de l'unité de compilation (le fichier).

Les répertoires contenant les packages doivent être présents dans la variable d'environnement **CLASSPATH**.

En dehors du package, les noms des classes sont: packageName.className.

L'instruction **import packageName** permet d'utiliser des classes sans les préfixer par leur nom de package.

Les API sont organisées en package (java.lang, java.io, javax.swing,....)

Droits d'accès et paquetage

En Java, il y a quatre types de droits d'accès aux méthodes et aux champs d'un objet d'une classe. Autrement dit, la portée de la visibilité des méthodes et champs est assurée par les mots clés: **private**, **protected**, **vide** (**droit de paquetage**), **et public**.

Nous décrivons ici, la notion de droit d'accès parallèlement à la notion de paquetage. Nous séparons la visibilité des <u>champs</u> et celle des <u>méthodes</u>.

Paquetage et visibilité des champs (encapsulation des membres)

```
class c1{
                                                class c4 extends c1{
 public
             int a;
             int b;
                                                      a c
 protected int c;
 private
            int d;
                                                     package B
           package A
class c2 extends c1{
                         class c3 {
                                                class c5 {
       a b c
                          a b c
                                                            a
```

Exemple d'accès aux membres

```
ndong/classes/ graph/2D/ Circle.java
package graph.2D;
public class Circle {
.....
}
```

```
ndong/classes/ graph/3D/ Sphere.java
package graph.3D;
public class Sphere {
.....
}
```

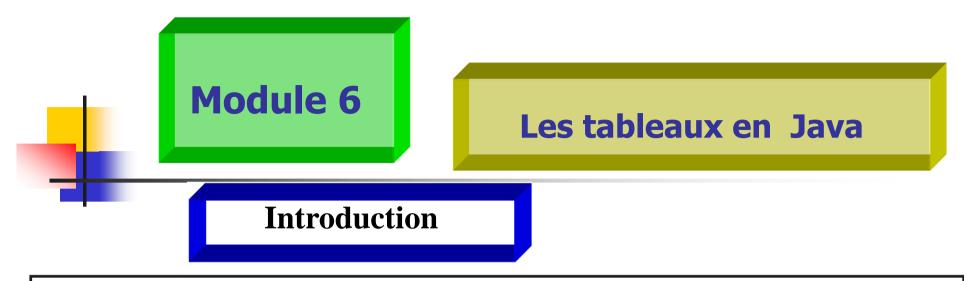
```
ndong/classes/ TestPackage/ MainClass.java
package testpackage; import graph.3D.Sphere, //nécessaire

import graph.2D.*;
public class MainClass {
public static void main (String args [ ]) {
    graph.2D.Circle c1= new graph.2D.Circle (50);
    Circle c2 = new Circle (80);
    graph.3D.Sphere s1 = new graph.3D.Sphere (100); //OK

Sphere s2 = new Sphere (50), // error: class testpackage.Sphere not found
}
```

Paquetage et visibilité des méthodes

```
class c1{
                                              class c4 extends c1{
 public void f();
                                                    f() h()
        void g();
 protected void h( );
 private
           void k();
                                                   package B
           package A
class c2 extends c1{
                        class c3 {
                                               class c5 {
                         f() g() h()
     f() g() h()
                                                          f()
```



Les tableaux sont des structures de données regroupant plusieurs valeurs de même type.

Ou encore on parle de tableaux pour désigner un ensemble d'éléments de même type désignés par un nom unique, chaque élément étant repéré par un indice précisant sa position au sein de l'ensemble .

Les tableaux constituent des **collections d'informations homogènes**, c'est-à-dire, de valeurs primitives ou d'objets de même type.

Les éléments d'un tableau peuvent être :

- des primitives (scalaires) (float, int, char, etc.),
- des références d'objets (String, Object),
- des références de tableaux.

La taille d'un tableau est fixée d'une façon permanente suite à la déclaration du tableau et à l'allocation de ressources systèmes pour ce dernier.



Introduction

La *taille* d'un tableau est donc **fixée lors de sa création** et ne peut plus être changée pendant toute la durée de sa vie.

Une solution est de créer un tableau d'une taille donnée, et, lorsque celui-ci est saturé, en créer un nouveau et déplacer toutes les références de l'ancien. Tableau dans le nouveau. C'est précisément ce que fait la classe **ArrayList** ou la classe **Vector**, qui seront étudiées plus loin dans ce cours.



Introduction

Les tableaux sont des objets:

Indépendamment du type de tableau qu'on utilise, un identifiant de tableau est en fait une référence sur un <u>vrai objet créé</u> dans le segment. C'est l'objet qui stocke les références sur les autres objets, et il peut être créé soit implicitement grâce à la syntaxe d'initialisation de tableau, soit explicitement avec une expression <u>new</u>. Une partie de l'objet tableau (en fait, la seule méthode ou champ auquel on peut accéder) est le membre en lecture seule <u>length</u> qui indique combien d'éléments peuvent être stockés dans l'objet. La syntaxe « [] » est le seul autre accès disponible pour les objets tableaux.

Déclaration et création de tableaux

Déclaration.

```
type identificateur [];// on peut déclarer un tableau comme ceci

type [] identificateur;// ou comme cela

Exemples:

int t []; // t est destiné à contenir la référence à un tableau d'entiers.

// on peut aussi écrire int [] t
Object [] obj; // obj est destiné à contenir la référence à un tableau d'objets
```

•

En fait la différence entre les deux formes de déclaration devient perceptible lorsque l'on déclare plusieurs identificateurs dans une même instruction. Ainsi :

int [] t1,t2; // t1 et t2 sont des références à des tableaux d'entiers

int t1 [], n, t2 []; // t1 et t2 sont de tableaux d'entiers, n est une variable entière

ATTENTION

La taille d'un tableau n'est spécifiée qu'à partir du moment de son utilisation dans le programme. Ainsi, la mémoire ne sera allouée que lorsque cela sera nécessaire.

C'est pourquoi, lors de la déclaration du tableau vous ne pouvez pas faire:

int t [12];// Erreur: pas de dimension à la déclaration.



Création.

On crée un tableau comme on crée un objet, c'est-à-dire en utilisant l'opérateur **new**. On précise à la fois le type des éléments, ainsi que leur nombre (dimension ou taille du tableau).

En d'autres termes la définition d'une référence d'un tableau, c'est-à-dire la spécification de la taille du tableau référencé par la variable, s'accomplit comme ceci:

identificateur = new *type* [*taille*]; //le système alloue un emplacement //mémoire pour un tableau de *taille* éléments de type *type* .

Exemples:

t = new int [10]; // la variable t fait référence à un tableau de 10 // valeurs entières bien initialisées.

La <u>déclaration</u> peut se combiner à la <u>définition</u> du tableau produisant une instruction plus compacte.

Dimension obligatoire

String [] tabcar = new String [14];

Dans la création d'un tableau, il faut obligatoirement mentionner la taille du tableau.

Par défaut, les valeurs de chaque élément d'un tableau sont égales à :

- 0 pour des entiers (int, short, ...),
- 0.0 pour des nombres à virgule flottante (double, float),
- u0000 pour des caractères (char),
- false pour des booléens (boolean),
- null pour des objets (Object, String).

Remarque



On peut aussi créer un tableau en associant l'opérateur new et un initialiseur.

Voici un exemple:

```
int tab [] = new int []{1,2,3};
String t[] = new String[]{"java","langage objet"};
```





On ne peut pas créer un tableau avec une taille négative.

Une instruction telle que:

Point p [] = **new Point** [-5];

déclenche une exception java.lang.NegativeArraySizeException, laquelle, si elle n'est pas interceptée et traitée provoque l'arrêt brutal du programme (on verra comment traiter les exceptions).

De même, on ne peut accéder à un indice de tableau trop grand (ie accès en dehors des bornes ou limites du tableau).

Avec: int tab[] = new int [10]; l'instruction tab[10] = 12; déclenche une exception java.lang.ArrayIndexOutOfBoundsException .(en fait les indices des éléments d'un tableau varient de 0 à taille -1).

Création avec un initialiseur

Les tableaux peuvent être initialisés par l'intermédiaire d'une liste de valeurs séparées par une virgule et compris entre des accolades .
type [] identificateur = { valeur1, ..., valeurN };

type identificateur [] = { valeur1, ..., valeurN };

Exemples:
int [] notes = $\{10, 9, 12, 14, 16, 15, 17, 20, 19, 18\}$;
int notes [] = $\{10, 9, 12, 14, 16, 15, 17, 20, 19, 18\}$;
L'utilisation d'un initialiseur n'est utilisable que dans une déclaration.
int [] notes;

notes = $\{10, 9, 12, 14, 16, 15, 17, 20, 19, 18\}$; //interdit





En Java, on peut utiliser un tableau de deux façons différentes :

- en accédant individuellement à chacun de ses éléments
- en accédant globalement à l'ensemble du tableau.

L'accès individuel aux éléments d'un tableau est réalisé en utilisant ses indices, soit les numéros de chacun de ses éléments, en sachant que le premier commence à l'indice 0. Exemples:

```
String s[] = new String [10];

s[2] = new String ("Bonjour");//place la chaîne "Bonjour" dans le

// 3eme élément du tableau

double d [] = new double [6];

d[5]++; // incrémente de 1 le dernier élément du tableau
```

Utilisation d'un tableau

La manipulation globale d'un tableau se fait par affectation de tableaux. Il est possible d'affecter une variable de type tableau à une autre variable, à condition qu'elles soient déclarées <u>avec le même</u> type de composantes.

```
int [] tab = new int [3];

for(int i = 0; i < 3; i++) tab [i] = i;

int [] t = new int [2];

for(int i = 0; i < 2; i++) t[i] = i+10;

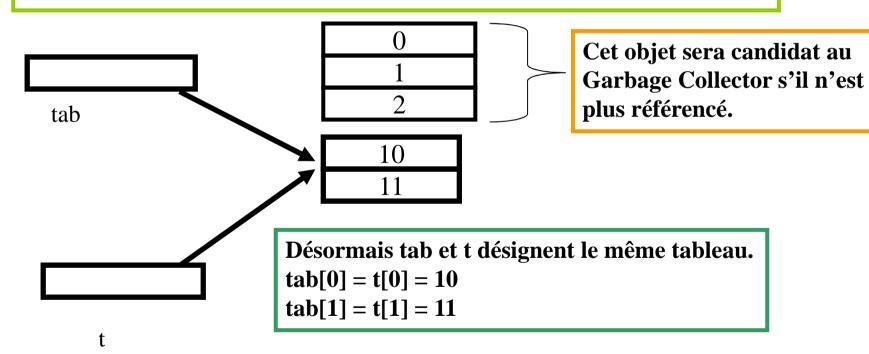
tab
```



Utilisation d'un tableau

Maintenant avec l'affectation: tab = t;

On se retrouve donc dans la situation suivante:





Utilisation d'un tableau

Remarque importante

```
public class Tab01 {
  public static void main (String[] args) {
    int t[] = new int [3];
    t[0]=1; t[1]=2; t[2]=3;
    int d[] = new int[3];
    d[0] =1; d[1] =2; d[2] =3;
    System.out.println (t.equals (d));
    System.out.println (t = = d);
    t = d;// t et d designent desormais le meme tableau
    System.out.println (t.equals(d));
    System.out.println (t.equals(d));
    System.out.println (t = = d);
    }
}
```

Même si deux tableaux contiennent les mêmes éléments (donc même contenu) et sont créés avec deux new identiques, il y deux espaces mémoires différents créés, donc ils ne désignent pas le même tableau..

Utilisation d'un tableau

Taille d'un tableau

On accède à la taille du tableau avec le mot clé lenght.

NB: il ne faut pas confondre cette variable avec la méthode lenght() de la classe String qui donne la longueur d'une chaîne de caractères.

float notes[] = new float [100];// notes.lenght vaut 100.

Pour parcourir le tableau on peut faire:

for(int i = 0; i < notes.lenght ;i++) //ou bien

for(int i = 0; i < 100; i++)



Tableau en argument ou en valeur de retour

Comprenez la transmission d'un tableau en argument ou en valeur de retour d'une méthode comme celle réalisée avec les objets.

Lorsqu'on transmet un nom de tableau en argument d'une méthode, on transmet en fait (une copie de)la référence au tableau .La méthode agit alors directement sur le tableau concerné et non sur une copie .

Exemple de tableau en argument et en retour

```
public class TabInverse {
/*méthode retournant sous forme de tableau
 l'inverse d'un tableau transmis en argument */
 public static int[ ] inverseTab (int t[ ])
 { int tabres[] = new int [t.length];
   for (int i = t.length - 1; i > = 0; i--)
      tabres [ t.length - i - 1] = t[i];
          return tabres;
/*méthode affichant les éléments du tableau renversé*/
 public static void afficheTab( int t[ ])
 { for ( int i = 0; i < t.length; i++)
  System.out.print ( t[ i ]+ " ");
                               Mr NDONG. FST/UCAD
```

Classe de teste pour l'exemple précédent

500 6 71 80 1



Tableau dynamique

Il est possible d'augmenter la taille d'un tableau une fois que celui-ci est plein et qu'on souhaite y ajouter des éléments.

On utilise pour cela un tableau tampon et la méthode statique **arraycopy** de la classe **System**. Voici un exemple:

On verra au chapitre sur les <u>Collections</u> qu'on aura plus besoin de recourir à cette méthode qu'il faut d'ailleurs utilisée avec précaution et modestie.

Algorithmes de Tri

L'intérêt d'un algorithme de tri est de trier les éléments d'un tableau selon un critère d'ordre donné.

Un exemple de critère est:

- par ordre croissant (pour les nombres),
- par ordre lexicographique (pour les chaînes).

Pour trier les valeurs d'un tableau, il va être nécessaire de permuter les valeurs contenues dans les différentes cases du tableau. Pour cela, une fonction de permutation, qui sera appelée "echanger", doit être écrite. Cette fonction prend en argument un tableau et deux entiers i et j. Elle récupère la valeur contenue dans la iième case du tableau, affecte à cette case la valeur contenue dans la jième case, puis affecte à la jième case l'ancienne valeur de la iième case.





Soit l'exemple suivant :

Considérons le tableau T suivant [1;4;3;2;5] dans lequel la permutation des valeurs contenues dans la première et la troisième case va être effectuée (la première case du tableau portant l'indice 0). Soit i = 1 et j = 3.

Etape 1 : Mémorisation de la valeur contenue dans la case i=1 : M=T(i)=T(1)=4

Etape 2 : Affectation à la iième case de la valeur de la jième case: T(i)=T(j)=T(3)=2.

Soit T=[1;2;3;2;5]

Etape 3 : Affectation à la jième case de la valeur contenue dans la mémoire M : T(j)=M.

Soit T = [1; 2; 3; 4; 5]

C'est ce qu'il fallait obtenir.



Code de la méthode echanger

```
\label{eq:code_control_control_control_control_control_control_control_control_control_control_control_control_control_control_control_control_control_control_control_control_control_control_control_control_control_control_control_control_control_control_control_control_control_control_control_control_control_control_control_control_control_control_control_control_control_control_control_control_control_control_control_control_control_control_control_control_control_control_control_control_control_control_control_control_control_control_control_control_control_control_control_control_control_control_control_control_control_control_control_control_control_control_control_control_control_control_control_control_control_control_control_control_control_control_control_control_control_control_control_control_control_control_control_control_control_control_control_control_control_control_control_control_control_control_control_control_control_control_control_control_control_control_control_control_control_control_control_control_control_control_control_control_control_control_control_control_control_control_control_control_control_control_control_control_control_control_control_control_control_control_control_control_control_control_control_control_control_control_control_control_control_control_control_control_control_control_control_control_control_control_control_control_control_control_control_control_control_control_control_control_control_control_control_control_control_control_control_control_control_control_control_control_control_control_control_control_control_control_control_control_control_control_control_control_control_control_control_control_control_control_control_control_control_control_control_control_control_control_control_control_control_control_control_control_control_control_control_control_control_control_control_control_control_control_control_control_control_control_control_control_control_control_control_control_control_control_control_control_control_control_control_control_control_contr
```

```
Code source "echanger"
private static void echanger(int tab[],int i ,int j)
{
   int tampon = tab [i];
   tab [i] = tab [j];
   tab [j] =tampon;
}
```



Algorithme de tri bulle

L'algorithme du tri bulle - ou bubble sort - consiste à regarder les différentes valeurs adjacentes d'un tableau, et à les permuter si le premier des deux éléments est supérieur au second. L'algorithme se déroule ainsi : les deux premiers éléments du tableau sont comparés, si le premier élément est supérieur au second, une permutation est effectuée. Ensuite, sont comparés et éventuellement permutés les valeurs 2 et 3, 3et 4 jusque (n-1) et n. Une fois cette étape achevée, il est certain que le dernier élément du tableau est le plus grand. L'algorithme reprend donc pour classer les (n-1) éléments qui précédent. L'algorithme se termine quand il n'y a plus de permutations possibles. Pour classer les n valeurs du tableau, il faut, au pire, effectuer l'algorithme n fois.

Exemple de tri bulle

Évolution du tableau au fil de l'algorithme (en vert, les éléments qui sont comparés, et éventuellement permutés, pour passer à la ligne suivante).

Premier parcours	5	3	1	2	6	4
•	3	5	1	2	6	4
	3	1	5	2	6	4
	3	1	2	5	6	4
	3	1	2	5	6	4
Deuxième parcours	3	1	2	5	4	6
	1	3	2	5	4	6
	1	2	3	5	4	6
	1	2	3	5	4	6
Tableau trié	1	2	3	4	5	6

L'algorithme se termine car il n'y a plus de permutations possibles. Ce fait sera constaté grâce à un dernier parcours du tableau ou aucune permutation n'a lieu.

Code source de tri bulle

```
tri_bulle (tableau T)
  debut
    entier longueur, i
    booleen inversion
    longueur<-taille(T)
  faire
       inversion=faux
       pour i=0 à (longueur-1)
         si T(i)>T(i+1)
            echanger (T,i,i+1)
            inversion<-vrai
         fin si
         longueur<-longueur-1
               fin pour
    tantque inversion
  fin
```

```
public static void triBulle( int tableau[])
     { int longueur = tableau.length;
       boolean inversion;
     do{
       inversion = false;
         for ( int i = 0; i < longueur-1; i++)
             if (tableau[i]>tableau[i+1])
             { echanger (tableau,i,i+1);
               inversion = true;
        longueur--;
     while (inversion);
    MI NOONG, 131/UCAD
```

Algorithme de tri par sélection

Présentation

Le tri par sélection est l'un des tris les plus instinctifs. Le principe est que pour classer n valeurs, il faut rechercher la plus grande valeur et la placer en fin de liste, puis la plus grande valeur dans les valeurs restantes et la placer en avant dernière position et ainsi de suite...

Considérons un tableau à n éléments. Pour effectuer le tri par sélection, il faut rechercher dans ce tableau la position du plus grand élément. Le plus grand élément est alors échangé avec le dernier élément du tableau. Ensuite, on réitère l'algorithme sur le tableau constitué par les (n-p) premiers éléments où p est le nombre de fois où l'algorithme a été itéré. L'algorithme se termine quand p = (n-1), c'est à dire quand il n'y a plus qu'une valeur à sélectionner ; celle ci est alors la plus petite valeur du tableau.

Algorithme de tri par sélection Exemple

Étapes de l'évolution du tableau au fil de l'algorithme. En vert, les valeurs déjà traitées.

5	3	1	2	6	4
5	3	1	2	4	6
4	3	1	2	5	6
2	3	1	4	5	6
2	1	3	4	5	6
1	2	3	4	5	6

Code source Tri par sélection (algorithmique)

```
tri_selection ( tableau T)
  debut
     entier longueur, maxi, i
     longueur<- taille(T)
 tantque (longueur>0) faire
       //recherche de la position du plus grand élément dans le tableau non encore trié
        maxi<-0;
       pour i = 1 à (longueur-1) faire
          \mathbf{si} \ \mathbf{T}(i) > \mathbf{T}(\max i) \ \mathbf{alors}
             maxi <-i
          fin si
        fin pour
//echange du plus grand élément avec le dernier
        echanger( T,maxi,longueur-1)
//traitement du reste du tableau
        longueur<-longueur-1
     fin tantque
                                             Mr NDONG. FST/UCAD
  fin
```

Code source Tri par sélection

(Java)

```
Landic static void triSelection (int tableau [])
 int longueur=(tableau.length-1);
 while (longueur>0)
    //on recupere la position du plus grand élément du tableau non encore trié
    int maxi = 0;
    for ( int i = 1; i < = longueur; i++)
      if (tableau[i]>tableau[maxi]) maxi = i;
    echanger (tableau, maxi, longueur); //on met le plus grand élément à la fin
    longueur--; //et on traite le reste du tableau !!!
```

Exemple de programme de tri par sélection

```
public class TabTrieSelection {
 public static void echange (int t [ ], int i, int j)
 \{ int res = t [j] ; \}
      t[j] = t[i];
      t[i] = res;
 public static int indiceMIN (int t[],int imin, int imax)
 { int res = imin;
   for ( int i = imin + 1; i \le imax; i++)
    if (t[i]< t[res]) res=i;
    return res;
 public static void tabTrie ( int t [ ])
 { for (int i=1;i< t.length ;i++)
         int dep = i-1;
              int j = indiceMIN (t,dep,t.length-1):
        echange(t,dep ,j);
                                            Mr NDONG. FST/UCAD
```

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

}
  public static void affiche(int s[])
{ for (int i=0;i< s.length ;i++)
System.out.println(s[i]+" ");
  }
} // fin de TabTrieSelection</pre>
```

ATTENTION: ici on récupère la position du plus petit élément, contrairement à ce qui est fait dans l'algorithme donné pour le tri par sélection.

Tableau à plusieurs indices

Introduction

Les tableaux vus jusqu'ici sont des *tableaux à une dimension* : conceptuellement tous les éléments se trouvent dans une seule ligne (ou colonne).

Les tableaux à plusieurs dimensions sont utiles dans la modélisation des données, mais ce sont les tableaux à deux dimensions qui sont de loin les plus utilises en informatique. Nous concentrons notre étude à leur cas.

Un *tableau `a deux dimensions*, ou *matrice*, représente un rectangle compose de lignes et de colonnes. Chaque élément stocké dans le tableau est adressé par sa position, donnée par sa ligne et sa colonne.

En Java, si tab est un tableau `a deux dimensions, l'élément de ligne i et colonne j est désigne par tab[i][j].

Déclaration

Pour déclarer un tableau à deux dimensions, on peut utiliser l'une de ces trois déclarations qui sont équivalentes :

```
int t[][]; // tableau d'entiers à deux dimensions
```

int [] t [] ; //idem
int [] []t ; //idem

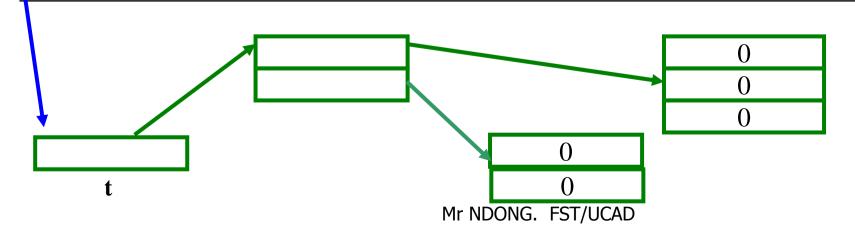
Elles déclarent que t est une référence à un tableau, dans lequel chaque élément est lui-même une référence à un tableau d'entiers. Pour l'instant, aucun tableau de cette sorte n'existe encore.



Considérons l'instruction:

 $-int[][]t = {new int [3], new int [2]};$

L'initialiseur de t comporte deux éléments dont l'évaluation crée un tableau de 3 entiers et un tableau de 2 entiers . On aboutit à cette situation (les éléments des tableaux sont, comme d'habitude, initialisés à 0) :





Dans ces conditions, on voit que:

- ✓ la notation t [0] désigne la référence au premier tableau de 3 entiers
- ✓ la notation t[0][1] désigne le deuxième élément de ce tableau
- ✓ la notation t[1] désigne la référence au second tableau de 2 entiers
- ✓ la notation t[1][i-1] désigne le ième élément de ce tableau .
- ✓ 1'expression *t.length* vaut 2
- ✓ 1'expression t[0].length vaut 2
- ✓ 1'expression t[1].length vaut 3



Second exemple:

On peut aboutir à une situation très proche de la précédente en procédant ainsi : int [][];

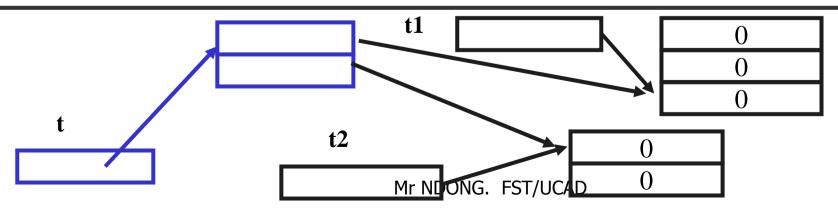
t = new int [2][3]; // creation d'un tableau de deux tableaux d'entiers

int [] t1 = new int [3]; // t1 = reference à un tableau de 3 entiers

int [] t2 = new int [2]; // t2 = reference à un tableau de 2 entiers

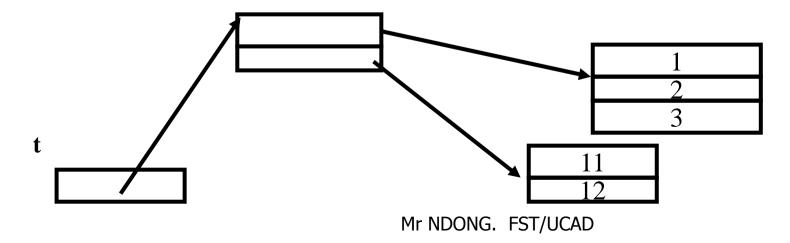
t[0] = t1; t[1] = t2; // on range ces deux references dans t

La situation peut être illustrée ainsi :





Dans le premier exemple, nous avons utilisé un initialiseur pour les deux références à introduire dans le tableau t ;autrement dit, nous avons procédons comme pour un tableau à un indice. Mais, les initialiseurs peuvent tout à fait s'imbriquer, comme dans cet exemple :





Tableaux réguliers

Rien n'empêche que dans un tableau toutes les lignes aient la même taille. Par exemple si l'on souhaite disposer d'une matrice de NLIG lignes et de NCOL colonnes, on peut procéder comme suit:

```
int tab [][] = new int [NLIG][];
```

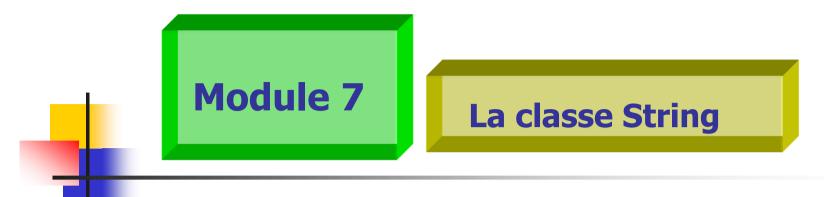
Et faire:

```
for (int i = 0; i<NLIG;i++) tab[i] = new int [NCOL];
```

Mais on peut écrire plus simplement:

```
int tab [][] = new int [NLIG][NCOL];
```

Maintenant il sera possible de parcourir ce tableau sans recourir à la variable length, comme ceci:



La classe **String** permet de manipuler « les chaînes de caractères ».

```
String chaine ;// declaration d une reference chaine
// à un objet de type String
chaine = "ca cest un objet chaine de caractere" ;//un objet
//chaine référencé par la variable chaine
```

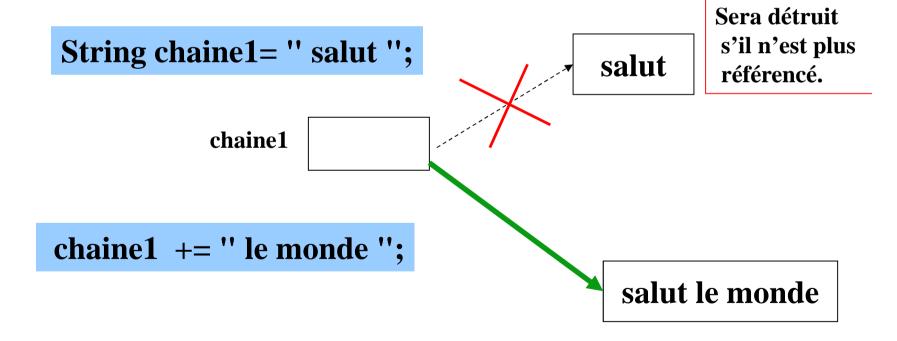
La classe String possède plusieurs constructeurs dont:

```
String(); // pour construire une chaine vide
String (String original ); // crée un objet contenant la chaine original
```



Fonctionnalités

Un objet de type String n'est pas modifiable.



ATTENTION

L'objet n'a pas été modifié, c'est simplement la référence qui change.

Les méthodes de la classe String (1/4)

```
/*retourne la longueur de l'objet String.*/
int length ()
/*retourne un nouvel objet String résultant du remplacement de toutes les
occurrences d'un caractère donnée par un autre caractère. */
String replace(char oldChar, char newChar)
/*remplace dans une chaîne de caractères, chaque sous-chaîne qui correspondent
à l'expression régulière fournie, par une chaîne de caractères de remplacement*/
String replaceAll (String origine, String remplacement)
/*teste si l'objet String démarre au préfixe spécifié. */
boolean startsWith (String prefix)
/*retourne une nouvelle chaîne de caractères qui est une sous-chaîne de l'objet String
par l'intermédiaire d'un intervalle commençant à l'index spécifié jusqu'à la fin. */
String substring (int beginIndex)
/*retourne une nouvelle chaîne de caractères qui est une sous-chaîne de l'objet Strin
par l'intermédiaire d'un intervalle spécifié */
String substring(int beginIndex, int endIndex) FST/UCAD
                                                                               206
```

Les méthodes de la classe String (2/4)

```
/*retourne la caractère positionné à l'index spécifié. */
char charAt (int index)
/*compare l'objet String à un autre objet o. Retourne 0 en cas d'égalité
-1 en cas d'infériorité et 1 en cas de supériorité */
int compareTo (Object o)
/*comparaison lexicographique de deux chaines. Retourne 0 en cas d'égalité
-1 en cas d'infériorité et 1 en cas de supériorité */
int compareTo(String anotherString)
/*compare deux chaînes lexicographiquement en ignorant la casse de caractères*/
int compareToIgnoreCase (String str)
/*concatène l'objet String à une autre chaîne de caractères. */
String concat (String str)
/*retourne true si et seulement si l'objet String représente la même séquence de
caractères comme l'objet StringBuffer spécifié. */
boolean contentEquals (StringBuffer buff)
```

Mr NDONG. FST/UCAD

207

Les méthodes de la classe String (3/4)

```
/*retourne un nouvel objet String qui représente la séquence de caractères
dans le tableau spécifié. */
static String copyValueOf (char[ ] data)
/*teste si la fin de l'objet String correspond au suffixe spécifié. */
boolean endsWith (String suffix)
/*compare le contenu de deux chaînes de caractères entre elles. */
boolean equals (Object anObject)
/*compare l'objet String à un autre objet de même type en ignorant la casse
de caractères. */
boolean equalsIgnoreCase (String anotherString)
/*retourne l'index à l'intérieur de l'objet String de la première occurrence du
caractère spécifié */
int indexOf (int ch)
/*retourne l'index à l'intérieur de l'objet String de la première occurrence du
caractère spécifié à partir d'une certaine position */
                                   Mr NDONG. FST/UCAD
                                                                             208
int indexOf(int ch, int fromIndex)
```



Les méthodes de la classe String (4/4)

```
/*retourne l'index à l'intérieur de l'objet String de la première occurrence de
la sous-chaîne spécifiée. */
int indexOf (String str)
/*retourne l'index à l'intérieur l'objet String de la première occurrence de la
sous-chaîne spécifiée à partir d'une certaine position. */
int indexOf(String str, int fromIndex)
/*retourne l'index à l'intérieur de l'objet String de la dernière occurrence
du caractère spécifié */
int lastIndexOf (int ch)
/*retourne une chaîne de caractères issue de l'objet String. */
String toString()
/*retourne un représentation sous forme d'un objet String de la valeur
de type int passée en argument */
static String valueOf (int i)
```



Utilisation de quelques méthodes (1/2)

```
String ch = " Le langage Java est vraiment très puissant ";
ch.length ();// longueur de la chaine est 43 (espaces compris)
ch.substring (11);
                                         "Java est vraiment très puissant ";
ch.substring (0,16);//attention: le caractere à l'indice 16 n'est pas extrait
                                         "Le langage Java ";
ch.toUpperCase ();// il existe aussi toLowerCase
                      "LE LANGAGE JAVA EST VRAIMENT TRÈS PUISSANT"
        bonjour \n'';
ch = "
                                                               → ''bonjour''
ch.trim() //supprime les espaces de début et de fin dans la chaine
                                    Mr NDONG. FST/UCAD
                                                                               210
```

Utilisation de quelques méthodes (2/2)

```
String s= " java ";
String ch = " java ";
s.equals (ch); // ici renvoie true car equals est redéfinie dans String
                                                                       true
s.charAt (0); // renvoie le caractere à la position 0 donc j
char c [ ] = s.toCharArray( );// renvoie un tableau de caracteres
                            → c vaut { 'j','a','v','a'}
ch.indexOf (a); // l'indice de la 1ère occurrence trouvée
                      renvoie la valeur 1
ch.indexOf (a,2); // l'indice de la 1ère occurrence trouvée à partir de 2
                      renvoie la valeur 3 CAD
                                                                          211
```



La méthode to Upper Case ()

La méthode toUpperCase de la classe String permet de convertir tous les caractères d'une chaîne en majuscules.

Pour convertir certains caratères d'une chaîne en majuscules, utilisez la méthode public static char to Upper Case (char c) de la classe Character.

Par exemple:

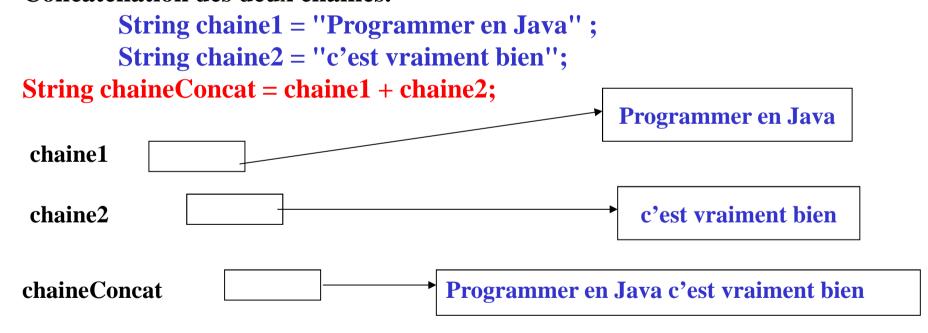
```
String s = "bonjour ";
for (int j = 0; j < s.length (); j++)
if (s.charAt (j) == 'o')
{
    char c = Character.toUpperCase (s.charAt (j) );
    s = s.replace (s.charAt (j), c );
}
System.out .print (" affichage: "+s); // affichage : "bOnjOur "</pre>
```

Il existe aussi la méthode public static char toLowerCase (char c)



L'opérateur +

L'opérateur + permet de concaténer deux chaînes de caractères. Il est défini lorsque ses deux opérandes sont des chaînes. Il renvoie un résultat qui est la Concaténation des deux chaînes.





L'opérateur +

L'opérateur + est utilisé lorsque ses deux opérandes sont de type String. Mais, il est possible de mélanger des expressions de type chaine et de type primitif. Dans ce cas, il y a conversion (formatage) de la valeur de type primitif en chaîne.

En définitive, lorsque l'opérateur + possède un opérande de type String, l'autre est automatiquement converti en String.

Lorsque l'opérateur + possède deux opérandes, l'un de type String, l'autre peut être de n'import quel type primitif, mais aussi de type objet. Dans ce dernier cas, il y a conversion de la valeur de l'objet en chaine et ceci est réalisé grâce à la méthode toString de la classe de l'objet qu'il faut souvent redéfinir.



L'opérateur +=

L'opérateur += défini dans le cadre des opérations arithmétiques binaires est également défini pour les chaînes. Cet opérateur n'est pas défini si son deuxième

i = 3

opérande est chaine alors que le premier ne l'est pas.

Seront candidat au ramasse miettes.

ch ="note 0123"

qui permet de modifier directement la valeur d'une chaîne de caractères.



= = et equals

```
String ch = "note";
String s = "note";
                                                     true
System.out.print (ch = = s); // affiche la valeur
                              // = = teste les références des chaînes
 Une chaine n'étant pas modifiable, une seule chaine est créée et
référencée par ch et s. On parle ici d'une fusion des chaînes identiques.
String ch = "bonjour";
String s = "bon";
      s + = "jour";
System.out.print (ch = = s); // affiche la valeur
                                               false
Vu l'utilisation non optimisée de = =, pour comparer deux chaînes il faut
utiliser la méthode equals qui compare le contenu de deux chaînes.
Cette méthode est celle de la classe Object mais redéfinie dans la classe String.
```



= = et equals



Conversions chaînes et types primitifs

La classe String possède une méthode statique valueOf surdéfinie avec un argument de chaque type primitif qui permet de convertir n'importe quel type primitif en chaîne de caractères.



Conversions chaînes et types primitifs

Il est possible de convertir une chaine dans un type primitif en recourant aux classes enveloppes définies pour chaque type primitif.

```
String ch = " 3521 ";
int n = Integer.parseInt (ch);// n contient la valeur entiere 3521
On dispose aussi des méthodes:
```

Byte.parseByte
Short.parseShort
Integer.parseInt
Long.parseLong
Double.parseDouble
Float.parseFloat



```
En Java, on peut convertir un tableau en chaîne et vice versa:
char [] mot = { 'b','o','n','j','o','u','r'};
/*on construit une chaine à partir d'un tableau de caracteres*/
String ch = new String (mot); // ch = "bonjour"
/*constructeur avec le premier caractère et le nombre de caracteres*/
String ch2 = new String (mot, 3,4);// ch = "jour"

String ch = "bonjour";
char [] mot = ch.toCharArray ();

mot = { 'b','o','n','j','o','u','r'};
```



La classe StringBuffer

Les objets de la classe String ne sont pas modifiables.

La modification d'une chaîne n'est possible qu'en créant une nouvelle chaîne, ce qui n'est pas optimale lorsqu'on manipule des chaînes assez intenses.

C'est pourquoi, Java propose la classe **StringBuffer** qui permet de manipuler des chaînes tout en ayant la possibilité d'en modifier la valeur sans créer de nouvelles chaînes.

