Programmation orientée objet

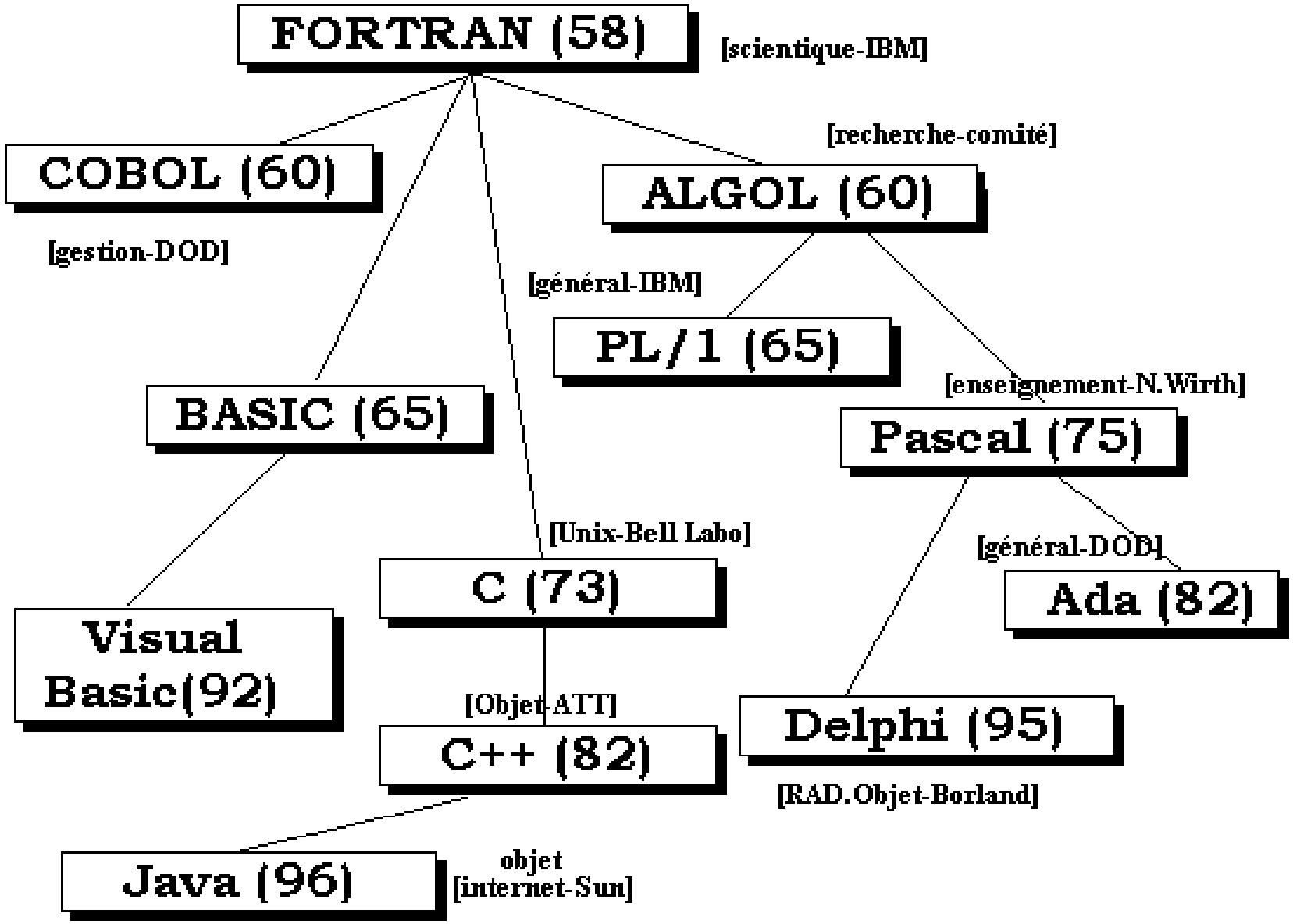
Mme Soumia ZITI-LABRIM
s.ziti@fsr.ac.ma
Université Mohamed V
Faculté des Sciences
Département Informatique