```
/*pour optimiser la creation precedente de la chaine "note 01234" */
String ch = "note";
StringBuffer sb = new StringBuffer (ch) ;// on transmet ici un objet String
for (int i =0; i < 5;i++)
sb.append (i); //on rajoute a la fin du StringBuffer vide
ch = sb.toString (); // on convertit le StringBuffer en String
System.out.println (ch);
```

La classe StringTokenizer

Cette classe n'a aucun lien direct avec la classe String, elle se trouve d'ailleurs dans le paquetage java.util. Elle apporte un rôle dans la manipulation des chaînes en facilitant la division de chaînes en souschaînes selon un nombre de « délimiteurs ».

Cette méthode divise une chaîne en différents éléments appelés tokens.

On trouve 5 tockens : Java | est | un | langage | interessant



L'Héritage en Java

Héritage

possibilité de réutilisation des composants logiciels.

classe dérivée Héritage — définir une (nouvelle classe) d'une

(déjà existante). classe de base

La classe dérivée hérite donc de toutes les fonctionnalités de sa classe de base: champs et méthodes. Elle peut avoir des caractéristiques propres et redéfinir des caractéristiques héritées.

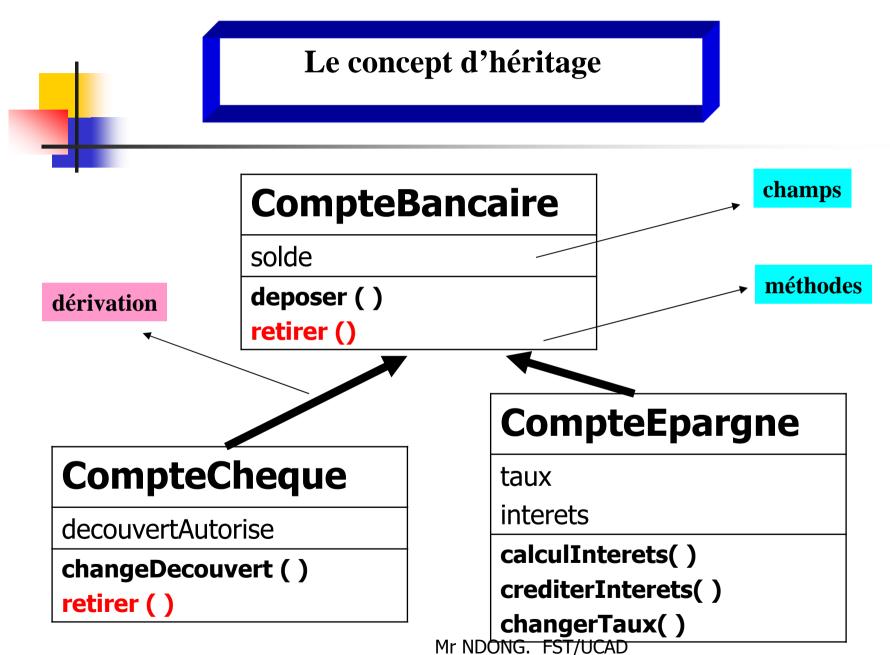
Mr NDONG. FST/UCAD

Le concept d'héritage

Supposons disposer d'une classe CompteBancaire:

```
allndong.compte;
package
public class CompteBancaire {
 double solde;
    CompteBancaire (double solde )
          { this.solde = solde;
   void deposer ( double montant)
      { solde +=montant;
    void retirer (double montant)
     { if (solde >=montant;
           solde -= montant;
```

On se propose de spécialiser la gestion des comptes. On crée alors une classe CompteChèque et une autre classe CompteEpargne qui dérivent de la classe CompteBancaire.



Le concept d'héritage



```
allndong.compte;
package
                            extends | CompteBancaire {
public class CompteCheque
               double decouvertAutorise;
CompteCheque (double solde, double decouvertAutorise)
          { super (solde);
           this.decouvertAutorise = decouvertAutorise;
   void retirer (double montant) // methode redefinie
      if (solde + decouvertAutorise >=montant ;
           solde -=montant ;
                                     Mr NDONG FST/UCAD
```

En Java, on utilise la mention extends pour signaler au compilateur que la classe CompteCheque dérive de la classe CompteBancaire.

Ici, on rajoute un nouveau champ decouvertAutorise. Et la méthode retirer est redéfinie.





```
allndong.compte;
package
public class CompteEpargne | extends | CompteBancaire {
               double taux;
CompteEpargne (double solde, double taux)
          { super (solde);
           this. taux = taux;
// pas de methode retirer
```

Ici, on rajoute un nouveau champ taux.





Avec l'instruction:

CompteEpargne ce = new CompteEpargne (20000, 0.05);

On peut bien faire:

ce.retirer (10000);

malgré que la méthode retirer n'est pas définie dans la classe CompteEpargne.

Un objet d'une classe dérivée accède aux membres publics de sa classe de base, exactement comme s'ils étaient dans la classe dérivée elle-même.

Une méthode d'une classe dérivée n'a pas accès aux <u>membres</u> privés de sa classe de base.



La construction d'un objet dérivé est intégralement prise en compte par le constructeur de la classe dérivée.

Par exemple, le constructeur de CompteCheque:

- initialise le champ decouvertAutorise (déjà membre de CompteCheque);
- appelle le constructeur de Compte pour initialiser le champ solde (hérité) dans l'initialisation de champs d'un objet dérivé , il est fondamental et très important de respecter une contrainte majeure:

Si un constructeur d'une classe dérivée appelle un constructeur d'une classe de base, il doit <u>obligatoirement</u> s'agir de la **première instruction du constructeur** et ce dernier est désigné par le mot clé **super.**





Nous avons mentionné au chapitre 5 qu'il est possible d'appeler dans un constructeur un autre constructeur de la même classe, en utilisant le mot clé *this* comme nom de méthode. Comme celui effectué par *super*, cet appel doit correspondre à la première instruction du constructeur.

Dans ces conditions, on voit bien qu'il n'est pas possible d'exploiter les deux possibilités en même temps . Autrement dit, dans un constructeur d'une classe dérivée il n'est pas possible d'appeler en même temps un constructeur de la même classe et un constructeur d'une classe de base.





```
package allndong.compte;
class CompteBancaire {
 double solde;
  void retirer (double montant)
     { if (solde >=montant;
           solde -=montant ;
void imprimeHistorique( )
{ System.out.print (" solde ="
+solde);
```

```
package allndong.compte;
class CompteCheque extends CompteBancaire {
double decouvertAutorise;
// methode redefinie
     void retirer (double montant)
    { if (solde + decouvertAutorise >=montant;
           solde -= montant;
void imprimeHistoriqueCheque( )
{ System.out.print (" solde =" +solde + " " +
"decouvertAutorise=" +decouvertAutorise);
  MI NOONO. ISTOCAD
```



Redéfinition de membres (2/4)

Avec:

CompteBancaire cb; CompteCheque cc;

l'appel : cb.retirer (20000);

appelle la méthode retirer de CompteBancaire.

l'appel : cc.retirer (20000);

appelle la méthode retirer de CompteCheque.

On se base tout simplement sur le type de l'objet pour déterminer la classe de la méthode appelée.

Pour bien voir l'intérêt de la redéfinition des méthodes, examinons la méthode imprimeHistorique de la classe CompteBancaire qui permet d'afficher le solde pour un compte et la méthode imprimeHistoriqueCheque de la classe CompteCheque qui affiche non seulement le solde (qui est un membre hérité) mais aussi le decouvertAutorise. Dans cette dernière, il y a une information qui est déjà prise en compte dans la méthode imprimeHistorique. La situation précédente peut être améliorée de cette façon:





```
package allndong.compte;
class CompteBancaire {
 double solde;
  void retirer (double montant)
     { if (solde >=montant;
           solde -=montant ;
void imprimeHistorique( )
{ System.out.print (" solde ="
+solde);
```

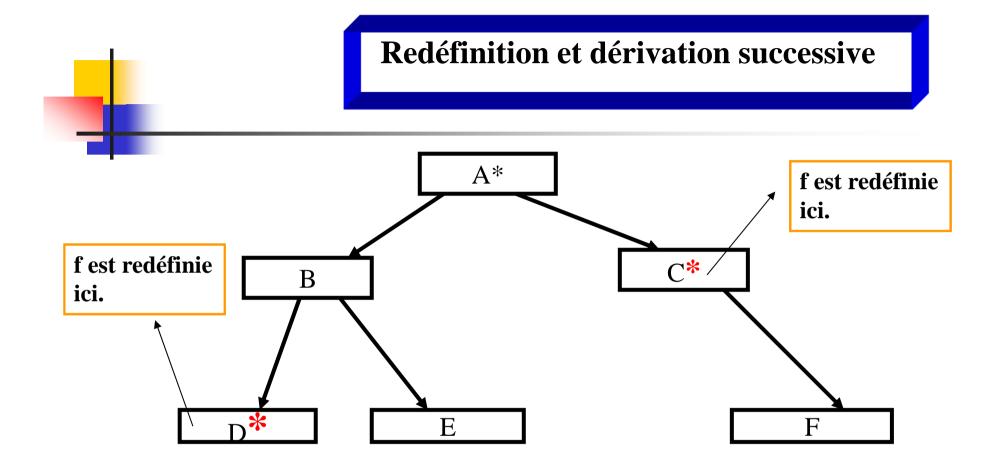
```
allndong.compte;
package
class CompteCheque extends CompteBancaire {
double decouvertAutorise;
// methode redefinie
void retirer (double montant)
{ if (souce + accouvertAutorise >= montant ;
            solde -= montant;
void imprimeHistoriqueCheque( )
      imprimeHistorique();
System.out.print (" et le decouvertAutorise ="
+decouvertAutorise);
  Mr NDONG. FST/UCAD
                                                 23
```



```
package allndong.compte;
class CompteBancaire {
 double solde;
  void retirer (double montant)
     { if (solde >=montant;
           solde -=montant ;
 void imprimeHistorique( )
{ System.out.print (" solde ="
+solde);
```

super obligatoire.

```
package
          allndong.compte;
class CompteCheque extends CompteBancaire {
double decouvertAutorise;
// methode redefinie
void retirer (double montant)
{ if (souce + accouvertAutorise >= montant ;
            solde -= montant;
   void imprimeHistorique
     Super.imprimeHistorique();
System.out.print (" et le decouvertAutorise ="
+decouvertAutorise);
  Mr NDONG. FST/UCAD
                                                 23
```



L'appel de f conduira, pour chaque classe, à l'appel de la méthode indiquée en regard :

class A : méthode f de A | class D: méthode f de D

class B: méthode f de A | class E: méthode f de A

class C: méthode f de C class F: méthode f de C Mr NDONG. FST/UCAD





```
package allndong.compte;
class Calcul {
public double division (int a) // 1
    // instructions
class CalculXXX extends Calcul {
public float division (float a) // 2
    // instructions
```

```
Calcul c; CalculXXX cx; int n; float p;

c.division (n); // appel de 1

c.division (p); // erreur de compilation

cx.division (n); // appel de 1

cx.division (p); // appel de 2
```

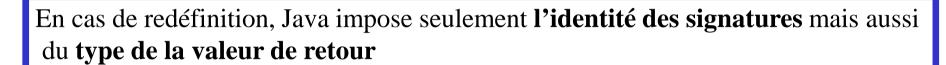
La recherche d'une méthode acceptable ne se fait

qu'en remontant la hiérarchie d'héritage, jamais

en la descendant .C'est pourquoi l'appel

c.division (p); ne peut être satisfait.





Signatures identiques Valeurs de retour identiques package allndong.compte; package allndong.compte; **class** CompteBancaire { **class** CompteCheque extends CompteBancaire { double solde; double decouvertAutorise; void retirer (double montant) void retirer (double montant) { if (solde >=montant; { if (solde + decouvertAutorise >=montant; solde -=montant ; solde -=montant ; }} Mr NDONG. FST/UCAD 23



La redéfinition d'une méthode ne doit pas diminuer les droits d'accès à cette méthode.

Par contre, elle peut les augmenter.

```
class A
{ public void f ( int n){.....}
}
class B extends A
{// impossible de mettre private
private void f ( int n) {....}
}
```

```
class A
{ private void f ( int n){.....}
}
class B extends A
{// augmente les droits d acces: possible
public void f ( int n) {....}
}
```

Règles sur :

la Redéfinition et la Surdéfinition

Si une méthode d'une clase dérivée a la même signature qu'une méthode d'une classe ascendante :

- les valeurs de retour des deux méthodes doivent être exactement de même type,
- le droit d'accès de la méthode de la classe dérivée ne doit pas être plus élevé que celui de la classe ascendante,
- la clause *throws* de la méthode de la classe dérivée ne doit pas mentionner des exceptions non mentionnées dans la clause throws de la méthode de la classe ascendante(la clause *throws* sera étudiée ultérieurement).

Si ces trois conditions sont remplies, on a affaire à une redéfinition. Sinon, il s'agit d'une erreur.

Dans les autres cas, c'està-dire lorsqu'une méthode d'une classe dérivée a le même nom qu'une méthode d'une classe ascendante, avec une signature différente, on a affaire à une *surdéfinition*





- 1 Une méthode de classe (*static*) ne peut pas être redéfinie dans une classe dérivée. Cette restriction va de soi puisque c'est le type de l'objet appelant une méthode qui permet de choisir entre la méthode de la classe de base et celle de la classe dérivé.
- 2 Les possibilités de redéfinition d'une méthode prendront tout leur intérêt lorsqu'elles seront associées au polymorphisme que nous allons étudié.

Bien que cela soit d'un usage peu courant, une classe dérivée peut définir un champ portant le même nom qu'un champ d'une classe de base ou d'une classe ascendante. Ce phénomène est appelé duplication de champs.

Duplication de champs

```
class A
{ public int resultat;

// instructions
}
```

On utilise le mot clé **Super** pour accéder à un champ de la super classe





Surclassement

La réutilisation de code est un aspect important de l'héritage, mais ce n'est peut être pas le plus important.

Un autre aspect fondamental est la relation qui relie une classe à sa super classe.

Une classe B qui hérite d'une classe A peut être vue comme un sous-type (sous - ensemble) du type défini par la classe A.

CompteBancaire



CompteCheque

Un CompteCheque est un CompteBancaire

L'ensemble des compte chèque est inclus dans l'ensemble des compte bancaires

Mr NDONG. FST/UCAD



Surclassement

Tout objet instance de la classe B peut être aussi vu comme une instance de la classe A.

Cette relation est directement supportée par le langage JAVA.

« à une référence déclarée de type A il est possible d'affecter une valeur qui est une référence vers un objet de type B (surclassement ou upcasting) »

CompteBancaire



CompteCheque

CompteBancaire cb;

cb = **new CompteCheque**(**100000**, **50000**);

« plus généralement, à une référence d'un type donné, il est possible d'affecter une valeur qui correspond à une référence vers un objet dont le type effectif est n'importe quelle sous-classe directe ou indirecte du type de la référence ».



Surclassement

Lorsqu'un objet est « sur-classé » il est vu comme un objet du type de la référence utilisée pour le désigner.

« ses fonctionnalités sont alors res<u>treintes à celles proposées par la classe du type de la</u>

référence ».

CompteBancaire solde deposer() retirer()



CompteCheque

decouvertAutorise

changeDecouvert ()

retirer ()

```
CompteCheque CC = new CompteCheque(100,50);
CompteBancaire Cb;
cb = CC; // surclassement
cb.retirer (50);
cc.retirer(25);
cb.deposer (500);
cc.deposer (250);
cb.changedecouvert();
cc.changeDecouvert();
Mr NDONG, FST/UCAD
```



Résolution des messages

« Que va donner cb.retirer (50)? »

```
CompteBancaire cb = new
CompteCheque (500,100);

cb.retirer (50)

CompteCheque

void retirer (double montant)
solde -=montant;

CompteCheque
void retirer (double montant)
solde retirer (double montant)
```



Résolution des messages

```
CompteBancaire cb = new
CompteCheque (50000,100);

cb.retirer (50)
```

CompteCheque

Mr NDONG. FST/UCAD



Résolution des messages

VRAIMENT A RETENIR

9

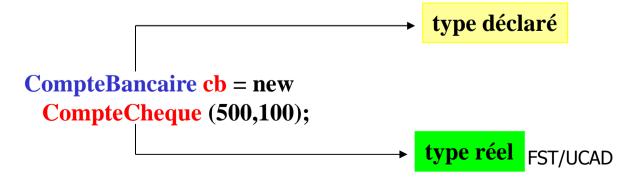
Lorsqu'une méthode d'un objet est accédée au travers d'une référence « surclassée », c'est la méthode telle qu'elle est définie au niveau de la classe effective de l'objet qui est réellement invoquée et donc exécutée.

Mécanisme de résolution des messages

Les messages sont résolus à l'exécution.

- « la méthode à exécuter est déterminée à l'exécution (run– time) et non pas à la compilation ».
- « la méthode définie pour le type réel de l'objet recevant le message est appelée et non pas celle définie pour son type déclaré ».

Ce mécanisme est désigné sous le terme de liaison dynamique (dynamic binding, late binding ou run-time binding)..



Vérifications statiques

A la compilation: seules des vérifications statiques qui se basent sur le type déclaré de l'objet (de la référence) sont effectuées.

La classe déclarée de l'objet recevant le message doit posséder une méthode dont la signature correspond à la méthode appelée.

A la compilation: il n'est pas possible de déterminer le type exact (réel) de l'objet récepteur du message.

Vérification statique : garantit dès la compilation que les messages pourront être résolus au moment de l'exécution. Elle permet de déterminer (figer) simplement la signature et le type de la valeur de retour de la méthode qui sera exécutée.

Choix des méthodes, sélection du code

Le choix de la méthode à exécuter est effectuée statiquement à la compilation en fonction du type des paramètres.

La sélection du code à exécuter est effectuée dynamiquement à *l'exécution* en fonction du type effectif de l'objet récepteur du message.



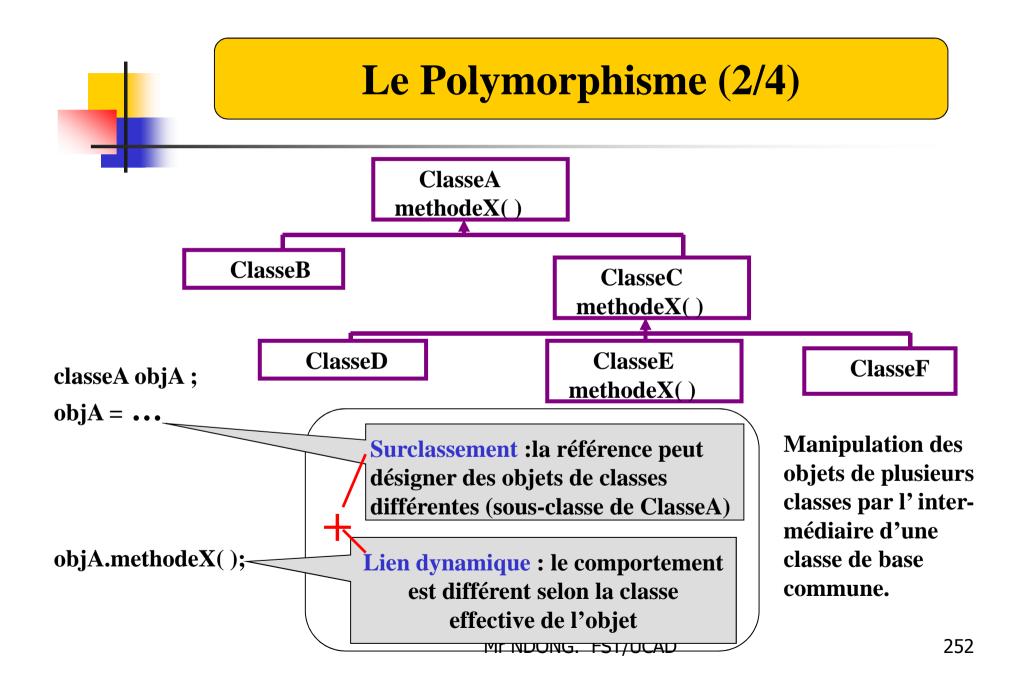
Le Polymorphisme (1/4)

Le **surclassement** et la **liaison dynamique** (ligature dynamique) servent à mettre en œuvre le **polymorphisme**.

Le terme polymorphisme décrit la caractéristique d'un élément qui peut prendre plusieurs formes, à l'image de l'eau qui peut être à l'état liquide, solide ou gazeux. En programmation Objet, on appelle polymorphisme:

- ➤ le fait qu'un objet d'une classe puisse être manipulé comme s'il appartenait à une autre classe
- ➤ le fait que la même opération puisse se comporter différemment sur différentes classes de la hiérarchie.

Le polymorphisme est la troisième caractéristique essentielle d'un langage orienté Objet après l'abstraction des données (encapsulation) et l'héritage.





Le Polymorphisme (3/4)

Etudiant

```
public void affiche()
{ System.out.print (
" Nom:" +nom+ ""
" Prenom:" +prenom+ ""
" Age:" +age+ "" );
```

EtudiantSportif

```
public void affiche()
{ super.affiche();
    System.out.print (
"Sport:" +sport+ "" );
```

EtudiantEtranger

```
public void affiche()
{ super.affiche();
System.out.print(
" Pays:" +pays+ "" );
```

Le Polymorphisme (4/4)

```
public class GroupeTD {
                                         nbEtudiants = 3
Etudiant liste [] = new Etudiant [30];
                                         À l'appel de afficherListe () on a :
static int nbEtudiants;
                                         // appel la méthode de Etudiant
public static void ajouter (Etudiant e)
                                         liste[0].affiche();
{ if (nbEtudiants < liste.length)
                                         // appel la méthode de EtudiantSportif
   liste [nbEtudiants ++] = e;
                                         liste[1].affiche();
public static void afficherListe(){
                                         // appel la méthode de EtudiantEtranger
for (int i = 0; i < nbEtudiants; i++)
liste[i].affiche();
                                         liste[2].affiche();
public static void main (String args [])
{ ajouter (new Etudiant (" Sene ", " Pierre ", 12));
 ajouter (new EtudiantSportif ("Fall", "Fatou", 5, "Natation"));
 ajouter (new EtudiantEtranger (" Ndiaye ", " Moussa ", 20," Senegal "));
afficherListe(); } }
                                        MI NDONG. FST/UCAD
```

Polymorphisme:



```
public class ClasseXXX
{ public void methodeXX
```

```
ClasseXXX objX = ....;
ClasseYYY objY = ....;

float e;
// appel de 1
objX.methodeXXXX ( e );
// appel de 3
objY.methodeXXXX ( e );

objX = objY;
// appel de 2 car poly...
objX.methodeXXXX ( e );
```



La super classe Object

En Java, toute classe dérive *implicitement* de la classe Object. Elle est donc la classe de base de toute autre classe.

```
Une déclaration comme celle-ci :
```

```
class Point { ...}
```

est équivalente à:

class Point extends Object { ...}

Quelles en sont les conséquences ?



Référence de type Object

Une variable de type Object peut toujours être utilisée pour référencer n'importe quel objet d'un type class quelconque.

« ceci peut être utile dans le cas où on manipule des objets sans connaître leur type exact. Cela peut être le cas avec les Collections ». Object o;

```
Point p = new Point(....);
Pointcol pc = new Pointcol (...);
On peut faire:
o = p;
o = pc;
((Point )o).deplace (...); // OK
o.deplace (...); // erreur car deplace n existe pas dans Object
```

Méthodes



equals et toString de Object

La méthode toString fournit une chaîne de caractères (ie un objet de la classe String) précisant:

- le nom de la classe de l'objet concerné, suivi du signe @
- l'adresse de l'objet en hexadécimal,

Point a = new Point (1,2);

System.out.println (" a = "+ a.toString());

a = Point@fc17aedf

NB: le plus souvent, vous aurez à redéfinir cette méthode.

Nous verrons cette méthode dans le module sur la classe String.



equals (3/1)

La méthode **equals** se contente de comparer les **adresses** de deux objets.

Avec:

```
Object o1 = new Point (1,2);
Object o2 = new Point (1,2);
```

O1.equals (o2) renvoie la valeur false.

En effet cette méthode est définie dans **Object** comme ceci:

```
public boolean equals (Object o)
{ return this == o;// == teste les références
}
```

Vous pouvez aussi redéfinir cette méthode à votre convenance.



equals (3/2)

```
class Point
{......
boolean equals ( Point p)
    { return ((p.x = = x) &&(p.y = = y));
    }
}
Avec :
Point a = new Point (1,2);
Point b = new Point (1,2);
```

Problème:

```
En revanche, avec :

Object o1 = new Point(1,2);

Object o2 = new Point(1,2);

l'expression o1.equals(o2)

aura la valeur false car on aura

utiliser la méthode equals de la

classe Object et non celle

de Point (règles du polymor -

-phisme).
```

a.equals (b); renvoie cette fois la valeur true



equals (3/3)

Il faut toujours prendre la peine de redéfinir convenablement cette méthode. Pour résoudre le problème posé, on peut améliorer la redéfinition de la méthode comme ceci:

```
class Point
{....
public boolean equals (Object o)
 { if (!o instanceof (Point))
        return false;
     else Point p = (Point) o; // sousclassement (downcasting)
               return ((this.x = = p.x) && (this.y = = p.y));
```



Quelques définitions

Une méthode déclarée final ne peut pas être redéfinie dans une classe dérivée.

Une classe déclarée final ne peut plus être dérivée.



Une classe abstraite ne peut instancier aucun objet. Une telle classe ne peut servir qu'à une dérivation (héritage).

Dans une classe abstraite, on peut trouver :

- des **méthodes** et des **champs**, dont héritera toute classe dérivée.
- des **méthodes abstraites**,(avec *signature* et *type de valeur de retour*). Le recours aux classes abstraites facilite largement la Conception Orientée Objet. En effet, on peut placer dans une classe abstraite **toutes les fonctionnalités** dont on souhaite disposer pour toutes ses descendantes :
- soit sous forme d'une implémentation complète de méthodes(non abstraites) et de champs(privés ou non) lorsqu'ils sont communs à toutes ses descendantes,
- soit sous forme d'interface de méthodes abstraites dont on est alors sûr qu'elles existent dans toute classe dérivée instanciable.



Généralités

On définit une classe abstraite en Java en utilisant le mot clé abstract devant le nom de la classe:

Avec cette classe:

- on peut écrire : Forme f ;// declaration d'une reference de type forme
- par contre on ne peut écrire : f = new Forme ();// INTERDIT



Généralités

Maintenant si on se retrouve dans la situation suivante:

```
class Rectangle extends Forme
{
    // ici on redefinit TOUTES les méthodes héritiées de
    // Forme
}
Alors on peut instancier un objet de type Rectangle et placer sa
référence dans une variable de type Forme (polymorphisme):
```

Forme f = new Rectangle ();// OK car polymorphisme

Exemple

```
package allndong;
abstract class Forme
{ public abstract double perimetre(); }// fin de Forme
class Circle extends Forme {private double r; //...constructeur à definir
                 public double perimetre ( ) { return 2 * Math.PI * r ; }
               }//fin de Circle
class Rectangle extends Forme { private double long, larg; //constructeur à définir
        public double perimetre() { return 2 * (long + larg); }
}//fin de Rectangle
            dans le main d'une classe de test */
Forme [] formes = {new Circle(2), new Rectangle(2,3), new Circle(5)};
double somme_des_perimetres = 0;
for (int i=0; i< formes.length; i++)
         somme_des_perimetres += formes[i].perimetre ( );
```



Quelques remarques importantes (1/2)

- Une classe abstraite est une classe ayant au moins une méthode abstraite.
- Une méthode abstraite ne possède pas de définition.
- Une classe abstraite ne peut pas être instanciée (new).
- Une classe dérivée d'une classe abstraite ne redéfinissant pas toutes les méthodes abstraites est elle-même abstraite.
- Une méthode abstraite ne doit pas être déclarée final, puisque sa vocation est d'être redéfinie. De même une classe abstraite ne doit pas être final.
- Une méthode abstraite ne doit jamais pas déclarée private.
- Une méthode abstraite ne doit jamais pas être déclarée static.



Quelques remarques importantes (2/2)

✓ Dans l'en-tête, il faut obligatoirement mentionner le nom des paramètres formels (muets):

```
abstract class A
{
    void g ( int  ) //nom d argument muet obligatoire sinon erreur de compilation
}
```

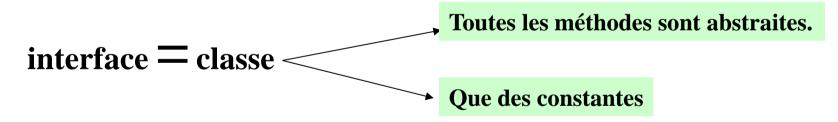
- ✓ Une classe dérivée d'une classe abstraite n'est pas obligée de redéfinir toutes les méthodes abstraites de sa classe de base(elle peut même n'en redéfinir aucune). Dans ce cas, elle reste simplement abstraite.
- ✓ Une classe dérivée d'une classe non abstraite peut être déclarée abstraite et/ou contenir des méthodes abstraites.



Interfaces

➤ Une interface correspond à une classe où **TOUTES** les méthodes sont abstraites. Une classe peut implémenter (implements) une ou plusieurs interfaces tout en héritant (extends) d'une classe.

Une interface peut hériter (extends) de plusieur (s) interface(s).



Généralités

```
package allndong;
                                                      Les modificateurs
                       ici on ne peut pas mettre.
                                                      private et protected sont
interface Operation
                                                      interdits.
                                                      Toute variable déclarée ne
{ /*constantes*/
                                                      peut être qu'une constante
 public double nombre =100;
                                                      (donc ayant une valeur).
        final float x = 1000;
 /* que des methodes abstraites*/
 public double addition( );
 public float division (float a, float b); //private et protected interdit
      abstract double multiplication (); //abstract non obligatoire
//fin de Operation
```

Dans la définition d'une interface seuls les droits d'accès public et doit de paquetage (vide) sont autorisés.



Utilisation d'une interface

Les interfaces sont faites pour être implémenter.

Une contrainte dans l'implémentation d'une interface:

il faut obligatoirement <u>redéfinir toutes</u> les méthodes de l'interface

Aucune définition de méthode ne peut être différée comme dans le cas des classes abstraites.

Lorsque que vous implémentez une interface, vous ne redéfinissez que les méthodes de l'interface, les constantes sont directement utilisables (vous n'avez pas besoin de les mentionner dans la classe qui implémente).



Exemple de mise en oeuvre

```
package allndong;
abstract class Forme{ public abstract double perimetre();
                                                                       }// fin de Forme
 interface Dessinable { public void dessiner ( ); }
                                                         {private double r;
class Circle extends Forme
                              implements Dessinable
                   public double perimetre () { return 2 * Math.PI * r; }
                public void dessiner(){ //instructions de dessin d un cercle}
                }//fin de Circle
class Rectangle extends Forme implements Dessinable public double perimetre() { return 2 " (long + larg); }
                                                            private double long, larg;
                 public void dessiner ( ){ //instructions de dessin d un rectangle}
}//fin de Rectangle
              dans le main d'une classe de test */
Dessinable [] dessins = {new Circle (2), new Rectangle(2,3), new Circle(5)};
for (int i=0; i< dessins.length; i++)
          dessins[i].dessiner();
```

Diverses situations avec les interfaces

On dispose de deux interfaces : interface I1 {...}

interface I2 {...}

Le fait de pouvoir implémenter plusieurs interfaces peut résoudre le problème de la dérivation multiple connue dans les autres langages objets comme (C++)

Vous pouvez avoir:

interface I3 extends I1 {/....} //derivation dune interface

class A implements I2/{ ...}//implementer une seule interface

class B implements I1, I2, I3 {} //implementer plusieurs interfaces

class C extends A implements I3 {....} // derivation d une classe et // implementation d une interface



Les classes enveloppes

Il existe des classes nommées *Boolean*, *Byte*, *Character*, *Short*, *Integer*, *Long*, *Float* et *Double*, destinées à manipuler des valeurs de type primitif en les encapsulant dans une classe. Cela permet de disposer de méthodes et de compenser le fait que les types primitifs ne soient pas des classes.

Toutes ces classes disposent d'un constructeur recevant un argument d'un type primitif :

```
Integer objInt = new Integer (5); // objInt contient la référence à un // objet de type Integer encapsulant la valeur 5
```

Elles disposent toutes d'une méthode de la forme *xxxValue* (*xxx* représentant le nom du type primitif) qui permet de retrouver la valeur dans le type primitif correspondant :



Exemple

```
Integer objet_n = new Integer (12);
Double objet_x = new Double (5.25);
int n = objet_n _intValue ();  // objet_n contient 12
double x = objet_x _.doubleValue ();  // objet_x contient 5.25
```

Nous verrons aussi dans le chapitre consacré aux chaînes (classe String), qu'elles disposent d'une méthode *toString* effectuant la conversion de la valeur qu'elles contiennent en une chaîne, ainsi que d'une méthode de la forme *parseXXX* permettant de convertir une chaîne en un type primitif.



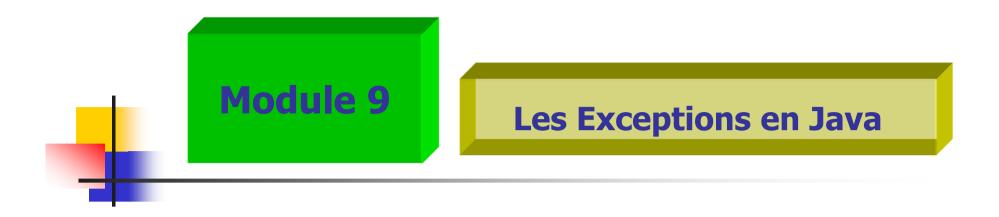
Exercice

On donne l'interface suivante:

```
interface Calculatrice{
/*calcule et renvoie la racine carrée de nombre*/
public double racineCarre(double nombre);
/*affiche le resultat renvoye par la methode racineCarre*/
public void afficheRacine();
/*calcule et renvoie val à la puissance degre*/
public double puissance(int val, int degre);
/*affiche le resultat renvoye par la methode puissance*/
public void affichePuissance();
}
```

Créer une classe qui utilise correctement cette interface.

Mr NDONG. FST/UCAD



Lorsqu'un programme traite des données, il peut arriver deux types de situations gênantes :

- on peut lui demander d'enlever un élément d'une liste vide. Il est possible de traiter ce problème tout de suite, en disant que le résultat, dans ce cas, est une liste vide ;
- on peut aussi demander la valeur du premier élément d'une liste vide. Dans ce cas, on ne peut pas répondre. La seule chose possible est de dire qu'il y a une erreur à cet endroit, et charge à d'autres d'essayer de réparer cette erreur.

Dans le premier type de situations, il est possible de modifier le code en séparant les cas (içi, liste vide ou liste non vide), pour éliminer le problème.

Le deuxième cas est plus gênant, car on ne peut pas le traiter au niveau actuel. Il faut arrêter de faire ce qui était en cours, et signaler l'erreur. On appelle cela une **Exception**.

Introduction

En Java, il existe deux classes intéressantes dérivées de la classe *Throwable*: la classe *Error*, d'où proviennent les erreurs système souvent irrécupérables représentées par les classes suivantes :

VirtualMachineError: indique de graves problèmes sur la machine virtuelle Java,

AWTError: indique une sérieuse erreur du système de gestion de l'interface

utilisateur AWT (Abstract Window Toolkit),

ThreadDeath: indique l'arrêt inopportun d'un Thread,

LinkageError: indique des problèmes de liaisons entre des classes.

Les exceptions provenant de la classe *Error* dépassent souvent le domaine de compétence d'un programmeur. Toutefois, certaines peuvent être gérées, à

l'image d'une erreur du type *OutOfMemoryError* (plus de mémoire).

la classe *Exception*, d'où sont issues les fameuses exceptions déclenchées par le compilateur.

Les exceptions sont donc de véritables objets créés suite à la détection d'une anomalie dans le déroulement du programme.

Contrairement à celles de la classe **Error**, les exceptions de la classe **Exception** peuvent et dans la plupart des cas, doivent être interceptées.

Vue générale sur les exceptions(1/2)

```
float division (float a, float b) {
return a/b;
}
```

La méthode **division** ne semble pas poser de problèmes. On peut cependant remarquer qu'il peut y avoir un problème si b = 0. C'est d'autant plus gênant que cela n'embête pas Java, qui va rendre une valeur. Autrement dit, on va continuer le calcul avec une valeur qui n'a aucun sens, et sans le savoir. Dans le cas précédent, il n'y avait pas d'autre possibilité que de s'arrêter. Dans ce cas-ci, il est possible de continuer, mais ce n'est pas souhaitable. Il faut signaler qu'il y a eu un problème à cet endroit. Il est donc préférable, là encore, d'utiliser une **Exception**.

Une Exception est un problème qu'il n'est pas possible de traiter immédiatement.



Vue générale sur les exceptions(2/2)

Une bonne gestion des exceptions en Java permet :

- de dissocier la détection d'une anomalie de son traitement,
- de séparer la gestion des anomalies du reste du code, donc de contribuer à la lisibilité des programmes.

Une exception est déclenchée par une instruction *throw* comportant une expression qui est un objet de type classe . Il y a alors branchement

à un ensemble d'instructions nommé « **gestionnaire d'exception** ». Le choix du bon gestionnaire est fait en fonction de l'objet mentionné à *throw* (de façon comparable au choix d'une fonction surdéfinie).

L'instruction throw

```
public class Except01 {
/*methode qui declenche une exception*/
                                              throws DivisionParZero
public static float division( float a, float b)
 \{ if (b = 0) throw new DivisionParZero("denominateur nul"); \}
  else return a/b;
                                         throws DivisionParZero
 public static void main(String[] args)
  System.out.println (division(1,0));
  System.out.println ("Merci!");
                                                     Ceci car on appelle la méthode
                                                     division qui est susceptible de
                                                     gégérer une exception.
                            extends Exception
class DivisionParZero
{DivisionParZero (String mes)
 { super(mes); }
                                     Mr NDONG. FST/UCAD
                                                                                 281
```



Remarques importantes

Au niveau de la méthode division, la clause throws DivisionParZero est obligatoire. Elle précise que la méthode est susceptible de déclencher une exception de type DivisionParZero.

Il faut bien noter que la méthode ne fait que déclencher; elle ne traite pas l'exception.

Autre remarque très importante:

la clause *throws* d'une méthode doit mentionner au moins la réunion de toutes les exceptions mentionnées dans les clauses *throws* des méthodes appelées. C'est pourquoi la méthode main (appelant la méthode division) mentionne dans sa clause throws l'exception DivisionParZero.

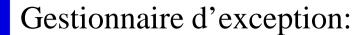




Une bonne gestion des exceptions doit toujours permettre à l'utilisateur, s'il le désire de pouvoir continuer l'exécution du programme après détection d'une anomalie. Dans l'exemple précédent de la classe Except01, la méthode division déclenche bien une exception q'elle ne traite pas et la méthode appelante (ie la méthode main aussi ne la traite pas non plus. C'est pourquoi le programme ne se poursuit pas pour exécuter l'instruction System.out.println ("Merci!"). L'intérêt de l'exemple précédent est simplement d'avoir des informations plus parlantes sur la nature de l'exception.

En java, si une méthode déclenche une exception qu'elle ne traite pas, la ou les méthodes appelantes doivent la traiter (avec un gestionnaire d'exception.

Traitement des exceptions avec un GESTIONNAIRE D'EXCEPTION qui donne des informations précises sur la cause, la localisation d'une erreur...





bloc try.....catch

Voyons maintenant comment procéder pour gérer convenablement les éventuelles exceptions de type **DivisionParZero** que son emploi peut déclencher. Pour ce faire , il faut :

-inclure dans un bloc dit « bloc try » les instructions dans lesquelles on risque de voir déclenchée une telle instruction; ce bloc se présente ainsi :

```
try {
     // instructions
}
```

- faire suivre ce bloc de la définition des différents gestionnaires d'exception :

```
catch (DivisionParZero e)
{ //instructions
}
```

Exemple avec gestionnaire d'exception

```
public class Except02 {
/*methode qui declenche une exception*/
                                                throws DivisionParZero
public static float division( float a, float b)
\{ if (b = 0) throw new DivisionParZero (); \}
  else return a/b; }
/* là on traite vraiment l'exception*/
public static void main(String[] args) {
                                              Ici le gestionnaire d'exception
  try { System.out.println (division(1,0));
                                              ne fait rien du tout (ie aucune
         System.out.println ("Merci!");
                                               <mark>information n'est transmise au</mark>
                                              gestionnaire).
 catch (DivisionParZero e)
     { System.out.println(" le denominateur ne peut pas etre nul");
class DivisionParZero
                        extends Exception
{}
                                       Mr NDONG. FST/UCAD
```





Là aussi à l'exécution, l'exception causée par la division par zéro est lancée dans la méthode division et traitée par le bloc catch (DivisionParZero e) dans la méthode appelante. Mais, comme précédemment l'instruction : System.out.println(« Merci! ») ne sera pas exécutée. Ceci parce que tout simplement dès qu'on sort du bloc try pour entrer dans le bloc catch, on ne peut plus revenir dans le try.

Si on veut continuer l'exécution des instructions après l'exception, il faut utiliser l'instruction finally après catch. Cette instruction veut dire qu'il y ait exception ou pas les instruction se trouvant dans ce bloc seront exécutées. On écrit:



Comment gérer plusieurs exceptions

Dans l'exemple précèdent, on ne gérait qu'un seul type d'exception (la division par zéro). Maintenant, nous allons voir comment prendre en compte plusieurs exceptions dans le programme.

On rajoute, par exemple, une exception provoquée par une taille négative d'un tableau d'entiers.

On verra en même temps comment transmettre de l'information au gestionnaire d'exception.

En fait il faut exploiter les méthodes de la classe Exception.

Exemple avec plusieurs gestionnaire d'exception

```
public class Except03 {
public static int [] suite( int n) throws TailleTableauNegative {
int tab[];
if (n < 0) throw new TailleTableauNegative (); tab = new int [n];
 for(int i = 0; i < tab.length; i++) tab[i] = i;
   return tab; }
 public static float division(float a, float b) throws DivisionParZero
    if (b = = 0) throw new DivisionParZero();
  return a / b;
 public static void main(String[] args) {
  try { System.out.println(suite(-5)); System.out.println(division(1,2));
  catch (TailleTableauNegative ex)
    { System.out.println("erreur: "+ex.getMessage());
  catch(DivisionParZero e)
  {System.out.println("message d'erreur: "+e.getMessage());
  } }}
class TailleTableauNegative extends Exception{ }
class DivisionParZero extends Exception PQNG. FST/UCAD
```

Commentaires



Dans cet exemple, le choix du bon gestionnaire est toujours réalisé en examinant le type de l'objet transmis au bloc catch (se rappeler que lancer une exception c'est produire un objet de type classe d'exception.

Si, en parcourant un bloc try une exception est rencontrée mais n'est traitée par aucun bloc catch, alors c'est la classe standard de l'exception (faisant partie de l'API) qui est invoquée par le compilateur pour la traiter.

En réalité, lorsque vous créez des classes d'exception, vous ne faites que personnaliser une classe d'exception de l'API dans le but d'avoir des informations plus parlantes quant à la nature de l'exception.

Aussi, on n'est pas sensé, à priori, connaître les classes de bases de l'API pour la gestion des exceptions (exemple savoir qu'il y a une classe ArithmeticException qui gère les erreurs dues à des calculs algébriques impossibles ou erronées). Mais, il faut savoir qu'il y a une super classe Exception qui englobe la gestion de toutes formes d'exception (et il est toujours possible de dériver de cette classe pour gérer une exception quelconque).



Les exceptions standards (API)

ServerNotActiveException

Exception de serveur non actif pour une opération à distance.

SQLException:

Exception SQL: Structure Query Language (BatchUpdateException, SQLWarning).

No Such Method Exception

Exception de méthodes introuvables.

ClassNotFoundException

Exception de classe chargée avec un nom erroné.

 ${\bf BadLocation Exception}$

Exception de mauvaise localisations d'une ressource.

Exemple avec exceptions de l'API

```
public class Except03 {
public static int [] suite(int n) throws NegativeArraySizeException {
int tab[] = new int [n];
if (n < 0) throw new NegativeArraySizeException();
 for (int i = 0; i < tab.length; i++) tab[i] = i;
  return tab; }
public static float division (float a, float b) throws ArithmeticException
    if (b = = 0) throw new ArithmeticException();
                                                                Pile d'exécution
  return a / b;
                                                                du programme
 public static void main(String[] args) {
  try { System.out.println(suite(-5)); System.out.println(division(1,2));
  catch (NegativeArraySizeException ex)
   { System.out.println("erreur: "+ex.getMessage()); ex.printStackTrace(); }
  catch(ArithmeticException e)
  {System.out.println("message d'erreur: "+e.getMessage());
  }
```

Méthode déclenchant et traitant une exception

4

Dans l'exemple précèdent, les exceptions déclenchées dans les méthodes suite et division sont traitées dans la méthode main. Le traitement est dans ce cas <u>différé</u>.

Voyons maintenant comment déclencher et traiter simultanément ces mêmes exceptions dans les dites méthodes.

Exemple de déclenchement et traitement simultanés

```
public class Except03 {
public static int [] suite( int n) {
int tab[] = null;
try { tab = new int [n];
       for (int i = 0; i < tab.length; i++) tab[i] = i;
catch (NegativeArraySizeException e) { System.out .println ("Erreur sur la taille");
                 e.printStackTrace(); }
return tab; }
public static float division (float a, float b)
    float resultat = 0;
                  resultat = a / b;
    try {
    catch (ArithmeticException ex) { System.out .println("Erreur arithmetique");
                 ex.printStackTrace();
    return resultat; }
 public static void main(String[] args) {
  System.out.println(suite (-5)); System.out.printhp(division(1,2));
                                                                                  293
```



Threads Java

Un thread est un ''processus léger'' en cours d'exécution dans un programme. Cette unité d'exécution fonctionne de façon autonome parallèlement à d'autres threads (ie à d'autres traitements).

Les threads sont donc des traitements qui vivent au sein d'un même processus.

Dans un système *monoprocesseur,à* chaque tâche, il est attribué des *quantum* de temps au cours desquels les ressources systèmes leur sont entièrement données pour accomplir leurs traitements.

Les threads partagent la même mémoire contrairement aux processus.

Le grand avantage de l'utilisation des threads est de pouvoir répartir différents traitements d'un même programme en plusieurs processus distincts pour permettre leur exécution « simultanée ».

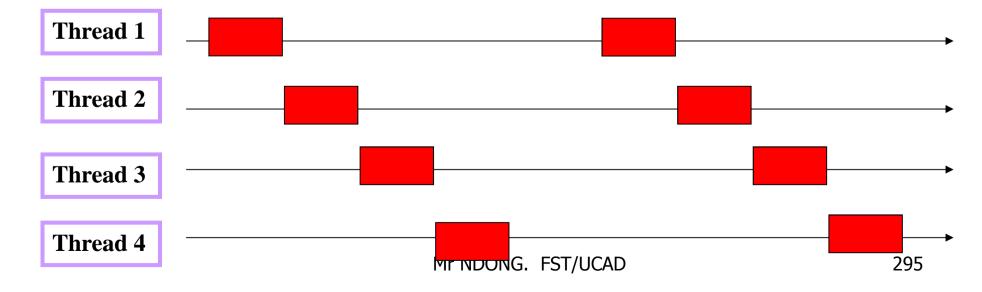
Donc l'idée des threads est de partager le temps CPU entre plusieurs tâches que le programme effectue simultanément.

La classe java.lang.Thread et l'interface java.lang.Runnable sont les bases du développement des threads.

Mr NDONG. FST/UCAD

Principes de base des threads

Les threads peuvent être créés comme instance d'une classe dérivée de la classe Thread. Elles sont lancées par la méthode start () (pour allouer les ressources système nécessaires), qui demande à l' <u>Ordonnanceur</u> de threads de lancer la méthode run () du thread. La méthode run () doit être nécessairement implantée dans le programme. Le schéma ci-après illustre les temps d'exécution et de latence des threads:



Types de threads



En Java, on distingue deux grandes catégories de threads: ceux dits utilisateurs et les démons.

Les threads utilisateurs se terminent lorsque les instructions dans le corps de leur méthode run () sont <u>toutes</u> exécutées.

Un thread démon continue <u>indéfiniment</u> si aucune précaution n'a été prise pour l'arrêter.

Nous commencerons par étudier les threads dits *utilisateurs* pour terminer sur un bref aperçu portant sur les threads *démons*.

Cycles de vie d'un Thread (1/2)



Un thread peut être dans 4 états différents:

État nouveau

C'est l'état initial après l'instanciation du thread. Le thread est opérationnel mais n'est pas encore actif. Un thread prend toujours cet état après son instanciation.

État exécutable

Un thread est dans cet état à partir du moment où il a été lancé par la méthode *start ()* et le reste tant qu'il n'est pas sorti de la méthode *run ()*. Le système donnera du temps d'exécution à votre thread dès qu'il le pourra.

État en attente

Il s'agit de thread qui n'exécute aucun traitement et ne consomme aucune ressource CPU. Il existe plusieurs manières de mettre un thread en attente:

- appeler la méthode Thread.sleep (long temps_en_millisecondes)
- appeler la méthode wait ()
- appeler une ressource bloquante (flux, accès base de données, 297)
- accéder à une instance sur laquelle un <u>verrou</u> a été posé





Il s'agit d'un thread qui est sorti de sa méthode *run* () soit de façon naturelle, soit de manière subite (exception non interceptée).

Regardons maintenant comment créer des threads avec l'instanciation d'objets de la classe java.lang.Thread.

Nous verrons par la suite comment créer également un thread avec *l'interface* Runnable. Cette dernière permettra de créer des threads d'une classe qui dérive déjà d'une autre classe (l'héritage multiple étant interdit en Java).

En programmation WEB (avec les applets) il faudra recourir à cette interface.





La classe Thread dispose de plusieurs constructeurs que vous pouvez utiliser aisément pour instancier des objets threadés:

```
/*crée un Thread dont le nom est généré automatiquement (aléatoirement)*/
public Thread ()
/*target est le nom de l'objet dont la méthode run () est utilisée pour
lancer le thread*/
public Thread (Runnable target )
/*on précise l'objet et le nom du thread*/
public Thread (Runnable target, String name)
/*on ne précise que le nom du thread*/
public Thread (String name)
```

Il existe aussi des constructeurs de groupe de Threads que nous allons voir plus tard.

Méthodes de la classe Thread



```
/*met fin au thread brutalement, à n'utiliser qu'en dernier recours*/
void destroy ()
/*renvoie la priorité du thread*/
int getPriority()
/*retourne le nom du thread*/
String getName()
/*pour interrompre le thread*/
void interrupt()
/*teste si le thread courant a été interrompu*/
static boolean interrupted ( )
/*attendre la mort du thread*/
void join () | void join (long millis) | void join (long millis, int nanos)
/*redémarre le thread :cette méthode est DEPRECIEE*/
void resume ()
/*méthode contenant le code à exécuter par le thread*/
void run ()
```

Méthodes de la classe Thread

```
/*changer la priorité du thread*/
void setPriority (int newpriotity)
/*mettre en veille le thread*/
static void sleep (long millis) | static void sleep (long millis, int nanos)
/*démarre l'exécution du thread*/
void start ()
/*renvoie le nom du thread, sa priorité et le groupe auquel il appartient*/
String toString ()
/*renvoie un booléen qui indique si le thread est actif ou non*/
boolean isAlive ()
/*renvoie un objet qui encapsule le groupe auquel le thread appartient*/
ThreadGroup getThreadGroup ()
/*indique à l'interpréteur que le thread peut être suspendu pour permettre à
d'autres threads de s'exécuter*/
void yield ( )
```

Premier Thread avec java.lang.Thread

(programme simple qui simule l'exécution de deux threads)

```
public class FirstThread
                         extends Thread
                                                                           sortie
   FirstThread (String name) {
                                                                          0 thread 1
        super (name );
                                                                            thread 2
                                                                            thread 1
                        { // code à exécuter par chaque thread
  public void | run ( )
                                                                            thread 2
   try { for (int i = 0; i < 5; i++)
                                                                          2 thread 1
         { System.out .println (i+" "+ this.getName ());
                                                                           thread 2
          Thread.sleep ((int) Math.random ()*10);// mise en attente
                                                                          3 thread 1
                                                                           thread 2
                                                                           thread 1
   catch (InterruptedException e) { }
                                                                          4 thread 2
 public static void main(String [ ] args) {
 new FirstThread("thread 1").start(); //le code lancé par start() est le code de run()
 new FirstThread("thread 2").start();
                                      Mr NDONG. FST/UCAD
                                                                                   302
```



Un appel de la méthode start () dans une instruction de la forme new FirstThread("thread 1").start (); assure qu'un thread sera bien pris en compte par la machine virtuelle et par le système d'exploitation <u>puis lance l'exécution</u> de la méthode run () de l'objet thread correspondant.

L'usage de la méthode statique sleep (long millis) nous permet de voir que les deux threads s'exécutent en apparente simultaneité.

Cette méthode peut lever une exception de type InterruptedException qu'il faut donc intercepter et capturer.

La méthode start () ne peut être appelée <u>q'une seule fois</u> pour un thread donné, sinon une exception de type <u>IllegalThreadStateException</u> est levée.

Il est possible d'appeler la méthode *run* () pour chaque thread mais cela entraîne l'exécution complète du *thread 1* puis celle complète du *thread 2*. l'appel de *sleep* entraînerait alors l'exécution d'autres threads autres que ceux –ci, donc ralentissement de l'exécution de notre programme. Mr NDONG. FST/UCAD 303

Deuxième Thread avec java.lang.Runnable

(le premier exemple réalisé ici avec l'interface Runnable)

```
implements Runnable
public class SecondThread
    String name;
SecondThread(String name){
                                this.name = name );
                        { // code à exécuter par chaque thread
  public void
   try { for ( int i = 0; i < 5; i++)
         { System.out .println (i+" "+ this.name );
          Thead.sleep ((int) Math.random ()*10); // mise en attente
   catch (InterruptedException e) { }
 public static void main(String [ ] args) {
SecondThread t1 = new SecondThread("thread 1");
SecondThread t2 = new SecondThread("thread 2");
 Thread thread 1 = \text{new} Thread (t1);
                                       thread1.start();
 Thread thread 2 = new Thread (t2); Mthread 2.startucas }
```

sortie

```
0 thread 1
0 thread 2
1 thread 1
1 thread 2
2 thread 1
2 thread 2
3 thread 1
3 thread 2
4 thread 1
4 thread 2
```



Avec cet deuxième exemple, pour lancer l'exécution d'un thread, nous sommes dans l'obligation d'instancier un objet de la classe implémentant Runnable:

SecondThread objetRun = new SecondThread ("thread 1");. 1

Mais cet objet n'est pas de type Thread, c'est pourquoi on ne peut pas lui appliquer directement la méthode start (), il faut absolument l'enroller dans un autre objet de type Thread pour pouvoir lancer son exécution via l'appel de start ().

```
Thread thread1 = new Thread (objetRun );
thread1.start();
```

Mais il est tout à fait possible, à partir de 1 de faire objetRun.start (). Il faut alors au préalable implémenter la méthode start () dans la classe comme ceci:

```
public void start () {
Thread t = new Thread (this);
t.start ();}
```

Interruption des Threads

(bloquer un thread pour laisser un autre continuer)

```
public class TestInterrupt extends Thread {
long attente;
TestInterrupt (String name, long attente)
{ super(name); this.attente = attente;}
public void run( ){
try { for (int i = 0; i < 10; i++)
{ System.out .println (i+ " "+this.getName ( ) ) ; Thread.sleep (attente) ; 5 thread 1
if (i = 4)
 {Thread.currentThread()..interrupt();
boolean try = Thread.currentThread().isInterrupted();
System.out .println (this.getName() +" "+"interrompu "+ trv);
          if (Thread.currentThread()..isInterrupted())
     { this.interrupted ( );
     System.out .println (this.getName () +" "+"redémarré");
   catch (InterruptedException e) { }
 public static void main(String[] args) {
new TestInterrupt("thread 1",5).start();
new TestInterrupt("thread 2",10),start(); }}
```

```
0 thread 1
0 thread 2
1 thread 1
1 thread 2
2 thread 1
3 thread 1
2 thread 2
4 thread 1
thread 1 interrompu true
3 thread 2
thread 1 redémarré
6 thread 1
7 thread 1
4 thread 2
8 thread 1
thread 2 interrompu true
9 thread 1
thread 2 redémarré
5 thread 2
6 thread 2
7 thread 2
8 thread 2
9 thread 2
                     306
```





L'interruption d'un thread se contente d'attirer son attention.

L'appel de la méthode *interrupt* () n'entraîne pas automatiquement l'interruption du thread *courant*. Il ne s'agit qu'une demande.

Cette méthode entre en vigueur que si le thread entre en sommeil ou et en attente (via l'appel de *sleep* (...) ou wait (...)).

Comme la méthode *interrupt()* ne fait que placer un indicateur de demande d'arrêt, il est tout à fait bon de connaître l'état de celui-ci à un instant donné en appelant la méthode *isInterrupted()*. Si la méthode *interrupt()* a été appelée pendant que le thread n'était pas en sommeil ou en attente, aucune

InterruptedException ne peut être générée.

La méthode non statique *isInterrupted* () permet de voir si le thread *correspondant* est effectivement interrompue et alors l'appel de *interrupted*() repositionne l'indicateur à <u>false</u> (pour redémarrer le thread).



Notes importantes (2/2)

Si un thread est bloqué, il ne peut pas déterminer s'il est interrompu. C'est à ce moment que la classe *InterruptedException* intervient. Lorsque la méthode *interrupt* est appelée sur un thread bloqué, l'appel bloquant (comme *sleep* ou *wait*) est terminé par une *InterruptedException*.

Pour savoir si un thread est couramment actif c'est-à-dire qu'il est soit exécutable, soit bloqué, utilisez la méthode boolean isAlive (). Elle renvoie true si le thread est exécutable ou bloqué et false si le thread est nouveau et pas encore exécutable ou si le thread est mort.

Vous ne pouvez pas déterminer si un thread actif est exécutable ou bloqué ou si un thread exécutable est réellement en cours d'exécution. Vous ne pouvez pas non plus faire la différence entre un thread qui n'est pas encore exécutable et un thread qui est déjà mort.



java.lang.ThreadGroup

Par défaut un thread appartient (est créé) au groupe (de threads) *courant* càd celui qui l'a créé.

Il faut savoir que le premier thread que l'on rencontre est la méthode *main*. Par défaut donc, un thread est créé dans ce groupe.

Mais il est possible de créer des groupes de threads autre que le groupe courant, et à chacun, associé un certain nombre de threads.

Dans ce cas, il sera plus facile d'interrompre un ensemble de threads, en interrompant simplement le groupe.

Pour cela, on crée une instance de la classe ThreadGroup avec l'un des constructeurs:

ThreadGroup (String name) // groupe de threads de nom name
ThreadGroup (ThreadGroup parent, String name) // sous-groupe d' un autre groupe
L'ajout d'un thread au groupe peut se faire avec le constructeur:
Thread (ThreadGroup g, String name)





Un thread démon est un thread qui n'a aucun autre but que de servir d'autres threads. L'exécution de tels threads peut se poursuivre même après l'arrêt de l'application qui les a lancés. On dit qu'ils s'exécutent <u>en tâche de fond</u>.

Une application dans laquelle <u>les seuls threads actifs</u> sont des <u>démons</u> est <u>automatiquement fermée</u>.

Un thread doit <u>toujours</u> être créé comme thread <u>standard</u>, puis il peut être transformé en thread démon grâce à un appel de la méthode <u>setDaemon</u> (<u>true</u>). Mais cet appel doit se faire <u>avant le lancement du thread</u> sinon une exception de type <u>IllegalThreadStateException</u> est levée.

Un thread démon dépend du thread parent qui l'a lancé et s'exécute en arrière plan de ce dernier.

Synchronisation de Threads

(utilisation d'un moniteur (ou sémaphore) pour l'accès à une ressource partagée)

La synchronisation permet de gérer les accès concurrents concernant la manipulation simultanée de données partagées. Elle permet de sauvegarder l'intégrité des données.

La synchronisation peut être analysée à deux niveaux:

- le premier niveau a trait à la manipulation de données partagées dans un bloc
- le second niveau permet de sécuriser l'accès au code d'une méthode

La synchronisation utilise la notion de verrou: à un instant donné, une seule méthode synchronisée peut accéder à un objet donné.

Ceci est dû au fait que pour chaque objet doté d'une méthode synchronisée, le système pose un verrou (ou clé) unique permettant l'accès à cet objet.

Le verrou est attribué à la méthode (ou bloc) synchronisé et restitué à la sortie de celui-ci.

Tant que le verrou n'est pas repris par l'environnement, aucune autre méthode synchronisée ne peut le recevoir et donc ne peut manipuler l'objet associé à l'appel de cette méthode.

Synchronisation de Threads: le Moniteur

(Attribution du verrou par le moniteur)

Le *moniteur* est utilisé pour synchroniser l'accès à une ressource partagée (qui peut être un segment de code donné). Un thread accède à cette ressource par l'intermédiaire de ce moniteur.

Ce dernier est attribué à un seul thread à la fois.

Pendant que le thread exécute la ressource partagée aucun autre thread ne peut y accéder.

Le thread libère le moniteur dès qu'il a terminé l'exécution du code synchronisé ou bien s'il a fait appel à la méthode wait () de l'objet.

Il faut faire très attention aux méthode <u>statiques</u>. Synchroniser de telles méthodes revient à bloquer le moniteur de la classe ce qui peut être source de performances médiocres dans certains cas, puisque vous bloquez l'accès à toute la classe.

Un thread détient le moniteur s'il exécute une méthode synchronisée, un bloc synchronisé ou une méthode statique synchronisée d'une classe.





Ici, il s'agit d'interdire à deux threads différents d'accéder simultanément à un même objet, en plaçant le mot clé synchronized pour le bloc concerné.

Dans une instruction telle que:

synchronized (expression) { // du code}

où <u>expression</u> repère un <u>objet</u> quelconque, le système Java pose un <u>verrou</u> sur cet objet pendant l'exécution du bloc. Aucun autre bloc <u>synchronized</u> sur cet objet APPARTENANT A UN AUTRE THREAD ne peut être exécuté et un tel bloc ne peut être lancé qu'après avoir obtenu ce <u>verrou</u>.

On définit ainsi une section critique.



Considérons une classe Compte permettant de gérer les comptes de clients dans une quelconque banque disposant de plusieurs agences.

Pour une gestion efficace et sécurisée des comptes client, il ne faudrait pas permettre par exemple que deux (au moins) transactions s'effectuent simultanément sur un même compte à un instant donné (exemple: le retrait et le dépôt).

Les opérations de retrait et de dépôt sont gérées séparément par deux threads distincts.

Notre travail sera de créer ces deux threads de telle sorte qu'ils ne pourront jamais s' exécuter simultanément sur un même compte.

Premier exemple Synchronisation de bloc (2/5)

(Compte.java)

```
package thread.compte;
        public class Compte {
                 private double solde;
                 private double decouvert;
             public Compte(double solde, double decouvert){
                            this.solde = solde;
                            this.decouvert = decouvert;
             public void deposer (double montant){
                           this.solde += montant;
             public void retirer (double montant){
                           if (montant + decouvert <= solde)</pre>
                                    this.solde -= montant;
            public double getSolde(){
                        return solde;
 } }
```

Premier exemple Synchronisation de bloc (3/5)

(ThreadCompteDepot .java)

```
package thread.compte;
public class ThreadCompteDepot extends Thread {
 private Compte c;
 private double depot;
 public ThreadCompteDepot (String name, Compte c, double depot){
 super (name);
 this.c =c;
 this.depot = depot;
 public void run ( ){
  synchronized (c) { // fondamental: on pose un verrou sur le compte
   try {
   System.out.print (this.getName () + ":avant le depot: "); c.imprimeHistorique();
    c.deposer (this.depot);
    Thread.sleep (1000);
    System.out.print (this.getName() + ":apres le depot: "); c.imprimeHistorique();
                                        Mr.NDONG. FST/UCAD
    System.out.println("retrait avorte"); } } }}
   catch (InterruptedException e) {
```

Premier exemple Synchronisation de bloc (4/5)

(ThreadCompteRetrait.java)

```
package thread.compte;
public class ThreadCompteRetrait extends Thread {
 private Compte c;
 private double retrait;
 public ThreadCompteRetrait (String name, Compte c, double retrait){
super (name);
this.c = c;
 this.retrait = retrait;
public void run ( ){
  synchronized (c) { // fondamental: on pose un verrou sur le compte
   try {
   System.out.print (this.getName () + ":avant le retrait:"); S.O.P(c.getSolde ());
    c.retirer (this.retrait);
    Thread.sleep (1000);
    System.out.print (this.getName() + ":apres le retrait:"); S.O.P(c.getSolde ());
                                        System.out.println (" depot avorte"); } }} 317
   catch (InterruptedException e) {
```

Premier exemple Synchronisation de bloc (5/5)

(TestThreadsCompte .java)

```
Le choix du thread à exécuter
package thread.compte;
                                              en premier lieu est aléatoire,
                                              les deux threads étant de
public class TestThreadsCompte {
                                              même priorité.
 public static void main(String [ ] args)
Compte u = new Compte (5000,100);
ThreadCompteRetrait tcr = new ThreadCompteRetrait ("retrait",u,2000);
ThreadCompteDepot tcd = new ThreadCompteDepot ("depot",u,1500);
tcr.start();
                                 retrait:
                                          avant le retrait votre solde est: 5000.0
tcd.start(); }
                         BON
                                 retrait:
                                          apres le retrait votre solde est: 3000.0
                                 depot:
                                          avant le depot: votre solde est: 3000.0
                                 depot:
                                          apres le depot: votre solde est: 4500.0
```

retrait:avant le retrait votre solde est: 5000.0 depot:avant le depot: votre solde est: 3000.0 retrait: apres le retrait votre solde est: 4500.0 depot:apres le depot: votre solde est: 4500.0

Si vous ne synchroniser pas le Compte dans les deux threads, vous aurez un solde erroné.

Deuxième Exemple de bloc synchronisé

On considère une classe qui permet d'inverser les éléments d'un tableau d'entiers. Mais avant que l'inversion ne se fasse, les éléments du tableau doivent être *incrémentés* d'une valeur égale à l'indice de l'élément en cours. Et après, l'inversion pourra se faire après un certain délai.

On disposera d'une méthode *affiche* qui nous permettra d'envoyer sur la console les éléments du tableau inversé.

Mais ici, nous avons un problème à gérer:

Avant que la méthode *affiche* n'accède au tableau pour l'afficher, il faudra que la méthode qui se charge de l'incrémentation et de l'inversion finisse carrément son travail, sinon l'affichage sera complètement faux.

Regardons maintenant comment on peut passer d'un mauvais exemple vers un cas où les données seront correctes.

Exemple de bloc NON synchronisé (1/4)

(la classe qui gère l'inversion et l'affichage du tableau)

```
class TabInverse {
int t [ ];
int [ ] inverse (int tableau [ ])
{ t = new int [tableau.length];
for (int i = 0; i < tableau.length; i++) tableau [i]+=i;
 try {Thread.sleep (1000);} // pour marquer une pause entre l'incrémentation
 catch (InterruptedException er) { }// et l'inversion des éléments du tableau
   for (int i = 0;i < tableau.length ;i++)</pre>
    t [tableau.length -i-1] = tableau[i];
return t;
void affiche ( ){
for (int i = 0 < t.length; i++)
System.out .print (t[i]+"::::");
```

Exemple de bloc NON synchronisé (2/4)

```
/*cette classe permet de créer un thread qui n'accédera qu'à la methode inverse
Cet objet peut donc manipuler simultanément le tableau qu'un autre thread */
class EssaiSynchroInverse extends Thread{
TabInverse inv:
int tab[];
 public EssaiSynchroInverse (String name, TabInverse inv, int tab [])
 { super (name);
  this.inv = inv;
  this.tab = tab;
 public void run ( )
 {System.out .println (this.getName ());
  inv.inverse (tab);
  try {Thread.sleep (1000);}
  catch (InterruptedException er) { }
  System.out .println("FIN de "+this.getName());
}}
                                   Mr NDONG. FST/UCAD
```

Exemple de bloc NON synchronisé (3/4)

```
/*cette classe permet de créer un thread qui n'accédera qu'à la methode affiche
Cet objet peut donc manipuler simultanément le tableau qu'un autre thread */
class EssaiSynchroAffiche extends Thread{
TabInverse inv;
int tab[];
 public EssaiSynchroAffiche (String name, TabInverse inv, int tab[] )
  { super (name);
   this.inv = inv;
   this.tab = tab;
  public void run ( )
  { System.out .println (this.getName ( ) );
   inv.affiche();
   try {Thread.sleep(1000);}
   catch (InterruptedException er) { }
   System.out .println ("FINITION de "+ this.getName ());
                                Mr NDONG. FST/UCAD
}}
```

Exemple de bloc NON synchronisé (4/4)



```
/*pour tester l'accès simultané à un même objet tableau*/
public class TestSynchroBloc {
    static int t [] = {1,2,3,4,5};
    public static void main (String [] args) throws Exception{
        ThreadAff
        FINITION de ThreadAff
        TabInverse ti = new TabInverse ();
```

EssaiSynchroInverse es = new EssaiSynchroInverse("ThreadInv",ti, t); EssaiSynchroAffiche ess = new EssaiSynchroAffiche("ThreadAff",ti, t);

```
es.start();
es.join()
ess.start();
}
```

Ici, on crée deux threads <u>es</u> et <u>ess</u> qui accèdent au tableau t simultanément. Le thread <u>ess</u> va vouloir afficher un résultat qui n'est pas encore entièrement connu, puisque <u>es</u> n'a pas terminé l'incrémentation et l'inversion.

Le résultat affiché sera donc erroné.

Deuxième Exemple de bloc synchronisé

(voici comment il faut implémenter la classe TabInverse)

Résultat attendu

```
class TabInverse {
int t [ ];
                                                          ThreadInv
int [ ] inverse (int tableau [ ])
                                                          ThreadAff
{ t = new int [tableau.length];
                                                           9::7::5::3::1::
                                                          FINITION de ThreadAff
synchronized (t) {
                             // verrouillage du tableau
                                                          FIN de ThreadInv
for (int i = 0; i < tableau.length ;i++) tableau [i]+=i;
      {Thread.sleep (1000);}
 catch (InterruptedException er) { }
   for (int i = 0;i < tableau.length ;i++)</pre>
    t [tableau.length -i-1] = tableau[i];
return t;}
void affiche ( ){
synchronized (t) {
for (int i = 0 < t.length; i++)
                                 System.out .print (t[i]+"::::");
                                                                                  324
    }}
```





Chaque fois que deux threads s'exécutent en même temps, il faut souvent prendre des mesures adéquates pour qu'ils n'accèdent pas simultanément à une même variable.

Le principe <u>d'exclusion mutuelle</u> doit être assuré sur le partage simultané d'objet (pour assurer la cohérence des données) par l'utilisation de méthodes dites <u>synchronisées</u>. Ce principe est assuré lorsqu'une méthode est déclarée avec le mot clé <u>synchronized</u>. Une méthode synchronisée appartient à un <u>objet quelconque</u>, pas forcément à un thread.

Lorsqu'une méthode déclarée *synchronized*, est en cours d'exécution par un thread, tous les autres threads qui en auraient besoin doivent attendre la *fin* de son exécution.

Lorsque *synchronized* est utilisé comme modifieur de méthode d'instance, une instruction telle que: **synchronized** void method () {......} est équivalente à:

Exemple de Synchronisation de méthodes (1/4)

Considérons un exemple qui permet de réaliser deux opérations (addition et affichage) sur deux champs d'une instances d'un objet (syn) d'une classe *Synchro*. Il s'agit d'incrémenter la valeur du premier champ (de 1) et de faire la somme avec le deuxième champ. On souhaite que les deux champs sont accédés dans les deux méthodes et utilisés de façon concurrente par trois threads que nous créerons dans le *main*.

Regardons d'abord comment les valeurs des deux champs sont érronées et incohérentes si l'exclusion mutuelle n'est pas bien gérée: l'incohérence s'explique par le fait que le s trois threads manipulent les deux champs pèle mêle.

Nous verrons alors une version qui dégage une utilisation correcte de la valeur de ces variables.

Exemple de Synchronisation de méthodes (2/4)



```
class Synchro {
int n, som;
 public Synchro (int n,int som) { this.n =n; this.som =som;
 void addition () { // methode non synchronisée
System.out .print ("n++="+(n++) +" suivi de ");
 try { Thread.sleep (222) ;}
 catch (InterruptedException t){ }
 som += n; System.out .println(" et som="+som) ;
 void affiche () { // methode non synchronisée
 System.out .print("affiche: n=" +(++n)+" et ");
 try {Thread.sleep (222);}
 catch (InterruptedException t){ }
 System.out.println ("affiche :som= "+som);
                                    Mr NDONG. FST/UCAD
```

Exemple de Synchronisation de méthodes (3/4)

```
class TestAddition extends Thread {
Synchro syn;
public TestAddition (String t, Synchro syn) {super (t); this.syn =syn;}
public void run ( ){ System.out.print (this.getName ( )+" " );
             try { syn.addition ( ); Thread.sleep(522); }
             catch (InterruptedException er){ }
             System.out.println("FIN "+this.getName());
    } // fin de TestAddition
class TestAffiche extends Thread{
Synchro syn;
public TestAffiche (String t, Synchro syn){super (t); this.syn =syn;}
public void run (){     System.out.println("******"+this.getName ()+"******");
             try { syn.affiche ( ); Thread.sleep(52); }
             catch (InterruptedException er) { }
             System.out.println("fin "+this.getName()); }
} // fin de TestAffiche
                                 Mr NDONG. FST/UCAD
                                                                              328
```

Exemple de Synchronisation de méthodes (4/4)



```
public class TestSynchro {
  public static void main (String [ ] args) {
    Synchro sy = new Synchro (1,1);
    TestAffiche taf = new TestAffiche (" ThreadAffiche",sy);
    TestAddition tad1 = new TestAddition ("ThreadAdd1",sy);
    TestAddition tad2 = new TestAddition ("ThreadAdd2", sy);
    TestAddition tad3 = new TestAddition ("ThreadAdd3", sy);
    tad1.start ( );
    tad2.start ( );
    tad3.start ( );
    tad3.start ( );
}
```

Sortie de l'exemple

Pour rectifier le problème encouru, il faudra synchroniser les méthodes addition() et affiche() de la classe Synchro, en les déclarant avec le mot clé synchronized.

```
ThreadAdd1 n++=1
                                               ThreadAdd1 n++=1
suivi de ThreadAdd2 n++=2
                                               suivi de ThreadAdd2
                                               ***** ThreadAffiche*****
suivi de ***** ThreadAffiche*****
                                                                           exact
affiche: ++n= 4 et ThreadAdd3 n++= 4 suivi de
                                               ThreadAdd3 (et som = 3)
                                               n++= 2 suivi de et som=6
et som = 6
                                               affiche: ++n= 4 et affiche :som= 6
et som=11
                                               n++=4 suivi de
affiche :som= 11
                                               fin ThreadAffiche
                         Erronée: on s'
                                               FIN ThreadAdd1
et som = 16
                         attendait à avoir 3
fin ThreadAffiche
                                               et som=11
FIN ThreadAdd1
                                               FIN ThreadAdd2
FIN ThreadAdd2
                                               FIN ThreadAdd3
FIN ThreadAdd3
                                    Mr NDONG. FST/UCAD
                                                                             330
```

Amélioration de l'exemple

```
/*pour que les données ne sont plus érronées et falsifiées*/
class Synchro {
int n, som;
 public Synchro (int n, int som) { this.n =n; this.som =som;
   synchronized void addition () { // methode synchronisée, donc bloquant
System.out .print ("n++= "+n++ +" suivi de ");
 try { Thread.sleep(222) ;}
 catch (InterruptedException t){ }
 som+=n; System.out .println(" et som="+som);
   synchronized void affiche () { // methode synchronisée, donc bloquant
 System.out .print("affiche: n= " +(++n)+" et ");
 try {Thread.sleep (222);}
 catch (InterruptedException t){ }
 System.out.println ("affiche :som= "+som);
                                   Mr NDONG. FST/UCAD
```

Attente et notification:



les méthodes wait () et notifyAll ()

(ces méthodes doivent êtres lancées dans des blocs ou méthodes synchronisés)

Attention: c'est des méthodes de la super classe Object et non de la classe Thread.

Elles implantent des mécanismes de <u>demande</u> de verrou et d'avertissement de <u>libération</u> de ce verrou.

une méthode *synchronisée* peut appeler la méthode *wait*() de l'objet dont elle possède le verrou, pour:

- rendre le verrou à l'environnement qui peut alors le donner à une autre méthode synchronisée,
- mettre en attente le thread correspondant.

Une méthode *synchronisée* peut appeler la méthode *notifyAll* () d'un objet afin de prévenir tous les threads en attente sur cet objet et de leur donner la possibilité de s'exécuter.

NB: notifyAll() permet de débloquer un wait () sur un objet où notify() a été lancé.

La méthode notify () prévient <u>un seul</u> thread. Avant de faire un notify () ou notifyAll() il faut changer la condition de boucle qui mène au wait ().



Exemple: producteur - consommateur. (1/4)

Un <u>producteur</u> est un thread qui dépose des jetons numérotés dans un <u>chapeau</u> qui ne peut contenir <u>qu'un seul jeton</u>. Un <u>consommateur</u> prend ce jeton qui doit être présent dans le chapeau. Donc:

- le *producteur* doit s'arrêter de déposer des jetons lorsqu'il y en a déjà un et doit être informé qu'un jeton a été retiré.
- le *consommateur* ne peut pas prendre de jetons s'il n'y en a pas (arrêt du consommateur) et doit être informé lorsqu'un jeton a été déposé.

L'objet le plus à même pour avertir le *producteur* et le *consommateur* est le *chapeau* lui-même.

Exemple: producteur - consommateur. (2/4)

(fichier **Producteur.java**)

```
public class Producteur extends Thread{
  private Chapeau chapeau;
  private int number; // le numéro du jeton à déposer
 public Producteur (Chapeau chapeau, int number) {
        this.chapeau = chapeau;
        this.number = number;
public void run ( ){
for (int i = 0; i < 5; i++) {
     chapeau.put (i);
     System.out .println ("le producteur N° "+this.number +" a mis "+i);
     try { Thread.sleep(1000);}
     catch (InterruptedException e){ }
```

Exemple: producteur - consommateur. (3/4)

(fichier Consommateur.java)

```
public class Consommateur extends Thread {
private Chapeau chapeau;
private int number;
  public Consommateur (Chapeau chapeau, int number) {
     this.chapeau = chapeau;
     this.number = number;
public void run( ){
  int value = 0;
    for (int i = 0; i < 5; i++) {
       value = chapeau.get ( );
       System.out .println ("le consommateur N° "+this.number +" a pris "+value);
       try {Thread.sleep (1000);}
       catch (InterruptedException e){ }
                                      Mr NDONG. FST/UCAD
                                                                                  335
```

Exemple: producteur - consommateur. (4/4)



```
public class Chapeau {
private int contenu;
private boolean permis = false;
public synchronized int get ( ){
while (permis == false)
{ try {wait (); // rendre le verrou }
 catch (InterruptedException e){ }
permis = false; notifyAll();
return contenu; }
public synchronized void put (int value){
while (permis == true)
 { try { wait ( ); }
  catch (InterruptedException er) { } }
 contenu = value; permis = true;
 notifyAll ();// attribuer le verrou à un autre
             // qui peut alors s'exécuter
```

Exemple: producteur – consommateur: sortie

Voici la sortie du programme précédent avec un <u>seul</u> producteur et un <u>seul</u> consommateur.

```
le producteur N° 1 a mis 0
le consommateur N° 1 a pris 0
le producteur N° 1 a mis 1
le consommateur N° 1 a pris 1
le producteur N° 1 a mis 2
le consommateur N° 1 a pris 2
le producteur N° 1 a mis 3
le consommateur N° 1 a pris 3
le producteur N° 1 a mis 4
le consommateur N° 1 a pris 4
```

Exercice: producteur – consommateur

Reprendre l'exemple précédent en créant trois (3) producteurs et trois (3) consommateurs. Il y a un truc à faire.

```
/*une classe de test pour 3 producteurs et 3 consommateurs*/
public class Test3ProductCons {
    public static void main(String [ ] args) {
        Chapeau chap = new Chapeau ( );
        Producteur2 p1 = new Producteur2 (chap,1);
        Producteur2 p2 = new Producteur2 (chap, 2);
        Producteur2 p3 = new Producteur2 (chap, 3);
        Consommateur2 c1 = new Consommateur2 (chap, 1);
        Consommateur2 c2 = new Consommateur2 (chap, 2);
        Consommateur2 c3 = new Consommateur2 (chap, 3);
        p1.start ( ) ; p2.start ( ) ; p3.start ( ) ;
        c1.start ( );c2.start ( ); c3.start ( );
```

Exercice: producteur – consommateur: Corrigé (1/2)

(on ne modifie que le producteur et le consommateur)

(fichier Producteur2.java)

```
public class Producteur2 extends Thread{
  private Chapeau chapeau;
  private int number;
 public Producteur2 (Chapeau chapeau, int number) {
        this.chapeau = chapeau;
        this.number = number;
public void run ( ){
synchronized (chapeau) {// il faut absolument synchroniser ce bloc
for (int i = 0; i < 5; i++) {
     chapeau.put (i);
     System.out .println ("le producteur N° "+this.number +" a mis "+i);
     try { Thread.sleep(1000); }
     catch (InterruptedException e){ }
    } // fin du bloc synchronized
                                    Mr NDONG. FST/UCAD
```

Exercice: producteur – consommateur: Corrigé (2/2)

(fichier Consommateur2.java)

```
public class Consommateur2 extends Thread {
private Chapeau chapeau;
private int number;
  public Consommateur2 (Chapeau chapeau, int number) {
     this.chapeau =chapeau;
     this.number =number;
public void run ( ){
  int value = 0;
synchronized (chapeau) { // il faut absolument synchroniser ce bloc
    for (int i = 0; i < 5; i++) {
        value = chapeau.get ( );
        System.out .println("le consommateur N° "+this.number +" a pris "+value);
        try {Thread.sleep (1000);}
        catch (InterruptedException e){ }
     } // fin du bloc synchronized
                                    Mr NDONG. FST/UCAD
                                                                                340
```

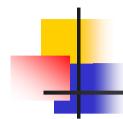
Exercice: producteur – consommateur: sortie

Voici un exemple de sortie du programme de test.