Généralité et Structures fondamentales de java

Plan

- **Caractéristiques de Java**
- **Structures fondamentales**
- **La programmation par objets en Java**
- **Les paquetages**
- **les entrées / sorties en Java**
- **Les exceptions**
- **Les collections en Java**

Histoire des langages



Les différentes versions de Java

- **De nombreuses versions de Java depuis 1995**
- **Java 1.0 en 1995**
  - Java 1.1 en 1996
- **Java 1.2 en 1999 (Java 2, version 1.2)**
  - Java 1.3 en 2001 (Java 2, version 1.3)
  - Java 1.4 en 2002 (Java 2, version 1.4)
  - Java 5.0 en 2004
  - Java 6.0 en 2006
- **Évolution très rapide et succès du langage**
- **Une certaine maturité atteinte avec Java 2**
- **Mais des problèmes de compatibilité existent**
  - entre les versions 1.1 et 1.2/1.3/1.4
  - avec certains navigateurs

Analyse du problème

- **Se poser les bonnes questions**
  - Quelles sont les objets qui interviennent dans le problème? Quelles sont les données, les objets, que le programme va manipuler?
  - Quelles vont être les relations entre ces objets? Quelles sont les opérations que je vais pouvoir effectuer sur ces objets?
- **Savoir être :**
  - **efficace** : quelle méthode me permettra d'obtenir le plus vite, le plus clairement, le plus simplement possible les résultats attendus ?
  - **paresseux** : dans ce que j'ai développé avant, que puis-je réutiliser ?
  - **prévoyant** : comment s'assurer que le programme sera facilement réutilisable et extensible ?

Caractéristiques du langage Java

- **Simple**
- **simplifications des fonctionnalités essentielles**
- **Familier**
- **Orienté objet**
- **Sûr**
- **Fiable**
- **Indépendant de l'architecture**
- **Multi-tâches**

Java, un langage de programmation

- **Applications Java** : programmes autonomes, "stand-alone"
- **Applets (mini-programmes)** : Programmes exécutables uniquement par l'intermédiaire d'une autre application
  - navigateur web : Netscape, Internet explorer, Hotjava
  - application spécifique : Appletviewer
- **Java est un langage interprété**
  - La compilation d'un programme Java crée du pseudo-code portable : le "byte-code"
  - Sur n'importe quelle plate-forme, une machine virtuelle Java peut interpréter le pseudo-code afin qu'il soit exécuté
- **Les machines virtuelles Java peuvent être**
  - des interpréteurs de byte-code indépendants (pour exécuter les programmes Java)
  - contenues au sein d'un navigateur (pour exécuter des applets Java)