```
le producteur N° 1 a mis 0
le consommateur N° 1 a pris 0
le producteur N° 2 a mis 0
le consommateur N° 1 a pris 0
le producteur N° 3 a mis 0
le consommateur N° 1 a pris 0
le producteur N° 3 a mis 1
le consommateur N° 1 a pris 1
le producteur N° 3 a mis 2
le consommateur N° 1 a pris 2
le producteur N° 1 a mis 1
le consommateur N° 2 a pris 1
le producteur N° 1 a mis 2
le consommateur N° 2 a pris 2
le producteur N° 1 a mis 3
le consommateur N° 2 a pris 3
le producteur N° 1 a mis 4
```

MI NOUNG. 131/UCAD

Priorité des Threads



Jusqu'à présent, nous n'avons manipulé que des threads de même priorité. En théorie, il est permis d'attribuer une certaine priorité à un thread. Pour cela, on utilise la méthode setPriority (int threadPriority) où le paramètre transmis en argument est une valeur entière comprise entre MIN.PRIORITY (correspondant à la valeur 1) et la valeur MAX.PRIORITY (correspondant à 10).

La priorité par défaut est la valeur NORM.PRIORITY (valeur 5). La priorité d'un thread est exploité par l'environnement de cette façon:

- lorsqu'il peut donner la main à un thread, l'environnement choisi celui de plus haute priorité parmi ceux qui sont dans l'état prêt; s'il y a plusieurs candidats le thread choisi dépendra de l'environnement;
- si un thread plus prioritaire que le thread en cours d'exécution devient prêt, on lui donne la main (le thread courant passant alors à l'état prêt).

Priorité des Threads: Exemple (1/2)



```
public class ThreadPriority extends Thread{
 public ThreadPriority (String name) {super (name); }
 public void run ( ){
    for (int i = 0; i < 5; i++)
    { System.out.println ( this.getName ( ) +" valeur de i = "+i);
    try { Thread.sleep (1000) ;}
    catch ( InterruptedException er ) { }
    System.out.println (this.getName () +" se termine ");
 public static void main(String[] args) {
  ThreadPriority t = new ThreadPriority ("thread # 1");
  ThreadPriority p = new ThreadPriority ("thread # 2");
  t.setPriority (7); p.setPriority (Thread.MAX_PRIORITY);
  t.start(); p.start();
                            Mr NDONG. FST/UCAD
```

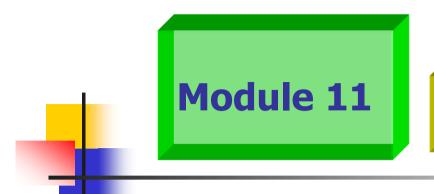
Priorité des Threads: Exemple (2/2)

```
thread # 1 valeur de i = 0
thread # 2 valeur de i = 0
thread # 1 valeur de i = 1
thread # 2 valeur de i = 1
thread # 1 valeur de i = 2
thread # 2 valeur de i = 2
thread # 1 valeur de i = 3
thread # 1 valeur de i = 3
thread # 2 valeur de i = 4
thread # 1 valeur de i = 4
thread # 2 valeur de i = 4
```

```
thread # 1 valeur de i = 0
thread # 2 valeur de i = 0
thread # 2 valeur de i = 1
thread # 1 valeur de i = 1
thread # 2 valeur de i = 2
thread # 1 valeur de i = 2
thread # 1 valeur de i = 3
thread # 2 valeur de i = 3
thread # 1 valeur de i = 4
thread # 1 valeur de i = 4
thread # 1 valeur de i = 4
thread # 1 se termine
```

Le thread thread# 2
étant de priorité
supérieure, son
exécution s'achève
avant celle du thread
thread # 1 lancé
en premier lieu.

Ce qu'on aurait si les deux threads étaient de même priorité (celle par défaut).



Les Swing GUI

Les classes graphiques Swing dérivent de la classe JComponent, qui hérite elle-même de la classe AWT (Abstract Window Toolkit) Container.

Quelle est la différence entre les composants AWT et les composants Swing?

Tous les composants Swing commencent par la lettre " J " . La classe JComponent et les contrôles GUI (Graphical User Interface) se trouvent dans le paquetage javax.swing .*

Les composants Swing se répartissent :

- en conteneurs de plus haut niveau (JFrame, JWindow, JApplet et JDialog)
- en conteneurs légers (les autres contrôles GUI Swing).





Les composants AWT sont des composants "lourds" c-à-d des contrôles produits par la machine virtuelle à destination du système d'exploitation. Si vous créez par exemple un bouton Button tiré du module java.awt sous Windows NT, la machine virtuelle génère un bouton NT et lui communique tous les paramètres nécessaires à son initialisation. L'aspect du bouton, comme des autres composants de ce type, dépend du système d'exploitation utilisé.

Les composants Swing sont des composants "légers" c-à-d directement dessinés par la machine virtuelle. Le composant aura toujours le même aspect quelque soit la plateforme utilisée. On trouve dans les Swing plus de fonctionnalités.

Pour les Swing, un conteneur de plus haut niveau se compose d'une "fenêtre visible ", la ContentPane, placée au dessus de la fenêtre native. Les composants GUI doivent se placer dans cette ContentPane.

Création d'une fenêtre Swing

```
import java.awt.*;
                                                                             _ | D ×
                                              🥰 Ma Fenêtre Swing
import javax.swing .*:
public class Swing01
                      extends JFrame
 public Swing01 (String titre) {
  this.setTitle (titre);
  this.setSize (250,200);
// nécessité de récupérer le ContentPane
Container contenu = this.getContentPane( );
contenu.setBackground (Color.yellow);
 public static void main( String [] args) {
  Swing01 fen = new Swing01("Ma Fenêtre Swing");
  fen.setVisible (true);
```

La même fenêtre en AWT

```
👸 Ma Fenêtre AWT
import java.awt.*;
                      extends Frame
public class AWT01
 public AWT01 (String titre) {
  this.setTitle (titre);
  this.setSize (250,200);
  this.setBackground (Color.yellow);
 public static void main( String [] args) {
AWT01 fen = new AWT01("Ma Fenêtre AWT");
  fen.setVisible (true);// pour rendre la fenetre visible
```

_ | D | X



Remarques

Les classes Color et Container sont présentes dans le module java.awt, c'est pourquoi il faut toujours importer ce package. Dans la gestion des interfaces graphiques, il ne s'agit pas simplement de construire des composants, mais il faut aussi pouvoir interagir avec eux en produisant des évènements. Il s'agit de la programmation évènementielle qui nécessitent les classes de gestion d'évènements présentées dans les modules java.awt.event .* et javax.swing.event .*

En somme, il faut importer au minimum, les quatre bibliothèques suivantes:

java.awt.*
java.awt.event .*
javax.swing .*
javax.swing.event .*

Ajout d'un composant léger: un JButton

```
import java.awt.*;
import javax.swing .*:
public class Swing02
                      extends JFrame
 public Swing02 (String titre) {
  this.setTitle (titre); this.setSize (250,200);
 Container contenu = this.getContentPane( );
contenu.setBackground (Color.yellow);
JButton bouton = new JButton ("Copier");
bouton.setBackground (Color.green);
contenu.add (bouton);
 public static void main( String [] args) {
 new Swing02("Ma Fenêtre Swing").setVisible (true);
 }}
                                       Mr NDONG. FST/UCAD
```

```
Ma Fenêtre Swing

Copier
```



Ajout du JButton

La création d'un bouton nécessite l'usage d'un constructeur de la classe JButton. Ici, on utilise le constructeur JButton (String intitule).

JButton bouton = new JButton ("Copier");

On donne une couleur au bouton avec la méthode setBackground (Color couleur) appliqué à l'objet bouton.

bouton.setBackground (Color.green);

Et on ajoute le composant à la partie contenu de la fenêtre native (le ContenPane) en utilisant la méthode add (Component comp) :

contenu.add (bouton);



Remarques sur l'ajout du JButton

A l'affichage de la fenêtre, il faut remarquer que seule la couleur verte (celle du bouton apparaît) et non celle de la fenêtre (couleur jaune).

En fait, le bouton occupe par défaut tout le ContenPane. Ceci s'explique du fait que chaque composant de plus haut niveau dispose de ce que l'on nomme un *gestionnaire de mise en forme* (Layout Manager) qui permet de disposer les différents composants dans le ContenPane.

Pour JFrame, le gestionnaire est la classe BorderLayout.

Avec ce gestionnaire, le composant occupe toute la fenêtre. Donc même si vous rajouter un deuxième bouton à la fenêtre, *il va se substituer au premier* et vous ne verrez donc que le dernier composant ajouté. Pour visualiser les deux composants, il faut indiquer leur position car BorderLayout place les composants au quatre points Cardinaux (North, West, East, South) et au centre (Center).

Le gestionnaire de JFrame: BorderLayout

```
import java.awt.*;
import javax.swing .*:
public class Swing03
                      extends JFrame {
 public Swing03 (String titre) {
  this.setTitle (titre); this.setSize (250,200);
 Container contenu = this.getContentPane( );
contenu.setBackground (Color.vellow);
JButton bouton = new JButton ("Copier");
bouton.setBackground (Color.green);
contenu.add (bouton, BorderLayout.SOUTH);
JButton bout = new JButton ("Effacer");
bout.setBackground (Color.green);
contenu.add (bout, BorderLayout.NORTH);
```

```
Effacer

Copier
```



Gestion de l'interface MouseListener

Nous allons implémenter l'interface MouseListener. Donc voir comment traiter un clic de souris sur la fenêtre. On va se contenter d'afficher les coordonnées du point où l'on clique.

En Java, tout évènement possède ce qu'on appelle une source. Il s'agit de l'objet ayant donné naissance à cet évènement : bouton, menu, fenêtre...

Pour traiter un évènement, on associe à la source *un objet de son choix* dont la classe implémente une interface particulière à *une catégorie d'évènement*. Cet objet est un *écouteur* de cette catégorie d'évènement.

Chaque méthode proposée par l'interface correspond à une catégorie d'évènement.



Gestion de l'interface MouseListener

Il existe une catégorie *d'évènement souris* qu'on peut traiter avec un *écouteur de souris*, c'est-à-dire un objet d'une classe implémentant l'interface MouseListener. Cette interface possède cinq méthodes:

mouseClicked, mouseEntered, mouseReleased, mouseExited et mousePressed.

Pour prendre en compte la gestion du clic, seul l'évènement clic nous intéresse et ce dernier correspond à la méthode mouseClicked. Mais comme on implémente une interface, on est obligé de redéfinir toutes les méthodes de cette dernière.

Voici comment il faut procéder:

Gestion de l'interface MouseListener

```
import java.awt.*; import java.awt.event.*; import javax.swing.*;
public class Swing04 extends JFrame implements MouseListener
  public Swing04(String titre) {
  this.setTitle(titre); this.setSize(250,200);
  Container contenu = this.getContentPane(); contenu.setBackground(Color.yellow);
  /*la fenetre est son propre ecouteur d' evenement souris*/
    this.addMouseListener ( this );
/*redefinition obligatoire de toutes les méthodes de l'interface*/
 public void mouseClicked( MouseEvent e)
 {System.out.println ("vous avez clique au point de coordonnes : "+e.getX()+" "+e.getY()
 public void mouseReleased( MouseEvent e)
 public void mouseExited( MouseEvent e)
 public void mousePressed( MouseEvent e)
 public void mouseEntered( MouseEvent e)
                                     Mr NDONG. FST/UCAD
                                                                                356
```



Commentaires

Dans l'exemple précèdent, nous avons choisi la fenêtre comme son propre écouteur d'évènement souris. C'est pourquoi, il est obligatoire de mentionner implements MouseListener dans l'en tête de la classe. Si vous l'omettez, il y a erreur de compilation.

La mention this.addMouseListener (this) associe un écouteur à la fenêtre principale. Si vous l'omettez, il n'y a pas erreur, seulement le clic sera sans effet.

Supposons maintenant, au lieu de considérer que la fenêtre soit son propre écouteur d'évènement souris, que son écouteur soit un objet quelconque.

Tout ce qu'il faut vraiment savoir ici est que la classe de cet objet doit implémenter l'interface MouseListener.

Voyons comment traiter l'exemple précèdent :

Personnalisation de l'objet écouteur (1/2)

```
import java.awt.*; import java.awt.event.*; import javax.swing.*;
public class Swing05 extends JFrame {
  public Swing05 (String titre) {
  this.setTitle(titre);
  this.setSize(250,200);
  Container contenu = this.getContentPane();
  contenu.setBackground(Color.yellow);
/* on cree un objet ecouteur de la fenetre*/
   EcouteurFenetre ecout = new EcouteurFenetre ();
/*l'objet ecout devient maintenant l'ecouteur d'evenement souris de la fenetre*/
   this.addMouseListener ( ecout );
}// fin de la classe Swing05
```



Personnalisation de l'objet écouteur (2/2)

```
class EcouteurFenetre implements MouseListener
/*redefinition obligatoire de toutes les méthodes de l'interface*/

public void mouseClicked( MouseEvent e)
{System.out.println ("vous avez clique au point de coordonnes : "+e.getX()+" "+e.getY());
}

public void mouseReleased ( MouseEvent e) { }

public void mouseExited ( MouseEvent e) { }

public void mousePressed ( MouseEvent e) { }

public void mouseEntered ( MouseEvent e) { }
```



Les classes Adaptateur (1/4)

On constate que dans l'exemple précèdent, nous n'avions eu besoin que de la méthode mouseClicked; pourtant on étais obliger de redéfinir les autres méthodes de l'interface puisque Java l'impose lors de l'implémentation d'une interface.

Comment faire donc pour n'utiliser que la méthode qui nous intéresse ici ?

Il existe une classe particulière appelée **MouseAdapter** qui implémente toutes les méthodes de l'interface MouseListener ceci:

class MouseAdapter implements MouseListener

Les classes Adaptateur (2/4)



Comme MouseAdapter est une classe et non une interface, on pourra désormais en dériver simplement ce qui nous permettra d'utiliser que les méthodes que nous souhaitons exploiter (en les redéfinissant).

Presque toutes les interfaces Listener disposent d'une classe Adapter.

Les interfaces Listener qui n'ont qu' un seul type d'évènement à traiter, donc une seule méthode ne disposent pas de classe adaptateur. Par exemple l'interface **ActionListener** qui gère la catégorie d'évènements action.

Voici comment on peut refaire le premier exemple en ne tenant compte que de la méthode mouseClicked.

Les classes Adaptateur (3/4)

```
import java.awt.*; import java.awt.event.*; import javax.swing.*;
public class Swing05 extends JFrame {
  public Swing05 (String titre) {
  this.setTitle(titre);
  this.setSize(250,200);
  Container contenu = this.getContentPane( );
  contenu.setBackground(Color.yellow);
/* on cree un objet ecouteur de la fenetre*/
   EcouteurFenetre ecout = new EcouteurFenetre ();
/*l'objet ecout devient maintenant l'ecouteur d'evenement souris de la fenetre*/
   this.addMouseListener ( ecout );
}// fin de la classe Swing05
```

Les classes Adaptateur (4/4)



class EcouteurFenetre extends MouseAdapter

/*on ne redefinit que la methode mouseClicked*/

```
public void mouseClicked( MouseEvent e)
 System.out.println ("vous avez clique au point de coordonnes : "+e.getX()+" "+e.getY());
    ATTENTION
```

Si vous utilisez ici la classe MouseAdapter au lieu de l'interface il ne sera plus possible de considérer que la fenêtre soit son propre écouteur. Car ceci impliquerait de dériver la classe Swing05 en même temps de JFrame et de MouseAdapter, ce qui est interdit.

Gestion de l'écouteur



```
public class Swing04 ex tends JFrame
  public Swing04(String titre) {
  this.setTitle (titre); this.setSize (250,200);
  Container contenu = this.getContentPane(); contenu.setBackground (Color.yellow);
  /*gestion de l ecouteur_ayec une classe anonyme*/
this.addMouseListener | ( | new MouseAdapter ( )
 { public void mouseClicked( MouseEvent e)
       System.out.println ("vous avez clique au point de coordonnes : "+e.getX()+" "+
                   e.getY());
```



Mettre fin à l'application (1/2)

Le simple clic sur le bouton de fermeture de la fenêtre ne permet de mettre fin à l'application. Il rend simplement la fenêtre invisible. Le clic de fermeture est équivalent à faire:

new Swing02("Ma Fenêtre Swing").setVisible (false);

Autrement dit le processus qui gère l'application tourne toujours en tâche de fond. Pour l'arrêter, il faut interrompre le compilateur, ce qui n'est pas optimal.

Il faut toujours gérer la fin de l'application par des instructions adéquates. Pour ce faire, on va voir un premier cas d'utilisation de la gestion des évènements avec la classe java.awt.event.WindowListener dans l'implémentation d'une classe anonyme.



Mettre fin à l'application (2/2)

```
import java.awt.*;
import javax.swing .*;
public class Swing03
                      extends JFrame {
 public Swing03 (String titre) {
  this.setTitle (titre); this.setSize (250,200);
 Container contenu = this.getContentPane( );
  contenu.setBackground (Color.yellow);
  /* pour mettre fin a l'application des qu'on clique sur le bouton de fermeture*/
  this.addWindowListener (new WindowAdapter ()
  { public void windowClosing (WindowEvent e)
      { System.exit (0);
  });
                                      Mr NDONG. FST/UCAD
```



Un bouton gère une catégorie d'évènement appelée action qu'on l'on traite avec un écouteur qui est un objet implémentant l'interface ActionListener. Cette dernière ne possède qu'une seule méthode :

public void actionPerformed (ActionEvent ev).

Comme illustration, nous allons considérer un bouton et deux zones de texte, l'une contenant un texte et l'autre étant vide;

Le clic sur le bouton entraînera la copie du contenu de la première zone de texte dans la seconde, et le vidange de celle-là.

On supposera que la fenêtre est l'objet écouteur des clics sur le bouton.

```
import java.awt.event.*; import java.awt.*; import javax.swing.*;
public class Swing06 extends JFrame
                                     implements ActionListener
JTextField texteinitial, textefinal;
JButton bouton;
public Swing06 (String titre) {
  this.setTitle(titre); this.setSize(250,100);
  Container contenu = this.getContentPane();
  contenu.setBackground (Color.yellow);
  bouton = new JButton("Copier");
  bouton.setBackground(Color.green);
   bouton.addActionListener (this);
  contenu.add(bouton,BorderLayout.SOUTH);
  texteinitial = new JTextField("texte initial",15);
  contenu.add( texteinitial, BorderLayout.NORTH );
  textefinal = new JTextField("",15);
  contenu.add( textefinal, BordeMLayONGCEN/PER);
```

368



```
/*redefinition de la methode actionPerformed*/
public void actionPerformed(ActionEvent e)
{ if (e.getSource () = = bouton) }
  { textefinal.setText( texteinitial.getText());
   texteinitial.setText(" ");
 public static void main(String[] args) {
  Swing06 fen = new Swing06("Ma Fenêtre Swing");
  fen.setVisible(true);
                   Mr NDONG. FST/UCAD
```



Pour déterminer la source du clic, on utilise la méthode getSource () qui fournit une référence de type Object sur l'objet ayant déclenché l'évènement. Au lieu d'utiliser la référence du bouton pour tester dans l'instruction if la source de l'évènement, on peut utiliser l'étiquette du bouton, appelée une chaîne de command . Dans ce cas, on n'utilise pas la méthode getSource () mais la méthode getActionCommand, de la façon suivante: public void actionPerformed(ActionEvent e) { /*on recupere l etiquette du bouton sur lequel on clique dans une chaine*/ **String etiquette = e.getActionCommand ()**; // on utilise equals pour comparer le contenu de deux chaines if (etiquette.equals (" Copier")) { textefinal.setText(texteinitial.getText()); texteinitial.setText(" "); Mr NDONG. FST/UCAD 370





Le rôle d'un gestionnaire de mise en forme est de permettre une disposition des composants selon le choix de l'utilisateur. Nous avons déjà vu le gestionnaire BorderLayout pour la fenêtre principale. Nous allons à présent explorer les autres types de gestionnaires.

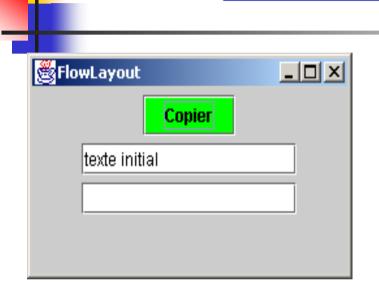
FlowLayout : représente les composants sur une même ligne, les uns à la suite des autres; s'il n'y a plus d'espace en fin de ligne, il passe à la ligne suivante.

CardLayout : permet de disposer des composants suivant une pile, à la manière d'un paquet de cartes, un seul composant étant visible à la fois,

BoxLayout : dispose les composants sur une seule ligne ou sur une seule colonne, GridBagLayout : dispose les composants sur une grille, la taille d'un composant dépend du nombre de cellules que le composant occupe.

GridLayout : dispose les composants sur une grille, les composants de même colonne ayant la même taille.

Exemples de mise en œuvre de FlowLayout



On associe un gestionnaire à un conteneur de haut niveau avec la méthode setLayout (LayoutManager)

```
public class Swing01 extends JFrame {
JTextField texteinitial;
JButton bouton;
JTextField textefinal;
 public Swing01(String titre) {
  this.setTitle(titre);
  this.setSize(250,150);
  Container contenu = this.getContentPane();
    contenu.setLayout (new FlowLayout ( ));
  bouton = new JButton("Copier");
  bouton.setBackground(Color.green);
 contenu.add(bouton);
  texteinitial = new JTextField("texte initial",15);
  contenu.add(texteinitial);
  textefinal = new JTextField("",15);
  contenu.add(textefinal);
                                             372
Mr NDONG. FST/ÙCAD
```

Exemples de mise en œuvre de GridLayout



Les zones de texte sont trop spacieuses; on verra qu'on peut avoir des zones plus rétrécies avec l'utilisation d'objets panneaux (JPanel)

```
public class Swing01 extends JFrame{
      JTextField texteprenom;
      JLabel prenom, nom;
      JTextField textenom;
      public Swing01(String titre) {
       this.setTitle(titre);
       this.setSize(250,150);
       Container contenu =this.getContentPane();
       contenu.setLayout(new GridLayout(2,2));
     prenom =new JLabel("prénom");
       nom = new JLabel("nom");
       texteprenom = new JTextField("texte initial",15
       textenom = new JTextField("",15);
contenu.add(prenom); contenu.add(texteprenom);
```

3/3

contenu.add(nom); contenu.add(textenom);

MFINDONG. FSI/UCAD

Exemples de mise en œuvre de CardLayout



Si on clique sur prem il affiche deux, si on clique sur deux, il affiche quat ...

```
public class Swing01 extends JFrame implements ActionListener{
 CardLayout pile; JButton prem,deux,trois,quat;
 public Swing01(String titre) {
  this.setTitle (titre); this.setSize (250,150);
  pile = new CardLayout (5,3); // hgap = 5 vgap = 3
  this.getContentPane ( ).setLayout (pile) ;
  prem = new JButton("premier"); prem.addActionListener (this);
  deux = new JButton("deuxieme"); deux.addActionListener (this);
  trois = new JButton("troisieme"); trois.addActionListener (this);
  quat = new JButton("quatrieme"); quat.addActionListener (this);
  this.getContentPane().add (prem, "Bouton"); //obligatoire
  this.getContentPane().add (deux,"Bouton");
  this.getContentPane().add (trois, "Bouton");
  this.getContentPane( ).add (quat, "Bouton");
 public void actionPerformed (ActionEvent e)
{ if (e.getSource ( ) == prem) pile.next (this.getContentPane());
 if (e.getSource ( ) == deux) pile. last (this.getContentPane());
if (e.getSource ( ) == trois) pile.first (this.getContentPane());
if (e.getSouppeNDONG.quest)/Upide.previous (this.getContentPane 37)4
```

Tour d'horizon de GridBagLayout

Des différents Layout Manager, Le gestionnaire GridBagLayout est le plus difficile à manier.

Il permet de disposer les composants selon une grille, ,ceux-ci pouvant occupés plusieurs cases.

Cette classe ne dispose que d'un seul constructeur sans paramètre:

GridBagLayout (). Pour associer ce gestionnaire à un Container conteneur : conteneur.setLayout (new GridBagLayout ());

Mais cette instruction ne suffit pas pour placer les composants au conteneur.

Tout composant à ajouter doit disposer d'un objet GridBagConstraints lequel spécifie comment faire le placement des différents composants:

/*The GridBagConstraints class provides the means to control the layout of components within a Container whose LayoutManager is GridBagLayout.*/

GridBagConstraints objetPlaceur = new GridBagConstraints ();

Cet objet objetPlaceur dispose alors de variables et de méthodes permettant de réaliser le placement des composants.

Mr NDONG. FST/UCAD

375

4

GridBagConstraints: variables et méthodes (1/2)

```
/*Specifies the alignment of the component in the event that it is smaller than the space
allotted*/
public int anchor
/*The component's resize policy if additional space available. */
public int fill
/*Number of columns (gridheight), of rows (gridwidth) a component occupies.*/
public int gridheight,
                                 public int gridwidth
/*Horizontal (gridx), vertical (gridy) grid position at which to add component. */
public int gridx,
                                 public int gridy
/*Specifies the outer padding around the component.*/
public Insets insets
/*Serves as the internal padding within the component in both the right and left direction*/
public int ipadx
/*Serves as the internal padding within the component in both the top and bottom
directions*/
                                         Mr NDONG. FST/UCAD
                                                                                      376
public intipady
```



GridBagConstraints: variables et méthodes (2/2)

/*Represents the percentage of extra horizontal space that will be given to this component if there is additional space available within the container*/.

public double weightx

/*Represents the percentage of extra vertical space that will be given to this component if there is additional space available within the container. */ public double weighty

Exemple de mise en œuvre de GridBagLayout Création d'un nouveau client Création d'un nouveau client Identité client Identité client Prénom: Prénom: Nom: Nom: Adresse: Adresse: Code Postal: Code Postal: mot de passe : NouveauBanquier ✓ NouveauBanquier Valider Valider Retour Retour

Le clic sur la case **NouveauBanquier** dans

donne la même fenêtre (2)

Code Exemple (1/7)

```
public class ExGridBagConstraints extends JFrame implements ActionListener{
  JLabel motpass;
  JCheckBox nbq;
  JPasswordField txtpass;
   Container c;
   GridBagConstraints gr;
public ExGridBagConstraints() {
 this.setResizable (false);
 this.setTitle ("Création d'un nouveau client");
 Dimension screensize = Toolkit.getDefaultToolkit().getScreenSize();
 Dimension framesize = this.getSize();
 if (framesize.width > screensize.width ) framesize.width = screensize.width;
 if (framesize.height > screensize.height ) framesize.height =screensize.height ;
   this.setLocation ((screensize.width -framesize.width )/3,
(screensize.height -framesize.height )/3);
Mr NDONG. FST/UCAD
```



Code Exemple (2/7)



Code Exemple (3/7)

```
/*ajouter le label nom*/
/*ajouter centrer le label Bienvenue*/
                                              gr = new GridBagConstraints();
 gr = new GridBagConstraints( );
                                              gr.gridx = 1; gr.gridy =12;
                                              gr.anchor =GridBagConstraints.WEST ;
gr.gridx =1; gr.gridy =4;
                                           gr.fill = GridBagConstraints.HORIZONTAL;
   gr.insets = new Insets(10,30,10,50);
                                              gr.ipadx =2;
   c.add (acc,gr);
                                              gr.insets = new Insets (0,20,2,0);
/*ajouter le label prénom*/
                                              c.add (nm,gr);
   gr = new GridBagConstraints();
   gr.gridx =1; gr.gridy =8;
   gr.anchor =GridBagConstraints.WEST ;
   gr.ipadx = 0;
   gr.insets = new Insets(40,20,2,0);
   c.add (pr,gr);
```

Code Exemple (4/7)

```
/*ajouter la zone pour le prenom*/
/*ajouter le label adresse*/
                                                 gr=new GridBagConstraints();
   gr = new GridBagConstraints();
                                                 gr.gridx =2;gr.gridy =8;
   gr.gridx =1;gr.gridy =16;
                                              gr.fill =GridBagConstraints.HORIZONTAL;
   gr.anchor = GridBagConstraints.WEST ;
                                                 gr.ipadx =100;
   gr.fill =GridBagConstraints.HORIZONTAL;
                                                 gr.insets = new Insets(40,0,2,0);
   gr.ipadx = 2;
                                                 c.add(txtpr,gr);
   gr.insets = new Insets(0,20,2,0);
   c.add (adr, gr);
/*ajouter le label code postal*/
    gr=new GridBagConstraints();
    gr.gridx =1;gr.gridy =20;
    gr.anchor = GridBagConstraints.WEST;
    gr.fill =GridBagConstraints.HORIZONTAL;
    gr.ipadx =2;
    gr.insets = new Insets(0,20,2,10);
    c.add (cp, gr);
```

Code Exemple (5/7)

```
/*ajouter la zone code postal*/
*ajouter la zone pour le nom*/
                                                  gr = new GridBagConstraints();
   gr = new GridBagConstraints();
                                                  gr.gridx =2;gr.gridy =20;
   gr.fill =GridBagConstraints.HORIZONTAL; gr.fill =GridBagConstraints.HORIZONTAL
   gr.gridx =2;gr.gridy =12;
                                                  gr.ipadx = 2;
   gr.ipadx = 2;
                                                  c.add(txtcp,gr);
   c.add(txtnm,gr);
                                              /*ajouter la case pour le nouveau banquier*/
/*ajouter la zone adresse*/
                                                  gr = new GridBagConstraints();
   gr = new GridBagConstraints();
                                                  gr.gridx =1;gr.gridy =28;
   gr.gridx =2;gr.gridy =16;
                                                  gr.anchor =GridBagConstraints.WEST;
   gr.fill =GridBagConstraints.HORIZONTAL; gr.fill =GridBagConstraints.HORIZONTAL
   gr.ipadx =2;
                                                  gr.ipadx = 2;
   c.add(txtadr,gr);
                                                  gr.insets = new Insets(0,20,2,0);
                                                  c.add(nbq,gr);
```

Code Exemple (6/7)

```
/*ajouter du label nouveau banquier*/
                                                    /*ajouter du bouton valider*/
   gr = new GridBagConstraints();
                                                        gr = new GridBagConstraints();
   gr.gridx =1;gr.gridy =24;
                                                        gr.gridx =1;gr.gridy =30;
   gr.anchor =GridBagConstraints.WEST;
                                                        gr.ipadx = 1;
   gr.fill =GridBagConstraints.HORIZONTAL;
                                                        gr.insets = new Insets(0,0,35,0);
   gr.ipadx = 2;
                                                        c.add(valider,gr);
   gr.insets = new Insets(0,20,2,0);
   motpass.setVisible (false);
                                                     /*ajouter du bouton retour*/
   c.add(motpass,gr);
                                                        gr = new GridBagConstraints();
/*ajouter d la zone mot de passe pour le nouveau
                                                        gr.gridx =1;gr.gridy =40;
banquier*/
                                                        gr.ipadx =1;
   gr = new GridBagConstraints();
                                                        gr.insets = new Insets(0,0,35,0);
   gr.gridx =2;gr.gridy =24;
                                                        c.add(retour,gr);
   gr.fill =GridBagConstraints.HORIZONTAL;
   gr.ipadx =2;
                                                          pack();
   txtpass.setVisible (false);
   c.add(txtpass,gr);
                                         Mr NDONG. FST/UCAD
                                                                                     384
```



```
public void actionPerformed (ActionEvent es)
{ if ( es.getSource() = =nbq)
    { motpass.setVisible( true) ;
        txtpass.setVisible( true) ;
    }
    if ( nbq.isSelected ( ) = = false)
    { motpass.setVisible( false) ;
        txtpass.setVisible( false) ;
    }
} // fin de la classe
```





Il se peut que, lors de la construction d'une interface graphique que le programmeur ne veuille utiliser aucun de gestionnaires prédéfinies. Cela voudra dire qu il prend ses propres dispositions pour ajouter les composants lui-même à l'endroit où il voudrait bien les placer.

Dans ce cas il faut signaler qu'on n'utilise aucun gestionnaire en faisant:

objet_conteneur.setLayout (null) ;

et après d'utiliser la méthode setBounds (int a, int b, int c, int d);

Où:

- a = abscisse du point d'insertion du composant,
- **b** = ordonnée du point d'insertion du composant,
- c = largeur du composant,
- **d** = hauteur du composant.

NB: cette technique demande beaucoup d'attention surtout avec l'usage de setBounds. Mr NDONG. FST/UCAD

Objet JPanel



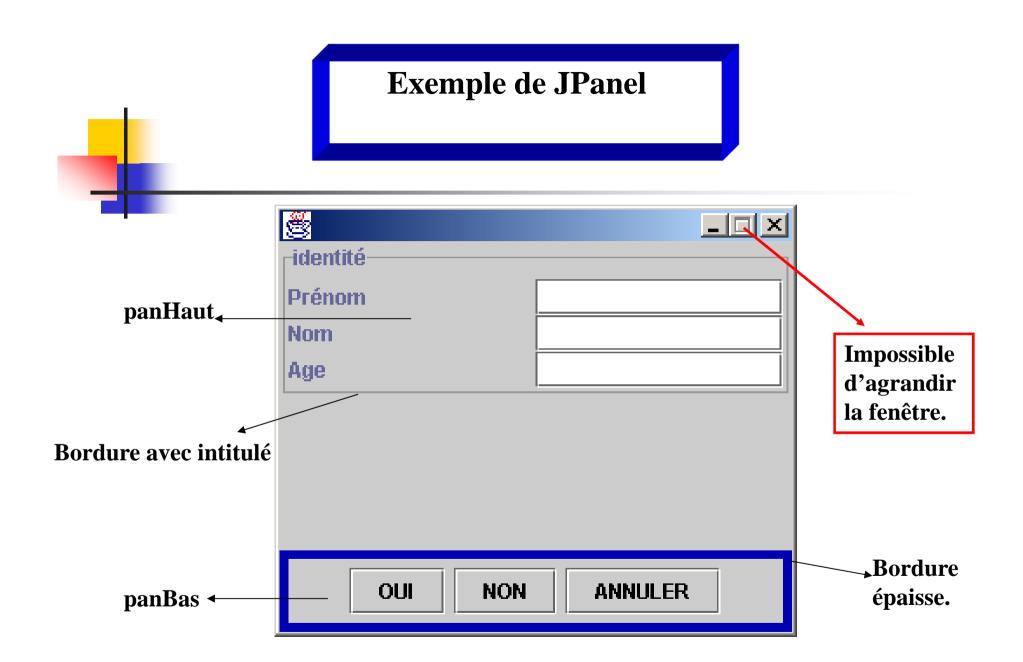
Si vous voulez rangés *en même temps et directement* dans un JFrame des composants suivant une grille avec par exemple GridLayout et d'autres composants selon une ligne horizontale avec FlowLayout, cela va s'avérer impossible puisque vous ne pouvez pas appliquer deux gestionnaires simultanément.

L'astuce qu'il faut utiliser est de créer deux panneaux, l'un pour le premier groupe de composants, le second pour le deuxième groupe.

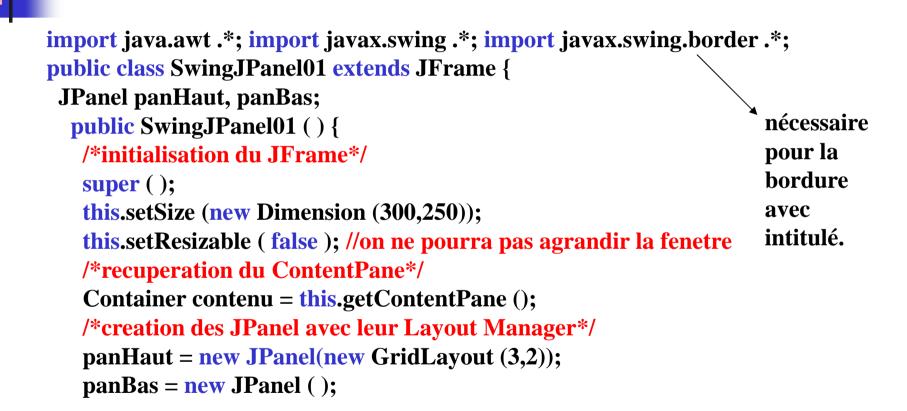
Les panneaux sont des conteneurs puisqu'ils servent à contenir des composants.

Un panneau est sorte de sous fenêtre sans titre, ni bordure.

Le gestionnaire par défaut de JPanel est FlowLayout.



Code Exemple de JPanel



Code Exemple de JPanel (suite)

```
/*ajout des pannneaux au ContentPane,l'un au nord, l'autre au sud*/
   contenu.add (panHaut, BorderLayout.NORTH);
   contenu.add(panBas, BorderLayout.SOUTH);
/*ajout de trois label et de trois zones de texte a panHaut*/
   panHaut.add( new JLabel ("Prénom")); panHaut.add (new JTextField());
   panHaut.add( new JLabel("Nom"));
                                         panHaut.add(new JTextField());
   panHaut.add (new JLabel("Age"));
                                         panHaut.add(new JTextField());
 /*ajout de trois boutons a panBas*/
   panBas.add ( new JButton("OUI"));
   panBas.add ( new JButton("NON"));
   panBas.add ( new JButton("ANNULER"));
 /*ajout d une bordure avec intiutlé a panHaut*/
   panHaut.setBorder ( new TitledBorder("identité"));
 /*ajout d une bordure epaisse a panBas*/
   Border b = BorderFactory.createLineBorder (Color.blue .darker (),5);
```

Mr NDONG. FST/UCAD

390

panBas.setBorder (b);

} }



Pour dessiner dans un panneau, il faut *redéfinir* la méthode **paintComponent** (appartenant à la classe **JPanel**). Il faut alors créer un panneau *personnalisé* c à d une classe dérivée de JPanel puisqu'il y a nécessité de <u>redéfinition</u>.

NB: lorsque vous redéfinissez *paintComponent*, prenez la peine d'appeler la méthode de la super classe par super.paintComponent (g) puisque celle-ci appelle la méthode *ComponentUI.update* () qui redessine le fond du composant s'il est opaque (*JPanel*). L'en-tête de la méthode à redéfinir est :

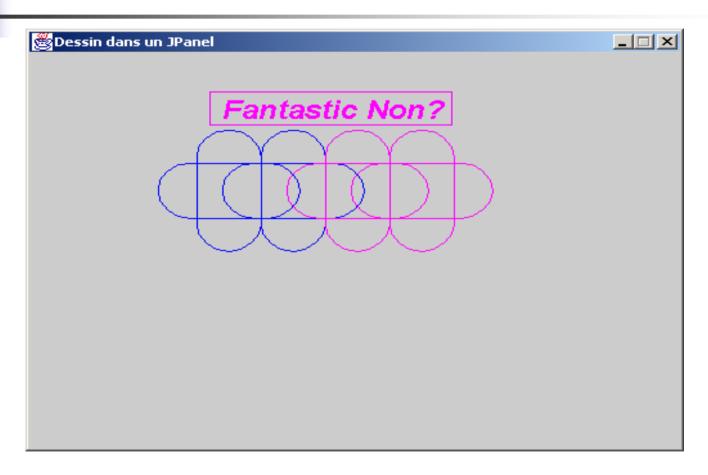
public void paintComponent (Graphics g)

L'argument g est ce que l'on nomme un contexte graphique, c'est un intermédiaire entre les commandes de dessin et leur réalisation effective.

Voici un exemple d'implémentation d'une classe qui redéfinit paint Component,



Exemple de dessin dans un JPanel



Exemple de dessin dans un JPanel

```
import java.awt.*; import javax.swing.*;
public class SwingDraw01 extends JFrame{
 JPanel pan;
 public SwingDraw01 (String titre) {
  super (titre);
  /*dimension de la fenetre en fonction de celle de l'ecran*/
  Toolkit tk =Toolkit.getDefaultToolkit();
  Dimension dim = tk.getScreenSize(); // on recupere les dimensions de l'ecran
  int larg = dim.width /2;
  int haut = dim.height /2;
  this.setSize (larg, haut);
  this.setResizable (false);
  /*recuperation du ContentPane */
  Container c = this.getContentPane();
/*creation et ajout du panneau a c*/
                            pan.setBackground ( new Color (200,150,200,150));
  pan = new
              Paneau ();
                                     Mr NDONG. FST/UCAD
  c.add (pan);
                                                                                  393
```

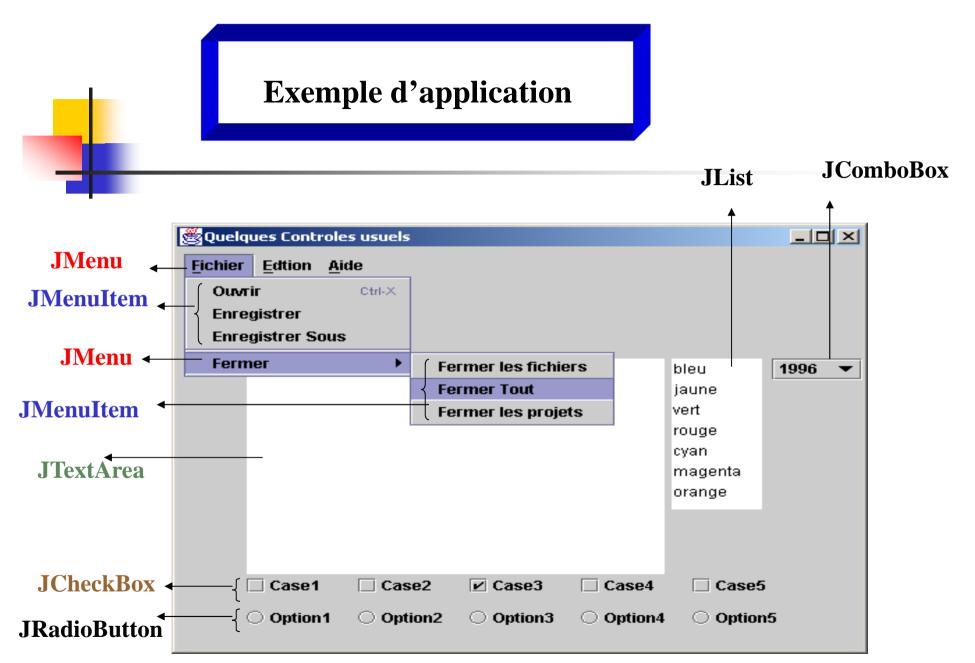
Exemple de dessin dans un JPanel (suite)

```
/*creation personnalisee d un panneau*/
Class Paneau extends JPanel
{ public void paintComponent( Graphics g) //on redefinit paintComponent
 { super.paintComponents (g); // pour redessiner le fond
 g.setColor (Color.blue.brighter ( ).brighter());
 g.drawRoundRect ( 100,100,110,50,50,50 );
 g.drawRoundRect (130,70,50,110,50,50);
 g.drawRoundRect (150,100,110,50,50,50);
 g.drawRoundRect (180,70,50,110,50,50); g.setColor (Color.magenta);
 g.drawRoundRect ( 200,100,110,50,50,50 );
 g.drawRoundRect (230,70,50,110,50,50);
 g.drawRoundRect (250,100,110,50,50,50);
 g.drawRoundRect (280,70,50,110,50,50);
 g.setFont ( new Font ("Arial", Font.ITALIC + Font.BOLD , 24));
 g.drawRect (140,35,180,30);
 g.drawString ("Fantastic Non?",150,60);
                                  Mr NDONG. FST/UCAD
```



Nous allons dans cette partie voir comment créer des contrôles comme des zones de texte sur plusieurs lignes (JTextArea), des cases à cocher (JCheckBox), des boutons d'options (JRadioButton), des boîtes de listes (JList) et listes combinées (JComboBox).

La création de menus est aussi mise en exergue.





Code Exemple d'application (1/5)

Code Exemple d'application (2/5)

```
Pour mettre la barre de menu
                                             complètement à gauche.
public SwingControls(String titre) {
  this.setSize (500,400); this.setTitle (titre);
  Container c = this.getContentPane();
  /*creation de paneaux avec leur gestionnaire*/
  JPanel panHaut = new JPanel(); panHaut.setLayout (new FlowLayout (0,5,5));
  JPanel panCentre = new JPanel();panCentre.setLayout ( null );
  JPanel panBas = new JPanel();
  /*ajout des paneaux au ContentPane*/
                                                hgap:interstice horizontal
  c.add (panHaut, BorderLayout.NORTH);
  c.add(panCentre, BorderLayout.CENTER);
  c.add(panBas, BorderLayout.SOUTH);
                                                        vgap:interstice vertical
  /*creation d une barre de menus et des menus*/
  JMenuBar barMenu = new JMenuBar();
  JMenu fichier = new JMenu ("Fichier"); fichier.setMnemonic ('F');
  JMenu edition = new JMenu ("Edtion"); edition.setMnemonic ('E');
                = new JMenu ("Aide");
  .IMenu aide
                                            aide.setMnemonic ('A');
                                                                            398
  JMenuItem ouvrir = new JMenuItem (GOTST/HCA)?
```



Code Exemple d'application (3/5)

```
/*un accelerateur CTRL X pour le menu ouvrir*/
ouvrir.setAccelerator (KeyStroke.getKeyStroke (KeyEvent.VK X, InputEvent.CTRL MASK
  JMenuItem enregistrer = new JMenuItem("Enregistrer");
  JMenuItem enregistrerSous = new JMenuItem("Enregistrer Sous");
  JMenu fermer = new JMenu("Fermer");
    JMenuItem fermerfics = new JMenuItem("Fermer les fichiers");
        JMenuItem fermertout = new JMenuItem("Fermer Tout");
        JMenuItem fermerproj = new JMenuItem("Fermer les projets");
        /*ajout de la barre de menus au panneau panHaut*/
        panHaut.add(barMenu);
        /*ajout des menus a la barre de menus*/
        barMenu.add(fichier,0); barMenu.add (edition); barMenu.add(aide);
        fichier.add(ouvrir); fichier.add (enregistrer); fichier.add(enregistrerSous);
        fichier.addSeparator(); fichier.add(fermer);
        fermer.add (fermerfics); fermer.add (fermertout); fermer.add (fermerproj);
```

Code Exemple d'application (4/5)

```
/*zone de texte sur plusieurs lignes*/
        aire = new JTextArea("Ça c'est une zone de texte sur plusieurs lignes.");
         aire.setBounds(50,70,300,200);
         panCentre.add(aire);
        /*les autres controles*/
        case1 = new JCheckBox("Case1");case2 = new JCheckBox("Case2");
        case3 = new JCheckBox("Case3",true);
        case4 = new JCheckBox("Case4");case5 = new JCheckBox("Case5");
         case1.setBounds (new Rectangle(50,270,80,20));panCentre.add (case1);
         case2.setBounds ( new Rectangle(130,270,80,20));panCentre.add (case2);
         case3.setBounds (new Rectangle(210,270,80,20));panCentre.add (case3);
        ButtonGroup groupe1 = new ButtonGroup();
         case4.setBounds (new Rectangle(290,270,80,20)); panCentre.add(case4);
         case5.setBounds (new Rectangle(370,270,80,20)); panCentre.add(case5);
         groupe1.add(case4); groupe1.add(case5);
```

Code Exemple d'application (5/5)

```
opbout1 = new JRadioButton("Option1");opbout2 = new JRadioButton("Option2");
opbout3 = new JRadioButton("Option3");
opbout1.setBounds(50,300,80,20); opbout2.setBounds(130,300,80,20);
opbout3.setBounds(210,300,80,20);
opbout4 = new JRadioButton("Option4");opbout5 = new JRadioButton("Option5");
opbout4.setBounds(290,300,80,20); opbout5.setBounds(370,300,80,20);
panCentre.add(opbout1); panCentre.add(opbout2); panCentre.add(opbout3);
panCentre.add(opbout4); panCentre.add(opbout5);
ButtonGroup groupe2 = new ButtonGroup();
ButtonGroup groupe3 = new ButtonGroup();
groupe3.add (opbout1); groupe3.add (opbout2); groupe3.add (opbout3);
groupe2.add (opbout4); groupe2.add (opbout5);
/*creation des listes*/
listesimple = new JList(couleurs); listesimple.setBounds (355,70,65,140);
panCentre.add(listesimple);
listecomplex = new JComboBox(annees); listecomplex.setBounds (427,70,65,20);
panCentre.add(listecomplex);} }
                                    Mr NDONG. FST/UCAD
                                                                               401
```





Dans l'utilisation des JCheckBox, il est possible de sélectionner plusieurs cases, si vous désirez interdire à l'utilisateur de cocher plus de deux cases à la fois (donc une seule case peut être sélectionnée) il faut utiliser un ButtonGroup et ajouter les composants à ce dernier. Ainsi une seule case pourra être sélectionnée à la fois.

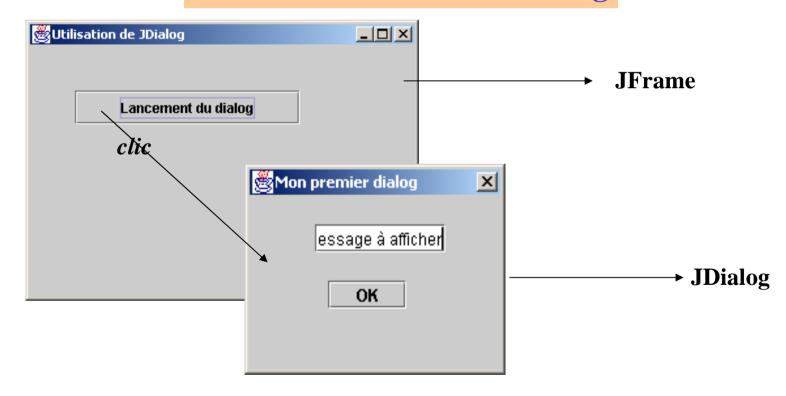
ATTENTION: le ButtonGroup n'est pas un composant, donc il ne peut pas être ajouté à un conteneur. Donc même si vous ajoutez des composants à un ButtonGroup, il faudra également ajouter ces mêmes composants au conteneur en question (un JPanel par exemple).

Les évènements liés aux **JCheckBox** et aux **JRadioButton** sont soit, **l'action de l'utilisateur** sur le composant soit connaître **l'état du composant** (sélectionné ou non).les interfaces qu'ils utilisent sont respectivement **ActionListener** contenant une seule méthode **void actionPerformed** (**ActionEvent e**) et **ItemListener** contenant aussi une seule méthode **void itemStateChange** (**ItemEvent e**).



Création de boîtes de dialogue

Utilisation de la classe JDialog

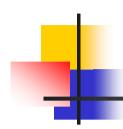


Code de l'interface (1/2)

```
public class SwingDialog extends JFrame implements ActionListener{
   JDialog dialog;
   JButton lancer, ok;
   public SwingDialog (String title) {
      this.setTitle( title);
      this.setSize(350,250);
      Container c = this.getContentPane();
      c.setLayout (null);
      lancer = new JButton ("Lancement du dialog");
      lancer.addActionListener (this);
      lancer.setBounds (40,40,200,30);
      c.add (lancer);
}
```

Code de l'interface (2/2)

```
public void actionPerformed(ActionEvent e)
 { if (e.getSource() = = lancer) lanceDialog ();
  if (e.getSource() = = ok) dialog.dispose ( );
 public void lanceDialog( )
 { dialog = new JDialog(this, "Mon premier dialog", true);
  dialog.setBounds (170,170,200,150);
  dialog.getContentPane( ).setLayout (null);
  JTextField text = new JTextField("Message à afficher");
  dialog.getContentPane().add (text).setBounds (50,20,100,20);
  ok = new JButton("OK");
  ok.addActionListener (this);
  ok.setBounds (60,60,60,20);
  dialog.getContentPane() .add (ok);
  dialog.setVisible (true);
```



Commentaires sur le JDialog

Dans l'instruction:

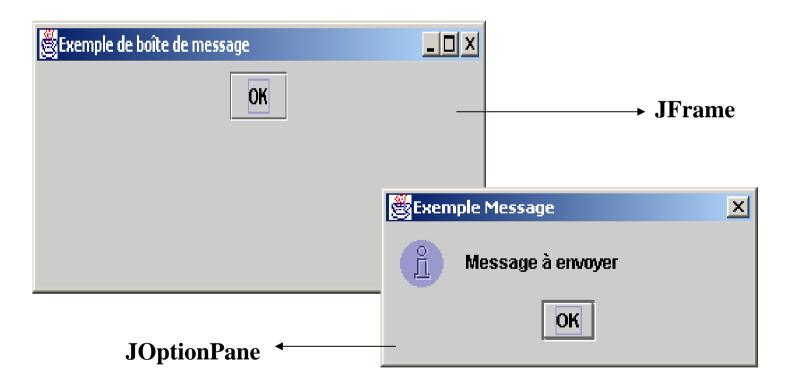
dialog = new JDialog(this, "Mon premier dialog", true); on a trois arguments: this désigne la fenêtre propriétaire (parent) c-à-d celle contenant le JDialog "Mon premier dialog " désigne le titre de la boîte de dialogue true la boîte de dialogue est modale c-à-d une fois lancée, l'utilisateur ne peut pas agir sur d'autres que ceux intégrés dans la boîte de dialogue.

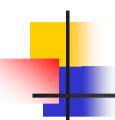
Remarque: il est possible (de la même façon qu'on utilise la classe JFrame) de créer une classe qui dérive de JDialog et d'y ajouter toutes les fonctionnalités dont on souhaite disposer.

Il est aussi possible de créer des boîtes de dialogue sans faire usage de la classe JDialog. C'est ce que nous allons voir dans le paragraphe suivant avec la classe JOptionPane.

La classe: javax.swing.JOptionPane

Les boîtes de Message: JOptionPane.showMessageDialog





Exemple message: JOptionPane.showMessageDialog

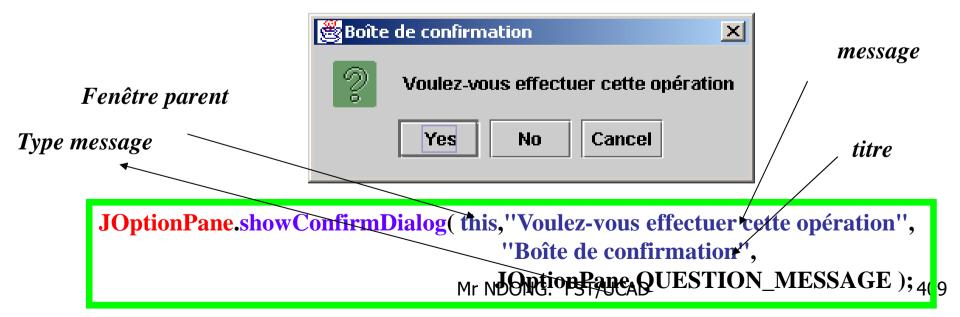
```
public class SwingMessage extends JFrame implements ActionListener{
JButton ouvre;
 public SwingMessage (String titre) {
  super(); this.setTitle(titre); this.setSize(400,150);
  this.getContentPane( ).setLayout( new FlowLayout());
  ouvre = new JButton("OK");
  ouvre.addActionListener (this);
                                                  Fenêtre parent
  this.getContentPane().add(ouvre);
                                                      Objet message
                                                                       Titre boîte
 public void actionPerformed(ActionEvent e)
 {if (e.getSource() = = ouvre)
  JOptionPane.showMessageDialog (this,"Message à envoyer","Exemple Message",
                                    JOptionPane.INFORMATION_MESSAGE, null);
                                                            Icon de la boîte ____
            Type du message
                                     Mr NDONG. FST/UCAD
                                                                                408
```

La classe: javax.swing.JOptionPane

Les boîtes de Confirmation: JOptionPane.showConfirmDialog

Une vraie application nécessite toujours le stockage de données sur disque ou sur tout autre média. Avant de faire des sauvegardes permanentes, il est aussi bon de demander une confirmation de la part de l'utilisateur.

Pour ce faire, en Java, on peut utiliser les boîtes de confirmation.





Les boîtes de confirmation apparaissent par défaut avec des boutons Yes, No et Cancel.

On peut souhaiter n'afficher que les boutons Yes et Cancel; dans ce cas utilisez la méthode showConfirmDialog (...) où le quatrième attribut permet de déterminer les boutons qui apparaîtront. Pour le cas évoqué, on fera

JOptionPane.showConfirmDialog (this,"Voulez-vous effectuer cette opération", "Boîte de confirmation", JOptionPane.OK_CANCEL_OPTION ,JOptionPane.QUESTION_MESSAGE);

Il se peut aussi qu'on veuille effectuer un certain traitement si l'on clique sur l'un des boutons Yes, No ou Cancel. Dans ce cas, récupérer la valeur renvoyée par la méthode showConfirmDialog sous une valeur de type entière (int). La valeur 0 correspond au clic sur Yes, 1 au clic sur No et 2 au clic sur Cancel.



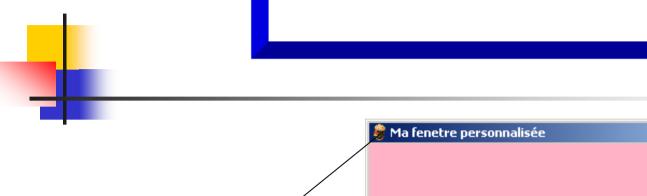
Remarques (2/2)

Il peut arriver qu'on veuille personnaliser le nom des boutons Yes, No et Cancel selon la langue utilisée.

Comment ferais t-on par exemple pour remplacer ces boutons par Oui, Non et Annuler?

Utilisez, pour ce faire la méthode showOptionDialog (...).

Personnalisation de l'icône d'un JFrame



Changer cette icône.

Exemple de code

```
ublic class TestJFrame4 {
public static void main(String argv[]) {
 JFrame f = new JFrame("Ma fenêtre personnalisée");
 f.setSize (400,400);
 JPanel b = new JPanel();
b.setBackground (new Color (255,25,0125,85));
f.getContentPane().add (b);
 /*rend le bouton de fermeture inactif*/
 f.setDefaultCloseOperation (WindowConstants.DO_NOTHING_ON_CLOSE);
 /*change l' icone de la barre de titre du JFrame*/
 ImageIcon image = new ImageIcon("d:/image000/ burger.gif");
 f.setIconImage (image.getImage());
 f.setVisible (true);
```

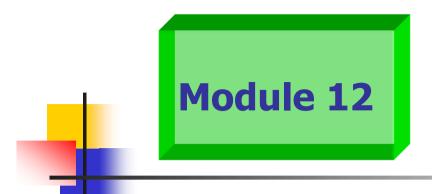
Remarque

Pour modifier l'icône de la fenêtre, vous pouvez utiliser un objet de la classe ImageIcon.

```
/*change l' icone de la barre de titre du JFrame*/
   ImageIcon image = new ImageIcon( "d:/image000/ burger.gif" );
   f.setIconImage (image.getImage());
   f.setVisible (true);
```

Vous pouvez aussi passer par la classe Toolkit:

```
// utilisation d'un toolkit pour l'affichage
    // d'une icone associée à la fenêtre
    Toolkit tk = Toolkit.getDefaultToolkit();
    Image JFrameIcon = tk.getImage( "d:/image000/ burger.gif" );
    setIconImage( JFrameIcon);
```



Entrées-Sorties: les flux

Un programme a souvent besoin d'échanger des informations pour recevoir d'une source des données ou pour envoyer des données vers un destinataire.

La source et la destination peuvent être de natures multiples: fichier, socket réseau, autre programme, etc....

La nature des données échangées peut également être diverse: texte, images, son, etc.

Entrée: le programme lit (reçoit) des données de l'intérieur,

Sortie: le programme écrit (envoie) des données vers l'extérieur.



Vue générale sur les flux

Un flux (stream) permet d'encapsuler des processus pour l'envoi et la réception de données.

Un flux traite toujours les données de façon séquentielle.

Par rapport à *la direction du flux*, on peut ranger les flux en deux familles:

- o les flux d'entrée (input stream)
- o les flux de sortie (output stream)

par rapport à *la nature des données* manipulées, on peut diviser les flux en deux grandes catégories:

- les flux de caractères
- les flux d'octets (ou flux binaires)
- Java définit des flux pour <u>lire</u> ou <u>écrire</u> des données mais aussi des classes pour traiter les données du flux. Ces classes doivent être associées à un flux de lecture ou d'écriture et sont considérées comme des <u>filtres</u>.



Vue générale sur les flux

Toutes les classes de manipulation de flux sont dans le paquetage java.io.

Remarque:

A partir du JDK 1.4 de nouvelles classes de flux sont introduites dans le paquetage java.nio. Nous y reviendrons lors de la programmation réseau avec les sockets.



Les classes de flux

L'utilisation de ces classes n'est pas chose facile, vu leur nombre élevé et la difficulté de choisir la classe convenable à un besoin précis.

Pour bien choisir une classe adaptée à un traitement donné, il faut comprendre la dénomination des différentes classes.

Le nom d'une classe est toujours composé d'un préfixe et d'un suffixe.

Il existe quatre suffixes possibles selon le type du flux (flux de caractères ou flux d'octets) et le sens du flux (flux d'entrée ou flux de sortie).



Les classes de flux: suffixes

flux	octets	caractères
entrée	InputStream	Reader
sortie	OutputStream	Writer



Les classes de flux

Il existe donc quatre hiérarchies de classes qui encapsulent des flux particuliers. Ces classes peuvent être séparées en deux séries de deux catégories différentes: les classes de lecture et d'écriture et les classes de manipulation

de caractères et d'octets.

- ✓ les sous classes de Reader sont des types de flux en <u>lecture</u> sur les caractères,
- ✓ les sous classes de Writer sont des types de flux en écriture sur les caractères,
- ✓ les sous classes de InputStream sont des types de flux en lecture sur les octets,
- ✓ les sous classe de **OutputStream** sont des types de flux en <u>écriture</u> sur les octets.



Les flux de caractères

Ces flux transportent des données sous forme de caractères et utilisent le format *Unicode* qui code les informations sur 2 octets.

Les classes qui gèrent les flux de caractères héritent d'une des deux classes abstraites **Reader** et **Writer**.

Il existe beaucoup de classes dérivées de celles-ci qui permettent de traiter les flux de caractères.

Un fichier texte sera considéré comme une séquence de *caractères* organisés en *lignes*.

Une fin de ligne est codée par:

- "\n" sous Unix et Linux
- "\r\n" sous MS-DOS et Windows
- System.getProperty ("line.separator") en Java.



Les flux de caractères

Il y a plusieurs classes de lecture /écriture.

BufferedReader / BufferedWriter

CharArrayReader / CharArrayWriter

FileReader / FileWriter

InputStreamReader / OutputStreamWriter

LineNumberReader

PipedReader / PipedWriter

PushbackReader

StringReader / StringWriter





Les flux dont le préfixe est:

File gère et manipule les fichiers,

CharArray gère et manipule le tableau de caractères en mémoire

String gère et manipule les chaînes de caractères

Pipe gère et manipule un pipeline entre deux threads

LineNumber: filtre pour numéroter les lignes contenues dans le flux

InputStream/OutputStream: filtre pour convertir des octets en caractères



La classe Reader

C'est une classe abstraite qui est la classe de base de toutes les classes qui gèrent des flux texte d'entrée.

quelques méthodes:

```
void close () // ferme le flux et libère les ressources qui lui étaient associées int read () // renvoie le caractère lu et -1 si la fin du flux est atteinte boolean ready() // indique si le flux est prêt à être lu boolean markSupported () // indique si le flux supporte la possibilité de marquer // des positions
void mark (int ) // permet de marquer une position dans le flux void reset () // retourne dans le flux à la dernière position marquée
```



La classe java.io.FileReader

Cette classe permet de gérer les fichiers texte d'entrée. Elle dispose des mêmes fonctionnalités que la classe abstraite Reader.

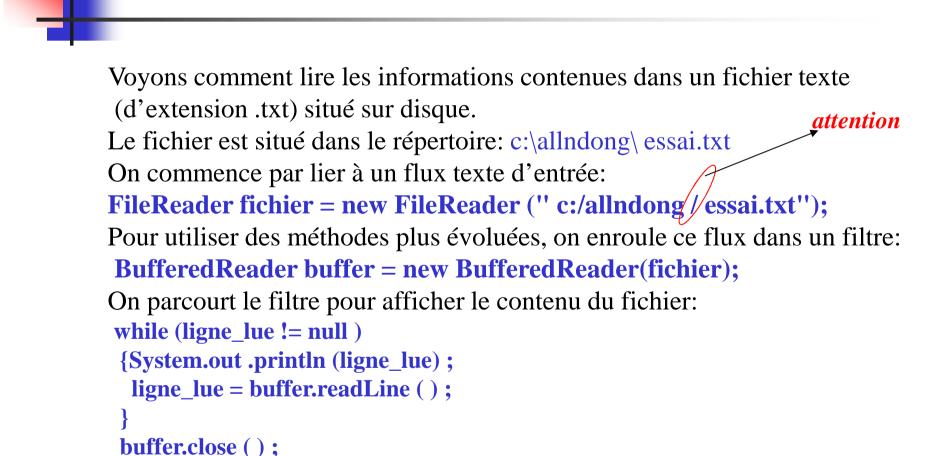
Mais les méthodes de cette classe sont rudimentaires, c'est pourquoi, il faut souvent la coupler à un objet de la classe **BufferedReader** qui possède des méthodes plus appropriées et complètes à la manipulation de fichiers textes en lecture.

La classe **BufferedReader** est doté d'un tampon et d'une méthode **String readLine** () pour la lecture d'une ligne.

Parmi les constructeurs de FileReader, on peut distinguer:

FileReader (File objectFile)
FileReader (String fileName)





fichier.close();

Lecture d'un fichier texte

```
import java.io .*; // pour utiliser FileReader et BufferedReader
public class TestFileReader {
 public static void main (String[] args) throws IOException{
FileReader fichier = new FileReader ("c:/allndong/essai.txt");
BufferedReader buffer = new BufferedReader (fichier);
String ligne_lue = buffer.readLine();
 while (ligne_lue != null )
                                              Pour les éventuelles exceptions
 { System.out .println (ligne_lue);
                                              sur les classes d'I/O.
   ligne_lue = buffer.readLine();
 buffer.close ( ); // on ferme le filtre
 fichier.close ( ); // on ferme le flux
                                            java est un langage interessant.
                                             N'est ce pas?
```



Remarques

Les deux instructions:

FileReader fichier = new FileReader ("c:/allndong/essai.txt"); BufferedReader buffer = new BufferedReader (fichier);

peuvent être remplacées par l'unique instruction:

BufferedReader buffer = new BufferedReader (new FileReader ("c:/allndong/essai.txt"));

Et dans ce cas on ne ferme que le filtre puisque contenant le flux, le faite de le fermer ferme en même temps le flux associé.

On verra une autre façon de créer le flux en utilisant la classe **File**.



La classe java.io.File

La classe **File** permet la gestion des fichiers et des répertoires (création, suppression, renommage, copie, etc....).

L'instruction File monfichier = new File ("memoire.txt");

crée un objet monfichier associé au fichier memoire.txt, mais elle ne crée pas le fichier texte (ie memoire.txt) en tant que tel.

On peut vérifier l'existence d'un fichier avec la méthode boolean exits ().

Si le fichier memoire.txt existe dans le système, on peut créer un objet de type File en mentionnant au constructeur un nom de répertoire:

File monfichier = new File ("c:/memoire.txt");

Si le fichier memoire.txt n'existait pas, il serait possible de le créer explicitement sur disque en utilisant la méthode createNewFile () qui renvoie un booléen (true si la création a réussi, false sinon):

monfichier.createNewFile();





```
import java.io .*;
public class TestFile {
 public static void main(String[] args) {
  File file = new File ("c:/memoire.txt");
  System.out .println (file.exists ()); // true car memoire.txt existe sur disque
  File f = new File("c:/memoire2.txt");
  System.out .println(f.exists());// false car memoire2.txt n'existe pas encore
  try {
       boolean creation = f.createNewFile ();// memoire2.txt est créé sur disque
  catch (IOException ex)
     {ex.printStackTrace();}
  System.out .println (f.exists ( ) );//true car memoire2.txtx existe maintenant
```

On protège la méthode createNewFile() des éventuelles erreurs de type IOException





Pour la création d'un fichier, on utilisera souvent l'un des deux constructeurs:

File (File fileName) // on transmet le nom simple du fichier

File (String pathName) // on transmet le répertoire complet du fichier

Si vous créez un fichier en spécifiant un nom de répertoire, il s'agit en fait d'un chemin qui peut être :

- absolu: c à d spécifié intégralement depuis la racine du système de fichiers;
- relatif: c à d se référer au répertoire courant.

Si vous créez un programme qui doit être déployé dans un environnement unique, utilisez les conventions en vigueur dans cet environnement:

- + sous Windows, les chemins absolus commencent par X (X étant le nom d' un lecteur, ex C:, D:, E:) ou par \
- +sous Unix, le chemin absolu commence par /



Constructeur de File

Si le programme doit être déployé dans un environnement multiplatforme, il faut tenir compte de la portabilité et pour cela il faut fournir le séparateur en vigueur dans chaque environnement concerné en utilisant la constante de type **String File.separator**.

NB: il est existe aussi deux autres constructeurs de la classe File que vous pouvez souvent utiliser:

File (File repertoire, String simpleNomFichier)

File (String repertoire, String simpleNomFichier)

Exemples



Création valable dans l'environnement Windows



Méthodes de la classe File

```
/*crée un nouveau fichier qui n'existait pas, renvoie true si la création réussie*/
                   boolean createNewFile()
/* essaie de supprimer le fichier, renvoie true si la suppression réussie
                   renvoie false si le fichier n'existe pas*/
                    boolean delete ()
/*crée un répertoire ayant le nom spécifié, renvoie true si la création s'est
              déroule correctement. Seul le dernier niveau du répertoire peut être créé*/
                   boolean mkdir ()
/* idem que mkdir mais avec possibilité de création d'éventuels
                    niveaux intermédiaires de répertoires*/
                   boolean mkdirs ()
/* renvoie true si le fichier correspondant existe*/
                   boolean exits ()
/*renvoie true si l'objet correspond à un nom de fichier (même si le fichier
                  en tant que tel n'existe pas*/
                    boolean isFile ( )
                                         Mr NDONG. FST/UCAD
                                                                                       434
```



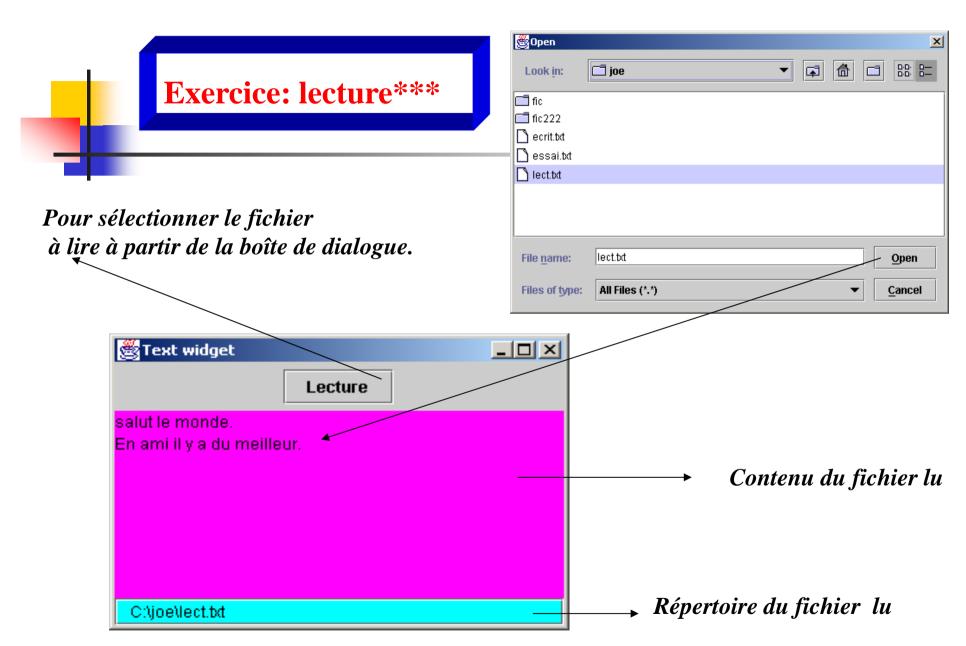


Méthodes de la classe File

Il est possible de connaître tous les répertoires et fichiers d'un répertoire donné en utilisant l'une des deux méthodes ci-dessous:

String [] list () // fournit les éléments du répertoire sous forme d'un tableau //de chaînes

File[] listFiles () // fournit les éléments du répertoire sous forme d'un tableau // de fichiers





Exercice : lecture***

Réaliser une classe qui permet de lire un fichier (.txt) situé sur disque à partir de l'interface graphique précédent. On utilisera les classes vues précédemment. On prévoira:

- une méthode void ouvrirFichierEnLecture () permettant d'ouvrir la boîte de dialogue du système de fichiers de Windows pour sélectionner le fichier à lire. Cette méthode appellera *lireFichier* pour lire le fichier sélectionné.
- une méthode void lireFichier (File nom) qui permet d'écrire le contenu du fichier lu dans l'interface graphique.

On ne peut lire que les fichiers qui existent et qu'on peut ouvrir en lecture.

Pour les boîtes de sélection de fichier sous Windows, utilisez la classe JFileChooser ou la classe FileDialog.



Exercice lecture***: **Corrigé express**

```
import javax.swing .*;
public class FileReading {
public void ouvrirFichierEnLecture() {
JFileChooser fileChooser = new JFileChooser ();//pour créer une boîte de dialogue
int retval = fileChooser.showOpenDialog (this); // pour ouvrir la boîte de dialogue
          if (retval == JFileChooser.APPROVE OPTION)
            File fichier = fileChooser.getSelectedFile (); // on sélectionne un fichier
            String chemin = fichier.getAbsolutePath (); // on extrait le chemin absolu
            annonce.setText (" "+chemin);
          if ( fichier.isFile( ) && fichier.canRead ( ))
              lireFichier(fichier); // appelle la méthode de lecture du fichier sélectionné
```





```
public void lireFichier (File nom)
           try { java.io.BufferedReader isr = new BufferedReader ( new
                FileReader (nom));
                String s = isr.readLine(); // lecture d'une ligne du fichier
                while (s!= null )
                            { texte.append(s+"\n");
                              s = isr.readLine();
               isr.close();
          catch (IOException e) {annonce.setText("Erreur de lecture du fichier");}
} //fin de FileReading
```

Exercice lecture***: Notes

```
On peut utiliser un objet de type <u>StringReader</u> pour la lecture de caractères.
Cette classe représente un flux de caractères dont la source est une chaîne de caractères.
  /*on peut remplacer la méthode lireFichier précédente par celle-ci*/
         public void lireFichier (File nom)
                   StringReader sr = new StringReader (nom+"");
          try {
                  BufferedReader br = new BufferedReader (sr);
                  String s = br.readLine();
              while (s!= null)
                            { texte.append(s+"\n");// lecture d'un fichier .txt
                              s = br.readLine();
             br.close();
             sr.close();
             catch (IOException e){ annonce.setText ("Erreur de lecture du fichier");}
                                      Mr NDONG. FST/UCAD
                                                                                    441
```



java.io.StreamTokenizer

La classe **StreamTokenizer** prend un flux d'entrée et l'analyse en jetons (« tokens »), autorisant les jetons à être lus à un moment.

Cette classe réalise un analyseur lexical qui considère plusieurs types de tokens:

- o **nombre**
- o mot (délimité par des espaces)
- o chaîne de caractères
- o commentaires
- o EOF (End Of File)
- o EOL (End Of Line)

La méthode nextToken() permet d'appeler l'analyseur.



champs

```
/*si le token courant est un nombre, ce champ contient sa valeur*/
double nval
/*si le jeton courant est un mot, sval contient une chaîne représentant ce mot*/
String sval
/*EOF représente une constante indiquant que la fin du flux a été lue*/
static int TT EOF
/*EOL représente une constante indiquant que la fin de la ligne a été lue*/
static int TT EOL
/*ce champ représente une constante indiquant qu'un nombre a été lu*/
static int TT_NUMBER
/*ce champ représente une constante indiquant qu'un mot a été lu*/
static int TT_WORD
/*après un appel à la méthode nextToken, ce champ contient le type du jeton
venant d'être lu*/
int ttype
```

java.io.StreamTokenizer: champs et méthodes

Méthodes

```
/*spécifie que l'argument de type char démarre un commentaire de ligne unique*/
void commentChar ( int ch)
/*détermine si les fins de ligne doivent être traitées comme des jetons*/
void eollsSignificant (boolean flag)
/*retourne le numero de ligne courante*/
int lineno ()
/* détermine si le mot doit être automatiquement converti en minuscule*/
void lowerCaseMode (boolean b)
/*analyse le prochain jeton à partir du flux d'entrée de l'objet StreamTokenizer*/
int nextToken ( )
/*spécifie que l'argument de type char est ordinaire dans l'objet StreamTokenizer*/
void ordinaryChar (int ch)
/*spécifie que tous les caracteres dans l'intervalle [low, hi]sont ordinaire dans l'objet
 StreamTokenizer*/
 void ordinaryChars (int low, int hi)
/* retourne un objet String représentant l'objet Stream Tokenizer courant*/
Mr NDONG. FST/UCAD
                                                                                   444
String toString()
```

Exercice StreamTokenizer

Ecrire une classe ArrayListStreamTokenizer qui permet de créer un vecteur dynamique (ArrayList) constitué de tous les mots d'un fichier texte (.txt). Le ArrayList sera formé de tous les mots séparés par des espaces blancs, apparaissant dans le fichier spécifié. On prévoira:

- une méthode ArrayList arrayListFromStreamTokenizer (File fic) qui prend en argument un fichier et stocke dans un vecteur tous ses mots.
- une méthode void printArray () qui affiche le vecteur obtenu.

Voici une classe de test:

```
public class TestStreamTokenizer {
  public static void main (String[] args) throws FileNotFoundException, IOException {
    File fichier = new File (''C:\|joe\\memoire.txt'');
    ArrayListStreamTokenizer essai = new ArrayListStreamTokenizer ();
    essai.arrayListFromStreamTokenizer (fichier);
    essai.printArray ();
}
```

Exercice StreamTokenizer: Corrigé (1/2)

```
public class AnalyseurStreamTokenizer {
ArrayList liste:
ArrayList arrayListFromStreamTokenizer (File fic) throws FileNotFoundException,
                     IOException
{ liste = new ArrayList ( );
BufferedReader br = new BufferedReader (new FileReader (fic));
 StreamTokenizer st = new StreamTokenizer (br);
 int unMotTrouve = StreamTokenizer.TT_WORD ; // un mot est lu (-3)
 while (unMotTrouve != StreamTokenizer.TT_EOF )
 { unMotTrouve =st.nextToken ( ); // jeton courant (si mot alors = -3)
  if (unMotTrouve == StreamTokenizer.TT_WORD )
    \{ String s = st.sval ; \}
     liste.add (s);
 return liste;
                                     Mr NDONG. FST/UCAD
```



Exercice StreamTokenizer: Corrigé (2/2)

```
void printArray(){
   System.out .println (liste.toString ( ) );
}
} // fin de la classe
```

Remarque:

Dans l'en-tête de la méthode arrayListFromStreamTokenizer, nous avons capturé par throws FileNotFoundException, IOException

les éventuelles exceptions qui seraient générées respectivement par l'emploi de FileReader et par l'appel de la méthode nextToken().



Filtre sur les noms de fichiers

Un filtre est un objet qui permet de ne prendre en compte qu'un certain nombre de <u>types</u> de fichiers.

Pour cela, on utilisera la méthode (de la classe File):

/*prend en argument un filtre et renvoie sous forme de tableau de chaînes tous les fichiers vérifiant les conditions du filtre*/
public String [] list (FilenameFilter filtre)

Toute classe qui définit un filtre sur un fichier doit implémenter l'interface FilenameFilter et donc redéfinir la seule méthode cette dernière:

/*là on définit les conditions du filtre*/
public boolean accept (File rep, String nom).