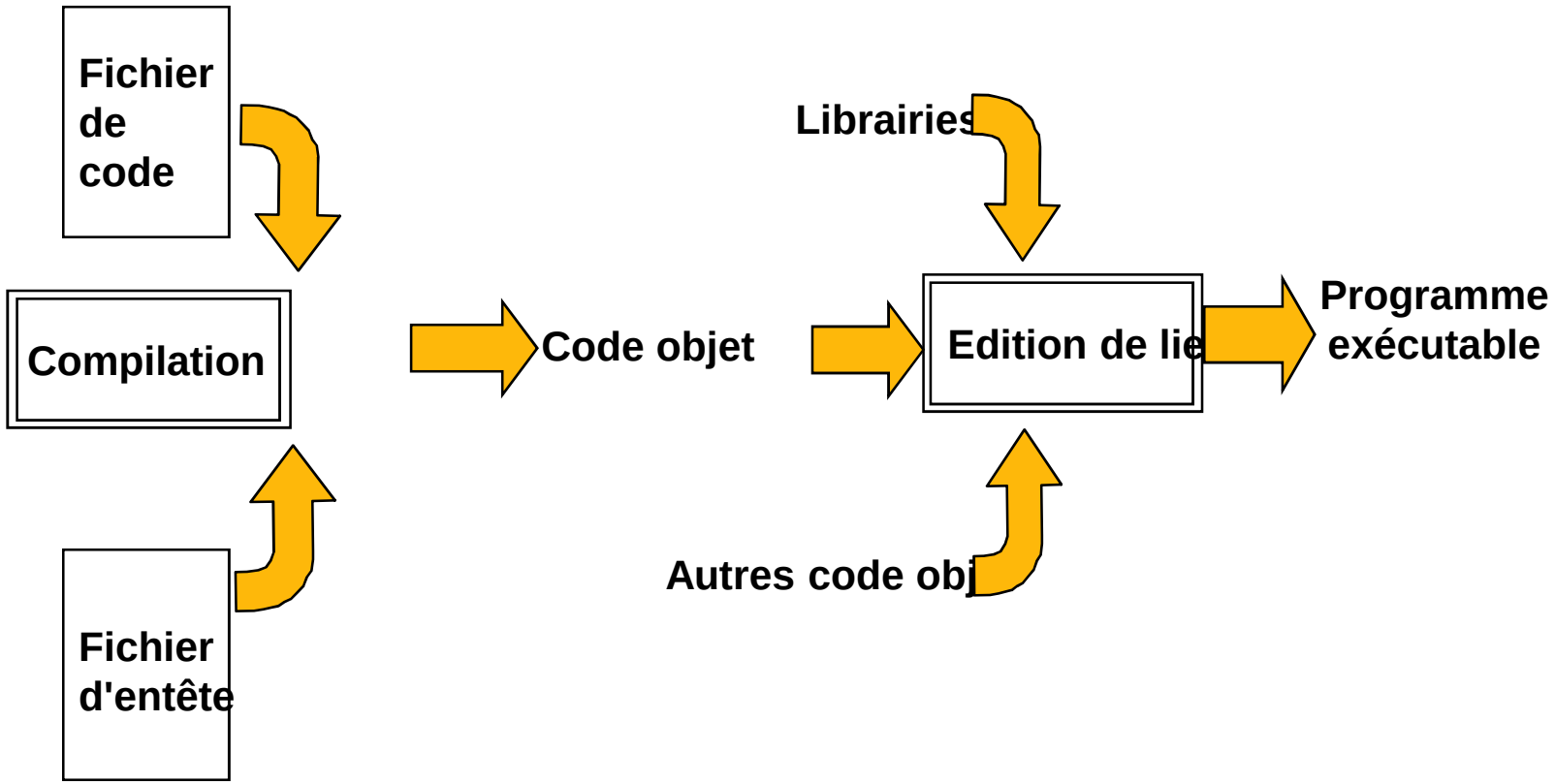
Java, un langage indépendant

- **Avantages :**
  - **Portabilité**
    - Des machines virtuelles Java existent pour de nombreuses plates-formes dont : Solaris, Windows, MacOS
  - **Développement plus rapide**
    - courte étape de compilation pour obtenir le byte-code,
    - pas d'édition de liens,
    - déboguagge plus aisé,
  - **Le byte-code est plus compact que les exécutables**
    - pour voyager sur les réseaux.

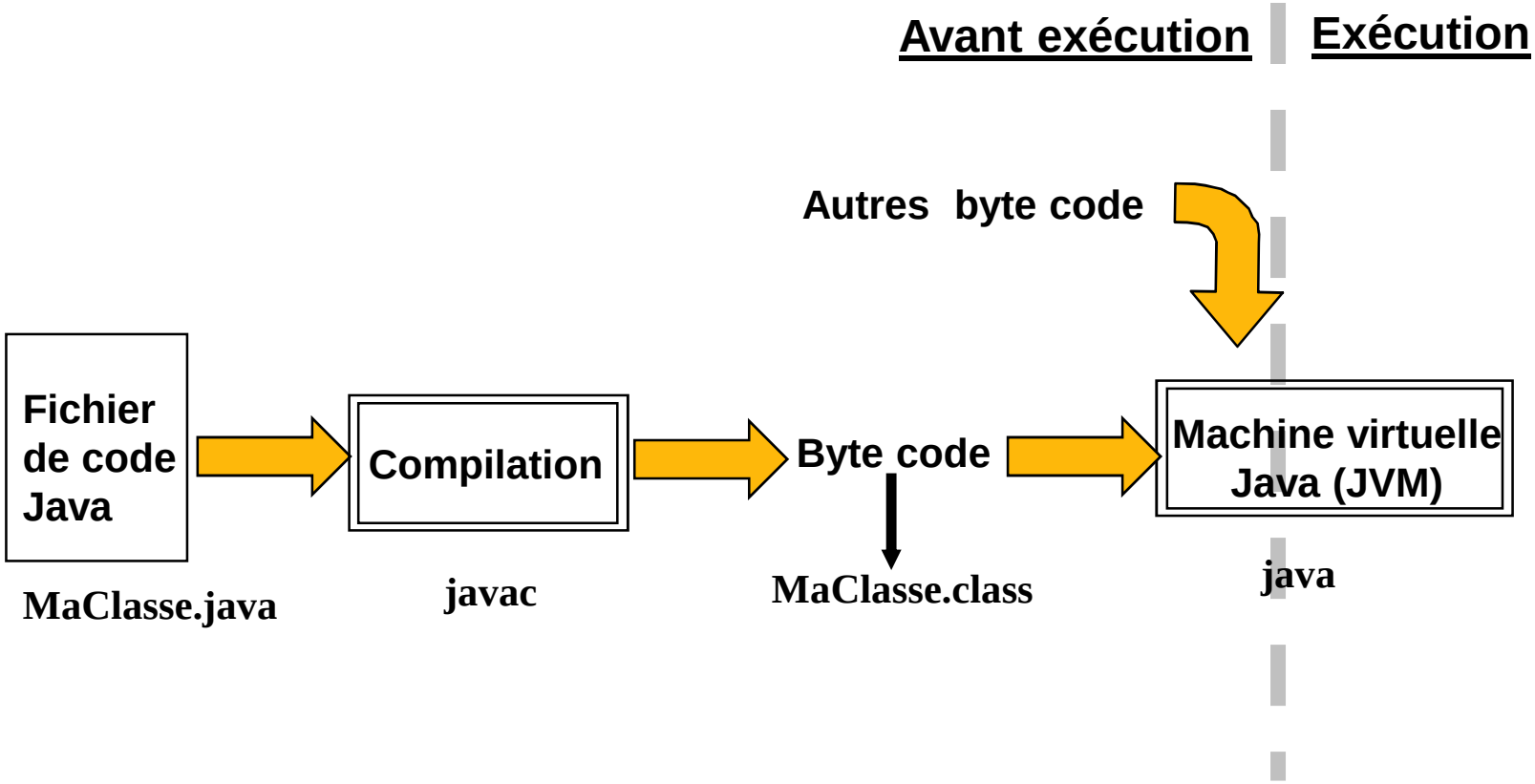
Java, un langage indépendant

- **Inconvénients :**
  - Nécessite **l'installation** d'un interpréteur pour pouvoir exécuter un programme Java
  - **L'interprétation** du code ralentit l'exécution
  - Les applications ne bénéficient que du **dénominateur commun** des différentes plate-formes
    - limitation, par exemple, des interfaces graphiques
  - Gestion **gourmande** de la mémoire

Langage compilé



Langage interprété



L'API de Java

- **Java fournit de nombreuses librairies de classes remplissant des fonctionnalités très diverses : c'est l'API Java**
  - API (Application and Programming Interface /Interface pour la programmation d'applications) : Ensemble de bibliothèques permettant une programmation plus aisée car les fonctions deviennent indépendantes du matériel.

- **Ces classes sont regroupées, par catégories, en paquetages (ou "packages").**

L'API de Java

- **Les principaux paquetages**
  - java.util : structures de données classiques
  - java.io : entrées / sorties
  - java.lang : chaînes de caractères, interaction avec l'OS, threads
  - java.applet : les applets sur le web
  - java.awt : interfaces graphiques, images et dessins
  - javax.swing : package récent proposant des composants « légers » pour la création d'interfaces graphiques
  - java.net : sockets, URL
  - java.rmi : Remote Method Invocation
  - java.sql : fournit le package JDBC