```
import java.io .*;
                           FilenameFilter
class Filtre implements
public boolean accept (File rep, String nom) {
/*on définit les conditions du filtre*/
if (rep.isDirectory () && nom.endsWith(".java")) return true;
else
                                                     return false:
public class FiltreTest {
public static void main (String args[]){
 new FiltreTest().affiche("."); // test: on filtre le repertoire racine (.)
public void affiche(String rep) {
 File fichier = new File (rep); // on yeut filtrer ce repertoire
 String nomFics [] = fichier.list ( new Filtre() );// on lui associe donc un filtre
 for (int i = 0; i < nomFics.length; i++)
  System.out.println (nomFics T) ONG. FST/UCAD
                                                                             449
```



La classe Writer

C'est une classe abstraite qui est la classe de base de toutes les classes de flux de caractères en écriture.

Elle définit quelques méthodes rudimentaires:

```
/*ferme le flux et libère les ressources qui lui étaient associées*/
void close ()
/*écrire le caractère en paramètre dans le flux*/
void write (int)
/* écrire le tableau de caractères dans le flux*/
void write (char [])
/*écrit le tableau de caractères tab, i = indice du premier caractère à écrire
        j = le nombre de caractères à écrire*/
void write (char [], int i, int j)
/*écrire la chaîne de caractères en paramètre dans le flux*/
write (String)
/*écrire une chaîne à partir du caractère i j étant le nombre de caractères à écrire*/
write (String, int i, int j)
```



La classe java.io.FileWriter

Cette classe gère les flux de caractères en <u>écriture</u>. Elle possède plusieurs constructeurs qui permettent un ou plusieurs caractères dans le flux:

/*Si le nom spécifié n'existe pas alors le fichier sera créé. S'il existe et qu'il contient des données, celles-ci seront écrasées*/

FileWriter (String nomfichier)

/* Idem que précédemment mais ici le fichier est précisé avec un objet de la classe File* /

FileWriter (File nomfichier)

/*le booléen permet de préciser si les données seront ajoutées au fichier (valeur true) ou écraseront les données existantes (valeur false)*/
FileWriter (String, boolean)



La classe java.io.FileWriter

Pour améliorer les performances des flux d'écriture sur un fichier texte, la mise en tampon des données écrites permet de traiter un ensemble de caractères plutôt que traiter caractère par caractère. La mise en tampon entraîne un gain de temps.

La classe BufferedWriter permet de tamponner un flux en écriture sur un fichier texte.

```
File fichierOuEcrire = new File (fileName);
FileWriter fw = new FileWriter (fichierOuEcrire);
BufferedWriter bw = new BufferedWriter (fw);
```

Et pour fermer les flux, on suit l'ordre inverse de leur création:

```
bw.close();
fw.close();
```

Ces trois instructions peuvent être rassemblées en une seule :

```
BufferedWriter bw =new BufferedWriter (new FileWriter (new File (fileName)));
```

```
Ici on ne ferme que le flux externe; ça suffit: bw.close ();
Mr NDONG. FST/UCAD
```

Ecrire dans un fichier texte

```
public class TestFileWriter {
public static void main(String[ ] args)throws IOException {
File fichier = new File("c:\\allndong\\ essai.txt");
FileWriter fw = new FileWriter( fichier.toString ( ),true);
BufferedWriter bw = new BufferedWriter (fw); // flux tamponne
int i = 0;
 while (i < 5)
  bw.newLine();// on cree une nouvelle ligne dans le fichier
  bw.write (" bonjour le monde"); // et on ecrit cette chaine
  i++;
 bw.close();
 wr.close();
```



Exercice: écriture***

Reprendre l'exercice lecture*** et ajoutez-y deux autres méthodes:

- -une méthode void ouvrirFichierEnEcriture () qui permet d'ouvrir la boîte de dialogue pour ouvrir un fichier (.txt) afin d'y écrire. Cette méthode appellera la méthode *ecrireFichier* pour écrire dans le fichier sélectionné.
- une méthode void ecrireFichier (File fic) qui permet d'écrire sur un fichier.

Il faut augmenter bien sûr un bouton <u>Écriture</u> dans l'interface graphique qui gère l'évènement lié à la sélection et l'écriture dans le fichier texte.



Exercice: écriture***: Corrigé

//METHODE POUR ECRIRE DANS UN FICHIER

```
public void ecrireFichier(File nom)
{
    try {
        BufferedWriter bw = new BufferedWriter(new FileWriter (nom+"",true));
        bw.write (texte.getText());
        texte.setText("");
        bw.close();
        }
        catch (IOException e){annonce.setText("Erreur d'écriture dans le fichier");}
}
```



Exercice : écriture***: Corrigé

//METHODE POUR SELECTION D' UN FICHIER OU ECRIRE

```
public void ouvrirFichierEnEcriture()
{ int retval = fileChooser.showSaveDialog (this);
  if (retval == JFileChooser.APPROVE_OPTION){
    File fichier = fileChooser.getSelectedFile();
    String ap = fileChooser.getSelectedFile().getAbsolutePath();
    annonce.setText(""+ap);
    ecrireFichier(fichier);
}
```

La classe java.io.LineNumberReader

(cette classe permet de numéroter les lignes d'un flux de caractères)

```
/** Exemple de copie d'un fichier texte dans un autre fichier avec numérotation des
    lignes. La première ligne porte le numéro 1.
 */
public class NumeroLigne {
 public static void main (String [] st) throws IOException {
File src = new File ("c:\\ joe\\ fex.txt"); // on copie ce fichier
File dest = new File ("c:\\ joe\\ newT.txt"); // dans celui-ci avec des numéros de lignes
LineNumberReader in = new LineNumberReader (new FileReader (src));
BufferedWriter out = new BufferedWriter (new FileWriter (dest));
   String s = in.readLine();
   while (s != null) {
    out.write (in.getLineNumber () + ": " + s + "\r\n");
                                                                // pour marquer une fin de
    s = in.readLine();
                                                                //ligne.
   in.close();
   out.close();
}}
```



La classe java.io.PrintWriter

Cette classe permet d'écrire dans un flux des données *formatées*. Le formatage autorise la manipulation simultanée *des données de types différents* dans un même flux.

Cette classe dispose de plusieurs constructeurs:

```
/*la paramètre fourni précise le flux, le tampon est automatiquement vidé*/
PrintWriter (Writer)
/*le booléen précise si le tampon doit être automatiquement vidé*/
PrintWriter (Writer, boolean)
/*la paramètre fourni précise le flux, le tampon est automatiquement vidé*/
PrintWriter (OutputStream)
/*le booléen précise si le tampon doit être automatiquement vidé*/
PrintWriter (OutputStream, boolean)
```

Cette classe s'applique à la fois pour les flux de sortie de caractères et d'octets.



La classe java.io.PrintWriter

La classe PrintWriter présente plusieurs méthodes print (anyType var) prenant un argument <u>var</u> de type <u>anyType</u> où <u>anyType</u> peut être n'importe quel type primitif (int, long, double, boolean,), type String, Object ou tableau de caractères (char []) permettant d'envoyer des données formatées dans le flux.

La méthode println () permet de terminer la ligne dans le flux en y écrivant un saut de ligne.

Il existe aussi plusieurs méthodes println (any Type var) faisant la même chose que la méthode print mais les données sont écrites avec <u>une fin de ligne</u>.

La classe PrintWriter présente un e méthode spéciale void flush () qui vide le tampon (sans que le flux ne soit fermé) en écrivant les données dans le flux. Les méthode write (...) de cette classe hérite de Writer.

Exemple PrintWriter

```
public class TestPrintWriter {
String fichierDest;
 public TestPrintWriter (String fichierDest) throws IOException {
 this.fichierDest = fichierDest:
 traitement ();
 private void traitement ( ) throws IOException
 { PrintWriter pw = new PrintWriter (new FileWriter (fichierDest));
  pw.println ("bonjour monsieur");
  pw.write (100); //écrire le caractère de code 100 c à d 'd'
  pw.println(); //mettre un saut de ligne
  pw.println ("votre solde est"+ 10000);
  pw.print ("nous sommes le:"+ new java.util.Date ( ));
  pw.close ( ); //fermer le flux pour que les données soit écrites
 public static void main (String[] args) throws IOException {
  TestPrintWriter tp = new TestPrintWriter("c:\\joe\\ecrit.txt");
 }}
```



Exercice avec PrintWriter

Utiliser la classe PrintWriter Réaliser une classe CopyFile qui permet de copier le contenu d'un fichier texte dans un autre fichier texte. On prévoira:

-une méthode void copieFichier (File src, File dest) permettant de copier le contenu du fichier <u>src</u> dans le fichier <u>dest</u>. La copie sera conforme (ie les données apparaîtront dans les deux fichiers exactement de la même manière)

on suppose que le fichier dest peut ne pas exister, dans ce cas, il faut prévoir:

- la méthode void createFile (File f) qui permet de créer le fichier f s'il n'existe pas encore. Avant de réaliser la copie dans copieFichier, il faut s'assurer que le fichier dest existe ou pas en appelant createFile qui créera éventuellement ce fichier.

```
/*une classe de test pour cet exercice*/
public class TestCopy {
public static void main(String[] args) {
 File s = new File ("c:\\joe\\source.txt");
  File d = new File ("c:\\joe\\dest.txt"); // si dest.txt n'existe pas, il est alors créé
  new CopyFile ().copieFichier (s,d); // et la copie est effectuée ici
                                      Mr NDONG. FST/UCAD
```

Exercice avec PrintWriter : Corrigé

```
public class CopyFile {

public File createFile (File f)
{ if (f.exists () = = false)
{ try { f.createNewFile ();
     }
     catch (IOException e)
     { e.printStackTrace ();}
}
return f;
}
```

```
public void copieFichier (File src, File dest)
{ dest = createFile(dest); // pour être sur que dest existera
try {
     FileReader ficsrc = new FileReader (src);
     BufferedReader b = new BufferedReader (ficsrc);
 PrintWriter pw=new PrintWriter(new FileWriter (dest));
 String lg = b.readLine();
 while (lg!= null)
 { pw.write (lg); // on écrit la ligne courante lue
   pw.println(); // on met un saut de ligne
   lg = b.readLine (); // on récupère la ligne suivante
pw.close();
b.close();
 catch (FileNotFoundException ee){ee.printStackTrace();
 catch (IOException er){er.printStackTrace();}
        Mr NDONG. FST/UCAD
                                                     462
```



Les flux d'octets

Ce type de flux permet le transport de données sous forme d'octets. Les flux de ce type sont capables de transporter toutes les données.

Les classes qui gèrent les flux d'octets héritent d'une des deux classes abstraites InputStream (flux binaire d'entrée) et OutputStream (flux binaire de sortie). Ces deux classes abstraites disposent de méthodes rudimentaires:

Pour OutputStream

void write (int) //écrit un octet
void write (byte []) // écrit un tableau d'octet
void close () //ferme le flux
void flush () //vide le tampon

Pour InputStream



Cette classe permet de gérer les flux binaires en <u>lecture</u> sur un fichier.

Cette classe possède <u>plusieurs constructeurs</u> qui peuvent tous lever une exception de type <u>FileNotFoundException</u>.

```
/*ouvre un flux en lecture sur un fichier de nom la chaîne fic*/
FileInputStream (String fic)
/*ouvre un flux en lecture sur un fichier dont le nom est un objet de type File*/
FileInputStream (File filename)
```

Cette classe hérite des méthodes de la classe *InputStream* mais elle définit aussi d'autres méthodes qui permettent de lire un ou plusieurs octets dans le flux.

La méthode int available () permet de connaître le nombre d'octets qu'il est encore possible de lire dans le flux.

Exemple de lecture d'un fichier binaire

```
im java.io .*;
public class TestFileInputStream {
 public static void main(String[] args) {
 File f = new File("c:\\joe\\entree.dat"); // un objet File sur le fichier à lire
 try { FileInputStream fis = new FileInputStream (f);// on associe un flux au fichier à lire
       byte [ ] tab = new byte [(int) f.length ( ) ];
       try { while (fis.read ( ) !=-1) // on s' arrete à la fin du fichier (-1)
             fis.read (tab,0,(int) f.length () -1); // on lit les octets qu'on stocke dans un tableau
             fis.close( );
      catch (IOException er){System.out .print("Erreur de lecture") ;}
      for (int i = 0; i < tab.length; i++)
           System.out .println (tab [i]);// on affiche chaque octet lu
 catch (FileNotFoundException ee) {System.out .println (ee.getMessage ( ) );}
                                          Mr NDONG. FST/UCAD
                                                                                          465
```





La méthode read () peut lever une exception de type IOException, d'où la nécessité de capturer les éventuelles exceptions q'elle peut générer par un gestionnaire try catch (IOException e).

Beaucoup de méthodes de lecture ou d'écriture dans un flux lèvent

une exception de ce type, donc à capturer.

Il en de même du constructeur FileInputStream qui peut lever une exception de type FileNotFoundException. Ici aussi, il faut la capturer.





On peut doter les flux binaires d'entrée d'un tampon qui permet d'avoir des opérations plus rapides puisque les données sont lues en rafales.

Pour ce faire, on utilise la classe **BufferedInputStream**.

Les octets sont lus dans un tableau tampon interne qui est créé simultanément avec la création de l'objet *BufferedInputStream*.

Il existe deux constructeurs de cette classe:

BufferedInputStream (InputStream in)

BufferedInputStream (InputStream in, int tailleTampon)

L'exemple précédent pouvez être amélioré en dotant le flux créé d'un tampon comme ceci:

File $f = new File ("c: \ \ joe \ entree.dat");$

FileInputStream fis = new FileInputStream (f);

BufferedInputStream bis = new BufferedInputStream (fis);



Cette classe permet de gérer les flux binaires en <u>écriture</u> sur un fichier.

Cette classe possède plusieurs constructeurs qui peuvent lever tous une exception de type : FileNotFoundException

/*ouvre un flux en écriture sur un fichier de nom la chaîne fic si le fichier n'existe pas, il sera créé, Si il existe et qu'il contient des données celles-ci seront écrasées*/

FileOutputStream (String fic)

/*ici le boolean précise si les données seront rajoutées au fichier (valeur true) ou écraseront les données existantes (valeur false)*/

FileOutputStream (String fic, boolean b)

/*ouvre un flux en écriture sur un fichier dont le nom est un objet de type File si le fichier n'existe pas, il sera créé, Si il existe et qu'il contient des données celles-ci seront écrasées*/

FileOutputStream (File filename)

Cette classe hérite des méthodes write (m) de la classe QutputStream

Exemple de copy de fichier binaire

(on réalise la copie d'un fichier dans un autre fichier avec usage de tampon)

```
Pour traiter un ensemble d'octets au lieu
public class TestFileOutputStream {
                                             de traiter octet par octet. Le nombre
void copyFile (File src, File dest){
                                               d'opérations est alors réduit.
try {
BufferedInputStream bis = new BufferedInputStream (new FileInputStream(src));
BufferedOutputStream bos = new BufferedOutputStream(new FileOutputStream(dest ));
while (bis.available () > 0) // tant qu'il reste des octets à lire
     bos.write (bis.read ( ) ); // on écrit l'octet lu
     bis.close(); bos.close();
catch (FileNotFoundException e) { System.out .println ("Erreur sur les flux") ;}
catch (IOException er) {System.out .println("Erreur de read ou write") ;}
 public static void main(String[] args) {
  File dest = new File("c:\\joe\\ecrit.dat");
  File src = new File("c:\\joe\\lecture.dat");
  new TestFileOutputStream( ).copyFile (src,dest) ;
 }}
                                       Mr NDONG. FST/UCAD
                                                                                     469
```





Les classes de flux utilisées ici lèvent toutes une exception de type FileNotFoundException.

Les méthodes read () et write(...) lèvent aussi une exception de type IOException.

Pour gérer ces exceptions, nous avons un bloc try suivi de deux blocs catch pour les deux types d'erreurs évoquées.

Pour gérer ces mêmes exceptions, au lieu de bloc try ... catch, on pourrait Mentionner après l'en-tête de la méthode:

throws FileNotFoundException, IOException.

Mais, il faut éviter souvent d'utiliser cette dernière possibilité pour des méthodes de l'API que vous redéfinissez. Chacune d'elles gère des exceptions spécifiques.

Par ailleurs, au lieu de deux blocs <u>catch</u>, on pouvait n'utiliser qu'un seul bloc catch de cette façon: <u>catch</u> (<u>Exception er</u>).

Classe java.io.DataInputStream

Classe java.io.DataOutputStream



Nous avons vu que la classe FileInputStream disposait de méthodes rudimentaires qui permettent de lire seulement un octet ou un tableau d'octet. Même associé à un tampon les performances se limitent tout simplement à une réduction du nombre d'opérations de lecture.

La classe **DataInputStream** dispose de méthodes plus évoluées (de lecture de tous les types de données) et l'un de ses constructeurs prend un argument de type **FileInputStream**.

Il en de même de la classe FileOutputStream qu'on peut lier dans un DataOutputStream pour disposer de méthodes plus évoluées d'écriture de données de n'importe quel type.

Ces deux classes améliorent donc considérablement la manipulation des fichiers binaires.

Exemple de copy de fichier binaire avec Data In/Out put Stream

```
public class TestCopy {
void copyFile (File src, File dest){ // pour copier src dans dest
try {
DataInputStream bis = new DataInputStream (new FileInputStream(src));
DataOutputStream bos = new DataOutputStream(new FileOutputStream(dest ));
while (bis.available () > 0) // tant qu'il reste des octets à lire
    { bos.writeChars (bis.readLine ( ) ); bos.writeChars("\n\r");}
bis.close(); bos.close();
catch (FileNotFoundException e) { System.out .println ("Erreur sur les flux") ;}
catch (IOException er) {System.out .println("Erreur de read ou write") ;}
 public static void main(String[] args) {
  File dest = new File("c:\\joe\\ecrit.dat");
  File src = new File("c:\\joe\\lecture.dat");
  new TestCopy( ).copyFile (src,dest) ;
 }}
                                      Mr NDONG. FST/UCAD
```

Classe java.io.RandomAcessFile

Cette classe gère les <u>fichiers à accès direct</u> permettant ainsi un accès rapide à un enregistrement dans un fichier binaire.

Il est possible dans un tel fichier de mettre à jour directement un enregistrement.

La classe RandomAccessFile enveloppe les opérations de lecture/écriture dans un fichier.

Elle implémente deux interfaces: DataInput et DataOutput.

Elle possède deux constructeurs:

RandomAccessFile (String nomfichier, String modeacces)

RandomAccessFile (File nomfichier, String modeacces)

Le mode d'accès est une chaîne qui est égale à:

```
+ "r": mode lecture
```

+ "rw": mode lecture/écriture.

Les constructeurs de cette classe lancent tous les exceptions:

- FileNotFoundException si le fichier n'est pas trouvé,
- IllegalArgumentException si le mode n'est pas "r" ou "rw",
- Security Exception si le gestionnaire de sécurité empêche l'accès aux fichiers dans le mode précisé.

Exemple de lecture/écriture dans un fichier

```
public class TestReadWrite {
void readWriting (String src, String dest ) throws FileNotFoundException, IOException{
RandomAccessFile raf = new RandomAccessFile (src,"r");// en lecture seule
RandomAccessFile rw = new RandomAccessFile (dest,"rw"); // en lecture/ ecriture
while (raf.read ( ) != -1)
  rw.writeChars (raf.readLine());
raf.close();
rw.close();
 public static void main(String [ ] args) throws FileNotFoundException,IOException {
  String fic1="c:\\joe\\lecture.dat"; // on copie ce fichier
  String fic2="c:\\joe\\ecrit.dat"; // dans celui-ci
  new TestReadWrite( ).readingWriting (fic1,fic2);
```

Exemple d'accès direct aux enregistrements

```
100 4
public class TestWriteInt {
 public static void main(String [ ] args) {
                                                                                200 8
 String fic = "c:\\joe\\destination.txt";
try { RandomAccessFile rw = new RandomAccessFile (fic,"rw");
                                                                                300 12
 /*ecriture dans le fichier dix entiers*/
                                                                                400 16
    for ( int i = 0; i < 10; i++)
   { rw.writeInt (i*100); }
                                                                                500 20
   /*lecture du fichier: accès aux enregistrements*/
   for (int i = 0; i < 10; i++)
                                                                                600 24
    { long pos = rw.getFilePointer(); // position courante du pointeur
      rw.seek (4*i);
                                                                                700 28
      System.out println (rw.readInt()+""+pos);
                                                                                800 32
rw.close();
                                            ième enregistrement.
                                                                                900 36
catch (Exception e){ }
                          Taille des données (un int est codé sur 4 octets)
                                                                                     475
                                      MIT INDUNG. FST/UCAD
}}
```



Entrées/Sorties standards

Il s'agit de flux relatifs à l'utilisation de la classe System du paquetage java.lang. Cette classe contient plusieurs méthode utilitaires et ne peut être instanciée.

Les entrées-sorties associées à cette classe se traduisent par l'utilisation de variables définissant trois types de flux:

- static PrintStream err pour le flux d'erreur standard,
- static InputStream in pour le flux d'entrée standard,
- static PrintStream Out pour le flux de sortie standard.

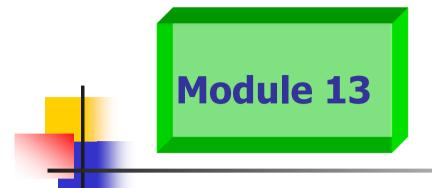
Le flux de sortie standard défini par la variable out est plus "connu" et est très facile à manier dans des instructions du genre:

System.out.println ("usage du flux de sortie pour l'affichage sur la console");

Exemple de flux standard

(exemple de lecture au clavier de type primitif)

```
public class LectureClavier {
 public static void main (String [ ] args) {
       InputStreamReader isr = new InputStreamReader (System. in );
BufferedReader dis = new BufferedReader (isr); // pour des méthodes plus évoluées
System.out .print ("saisir UN ENTIER");
String ligne_lue ="";
 try{ ligne_lue = dis.readLine(); // récupérer ce qui est entré au clavier
     try { int k = Integer.parseInt (ligne_lue); // et le convertir en entier
          System. out .println("valeur entière lue ="+k);
    catch (NumberFormatException g ){// pour les exceptions de parseInt (...)
                       System. err print ("Format de nombre incorrect");
catch (IOException e) {// pour les exceptions de readLine ( )
                        System. err .print ("Erreur lecture");}
}}
                                       Mr NDONG.
                                                                                   477
```



Collections et Algorithmes

Les Collections sont des objets qui permettent de manipuler des ensembles d'objets, ou encore de manipuler les *structures de données* telles que, les vecteurs dynamiques, les ensembles, les listes chaînées, les tables associatives.

Cet ensemble de bibliothèques du paquetage java.util a été introduit à la version 2 de Java pour amener un lot de performances sur notamment la simplicité, la concision, l'universalité, l'homogénéité et la flexibilité.

Ainsi les classes recouvrant les *vecteurs*, les *listes* et les *ensembles* implémentent une même interface: l'interface Collection.

Dans ce chapitre, nous commencerons par explorer l'univers des Collections et nous aborderons l'étude des algorithmes qui nous permettront de réaliser un certain nombre d'opérations sur les Collections, à savoir: la recherche de minimum, de maximum, le tri, la recherche binaire,...

Nous terminerons sur les tables associatives qui implémentent l'interface Map.

Présentation du framework collection (1/2)



Collection: interface implémentée par la plupart des objets qui gèrent des collections

Map: interface qui définit des méthodes pour des objets qui gèrent des tables

associatives sous forme clé/valeur.

Set: interface pour des objets qui n'autorisent pas de gestion des doublons dans

l'ensemble

List: interface pour des objets qui autorisent la gestion des doublons et un accès

direct à un élément.

SortedSet: interface qui étend l'interface Set et permet d'ordonner l'ensemble.

SortedMap: interface qui étend l'interface Map et permet d'ordonner l'ensemble.

NB: les interfaces List et Set étendent l'interface Collection, et SortedSet étend Set.

Le framework propose plusieurs objets qui implémentent ces interfaces et qui peuvent être directement utilisés:

Présentation du framework collection (2/2)



HashSet: Hashtable qui implémente l'interface Set

TreeSet: arbre qui implémente l'interface SortedSet

ArrayList: tableau dynamique qui implémente l'interface List

LinkedList: liste doublement chaînée qui implémente l'interface List

HashMap: Hashtable qui implémente l'interface Map

TreeMap: arbre qui implémente l'interface SortedMap.

Parallèlement, le framework définit des interfaces pour faciliter le parcours des collections et leur tri:

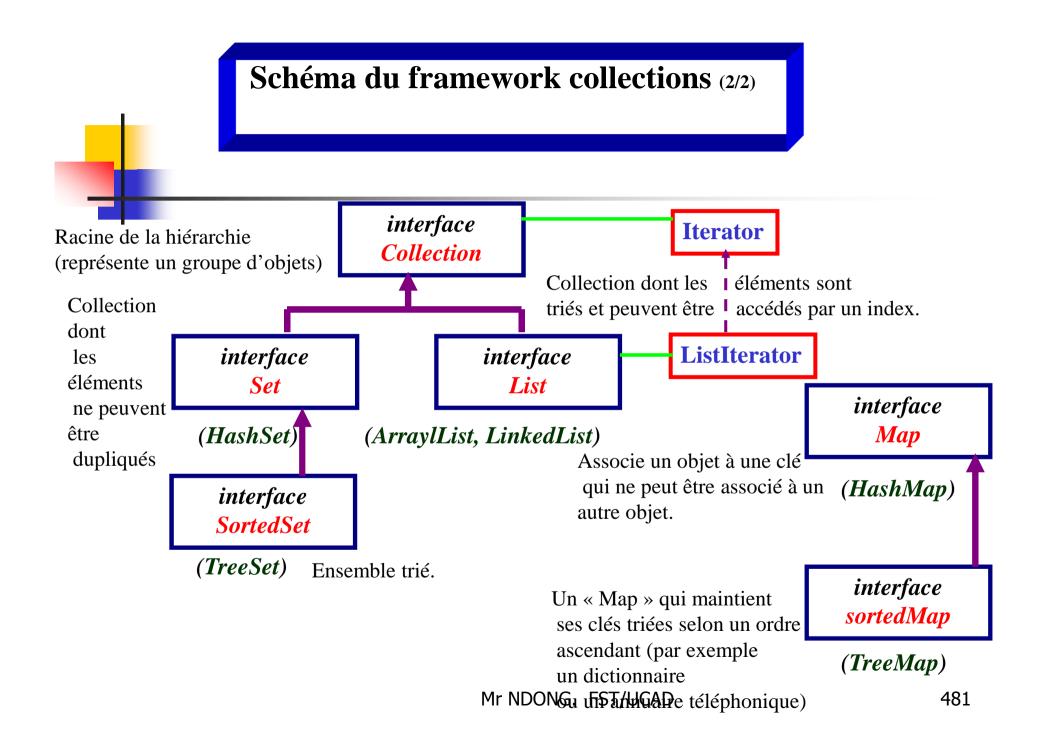
Iterator: interface pour le parcours des collections,

ListIterator: interface pour le parcours des listes dans les deux sens et modifier les

éléments lors de ce parcours

Comparable: interface pour définir un ordre de tri naturel pour un objet

Comparator: interface pour définir un ordre de tri quelconque





Concepts de collections

Qu'est ce qui est stocké dans les collections ?

Une collection peut stocker des éléments de type quelconque, mais ces éléments sont *obligatoirement* des objets.

Par exemple, un vecteur dynamique peut stocker des chaînes (String), des objets de type Float, Integer, des objets de type Compte, de type Point...

Mais il est très délicat de manipuler de telles collections, vu qu'il faudra très souvent recourir au casting (opérateur de cast) et à instanceof (pour trouver la classe d'un objet).

Donc il est préférable d'essayer de stocker dans une collection des objets de même type pour faciliter leur manipulation.

Dans l'introduction d'un élément dans une collection, on ne réalise pas de recopie d'objet, on se contente en fait d'introduire la référence à l'objet.

Ainsi, il est possible d'introduire la référence null dans une collection

L'interface java.util.Collection



Cette interface définit des méthodes pour des objets qui gèrent des éléments d'une façon assez générale.

/*ajoute l'élément fourni en paramètre à la collection. La valeur de retour indique si la collection a été mise à jour*/

boolean add (Object)

/*ajoute à la collection tous les éléments de la collection fournie en paramètre */
boolean addAll (Collection)

/*supprime tous les éléments de la collection*/

void clean ()

/*indique si la collection contient au moins un élément identique à celui fourni en paramètre (au sens de equals)*/

boolean contains (Object)

/*indique si tous les éléments de la collection fournie en paramètre sont contenus dans la collection*/

boolean contains All (Collection) Mr NDONG. FST/UCAD

L'interface java.util.Collection

Mr NDONG. FST/UCAD







Cette interface fournit des méthodes pour des objets capables de parcourir les données d'une collection:

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

Remarque: la méthode next () lève une exception de type java.util.NoSuchElementException, si elle est appelée alors que la fin du parcours des éléments est atteinte. Pour éviter de lever cette exception, il suffit d'appeler la méthode hasNext () avec le résultat de l'appel à la méthode next ().

Exemple de parcours d'un itérateur



```
/* construit un itérateur monodirectionnel sur la collection*/
       Iterator iter = objetcollection.iterator();
  /*vérifie si l'itérateur est ou non en fin de collection*/
  while (iter. hasNext ( ))
       /*renvoie l' Object désigné par l'itérateur */
        Object o = iter.next();
        System.out.println (" objet"+ o);
NB: la méthode remove () supprime l'élément renvoyé par le dernier appel de
next(). Ainsi, il est impossible d'appeler remove() sans un appel correspondant
de la méthode next(). On emploie remove () comme suit:
        iter.next();
                     // Si aucun appel à next() ne correspond à celui de remove(),
        iter.remove(); // une exception de type IllegalStateException est levée.
```

Mr NDONG. FST/UCAD

486





Cette interface définit un itérateur bidirectionnel en disposant de méthodes qui permettent de parcourir certaines collections (vecteurs dynamiques, listes chaînées) dans les deux sens et d'effectuer des mises à jour qui agissent par rapport à l'élément courant dans le parcours.

Cette interface dispose d'une méthode nommée **listIterator** qui fournit un objet qui est un itérateur bidirectionnel.

Il s'agit ici d'objet d'un type implémentant l'interface **ListIterator** (dérivée de **Iterator**) Cet objet dispose donc des méthodes next (), hasNext() et remove () mais aussi il dispose d'autres méthodes permettant d'exploiter le caractère bidirectionnel:

487

```
// ajoute un élément dans la liste à la position courante
void add(Object)
boolean hasPrevious() // indique si il reste au moins un élément à parcourir dans la
                      // liste dans son sens inverse
Object previous() // renvoie l'élément précédent dans la liste
void set (Object) // remplace l'élément courant par celui en paramètre
                                     Mr NDONG. FST/UCAD
```





On suppose qu'on dispose d'un objet l de type LinkedList

```
/* construit un itérateur bidirectionnel en fin de liste sur l'objet l*/
    ListIterator iter = l.listIterator (l.size());
/*vérifie si l'itérateur est ou non en fin de collection dans le sens inverse*/
while (iter. hasPrevious ())
{    /*renvoie l'Object précédent désigné par l'itérateur */
    Object o = iter.previous ();
    System.out.println (" objet"+ o);
}
```

NB: L'interface Iterator ne dispose pas de méthode d'ajout d'un élément à une position donnée.





Une liste est une collection ordonnée d'éléments qui autorise d'avoir des doublons. Étant ordonné, un élément d'une liste peut être accédé à partir de son index.

L'interface List étend l'interface Collection.

L'interface est renforcée par des méthodes permettant <u>d'ajouter</u> ou de <u>retirer</u> des éléments se trouvant à une position donnée

Les collections qui implémentent cette interface autorisent les doublons dans les éléments de la liste. Elles autorisent aussi l'insertion d'éléments null.

L'interface List propose plusieurs méthodes pour un accès aux éléments à partir d'un index.

La gestion de cet index commence par zéro.

Pour les listes, une interface particulière est définie pour assurer le parcours de la liste dans les deux sens et effectuer des mises à jour: l'interface ListIterator.

L'interface java.util.List

ListIterator listIterator() // renvoie un objet pour parcourir la liste

Object set(int i, Object o) //remplace l'élément à l'indice i par l'objet o

void add(int i, Object o) // ajoute à la liste l'objet o à la position i

Object get (int) // renvoie l'objet à la position spécifiée

int indexOf (Object o) //renvoie la position du premier o trouvé et -1 si l'élément

// n'est pas dans la liste

List subList (int, int) /*renvoie un extrait de la liste contenant les éléments

entre les deux index fournis (le premier inclus, le dernier

exclu); les éléments contenus dans la liste de retour sont

des références sur la liste originale; des mises à jour de

cette liste impactent la liste originale*/

int lastIndexOf (Object) //renvoie l'index du dernier élément fourni en paramètre

// ou -1 si l'élément n'est pas dans la liste

Le framework propose des classes qui implémentent l'interface List:

LinkedList et ArrayList.



Cette classe hérite de AbstractSequentialList et implémente donc l'interface List. Elle représente une liste doublement chaînée (introduction d'une référence à l'élément suivant et une référence à l'élément précèdent). On peut réaliser une pile à partir d'une LinkedList.

Les constructeurs

LinkedList () // pour créer une liste vide
LinkedList (Collection c) // créer une liste initialisée à partir des éléments
// de la collection c

Premier Exemple de liste chaînée

```
package ndong.collection;
import java.util .*; // package de base des collections
public class ExempleLinkedList01 {
 public static void main (String [ ] args) {
 LinkedList <String> list = new LinkedList <String> ( ) // cree une liste vide
 /*ajout d elements dans la liste*/
 list.add ("objet 1");
 list.add ("objet 2");
 list.add ("objet 3");
 /* iterateur bidirectionnel sur la liste*/
 ListIterator < String> iter = list.listIterator();
 /*parcours des elements de la liste*/
 while (iter.hasNext ( ) )
   { String o = iter.next();
    System.out .println ("element "+o);
```

Commentaires sur l'exemple de liste chaînée

Une liste chaînée gère une collection de façon ordonnée: l'ajout d'un élément peut se faire en fin de liste ou après n'importe quel élément. Dans ce cas l'ajout est lié à la position courante lors du parcours.

Les iterator peuvent être utilisés pour faire des mises à jour de la liste: une exception de type ConcurrentModificationException est levée si un iterator parcourt la liste alors q'un autre fait des mises à jour (ajout ou suppression d'un élément de la liste).

Pour gérer facilement cette situation, il faut mieux disposer d'un seul iterator s'il y a des mises à jour prévues dans la liste.

Voici quelques méthodes spécifiques de la classe LinkedList:

```
void addFirst( Object)// ajoute l'élément en début de listevoid addLast (Object)// ajoute l'élément en fin de listeObject getFirst ()// renvoie le premier de la listeObject getLast ()// renvoie le dernier de la listeObject removeFirst ()// supprime et renvoie le premier élémentObject removeLast ()// supprime et renvoie le dernier élément
```

La méthode toString () renvoie une chaîne contenant tous les éléments de la liste.

Il n'existe pas de méthode pour accéder CONCENTATION élément de la liste.

493

Deuxième Exemple de liste chaînée

```
package ndong.collection;
import java.util .*; // package de base des collections
public class ExempleLinkedList02 {
 public static void main(String[] args) {
 LinkedList <String> list = new LinkedList <String> (); // cree une liste vide
 /*ajout d elements dans la liste*/
 list.add (" a"); list.add (" b"); list.add (" c");
 /* iterateur bidirectionnel sur la liste//on se place en fin de liste*/
 ListIterator <String> iter = list.listIterator (list.size());
                                                                              [a,d
 /*parcours des elements de la liste en sens inverse*/
 while (iter.hasPrevious ( ) )
   { String s = iter.previous ( ) ; System.out .println ( s );
    if (s.equals ("b"));
                     { iter.remove(); break;
System.out .println( list. toString( ) ); // affiche les éléments de la liste
                                         Mr NDONG. FST/UCAD
                                                                                         494
```

Les Tableaux redimensionnables: la classe java.util.ArrayList

Cette classe représente un tableau d'objets dont la taille est dynamique. Elle hérite de la classe AbstractList donc elle implémente l'interface List.

Le fonctionnement de cette classe est identique à celle de la classe Vector. La différence avec la classe Vector est que cette dernière est multi-thread (toutes les méthodes sont synchronisées).

Pour une utilisation dans un seul thread, la synchronisation des méthodes est inutile et coûteuse. Il est donc préférable d'utiliser un objet de la classe ArrayList.

Les constructeurs

ArrayList () // vecteur dynamique vide
ArrayList (Collection c) // vecteur dynamique contenant tous les éléments de c



ArrayList: les méthodes

```
boolean add (Object )
                                                                   Il ajoute un element en fin de tableau
boolean addAll (Collection ) // ajoute tous les éléments de la collection en fin de tableau
boolean addAll (int, Collection ) // ajoute tous les éléments de la collection à partir de la
                                                                                          Il position indiquée
 void clear ()
                                                                             Il supprime tous les éléments du tableau
 void ensureCapacity (int) //permet d'augmenter la capacité du tableau pour s'assurer
                                                                        Il qu'il puisse contenir le nombre d'éléments passé en paramètre
 Object get( int)
                                                                      Il renvoie l'élément du tableau dont la position est précisée
 int indexOf (Object o) // renvoie la position de la première occurrence de l'élément o
 boolean isEmpty () // indique si le tableau est vide
 int lastIndexOf (Object o) // renvoie la position de la dernière occurrence de l'élément o
Object remove (int) // supprime dans le tableau l'élément dont la position est indiquée
 void removeRange (int i ,int j) //supprime tout élément entre i (inclus) et j (exclu)
 Object set (int, Object) // remplace l'élément à la position indiquée par celui en paramètre
 int size ()
                                                     Il renvoie le nombre d'élément du tableau
 void trimToSize( )
                                                             Il ajuste la capacitión dipoto de a postuno superior la capacitión dipoto de la capacitión dipoto de la capacitión de la capacitición de la capacitión de la capacitión de la capacitión de la ca
                                                                                                                                                                                                                                              496
```

ArrayList: quelques remarques

Chaque objet ArrayList gère une capacité qui est le nombre d'éléments qu'il est possible d'insérer avant d'agrandir le tableau. Cette capacité a une relation avec le nombre d'élément de la collection.

Lors de l'ajout, cette capacité et le nombre d'élément de la collection détermine si le tableau doit être agrandi.

L'agrandissement de cette capacité s'effectue avec la méthode ensureCapacity(int).

Premier Exemple de ArrayList