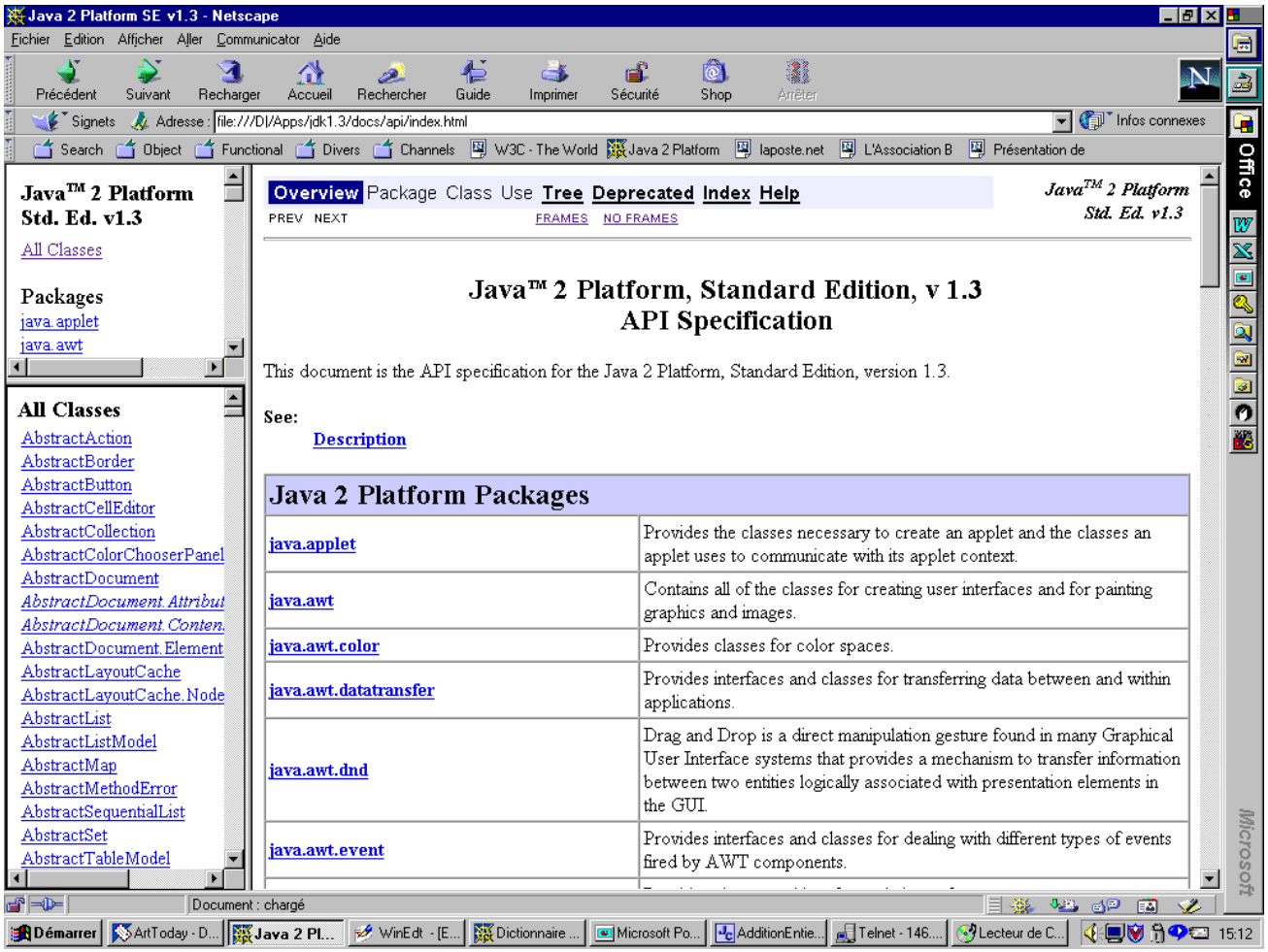
L'API de Java

- **Pour chaque classe, il y a une page HTML contenant :**
  - la hiérarchie d'héritage de la classe,
  - une description de la classe et son but général,
  - la liste des attributs de la classe (locaux et hérités),
  - la liste des constructeurs de la classe (locaux et hérités),
  - la liste des méthodes de la classe (locaux et hérités),
  - puis, chacune de ces trois dernières listes, avec la description détaillée de chaque élément.