```
putter class TestArrayList {
                                                   Résultats:
public static void main(String[] args) {
                                                   0123456789
 ArrayList c = new ArrayList();
                                                   [0, 1, 3, 4, 6, 7, 9]
 /*ajout de dix element de type Integer*/
                                                   [0, 1, 200, 100, 3, 300, 4, 6, 7, 9]
 for (int i = 0; i < 10; i++) c.add (new Integer(i));
 /*affiche des element par accès direct*/
 for (int i = 0; i < c.size(); i++)
 System.out .println();
/* suppression d'element aux positions indiquées*/
c. remove(2); c. remove(4); c. remove(6); // c.remove(8)
                                                              Après suppression de
                                                             2, 5 \text{ et } 8 \text{ c.size}() = 7
/*affiche de la liste */
System.out .print (c.toString() );
                                                             donc si vous faîtes
/*ajout de dix element de type Integer*/
                                                              c.remove (8); vous obtenez
c.add (2,"100") ;c.add(2,"200") ;c.add(5,"300") ;
                                                              Une exception
                                                              ArrayIndexOutOfBounds -
 l*réaffiche de la liste */
                                                             -Exception.
System.out .print(c.toString()
                                       Mr NDONG. FST/UCAD
```

Deuxième exemple de ArrayList

```
public class TestArrayList02 {
  public static void main(String[] args) {
          ArrayList c = new ArrayList();
      /*ajout de dix element de type Integer*
                                               [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
        for (int i = 0; i < 10; i++)
                                               [500, 1, 500, 3, 500, 5, 500, 7, 500, 9]
          c.add (new Integer ( i )) ;
        /*affiche toute la liste*/
        System.out .println (c.toString ( )
       /*iterateur sur le tableau*/
       ListIterator iter = c.listIterator();
       while (iter.hasNext ( ) )
       { Integer k = (Integer) iter.next() ;
          int i = k.intValue();
          if (i\%2 = 0) iter.set (new Integer(500));
       /*affiche toute la liste*/
          System.out .print (c.toString (,) N
                                                                                          499
```



Les ensembles (Set)

Un ensemble est une collection non ordonnée qui n'autorise pas l'insertion de doublons.

L'interface java.util.Set

Cette interface définit les méthodes d'une collection qui n'accepte pas de doublons dans ses éléments. Elle hérite de l'interface Collection mais ne définie pas de nouvelles méthodes.

Pour vérifier si un élément est déjà inséré dans la collection, utiliser la méthode **equals** () . Les **Objets** ajoutés à un **Set** doivent définir la méthode **equals** () pour pouvoir établir l'unicité de l'objet.

Deux classes implémentent l'interface Set: TreeSet et HashSet.

Le choix entre ces deux méthodes est lié à la nécessité de trier les éléments:

- ► les éléments d'un objet HashSet ne sont pas triés: l'insertion d'un nouvel objet élément est rapide
- ► les éléments d'un objet TreeSet sont tries: l'insertion d'un élément est longue.

Construction d'un objet de type HashSet





HashSet () // ensemble vide

HashSet (Collection) // ensemble contenant tous les

Il éléments de la collection

HashSet utilise une technique dite de *hachage* pour déterminer l'appartenance d'un élément.

Les constructeurs de TreeSet

TreeSet () // ensemble vide

TreeSet (Collection) // ensemble contenant tous les
// éléments de la collection

TreeSet utilise un arbre binaire pour ordonner complètement les éléments.



La classe HashSet est un ensemble sans ordre de tri particulier. Les éléments sont stockés dans une table de hachage possédant une capacité.

La classe TreeSet est un arbre qui représente un ensemble trié d'éléments. Cette classe permet d'insérer des éléments dans n'importe quel ordre et de les restituer dans un ordre précis lors de son parcours.

L'implémentation de cette classe insére un nouvel élément dans l'arbre à la position correspondant à celle déterminée par l'ordre de tri. L'insertion d'un nouvel élément dans un objet *TreeSet* est donc plus lent mais le tri est directement effectué.

L'ordre utilisé est celui indiqué par les objets insérés s'ils implémentent l'interface Comparable pour un ordre de tri naturel ou fournir un objet de type Comparator au constructeur de l'objet TreeSet pour définir l'ordre de tri.

Exemple de HashSet

```
ajout de 100 réalisé
public class TestHashSet {
                                                            l'élément joie est déjà présent
  public static void main(String[] args) {
                                                            ajout de 200 réalisé
 HashSet hs = new HashSet();
                                                            l'élément paix est déjà présent
 hs.add ("paix"); hs.add("joie");
                                                            ajout de 300 réalisé
String tab [] = {"100","joie","200","paix","300","vie"}
                                                            ajout de vie réalisé
  /*ajout d'élément dans l'ensemble*/
  for (int i = 0;i < tab.length ;i++)
                                                            [vie, joie, paix]
   { boolean ajout = hs.add ( new String (tab[i]));
      if (ajout) System.out .println( "ajout de " +tab[i]+ " réalisé" );
      else System.out .println( "l'élément " +tab[i]+ " est déjà présent" );
   Iterator iter = hs.iterator(); // un itérateur pour parcourir l'ensemble
   while (iter.hasNext() )
      {String s = (String) iter.next();
        if (s.equals("100")||s.equals("200")||s.equals("300")) iter. remove();
                                        Mr NDONG. FST/UCAD
                                                                                      503
 System.out.println(hs.toString()):
```

Caractéristiques de HashSet

Dans'l'implémentation des ensembles HashSet, il faut définir convenablement :

- la méthode **equals**: c'est elle qui permet de savoir l'appartenance d'un objet à un ensemble
- la méthode hashCode: elle est exploitée pour ordonnancer les éléments d'un ensemble au travers d'une table de hachage.

une table de hachage est une organisation des éléments d'une collection qui permet de retrouver facilement un élément de valeur donnée (la valeur d'un élément est formée de la valeur de ses différents champs).

Pour cela on utilise une méthode hashCode dite « fonction de hachage »qui, à la valeur d'un élément (existant ou recherché) associe un entier (un code). Pour retrouver un élément de la collection, on détermine son code de hachage code avec la méthode hashCode. Et il ne reste plus qu'à parcourir les différents éléments du seau pour vérifier si la valeur donnée s'y trouve (tester avec equals).





Cette interface définit une collection de type ensemble trié. Elle hérite de l'interface Set.

Le tri de l'ensemble peut être assuré par deux façons:

- o les éléments contenus dans l'ensemble implémentent l'interface Comparable pour définir leur ordre naturel
- o il faut fournir au constructeur de l'ensemble un objet Comparator qui définit l'ordre de tri à utiliser.

Cette interface définit plusieurs méthodes pour tirer parti de cet ordre.

java.util.SortedSet: les méthodes



```
/*renvoie l'objet qui permet de trier l'ensemble*/
```

Comparator comparator ()

/*renvoie le premier élément de l'ensemble*/

Object first ()

/*renvoie un sous-ensemble contenant tous les éléments inférieurs à celui fourni en paramètre */

SortedSet headSet (Object)

/*renvoie le dernier élément de l'ensemble*/

Object last ()

/*renvoie un sous ensemble contenant les éléments compris entre le premier élément inclus et le second élément exclus*/

SortedSet subSet (Object, Object)

/*renvoie un sous ensemble contenant tous les éléments supérieurs ou égaux à celui fourni en paramètre*/

SortedSet tailSet (Object) Mr NDONG. FST/UCAD



L'interface java.util.Map

Ce type de collection désignée sous le titre de table associative gère les informations sous forme de *paires* clé/valeur. Cette interface n'autorise pas de doublons.

L'intérêt des tables associatives est de pouvoir retrouver rapidement une <u>clé</u> donnée pour en obtenir l'information associée qui est <u>sa valeur</u>.

Deux types d'organisations sont rencontrées avec les ensembles:

- table de hachage: classe **HashMap**, **Hashtable**.
- arbre binaire: classe **TreeMap**.

Seule la clé est utilisée pour ordonnancer les informations :

pour *HashMap* on se servira du code de hachage des objets formant les clés; pour *TreeMap*, on se servira de la relation d'ordre induite par **compareTo** ou par un comparateur fixé à la construction.

interface java.util.Map: les méthodes

```
void clear( )
                                     Il supprime tous les éléments de la collection
boolean containsKey (Object o) // indique si la clé o est contenue dans la collection
boolean containsValue (Object o) //indique si la valeur o est dans la collection
Set entrySet ( ) // renvoie un ensemble contenant toutes les valeurs de la collection
Object get (Object o) // renvoie la valeur associée à la clé o
boolean isEmpty()
                               Il indique si la collection est vide
Set keySet()
                               Il renvoie un ensemble contenant les clés de la collection
Object put (Object c, Object v)
                                        Il insère la clé c et sa valeur v dans la collection
void putAll ( Map m)
                                       Il insère toutes les clés/valeurs dans la collection
Collection values () // renvoie tous les éléments de l'ensemble dans une collection
Object remove (Object o) // supprime l'élément de clé o
int size ()
                         Il renvoie le nombre d'élément de la collection
```





La classe HashMap n'est pas synchronisée; pour assurer la gestion des accès concurrents sur cet objet, il faut l'envelopper dans un objet Map en utilisant la méthode synchronizedMap de l'interface Collection.

La classe TreeMap implémente l'interface SortedMap. L'ordre des éléments de la collection est maintenu grâce à un objet de type Comparable.

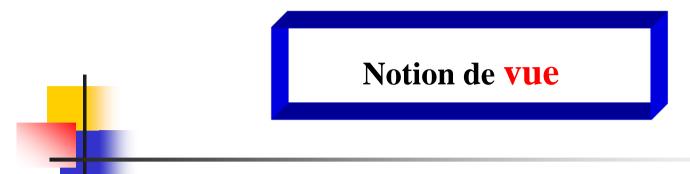
Exemple de HashMap

```
package ndong.collection;
import java.util.*;
public class TestHashMap {
  public static void main(String [ ] args) {
     HashMap\ table = new\ HashMap\ ();
    /*ajout d'éléments dans la table*/
    table.put (new Integer (1), "Livre Java"); table.put( new Integer (2), "Livre Oracle");
    table.put( new Integer (3), "Livre C++"); table.put( new Integer (4), "Livre Reseaux");
    l*recherche d'informations*/
     Object o = table.get ( new Integer(3)); // cherche la valeur associée à la cle 3
  /*suppression d'information*/
     Object cle = new Integer(4);
      table.remove (cle);
       System.out .println( "suppression de la valeur" ) ;
     else
    System.out .println("la cle" +val+ "n'existe pas"); }}
Mr NDONG. FST/UCAD
                                                                                           510
```

Remarques

Si la clé fournie à la méthode **put** existe déjà, la valeur remplacera l'ancienne (une clé donnée ne pouvant figurer qu'une seule fois dans une table).

put fournit en retour soit l'ancienne valeur si la clé existait déjà, soit la valeur **null**.



Les vues permettent de parcourir une table.

En théorie les classes HashMap et TreeMap ne disposent pas d'itérateurs. La méthode *entrySet* permet de « voir » une table comme un ensemble de paires. Une paire est un objet de type interface Map.entry réunissant deux objets.

Les méthodes getKey et getValue de Map.entry permettent d'extraire respectivement la clé et la valeur d'une paire.

Exemple avec vue

```
impor liava util *•
public class TestHashMap_Vue {
  public static void main (String [ ] args) {
    HashMap <Integer, String> table = new HashMap <Integer, String> ();
     /*ajout d'éléments dans la table*/
     table.put( new Integer(1), "Livre Java") ;table.put( new Integer(2), "Livre Oracle") ;
     table.put( new Integer(3), "Livre C++") ;table.put( new Integer(4), "Livre Reseaux") ;
     Set <Map.Entry<Integer, String>> entrees = table.entrySet(); //ensemble de paires
 Iterator < Map.Entry < Integer, String >> iter = entrees.iterator(); // iterateur sur les paires
     while (iter.hasNext())
     {Map.Entry<Integer, String> entree = iter.next(); // on est sur la paire courante
          Integer cle = entree.getKey();
          String valeur = entree.getValue();
if (cle!= null && cle. toString(). equals ( new Integer(2).toString() ))
           { table.put (new Integer(2), "Revue Technical Report");
     System.out .print(table.get( new Integer(2)) ); }}
                                                                                      513
```



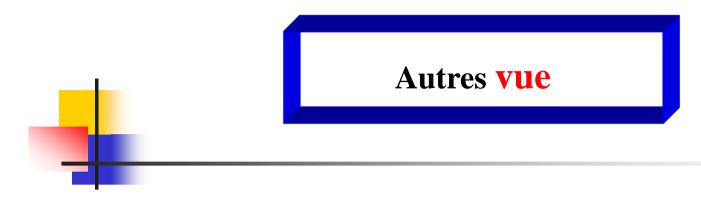


L'ensemble fourni pat *entrySet* n'est pas une copie des informations figurant dans la table. Il s'agit de ce que l'on nomme une **vue**.

Toute modification opérée sur la table se répercute sur la vue associée.

La suppression (par remove) de l'élément courant (paire) de la vue supprime du même coup l'élément correspondant de la table.

Il n'est pas permis d'ajouter directement des éléments dans la vue elle-même.



En plus de la vue précédente, on dispose de deux autres vues pour obtenir:

• l'ensemble des clés à l'aide de la méthode **keySet**: là on peut utiliser un itérateur pour parcourir l'ensemble,

Set cles = objetHashMap .keySet();

• la collection des valeurs à l'aide de la méthode values:

Collection valeurs = objetHashMap .values ();

Là encore, on peut parcourir la collection à l'aide d'un itérateur.

Là, on obtient une collection et non un ensemble car certaines valeurs peuvent bien évidemment apparaître plusieurs fois.

Les remarques précédentes s'appliquent aussi ici.

Second exemple vue

```
pellege tp_annee0405;
import java.util .*;
public class ExempleVue {
 public static void main (String [ ] args) {
 HashMap\ map = new\ HashMap\ (\ );
 map.put ("1","stock"); map.put ("2","livres");
 map.put ("3","sucre"); map.put ("4","huile");
 //collection des valeurs
 Collection valeurs = map.values ();
                                                Affichage 1 [huile, sucre, livres, stock]
  System.out .println ("Affichage 1"+valeurs)
                                                Element courant huile
  Iterator iter = valeurs.iterator():
                                                Element courant sucre
 while (iter.hasNext())
                                                Element courant livres
 { Object o = iter.next();
                                                Element courant stock
 if (o.toString() .equals ("huile"))
                                                Affichage 2 [sucre, livres, stock]
     { iter.remove();
   System.out .println ("Element courant" +0);
 } System.out .println ( "Affichage 2 'M+WARENES) FST/UCAD
                                                                                 516
```

Troisième exemple vue

```
package tp_annee0405;
import java.util.*;
public class ExempleVue2 {
 public static void main(String[] args) {
 HashMap map = new HashMap();
 map.put("1","stock"); map.put("2","livres");
 map.put("3","sucre"); map.put("4","huile");
 //ensemble des cles
                                               Aff1 [4=huile, 3=sucre, 2=livres, 1=stock]
 Set cles = map. entrySet();
                                               Element courant huile
                                               Element courant sucre
  System.out .println("Aff1 "+cles);
  Iterator iter = cles.iterator() :
                                               Element courant livres
                                               Element courant stock
 while (iter.hasNext ( ) )
                                               Affiche [4=riz, 3=sucre, 2=livres, 1=stock]
 {Map.Entry entree = (Map.Entry )iter.next()
 String s = (String) entree. getValue();
 if (s. equals ("huile"))
  entree.setValue ("riz");
  System.out .println( "Element courant" NDQNG. \System.out .print( "Affiche " +cles \frac{1}{2}\frac{7}{2}\}
```



Le Tri des Collections

L'ordre de tri des collections est défini grâce à deux interfaces:

- o **Comparable**
- o **Comparator.**





Tous les objets qui doivent définir un ordre naturel utilisé par le tri d'une collection doivent implémenter cette interface.

Certaines classes comme *String*, *File*, *Date* ou *Integer*, *Float*,... implémentent cette interface.

L'interface *Comparable* ne dispose que d'une seule méthode, **compareTo** qui fournit en résultat un ordre qu'on peut qualifier de naturel:

- -Ordre lexicographique pour les chaînes, noms de fichier ou la classe *Character*,
- -Ordre numérique pour les classes enveloppes numériques.

Cette méthode renvoie:

- + une valeur entière positive si l'objet courant est supérieur à l'objet fourni,
- + une valeur entière négative si l'objet courant est inférieur à l'objet fourni,
- + une valeur nulle si l'objet courant est égal à l'objet fourni.



Exercice avec Comparable

Réaliser une classe **TestComparable** qui implémente l'interface **Comparable** et permettant de trier les valeurs d'un HashMap.

On prévoira:

- la méthode void tri (Object []o) permettant de trier un tableau d'Object,
- la méthode **ArrayList triHashMap (HashMap map):** cette méthode reçoit en paramètre un HashMap et renvoie sous forme d'un ArrayList trié les valeurs du HashMap,

On redéfinira convenablement la méthode **CompareTo** de l'interface **Comparable**.

Pour implémenter la méthode tri, utilisez CompareTo.

Pour implémenter la méthode triHashMap, utilisez la méthode tri.

Solution (1/2)

```
package allndong.collection;
import java.util .*;
public class TestComparable
            implements Comparable{
public int compareTo (Object o)
if (o instanceof String)
         String d = (String)o;
         return this.compareTo (d);
        new ClassCastException ( );
else
        return -1;
```

```
ArrayList triHashMap (HashMap map)
   \{ int i = 0; 
        ArrayList liste = new ArrayList();
        Set c = map.entrySet();
        Object tab[] = new Object[c.size()];
        Iterator it = c.iterator();
        Map.Entry mp = null;
        while (it.hasNext ( ) )
         { mp = (Map.Entry) it.next(); }
         tab[ i ] = mp.getValue ( );
         i++;
         tri (tab);
         for ( int ii = 0;ii < tab.length; ii++)
         liste.add (tab[ii]);
         return liste;
ND CNG. FST/UCAD
                                           521
```

Solution (2/2)

```
public void tri (Object [ ] tab)
for ( int i = 0;i< tab.length-1; i++)
if (tab[i].toString( ) .compareTo(tab[i+1])>0)
          String temp = tab[i].toString();
          tab[i] = tab[i+1];
          tab[i+1] = temp;
          tri(tab);
```

```
-public static void main (String[] args) {
 TestComparable tt= new TestComparable();
HashMap map= new HashMap();
 map.put ("1","stock");
 map.put ("2", "livres");
 map.put ("11","sucre");
 map.put ("4","huile");
 map.put ("5","riz");
 map.put ("6","or");
 Collection val = map.values();
 System.out .println ("aff 1 "+ val);
 ArrayList h = tt.triHashMap (map);
 System.out .print("aff 2 "+ h.toString());
```

aff 1 [or, riz, huile, sucre, livres, stock] aff 2 [huile, livres, or, riz, stock, sucre]



L'interface java.util.Comparator

Cette interface représente un ordre de tri quelconque.

Elle permet de trier des objets qui n'implémente pas l'interface Comparable, ou de définir un ordre de tri différent de celui réalisé avec Comparable (cette interface représente un ordre naturel: il ne peut y en avoir qu'un).

Cette interface ne contient qu'une seule méthode: int compare(Object o, Object v)

Cette méthode compare les deux objets fournis en paramètre et renvoie:

- + une valeur entière positive si o est supérieur à v,
- + une valeur entière négative si o est inférieur à v,
- + une valeur nulle si o est égal à v.





Reprendre l'exercice précèdent en implémentant cette fois ci l'interface Comparator, au lieu de l'interface Comparable.



Les Algorithmes

La classe **java.util.**Collections propose plusieurs méthodes statiques pour Effectuer des opérations sur les collections: recherche de minimum, maximum, ...

Ces méthodes font un traitement polymorphique (elles appliquent le polymorphisme) car elles demandent en paramètre un objet qui implémente une <u>interface</u> et retourne une <u>collection</u>.

```
void copy (List, List) // copie les éléments de la seconde liste dans la première
Enumeration enumeration (Collection) // renvoie un objet Enumeration pour parcourir
// la collection
```

Object max (Collection) // renvoie le plus grand élément de la collection selon l'ordre // naturel des éléments

Object max (Collection, Comparator) //renvoie le plus grand élément de la collection // selon l'ordre naturel précisé par l'objet

Mr NDONG. AFSTOCAD

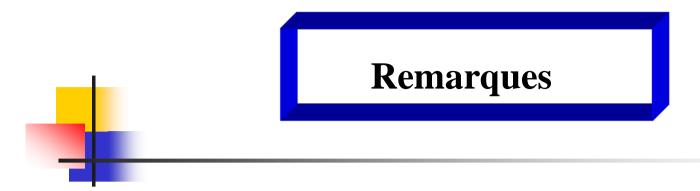


Les Algorithmes

```
Object min(Collection) //renvoie le plus petit élément de la collection selon l'ordre // naturel des éléments

Object min(Collection, Comparator) // renvoie le plus petit élément de la collection // selon précisé par l'objet Comparator void reverse (List) // inverse l'ordre de la liste fournie en paramètre void shuffle (List) // réordonne tous les éléments de la liste de façon aléatoire void sort (List) // trie la liste dans un ordre ascendant selon l'ordre naturel des // éléments

void sort(List,Comparator) // trie la liste dans un ordre ascendant selon l'ordre // précisé par l'objet Comparator
```



Pour utiliser la méthode **sort(List)**, il faut obligatoirement que les éléments inclus dans la liste implémentent tous l'interface Comparable sinon une exception de type **ClassCastException** est levée.

La classe Collections dispose de méthodes pour obtenir une version multi-trhead ou non modifiable des principales interfaces des collections: Collection, List, Map, Set, SortedMap, SortedSet.

- synchronizedXXX (XXX) pour obtenir une version multi-thread des objets implémentant l'interface XXX
- unmodifiableXXX (XXX) pour avoir une version non modifiable des objets implémentant l'interface XXX.

Exemple 1



```
import java.util .*;
public class TestUnmodifiable {
 public static void main(String args[])
 { java.util.List list = new java.util.ArrayList ();
   list.add( new Integer (100));
   list.add( new Integer (200));
   list.add( new Integer (300));
 java.util.List listun;
  listun = java.util.Collections .unmodifiableList (list);
 list.add ( new Integer (400));
 listun.add (new Integer (400)); // tentative illégale de modification de la liste l
 System.out .println ( list.toString ( ) );
```





La tentative de modification de la liste (non modifiable) lève une exception de type

 $java.lang. Unsupported Operation Exception \ .$





L'utilisation d'une méthode **synchronizedXXX** renvoie une instance de l'objet qui supporte la synchronisation pour les opérations <u>d'ajout</u> et de <u>suppression</u> d'élément.

Pour le parcours de la collection avec un objet itérateur, il est nécessaire de synchroniser le bloc de code utilisé pour le parcours.

Il est important d'inclure aussi dans ce bloc l'appel à la méthode pour obtenir l'objet de type Iterator utilisé pour le parcours.

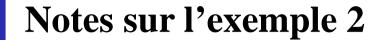




```
import java.util.*;
public class Testsynchronized {
  public static void main(String[] args) {
    List list = new LinkedList();
    list.add( new Integer (800)) ;
    list.add( new String ("livres")) ;
    list.add("revue");
    list.add("articles") ;
    list.add( new Integer(122)) ;
    list.add( new Object()) ;
```

Il faut définir un autre itérateur à cause des deux blocs synchronized.

```
list = Collections.synchronizedList(list);
       synchronized (list) {
         Iterator iter = list.iterator();
         while (iter.hasNext ( ) )
         {if (iter.next () instanceof Integer)
           iter.remove();
           list.add( null ) ; // illegal
         synchronized (list){
        Iterator it = list.iterator();
        while (it.hasNext ( ) )
         { if ( it.next ( ) instanceof String)
            it.remove();
         System.out .println(list.toString());
Mr NDONG. FST/UCAD
                                                531
```





Toute tentative d'ajout d'un élément au moment du parcours de l'itérateur provoque une exception de type:

java.util.ConcurrentModificationException.



Programmation WEB: Applets

Une Applet est une classe Java compilée (byte code) qui s'exécute dans un logiciel de navigation supportant java.

Cette classe doit dérivée soit de java.applet.Applet, soit de javax.swing.JApplet qui lui donne une certaine interface:

- init (), start (), paint (), stop (), destroy (),
- l'applet spécifie son propre comportement en redéfinissant ces méthodes
- Elle réside en général sur un poste serveur distant.
- -Elle est véhiculée dans une page HTML qui contient son URL.

Les applets permettent d'ajouter du dynamisme ou des possibilités à une page HTML

- > il faut une JVM sur le client
- > il faut qu'elle ait la bonne version
- il faut que le browser sache la lancer (grâce à un plugin)

Généralités

Lorsqu' un navigateur compatible Java (possédant une JVM) reçoit cette page HTML, il télécharge (par HTTP) le code de la classe et l'exécute sur le poste client (ie invoque ses méthodes init (), start (),)

C'est alors une véritable application, qui s'exécute dans la page HTML à l'intérieur du navigateur qui peut:

- construire et gérer une interface graphique,
- créer de nouveaux threads,
- ouvrir des connexions réseaux.

En local, l'interpréteur Java instancie un objet de cette classe et lance la méthode init () sur cet objet.



Invoquer une applet

(Intégration d'applets dans une page HTML)

Dans une page HTML, il faut utiliser le tag APPLET avec la syntaxe suivante:

```
< APPLET

CODE = "AppletTest.class"

WIDTH = 200

HEIGHT = 300 >

</ APPLET >
```

Le nom de l'applet est indiqué entre guillemets à la suite paramètre CODE. Les paramètres WIDTH et HEIGHT fixent la taille de la fenêtre de l'applet HTML. L'unité est le pixel.

Ce texte doit être intégré dans un fichier d'extension (.html). Il constitue le *minimum requis* par le browser.

Fichier HTML de lancement d'une applet

AppletTest.html

Le répertoire courant est:

- le répertoire local si l'applet est exécutée depuis un visualiseur d'applets;
- le répertoire correspondant à l'adresse URL à partir de laquelle a été chargé le fichier HTML, dans le cas d'un navigateur.

 Mr NDONG. FST/UCAD

 536

Extensions du fichier HTML

Nous venons de décrire le squelette minimal de la balise </applet >contenant les éléments obligatoire pour le lancement d'une applet.

Mais il existe des paramètres optionnels mais importants utilisables dans cette balise.

Les méthodes de la classe Applet: le cycle de vie (1/2)

(le cycle de vie d'une applet dépend de l'utilisation de ses méthodes)

- 1. La page est chargée
- (a) la balise <applet > est rencontrée,
- (b) le fichier archive est rapatrié,
- (c) le fichier code est rapatrié,
- (d) la classe est chargée,
- (e) l'applet est construite,
- (f) la méthode init () est lancée,
- (g) la méthode start () est lancée
- 2. La page est cachée

La méthode stop () est exécutée

3. La page est de nouveau visible

La méthode start () est réappelée

4. La page est détruite

La méthode destroy () est appelée

Les méthodes de la classe Applet: le cycle de vie (2/2)

Dans le cycle de vie d'une applet, quatre fonctions sont à l'œuvre de la naissance à la mort de celle-ci :

[le constructeur: pour l'instanciation de l'applet.]

public void init (): appelée une seule fois après construction/chargement de

la page, premier lancement de l'applet

public void start (): lancement de l'applet ou réapparition

de la page. Appelée après init () ou stop ()

public void stop (): disparition de la page, elle interrompt tous les processus

en cours

public void destroy (): fermeture de la page, perte de la page

En général, il n'est pas nécessaire de faire un appel <u>explicit</u> aux méthodes ci-dessus, le navigateur se charge d'appeler ces méthodes en fonction de l'état de la page HTML contenant l'applet.



Remarques

init () est la méthode principale, elle doit être absolument définie.

Le constructeur et init () semblent faire la même chose, mais ce n'est pas le cas. On peut définir les deux et appeler init () dans le constructeur.

init () ne devait être appelée qu'une seule fois.

init () sert à initialiser les données nécessaires à l'applet (composants graphiques notamment)

start () sert à réinitialiser les données de l'applet qui dépendent du moment où l'applet est affichée

stop () sert à arrêter l'applet

destroy () permet de nettoyer, de libérer les ressources.

Écrire une première applet: code de l'applet

(exemple d'applet très simple qui affiche l'heure et la date courantes)

```
import java.awt. *; import java.text . *; import java.util . *; import java.applet . *;
public class Applet1 extends Applet {
 /**Initialiser l'applet*/
 public void init(){
             ibInit(); }
  try {
  catch (Exception e) { e.printStackTrace ( ) ; }
 /**Initialiser le composant*/
 private void jbInit( ) throws Exception {
 this.setBackground (new Color (150,240,170));
 Calendar cal= Calendar.getInstance();
 int heure = cal.get (cal.HOUR); int minute = cal.get (cal.MINUTE);
 int seconde = cal.get (cal.SECOND );
 Label pan = new Label ("Date du jour:");
 TextField txt = new TextField (DateFormat.getDateInstance().format(new Date()));
 TextField tx = new TextField (heure+":"+minute+":"+seconde);
 this.add (pan); this.add (txt); this.add (txt); ; this.add (txt); this.add (txt);
                                                                                   541
 }}
```

Écrire une première applet: le fichier HTML

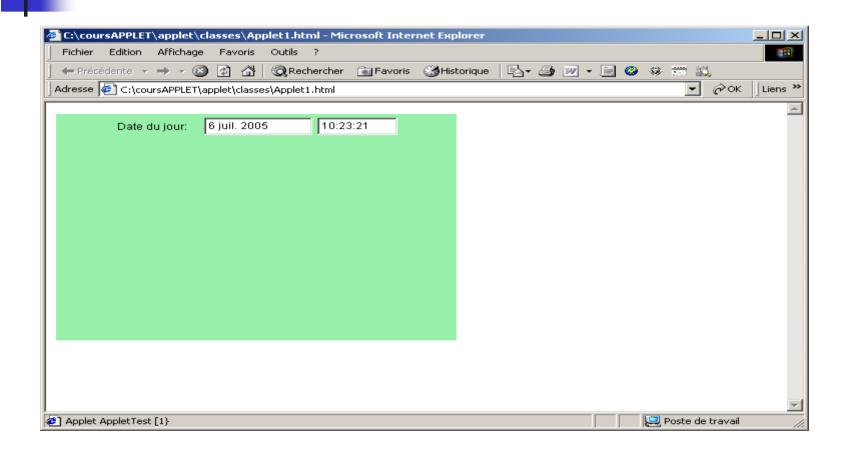


(le fichier Applet1.html associé à l'applet)

```
<html>
    <head>
    </head>
        <body>
               <applet
                codebase = "."
               code = "applet.Applet1.class"
                       = "AppletTest"
               name
                width = "400"
               height = "300"
               hspace = "0"
                vspace = "0"
                align = "top">
             </applet>
        </body>
</html>
```

Lancement de la page HTML

(l'heure est affichée de façon statique)



Deuxième applet: animation: code de l'applet (1/2)

(regardons maintenant un exemple d'affichage dynamique de l'heure courante)

```
import java.awt.*; import java.applet.*;
import java.text.*; import java.util.*;
public class Applet2 extends Applet implements Runnable {
TextField txt,tx,cpt; Label dat, sec;
int compteur;
Thread horloge; // pour affichage dynamique de la date, de l'heure et du compteur
DateFormat df,dh;
public Applet2 ( ) {
Font f = new Font ("helvetica", Font.BOLD + Font.PLAIN ,16);
dat = new Label ("Date & Heure"); dat.setFont (f);
sec = new Label ("Temps Connecté (secondes)"); sec.setFont (f) ;
txt = new TextField (10); tx = new TextField (10);
cpt = new TextField (8);
this.add (dat); this.add (txt); this.add(tx); this.add(sec); this.add(cpt);
df = DateFormat.getDateInstance();
dh = DateFormat.getTimeInstance();
compteur = 0;
                                    Mr NDONG. FST/UCAD
```

Deuxième applet: animation: code de l'applet (2/2)

```
public void init () {
  this.setBackground (new Color (130,30,160,60));
  tx.setText (dh.format(new Date ()));
  txt . setText ( df . format (new Date()));
  cpt . setText ( " " + compteur );
  }
  public void start () {
  horloge = new Thread( this );
  horloge. start ();
  }
  public void stop () {
  horloge. interrupt ();
  }
```

```
public void run () {
  while ( true ) {
    tx.setText (dh.format(new Date()) ) ;
    txt . setText ( df . format (new Date() ) ) ;
    cpt . setText ( " " + compteur++);
    try {
        Thread. sleep (1000);
      }
    catch (Exception e) { }
    }
}
// fin de la classe de l'applet
```

Deuxième applet: animation: fichier HTML

(Applet2.html)

```
<html>
      <head>
            <title> Page de test HTML </title>
      </head>
        <body>
               <applet
                     codebase = "."
                     code = "applet.Applet2.class"
                     name = " Com2"
                     width = "500"
                     height = "300"
                     hspace = "0"
                     vspace = "0"
                     align = "middle">
              </applet>
        </body>
                               Mr NDONG. FST/UCAD
</html>
```

Récupérer les Paramètres du fichier HTML dans l'applet

(communication fichier HTML – applet)

Il est possible de transmettre des informations à une applet depuis le fichier HTML.

Pour passer de la page HTML à l'applet, on utilise la balise **PARAM**> (ce tag doit être inséré entre la balise **APPLET**> et **APPLET**>)

Pour récupérer les paramètres utilisateurs définis dans la balise <PARAM> du fichier

HTMl, depuis l'applet, on utilise la méthode getParameter (String param).

Dans notre deuxième exemple, on pouvait faire de telle sorte qu'à chaque visite de la page par un utilisateur, qu'on affiche le message suivant:

"Merci pour cette visite", dans une zone de texte.

Il suffit d'ajouter dans le fichier HTML cette ligne:

```
<param name="message" value = "Merci pour cette visite">
```

et d'ajouter aussi dans la méthode init () de l'applet ces instructions:

```
String msg = this.getParameter ("message");
this.add (new TextField (msg));

Mr NDONG. FST/UCAD
```



Les méthodes graphiques

Pour réaliser des applets graphiques (ce qui très souvent le cas), Java dispose de la méthode public void paint (Graphics g).

Cette méthode est invoquée par le navigateur lors d'évènements nécessitant de (re)dessiner dans la surface occupée par l'applet.

cette méthode reçoit en argument un objet de la classe **Graphics** qui décrit l'envi - ronnement graphique courant.

Toutes les opérations de dessin peuvent être effectuées dans cette méthode. Il y a aussi les méthodes:

- public void repaint (): pour le rafraîchissement du programme graphique, en forçant l'appel de la méthode paint ().
- public void update (Graphics g):pour redessiner un bout de programme graphique en effaçant l'écran et en appelant paint().

Cette méthode entraîne des scintillements désagréables de l'écran. Il faut donc souvent la redéfinir: par exemple: public void update (Graphics g) { paint (g);}



Chargement d'images avec attente

L'insertion d'images dans les applets peut ralentir considérablement le chargement de pages web, si elle n'est pas optimisée. Souvent il faudra attendre que toutes les images soit chargées pour afficher complètement la page.

Dans d'autres cas, la page est affichée et les images sont chargées par morceaux, ce qui peut entraîner des frustrations chez le visiteur.

Regardons d'abord comment charger des images de la façon la plus simple avec possibilité d'attente.





On commence par créer un objet de type **Image**, qui servira à stocker et à manipuler l'image. <u>Attention</u>: <u>Image</u> est une *interface*.

Image image;

On charge l'image depuis son emplacement

```
image = getImage (getCodeBase ( ), "javalogo.gif") ;
```

getCodeBase () renvoie l'adresse Internet de l'applet; autrement dit elle donne l'endroit (URL) où est stockée l'applet (ie le fichier .class).

On peut aussi enregistrer l'image au même endroit que la page web (la page HTML). Dans ce cas, utilisez cette syntaxe:

```
image = getImage (getDocumentBase ( ), "javalogo.gif");
```



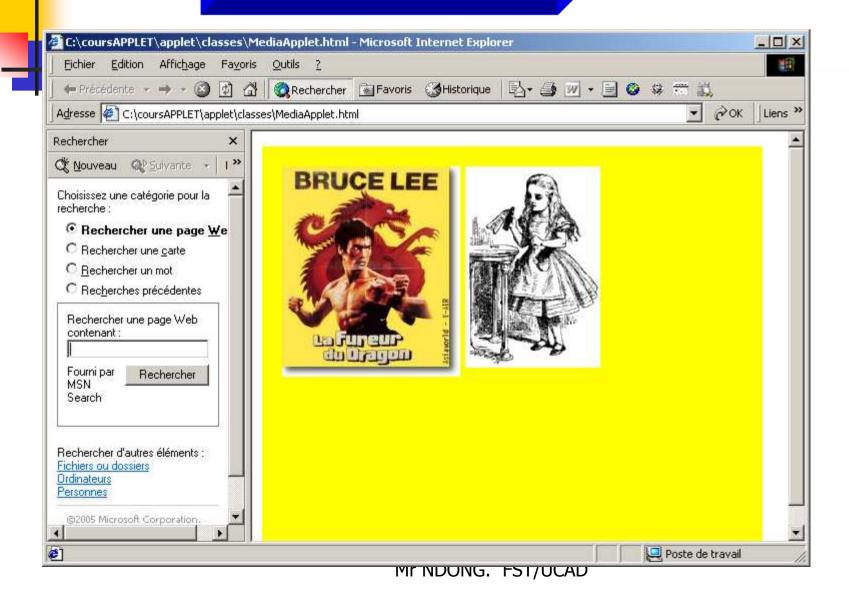
On dessine maintenant l'image dans le contexte graphique de l'applet en choisissant les coordonnées du coin supérieur gauche (x = 50, y = 50):

g.drawImage (image, 50, 50, this);

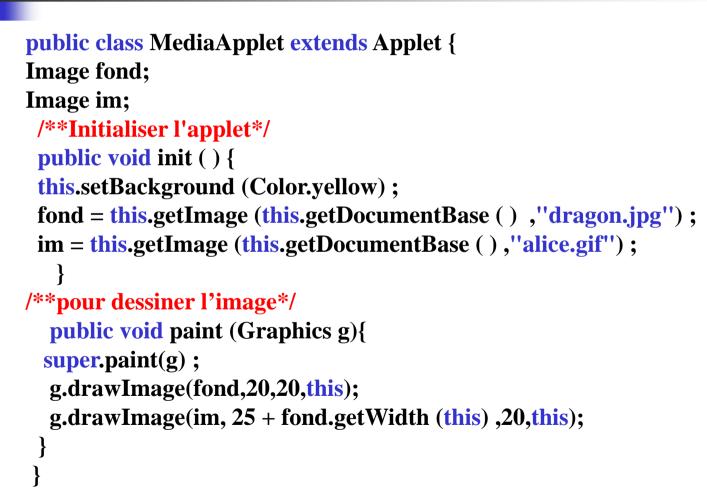
Le quatrième paramètre est un objet de type java.awt.image.ImageObserver.

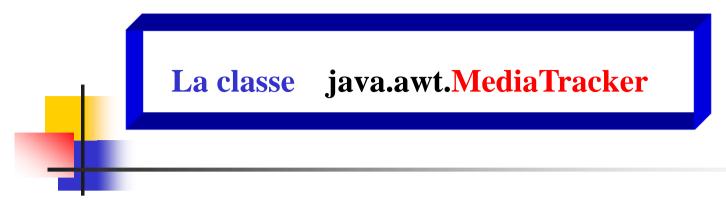
Au moment du chargement des images, vous pouvez afficher un message au niveau de la barre d'état de la page en utilisant la méthode showStatus (String message).

Exemple



Code Exemple





Dans certains cas, Java peut afficher des images alors que celles-ci *ne sont pas encore* complètement chargées via Internet. Le lecteur de la page HTML voit les images apparaître par morceaux à l'écran. Le lecteur patientera alors.

Souvent vous pouvez avoir besoin d'afficher des images *seulement* lorsqu'elles sont arrivées sur l'ordinateur client (du lecteur). Dans ce cas, vous pouvez effectuer le chargement des images en utilisant la classe MediaTracker.

En lui spécifiant les images à charger, cette classe peut vous indiquez les images qui sont arrivées, celles qui ont provoqué une erreur, l'état d'avancement du chargement, etc.

Exemple MediaTracker