L'API de Java

- **Les informations sur les classes de l'API se trouve :**
  - sous le répertoire jdk1.x/docs/api dans le JDK
    - les documentations de l'API se téléchargent et s'installent (en général) dans le répertoire dans lequel on installe java. Par exemple si vous avez installer Java dans le répertoire D:/Apps/jdk1.4/, vous décompresser le fichier zip contenant les documentations dans ce répertoire. Les docs de l'API se trouveront alors sous : D:/Apps/jdk1.4/docs/api/index.html
  - Sur le site de Sun, on peut la retrouver à http://java.sun.com/docs/index.html

L'API de Java


Java 2 Platform SE v1.3 - Netscape
Fichier Edition Afficher Aller Communicator Aide
Précédent Suivant Recharger Accueil Rechercher Guide Imprimer Sécurité Shop Arrêter
Signets Adresse : file:///D|/Apps/jdk1.3/docs/api/index.html
Search Object Functional Divers Channels W3C - The World Java 2 Platform laposte.net L'Association B Présentation de
Java™ 2 Platform Std. Ed. v1.3
All Classes
Packages
java.applet
java.awt
All Classes
AbstractAction
AbstractBorder
AbstractButton
AbstractCellEditor
AbstractCollection
AbstractColorChooserPanel
AbstractDocument
AbstractDocument.Attribut
AbstractDocument.Conten.
AbstractDocument.Element
AbstractLayoutCache
AbstractLayoutCache.Node
AbstractList
AbstractListModel
AbstractMap
AbstractMethodError
AbstractSequentialList
AbstractSet
AbstractTableModel
Overview Package Class Use Tree Deprecated Index Help
PREV NEXT FRAMES NO FRAMES
Java™ 2 Platform Std. Ed. v1.3
Java™ 2 Platform, Standard Edition, v 1.3
API Specification
This document is the API specification for the Java 2 Platform, Standard Edition, version 1.3.
See:
Description
Java 2 Platform Packages
java.applet
Provides the classes necessary to create an applet and the classes an applet uses to communicate with its applet context.
java.awt
Contains all of the classes for creating user interfaces and for painting graphics and images.
java.awt.color
Provides classes for color spaces.
java.awt.datatransfer
Provides interfaces and classes for transferring data between and within applications.
java.awt.dnd
Drag and Drop is a direct manipulation gesture found in many Graphical User Interface systems that provides a mechanism to transfer information between two entities logically associated with presentation elements in the GUI.
java.awt.event
Provides interfaces and classes for dealing with different types of events fired by AWT components.
Document : chargé
Démarrer ArtToday - D... Java 2 Pl... WinEdt - [E... Dictionnaire ... Microsoft Po... AdditionEntie... Telnet - 146.... Lecteur de C... 15:12
Office
Microsoft

Outil de développement : le JDK

- **JDK (Java Development Kit)** est environnement de développement fourni par Sun
- **Il contient :**
  - les classes de base de l'API java (plusieurs centaines),
  - la documentation au format HTML
  - le compilateur : **javac**
  - la JVM (machine virtuelle) : **java**
  - le visualiseur d'applets : **appletviewer**
  - le générateur de documentation : **javadoc**
  - etc.

Les commentaires

- /* commentaire sur une ou plusieurs lignes */
  - Identiques à ceux existant dans le langage C
- // commentaire de fin de ligne
  - Identiques à ceux existant en C++
- /** commentaire d'explication */
  - Les commentaires d'explication se placent généralement juste avant une déclaration (d'attribut ou de méthode)
  - Ils sont récupérés par l'utilitaire javadoc et inclus dans la documentation ainsi générée.

Instructions, blocs et blancs

- **Les instructions Java se terminent par un ;**
- **Les blocs sont délimités par :**

  { pour le début de bloc

  } pour la fin du bloc

  Un bloc permet de définir un regroupement d'instructions. La définition d'une classe ou d'une méthode se fait dans un bloc.