```
public class MediaTrackerApplet extends Applet {
Image fond; Image im;
                                  // pour suivre le chargement des images
 MediaTracker tracker = null;
/**Initialiser l'applet*/
 public void init(){
 this.setBackground (Color.yellow);
 fond = this.getImage (this.getDocumentBase ( ) ,"dragon.jpg");
 im = this.getImage (this.getDocumentBase ( ) ,"alice.gif") ;
/*initialiser le MediaTracker*/
   tracker = new MediaTracker (this);
/*demarrer le suivi du chargement des images*/
tracker.addImage (fond, 0);
tracker.addImage (im, 0);
```

Exemple MediaTracker



```
/**pour dessiner l'image*/
public void paint (Graphics g){

/*sortir si les images ne sont pas complètement chargées*/
if (!tracker.checkAll (true))
{
    g.drawString ("Chargement en cours...",10, 20);
    repaint (100);
    return;
}
g.drawImage (fond,20,20,this);
g.drawImage (im, 25 + fond.getWidth (this),20,this);
}
}
```

Notes



Dans l'instruction:

tracker = new MediaTracker (this);

le this représente le composant sur lequel on dessine l'image, ici il s'agit de la zone d'affichage de l'applet.

Dans l'instruction:

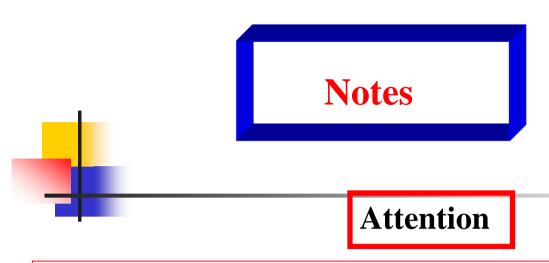
tracker.addImage (fond, 0);

le deuxième paramètre (0) représente l'importance de l'image (0 est la plus grande importance).

La méthode checkAll () vérifie l'état du MediaTracker ie si toutes les images sont arrivées ou non.

Ici si les images ne sont pas toutes arrivées, on demande de redessiner l'applet dans 100 millisecondes (avec repaint (100)) et on s'en arrête là (avec return). IL NE FAUT JAMAIS OUBLIER le repaint(...): par son appel, le MediaTracker prévient Java que tout est terminé.

En clair, il n' y a jamais de repaint () automatique et le message "Chargement en cours..." demeurera à l'ecran. FST/UCAD



La méthode checkAll () renvoie aussi true si le chargement des images a été interrompu, ou s'il y a eu des erreurs de chargement.

Le MediaTracker possèdent d'autres méthodes plus fines permettant de détecter les erreurs (par exemple, la méthode isErrorAny (), qui vaut true si une erreur se produit.

Dans l'instruction g.drawImage (fond,20,20,this);

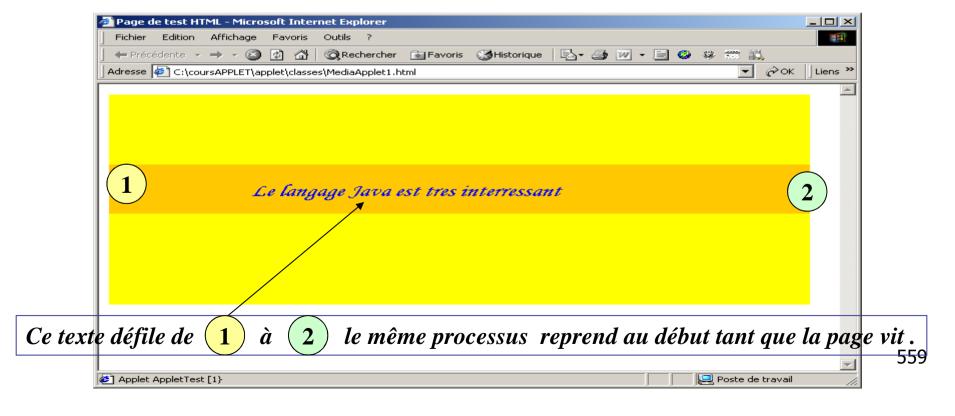
le dernier paramètre (ici this) représente un objet de l'interface ImageObserver.

Cette interface contient un e seule méthode, **imageUpdate** () qui est invoquée chaque fois qu'une portion d'image est chargée. Elle permet de gérer le téléchargement d'images à partir d'un réseau, en tâche de fond.

Applet avec images qui bougent

Pour réaliser des applets avec des images qui bougent ou du texte défilant, il faut faire usage des threads.

Voyons un exemple dans lequel nous construisons une applet contenant une bande où on fait défiler un texte de bout en bout et de façon continue.



Applet avec texte défilant (1/3)



```
public class MediaApplet1 extends Applet implements Runnable {
Panel pan;
Thread t;
    /*Initialiser l'applet*/
    public void init () {
    this.setBackground (Color.yellow);
    this.setLayout (null);
    pan = new Panel(null); pan.setBackground (Color.orange);
    pan.setBounds (0,100, this.getWidth (),70);
    this.add (pan);
}
```

Applet avec texte défilant (2/3)



```
public void paint (Graphics g){
   int i:
   for (i = 0; i < pan.getWidth(); i++)
       { g = pan.getGraphics();//on veut dessiner sur le panneau et non sur la zone de l'applet
      String s = "Le langage Java est très intéressant";
      g.setFont (new Font("Monotype Corsiva",Font.ITALIC ,24));
          if (s.length ( ) +i < pan.getWidth ( ) ){</pre>
            g.setColor (Color.blue ) ;
            g.drawString (s,i ,45);
            try {Thread.sleep (100);}
            catch (InterruptedException er){}
           pan.update (g);//fondamentale sinon les écritures e superposent au fur et à mesure de l'affichage
```

Applet avec texte défilant (3/3)



```
/*redéfinition de start*/
public void start ( ){
 if (t = = null){
    t = new Thread (this);
    t.start();
    repaint ();
/*redefinition de run*/
   public void run ( ){
       while (true) {
                     paint (this.getGraphics ( ) );
  }//fin de la classe
```





Les scintillements constatés dans la manipulation des animations sont souvent dus à une utilisation non optimale des méthodes repaint (), update () et paint ().

En effet, la méthode *repaint* (), dès que cela est possible, appelle la méthode *update* () qui, à son tour appelle *paint* (). *Update* () dessine le fond de l'écran, puis appelle *paint* (): c'est ce qui produit le tremblements.

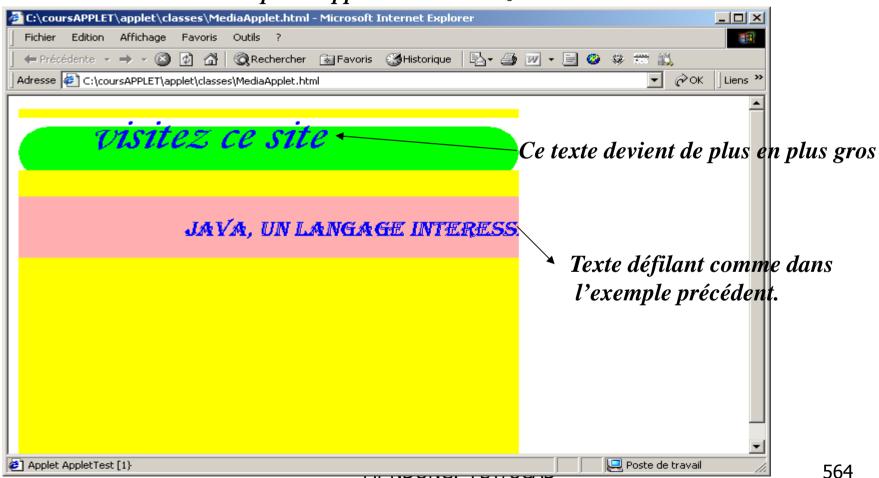
Pour résoudre ce problème, une première solution consiste à redéfinir la méthode *update* (). Dans le code de l'applet, vous écrivez:

```
public void update (Graphics g)
    { paint (g);
    }
```

Exercice: Applet avec plusieurs textes en mouvement

(nécessité d'utilisation de plusieurs threads)

Réaliser cette petite applet où vous avez deux textes en mouvement.



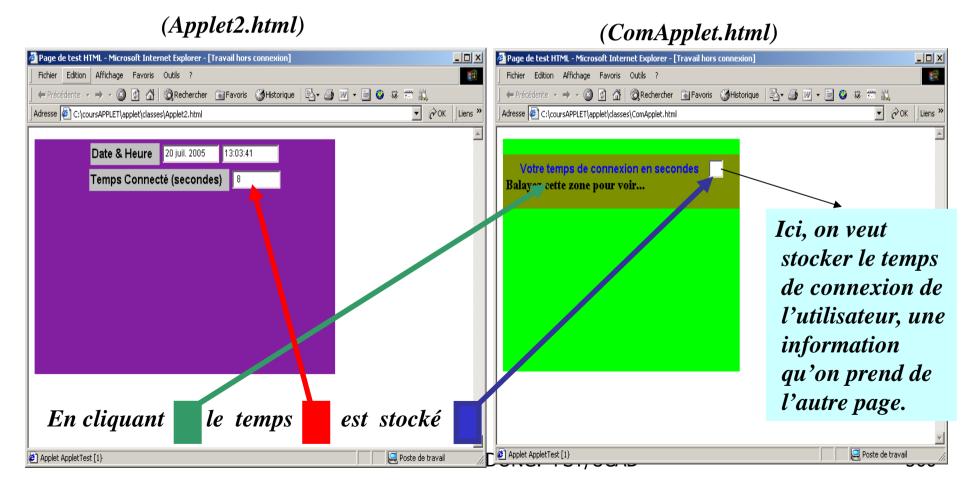


Communication entre applets

Java permet d'avoir dans une seule page HTML deux applets qui s'exécutent et communiquent entre elles.

Pour cela, il suffit d'écrire <u>un seul fichier HTML</u> dans lequel il y a les 2 applets. La communication sera rendu possible par l'exploitation du paramètre NAME de la balise <applet>.

Nous allons réunir ces deux pages web en une seule page.



Applet2.class est la première applet dans les exemples précédents (cf. diapos 493 à 495). Nous donnons ci-dessous le code de la seconde applet dont le byte code sera mis dans ComApplet.class, nous donnons également son fichier HTML (ComApplet.html).

ComApplet.html

```
public class ComApplet extends Applet {
Label 1;
TextField tx;
Panel pan;
/**Initialiser l'applet*/
 public void init(){
  try {
       jbInit();
  catch (Exception e) {     e.printStackTrace ( );   }
/*pour que le clic sur la zone de coordonnées (x,y) est un effet */
 public boolean mouseDown (Event e, int x, int y){
Applet2 app = (Applet2) this.getAppletContext () .getApplet( "Com2"
 tx.setText (app.compteur+"");
 return true;
                                 Mr NDONG. FST/UCAD
                                                                              568
```



```
public void paint (Graphics g){
    g = pan.getGraphics ();
    g.setFont (new Font("Times New Roman", Font.BOLD ,18));
    g.drawString ("Balayez cette zone pour voir...",5,45);
    this.repaint (); // pour rendre permanent le dessin
}

public void update (Graphics g){
    paint (g);
}
```



```
/**Initialiser les composants*/
private void jbInit () throws Exception {
    this.setLayout (null);
    this.setBackground (Color.green .brighter () .brighter ());
    l = new Label ("Votre temps de connexion en secondes");
    l.setFont (new Font("CASTELLAR",Font.PLAIN +Font.BOLD ,16));
    l.setForeground (Color.blue );
    tx = new TextField();
    pan = new Panel();pan.setBackground (new Color(125,145,0,15));
    pan.setBounds (0,20, this.getWidth () ,70);
    this.add (pan); pan.add (l);pan.add (tx);
}
```

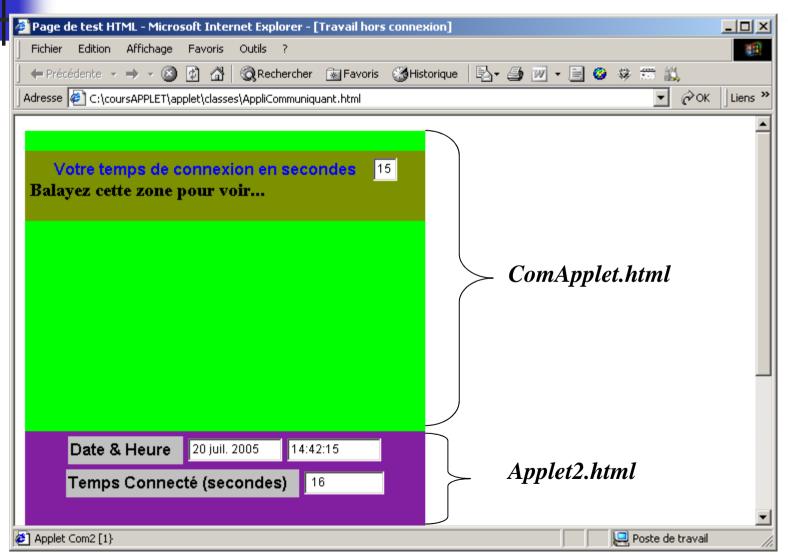
Nous avons Applet2.class et Applet2.html, et aussi Nous avons ComApplet.class et ComApplet.html.

Maintenant, nous allons construire la page qui contient ces deux applets: AppletComm. html

(le fichier AppletComm.html)

</body> </html>

(AppletComm.html)



Notes

Dans *Applet2.html* et *ComApplet.html*, il n'est pas nécessaire de mentionner le paramètre NAME, puisque les deux applets ne communiquent pas séparément. Mais l'utilisation de ce paramètre est obligatoire dans *AppletComm.html*.

La méthode getAppletContext() donne l'environnement (le contexte) actuel de l'applet. Elle rend un objet de type AppletContext qui possède une méthode getApplet (String s) qui permet de recupérer l'applet de nom s.

Cet objet possède aussi une méthode getApplets () qui permet de lister toutes les applets d'une page, sous forme d'un objet de type Enumeration (voir les <u>Collections</u>)

Module 15

Java DataBase Connectivity: JDBC

JDBC est une API Java (ensemble de classes et d'interfaces défini par SUN et les acteurs du domaine des BD) permettant d'accéder aux bases de données à l'aide du langage Java via des requêtes SQL (langage permettant de dialoguer avec un SGBDR).

Cette API permet d'atteindre de façon quasi transparente des bases Sybase, Oracle, Informix,... avec le même programme Java JDBC.

JDBC est fourni par le paquetage java.sql

Principe de JDBC



Java DataBase Connectivity (JDBC)

permet à un programme Java d'interagir

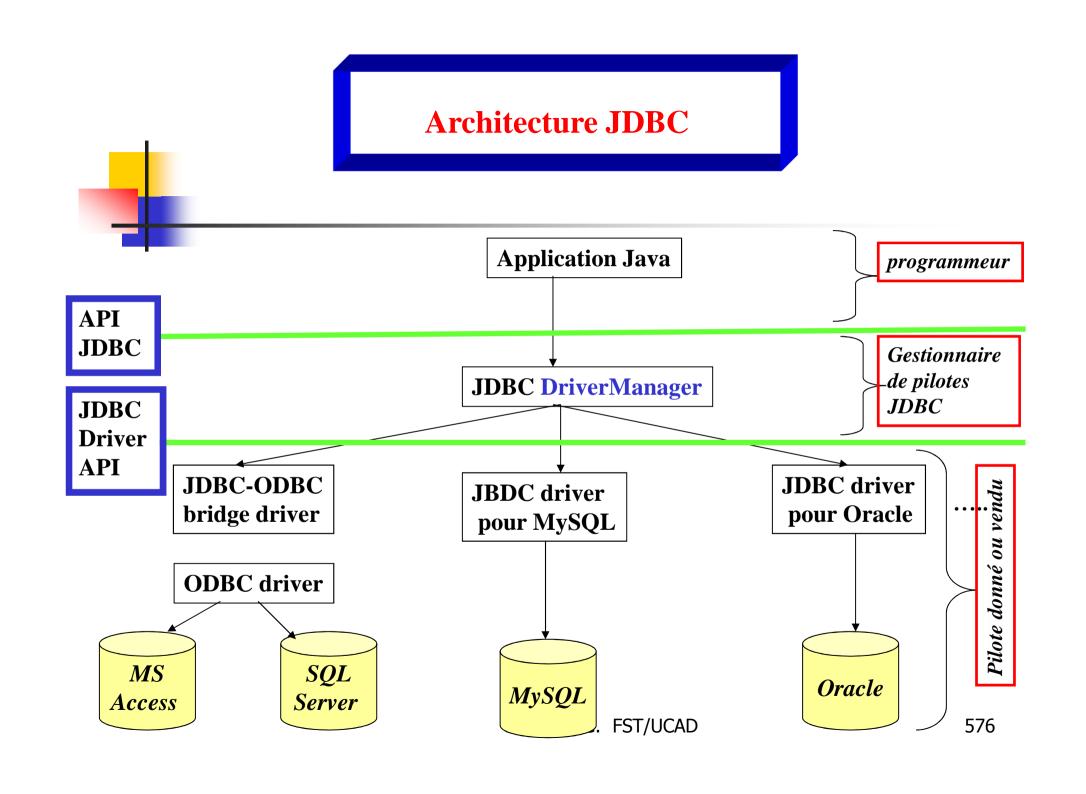
- ✓ localement ou à distance
- ✓ avec une base de données relationnelle

fonctionne selon un principe client/serveur

- ✓ client = le programme Java
- ✓ serveur = la base de données

principe

- ✓ le programme Java ouvre une connexion
- ✓ il envoie des requêtes SQL
- ✓ il récupère les résultats de la requête
- ✓..... (ici les traitements à faire sur les données recueillies)
- ✓ il ferme la connexion une fois les traitements terminés



Pilotes (drivers)

L'ensemble des classes qui implémentent les interfaces spécifiées par JDBC pour un SGBD (Système de Gestion de Bases de Données) particulier est appelé un pilote JDBC. Les protocoles d'accès aux bases de données étant propriétaires, il y a donc plusieurs drivers pour atteindre diverses BD.

Chaque BD utilise un pilote qui lui est propre et qui permet de convertir les requêtes dans le langage natif du SGBDR.

Les drivers dépendent du SGBD auquel ils permettent d'accéder.

Pour travailler avec un SGBD, il faut disposer de classes (driver) qui implémentent les interfaces de JDBC.

JDBC est totalement indépendant de tout SGBD: la même application peut être utilisée pour accéder à une base Oracle, Sybase, MySQL, etc.





Il existe quatre grandes familles de pilotes JDBC en Java

Type I: pont (ou passerelle) JDBC-ODBC

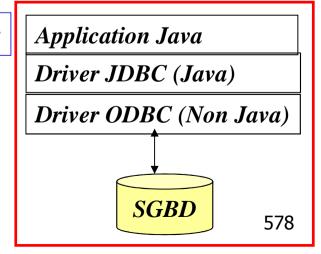
Le driver accède à un SGBDR en passant par les drivers ODBC (standard Microsoft: Open DataBase Connectivity) via un pont JDBC-ODBC:

Les appels JDBC sont traduits en appels ODBC

Ce type de pilote ne peut pas être utilisé par des applets puisque il ne permet

qu'un accès local.

Il est fourni par Sun: sun.jdbc.odbc.JdbcOdbcDriver

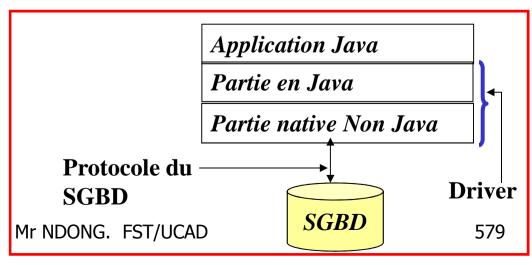


Mr NDONG. FST/UCAD

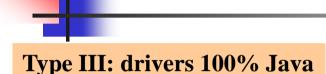


Ce sont des drivers partiellement écrits en Java et qui reposent sur des librairies propriétaires (des fonctions natives non Java : par ex. C) pour accèder au SGBD. Ils peuvent gérer des appels C/C++ directement avec la base.

Ne convient pas pour les applets (sécurité). Interdiction de charger du code natif dans la mémoire vive de la plateforme.





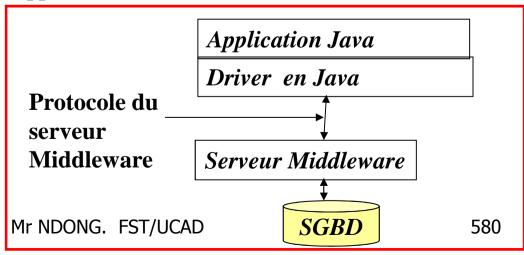


Ils communiquent localement ou à distance avec le SGBD selon un protocole réseau générique (Sockets). La communication se fait par une application intermédiaire (Middleware) sur le serveur.

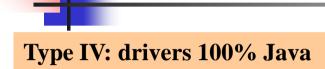
Le middleware accède par un moyen quelconque (par exemple JDBC si écrit en Java) aux différents SGBDR.

Ils sont portables car entièrement écrits en Java.

Donc appropriés pour les Applets et les Applications.







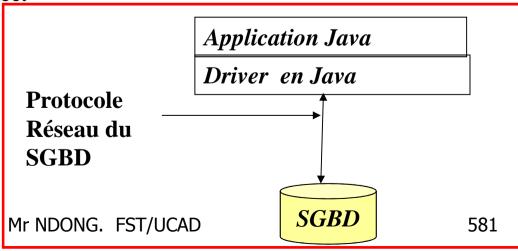
Ces drivers utilisent un protocole réseau propriétaire spécifique au SGBD.

- interagit avec la base de données via des sockets,
- généralement fourni par l'éditeur

Aucun problème d'exécution pour une *applet* si le SGBDR est installé au même endroit que le serveur WEB.

Sécurité assurée pour l'utilisation des sockets: une applet ne peut ouvrir une connexion

que sur la machine où elle est hébergée.





4

Un programme JDBC débute toujours par le chargement du pilote approprié pour la BD. Mais puisque le programme a la possibilité d'accéder à plusieurs types de BD, il peut avoir plusieurs pilotes.

C'est au moment de la connexion à la BD que le DriverManager choisit alors le bon pilote.

DriverManager = gestionnaire de tous les drivers chargés par un programme java.

Les URLs JDBC



(Localisation et accès à la BD: établissement d'une connexion avec la BD)

Après le chargement du driver, il faut <u>localiser</u> la BD en spécifiant son emplacement.

Pour chaque type de driver il existe un schéma de désignation de la base à atteindre avec JDBC:

Type I:

jdbc: odbc:source

source = source de données ODBC.

(ex: jdbc: odbc: employe)

Type II:

jdbc: protocole

où protocole = protocole spécifique et utilisant des méthodes natives.

Type III et IV:

jdbc: driver: adresse

ex: jdbc:msql://elios.lip6.fr/employe

jdbc:oracle: thin :@elios.lip6.fr:employe

Chaque URL est de la forme: jdbc: sous-protocole :base_de_donnée





(différente étapes lors de l'utilisation de JDBC)

1. Chargement du driver (= chargement de la classe du driver dans la JVM) Class.forName (String driverName);

Class.forName ("sun.jdbc.odbc.JdbcOdbcDriver");

Class.forName ("oracle.jdbc.driver.OracleDriver");

Class.forName ("org.gjt.mm.mysql.Driver");

Quand une classe <u>Driver</u> est chargée, elle doit créer une instance d'elle même et s'enregistrer auprès du *DriverManager*.

Certains compilateurs refusent la notation précédente et demandent :

Class.forName ("driver_name").newInstance ();

Cette étape 1 constitue l'enregistrement du driver JDBC.

Structure d'un programme JDBC (2/4)



Une fois le driver enregistré, une connexion peut être établie avec la BD. pour obtenir une connexion à un SGBD, il faut faire une demande à la classe gestionnaire de drivers:

Demande permise par la méthode getConnection (...) de la classe DriverManager Cette méthode peut prendre 3 arguments au plus:

- l' URL vers la BD
- le nom de l'utilisateur de la base
- son mot de passe cette méthode renvoie un objet de type Connection.

Connection con =
 DriverManager.getConnection ("jdbc:odbc:employe", "login", "passwd");

Structure d'un programme JDBC (3/4)

3. Création de requêtes SQL en utilisant l'objet de type Connection (étape 2). dans cette étape on crée en fait une zone de description de requêtes ie un espace où l'on pourra exécuter des opérations SQL. cette zone est un objet de la classe Statement que l'on crée par la méthode createStatement ().

Statement st = con.createStatement();

Il existe trois types de Statement:

Statement: requêtes statiques simples,

PreparedStatement: requêtes dynamiques pré compilées (avec paramètres d' I/O),

CallableStatement: procédures stockées.

Les deux derniers seront développés plus tard dans ce cours.

Structure d'un programme JDBC (4/4)



4. Envoi et exécution de requêtes.

Il existe trois types d'exécution de requêtes:

- > executeQuery (...): pour les requêtes SELECT qui retournent un ResultSet (tuples),
- > executeUpdate (...): pour les requêtes INSERT, UPDATE, DELETE, CREATE TABLE DROP TABLE qui retourne un entier (int) désignant le nombre de tuples traités.
- > execute (): pour les procédures stockées (cas rares).

Les méthodes executeQuery () et executeUpdate() de la classe Statement prennent comme argument une chaîne (String) indiquant la requête SQL à exécuter.

```
Statement st = con.createStatement ();
ResultSet rs = st.executeQuery ("select * from Client");
```

Notes 1



Le code SQL n'est pas interprété par Java.

- c'est le pilote associé à la connexion (et au finish par le moteur de la BD) qui interprète la requête SQL.
- si une requête ne peut s'exécuter ou qu'une erreur de syntaxe SQL a été détectée, l'exception SQLException est levée.

Le driver JDBC effectue d'abord un premier accès à la BD pour découvrir les types des colonnes impliquées dans la requête puis un deuxième pour l'exécuter.



Notes 2: Traitement des données retournées

L'objet ResultSet (retourné par l'exécution de executeQuery ()) permet d'accéder aux champs des enregistrements (tuples) sélectionnés.

Seules les données demandées sont transférées en mémoire par le driver JDBC. Il faut donc les lire manuellement et les stocker dans des variables pour un usage ultérieur.

La méthode next () de ResultSet permet de parcourir itérativement ligne par ligne l'ensemble des tuples sélectionnés.

Cette méthode:

- retourne false si le dernier tuple est lu, true sinon,
- chaque appel fait avancer le curseur sur le tuple suivant,
- initialement, le curseur est positionné avant le premier tuple

Exécuter next () au moins une fois pour avoir le premier.

while (rs.next ()) { //traitement tuple par tuple}

Impossible de revenir au tuple précédent ou de parcourir l'ensemble dans un ordre quelconque.

Mr NDONG. FST/UCAD



Notes 2: Traitement des données retournées

La méthode next () permet de parcourir le ResultSet du premier au dernier enregistrement. Mais il existe d'autres façons aussi de parcourir un ResultSet.

On peut parcourir le ResultSet ligne par ligne de façon itérative de la <u>fin</u> vers le <u>début</u>:

--- utilisez pour cela la méthode previous (), [while (rs.previous ()) { // ...}]

En <u>déplacement absolu</u>: on peut aller exactement à la nième ligne:

---utilisez alors la méthode absolute (int row), [while (rs.absolute (i)) { // ...}] vous pourrez faire usage des méthodes first (), last (),....

En <u>déplacement relatif</u>: on peut aller à la nième ligne à partir de la position courante du curseur:

--- utilisez la méthode relative (int row) et pour accéder à un enregistrement les méthodes afterLast (), beforeFirst (), ...



Notes 3: Traitement des données retournées

Les <u>colonnes</u> d'une table de la BD sont référencées par leur <u>numéro</u> (commençant par 1) ou par leur <u>nom</u>.

L'accès aux valeurs des colonnes se fait par des méthodes de la forme getXXX (...) permettant la lecture de données du type XXX dans chaque colonne du tuple courant.

int val = rs.getInt (3); // acces à la 3e colonne

String prod = rs.getString ("Produit") ;// acces à la colonne de nom Produit

Premier exemple complet d'utilisation de JDBC (1/3)

(exemple avec une source de données ODBC pour MS ACCESS)

```
import java.sql.*;
public class TestJDBC {
  public static void main (String [ ] args) {
 /** chargement du Driver ODBC-JDBC*/
 try {
      Class.forName ("sun.jdbc.odbc.JdbcOdbcDriver");
catch (Exception e) {
      System.out.println ("Erreur dans le chargement du driver"); System.exit (0);
/**Connexion à la base*/
Connection con = null;
try {
    con = DriverManager.getConnection ("jdbc: odbc: employe", "dba", "sql");
catch (SQLException e) {
               System.out.println ("Impossible de sonnecter à la BD"); }
                                                                                592
```

Premier exemple complet d'utilisation de JDBC (2/3)

(exemple avec une source de données ODBC pour MS ACCESS)

```
/**création d'une zone d'exécution de requêtes SQL*/
 Statement stmt = null;
try
         stmt = con.createStatement();
  catch (SQLException e) {
        System.out.println("Impossible de créer de Statement ");
/**exécution de requêtes et récupération des données demandées*/
ResultSet rs = null;
try { rs = stmt.executeQuery ("SELECT * FROM CLIENT ");
catch (SQLException e) {
        System.out.println(" Erreur de requête SQL ");
```



(exemple avec une source de données ODBC pour MS ACCESS)

```
/**parcours du résultat: affichage des données lues*/
try {
while (rs.next ( )) {
                   String prenom = rs.getString ("prenom");
                   int age = rs.getInt ("age");
                   System.out.println (prenom+" a "+age+" ans");
/** fermeture de tout lien avec la BD*/
finally { try{ rs.close ( );
             stmt.close();
             con.close ( );}
         catch (Exception e) { }
catch (SQLException e ){ }
}// fin de la classe
                             Mr NDONG. FST/UCAD
```

Correspondances Types données SQL/Java

Si vous écrivez une instruction telle que String prenom = rs.getString ("prenom"); cela signifie que vous êtes convaincu que la colonne de nom PRENOM a été créée sous SQL avec le type VARCHAR. Autrement cette instruction causerais une erreur de runtime.

Donc il est bon de savoir la correspondance entre types SQL et types Java pour pouvoir manipuler convenablement les données lues.

Types SQL	Types Java	Méthodes
CHAR/ VARCHAR	String	getString ()
INTEGER	int	getInt ()
TINYINT	byte	getByte ()
SMALLINT	short	getShort ()



Types SQL	Types Java	Méthodes
BIGINT	long	getLong ()
bit	boolean	getBoolean ()
REAL	float	getFloat ()
NUMERIC DECIMAL	java.math.BigDecimal	getBigDecimal ()
DATE	java.sql.Date	getDate ()
TIME	java.sql.Time	getTime ()
TIMESTAMP	java.sql.TimeStamp	getTimeStamp ()
FLOAT DOUBLE	double	getDouble ()

Second exemple complet d'utilisation de JDBC (1/3)

(utilisation de JDBC pour MySQL)

Pour travailler avec JDBC et MySQL, il faut commencer par installer le driver pour une base de données MySQL.

Ce driver se nomme Connector/J. Vous pouvez le récupérer à l'adresse http://www.mysql.com/products/connector-j

Ensuite décompressez le fichier jar ou le zip.

Et maintenant effectuez l'une des opérations suivantes:

- copier le fichier jar (mysql-connector-java-3.1.10-bin.jar par exemple) dans l'un des répertoires de votre variable CLASSPATH
- ajouter le répertoire contenant le fichier jar à votre CLASSPATH
- copier le fichier jar dans \$JAVA_HOME/jre /lib/ext. (pour un déploiement)

Et enfin dans le programme créez une librairie qui encapsule le fichier jar. Pour tester vous pouvez créer le programme suivant:

Second exemple complet d'utilisation de JDBC (2/3)

(utilisation de JDBC pour MySQL)

```
import java.sql .*;
public class TestJavaMySQL {
    static Connection con;
    static Statement st;
    static ResultSet rs;
    public static void main (String[] args) {

    try { /*chargement du driver*/
        Class.forName("com.mysql.jdbc.Driver").newInstance();
    }
    catch (Exception e){System.out .println("Erreur driver: "+e.getMessage());}
```

```
NB: vous pouviez aussi prendre comme driver:
Class.forName ("org.gjt.mm.mysql.Driver").newInstance();
```

Second exemple complet d'utilisation de JDBC (3/3)

(utilisation de JDBC pour MySQL)

```
try {con = DriverManager.getConnection ("jdbc:mysql: //localhost /Employe","root","");
catch (Exception ez) {System.out.println("Erreur de connexion "+ ez.getMessage ());}
try { st = con.createStatement();
catch (SQLException t){System.out.println ("Erreur de Statement "+t.getMessage());}
try {
    rs = st.executeQuery("select * from client");
      while (rs.next())
       { System.out .println(rs.getObject (1)+" "+rs.getObject(2)+" "+rs.getObject(3) ) ;
catch (Exception er) {System.out.println("Erreur ResultSet "+er.getMessage()); }
                      st.close();
 try { rs.close ( );
                                     con.close();
 catch (Exception d) { }
}}
```

L'interface java.sql.PreparedStatement

(pour l'utilisation efficace des requêtes pré compilées: SQL DYNAMIQUE)

Parfois, il est très utile et plus efficace de faire usage de l'objet PreparedStatement pour envoyer une instruction SQL à la base de données.

En effet, si vous voulez *exécuter* un objet <u>Statement plusieurs fois</u>, le temps d'exécution sera *réduit* si vous utilisez plutôt un objet <u>PreparedStatement</u>.

La fonctionnalité d'un objet PreparedStatement, contrairement à l'objet Statement, est de lui fournir une instruction SQL dès sa création. L'avantage est que l'instruction SQL sera directement envoyée au SGBD, où elle sera compilée.

Ainsi l'objet PreparedStatement ne contient plus seulement une instruction SQL, mais bien une instruction SQL pré compilée.

Cela signifie donc que quand le *PreparedStatement* est exécuté, le SGBD a juste à lancer l'instruction SQL sans avoir à le compiler à nouveau.

L'interface java.sql.PreparedStatement

Comment créer un objet PreparedStatement?

Un objet de type Connection est nécessaire lors de la création d'un objet PreparedStatement. Cet objet appelle la méthode prepareStatement de la même classe

PreparedStatement pst = objet_conn.prepareStatement ("UPDATE Compte SET solde = ? WHERE numC = ?);

Tout objet **PreparedStatement** est caractérisé par des <u>paramètres</u> indispensables représentés par les point d'interrogation (?).

La transmission de ces paramètres est réalisée par les méthodes setXXX (...) qui permettent de fournir des valeurs qui vont être utilisées à la place des ?.

NB: dans setXXX (...), XXX désigne n'importe quel type Java.

```
Après création, on peut transmettre effectivement des valeurs à l'objet pst : pst.setDouble (1,20000); // transmission de la valeur 20000 au premier ? pst.setInt (2, 703); // transmission de la valeur 703 au second ? pst.executeUpdate (); // exécution maintenant de la requête pst.close (); Mr NDONG. FST/UCAD
```

Exemple complet de PreparedStatement

```
import java.sql .*;
public class TestPreStat {
 public static void main (String [ ] args) {
try { Class.forName ("com.mysql.jdbc.Driver").newInstance();
   Connection con=
         DriverManager.getConnection ("jdbc:mysql://127.0.0.1/Employe","root","");
   PreparedStatement p =
            con.prepareStatement ("UPDATE compte SET solde =? where numC =?");
      p.setDouble (1,200000);
      p.setInt (2,702);
      p.setDouble (1,500000);
      p.setInt (2,703);
      p.executeUpdate();
      p.close();
      con.close();
catch (Exception er) { er.printStackTrape (IDGN)6.} FST/UCAD
```

Notes sur le PreparedStatement



Si vous réalisez des opérations de mises à jour (ex UPDATE) sur l'objet PreparedStatement, utilisez la méthode executeUpdate() pour effectuer la transaction dans la base.

Si vous réalisez des opérations de sélection (SELECT), utilisez la méthode executeQuery () pour effectuer la transaction dans la base.

Quelques fois, le type de la requête est indéfinie à priori (MàJ ou select): utilisez dans ce cas la méthode execute () qui renvoie un booléen (true si c'est une requête SELECT, false dans les cas de MàJ).

<u>ATTENTION</u>: les méthodes *executeUpdate* () et *executeQuery* () ne prennent aucun paramètre. C'est évident puisque la requête est déjà transmise à l'objet *PreparedStatement*.



JDBC 2.0 permet de réaliser des mises à jour de masse en regroupant plusieurs traitements pour les envoyer <u>en une seule fois</u> au SGBD. Ceci permet d'améliorer les performances surtout si le nombre de traitements est important.

Cette fonctionnalité n'est pas obligatoirement supportée par le pilote. La méthode supportsBatchUpdate () de la classe DatabaseMetaData permet de savoir si elle est utilisable avec le pilote. Elle renvoie un boolean.

Plusieurs méthodes ont été ajoutées à l'interface Statement pour pouvoir utiliser les mises à jour de masse :

Les mises à jour de masse (Batch Updates)



Méthodes	Rôle
void addBatch (String)	permet d'ajouter une chaîne contenant une requête SQL
int [] executeBatch ()	permet d'exécuter toutes les requêtes. Elle renvoie un tableau d'entier qui contient pour chaque requête, le nombre de mises à jour effectuées.
void clearBatch ()	supprime toutes les requêtes stockées

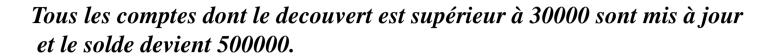
Lors de l'utilisation de *batch update*, il est préférable de positionner l'attribut autocommit à false afin de faciliter la gestion des transactions et le traitement d'une erreur dans l'exécution d'un ou plusieurs traitements.



Une exception particulière peut être levée en plus de l'exception SQLException lors de l'exécution d'une mise à jour de masse. L'exception SQLException est levée si une requête SQL d'interrogation doit être exécutée (requête de type SELECT).

L'exception BatchUpdateException est levée si une des requêtes de mise à jour échoue. L'exception BatchUpdateException possède une méthode getUpdateCounts () qui renvoie un tableau d'entier qui contient le nombre d'occurrences impactées par chaque requête réussie.

Les mises à jour de masse (Batch Updates):Exemple



```
public class TestBatchUpdate {
  public static void main(String [ ] args) {
           Class.forName ("org.gjt.mm.mysql.Driver").newInstance ();
try {
catch (Exception e) { }
try {
Connection con = DriverManager.getConnection
                                       ("jdbc: mysql://localhost/banque","root","");
con.setAutoCommit (false);
Statement st = con.createStatement();
st.addBatch ("UPDATE compte SET solde = 500000 where decouvert >= 30000 ");
      st.executeBatch();
      st.close();
      con.close();
    catch(Exception er){ er.printStackTrace ( ) ;}
                                     Mr NDONG. FST/UCAD
                                                                                 607
```

Les MétaDonnées

On peut avoir besoin quelque fois des informations sur la structure et le schéma de la base de données elle-même. Ces informations sont appelées des Métadonnées.

Pour obtenir ces renseignements, il faut utiliser un objet de la classe **DatabaseMetaData** à partir de l'objet **Connection**. La méthode **getMetaData** () de la classe **Connection** permet de créer cet objet.

DatabaseMetaData donneesBD = objet_con.getMetaData();

A partir de cet objet, cette classe fournit beaucoup de renseignements sur la BD. Par exemple:

```
DatabaseMetaData d = con.getMetaData();
String nomSGBD = d.getDatabaseProductName(); // nom du SGBD
String versionDriver = d.getDriverVersion(); // version du driver
String nomDriver = d.getDriverName(); // nom du driver
```

Les MétaDonnées

On peut aussi avoir des informations sur les objets ResultSet générés:

- nombre de colonnes contenues dans le ResultSet,
- noms des colonnes
- types des colonnes,
- etc.

Pour récupérer ces Métadonnées, il faut créer un objet de la classe ResultSetMetaData grâce à la méthode getMetaData () de la classe ResultSet.

ResultSetMetaData rsmd = objet_resultset.getMetaData ();

int columnCount = rsmd.getColumnCount(); //nombres de colonnes
String columLabel = rsmd.getColumnLabel(i); // nom de la colonne i
String columnType = rsmd.getColumnTypeName(i) // classe de la colonne i

NB: ces informations sont très utiles surtout lors de la création de modèles de table pour le chargement d'objets JTable.