- **Les espaces, tabulations, sauts de ligne sont autorisés. Cela permet de présenter un code plus lisible.**

Entrée d'un programme Java

- Un **programme Java = un ensemble de classes**.
- Une méthode **main** doit être définie dans une des classes.
  - Elle doit avoir la signature:

    public static void main(String[] args)
  - Le tableau <<args>> contient des paramètres passés par la commande d'exécution d'application (java MyAppli p1 p2 p3 \\args[]= {"p1", "p2", "p3"})
- En général, la méthode main sert à:
  - **Configurer le programme**:(créer des objets initiaux du programme
  - **Déclencher l'interaction** entre ses objets

Exemple

Fichier Hello.java

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

Compilation et exécution

- Le **nom du fichier** est nécessairement celui de la **classe** avec l'extension .java, en plus Java est sensible à la casse des lettres.

- **Compilation en bytecode java dans une console DOS:**

  javac Hello.java

  Générer un fichier Hello.class

- **Exécution du programme dans une console DOS sur la JVM :** java Hello

  Afficher de « Hello world » dans la console

Compilation et exécution

- Le **nom du fichier** est nécessairement celui de la **classe** avec l'extension .java, en plus Java est sensible à la casse des lettres.

- **Compilation en bytecode java dans une console DOS:**

  javac Hello.java

  Générer un fichier Hello.class

- **Exécution du programme dans une console DOS sur la JVM :** java Hello

  Afficher de « Hello world » dans la console

Identificateurs

- **On a besoin de nommer les classes, les variables, les constantes, etc. ; on parle d'identificateur.**

- **Les identificateurs commencent par une lettre, _ ou $**

  Attention : Java distingue les majuscules des minuscules

- **Conventions sur les identificateurs :**
  - Si plusieurs mots sont accolés, mettre une majuscule à chacun des mots sauf le premier.
    - exemple : uneVariableEntiere
  - La première lettre est majuscule pour les classes et les interfaces
    - exemples : MaClasse, UneJolieFenetre

Identificateurs

- **Conventions sur les identificateurs :**
  - La première lettre est minuscule pour les méthodes, les attributs et les variables
    - exemples : setLongueur, i, uneFenetre
  - Les constantes sont entièrement en majuscules
    - exemple : LONGUEUR_MAX

Les mots réservés de Java

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

Les types de bases

- **En Java, tout est objet sauf les types de base.**
- **Il y a huit types de base :**
  - un type booléen pour représenter les variables ne pouvant prendre que 2 valeurs : **boolean** avec les valeurs associées **true** et **false**
  - un type pour représenter les caractères : **char**
  - quatre types pour représenter les entiers de divers taille : **byte**, **short**, **int** et **long**
  - deux types pour représenter les réelles : **float** et **double**
- **La taille nécessaire au stockage de ces types est indépendante de la machine.**
  - Avantage : portabilité
  - Inconvénient : "conversions" coûteuses

Les entiers

- **byte** : codé sur 8 bits, peuvent représenter des entiers allant de -2^7 à $2^7 - 1$ (-128 à +127)
- **short** : codé sur 16 bits, peuvent représenter des entiers allant de -2^{15} à $2^{15} - 1$
- **int** : codé sur 32 bits, peuvent représenter des entiers allant de -2^{31} à $2^{31} - 1$
- **long** : codé sur 64 bits, peuvent représenter des entiers allant de -2^{63} à $2^{63} - 1$
- **Notation**
  - Entier normal en base décimal (2, 13…)
  - Entier au format long en base décimal (2**L**, 13**l**…)
  - Entier en valeur octale : (**0**7, **0**56…)
  - Entier en valeur hexadécimale (**0x**7, **0X**56A…)

Les réels

- **float** : codé sur 32 bits, peuvent représenter des nombres allant de -10^{35} à $+10^{35}$
- **double** : codé sur 64 bits, peuvent représenter des nombres allant de -10^{400} à $+10^{400}$
- **Notation**
  - Réel double précision (4.55, 4.55**D,** 9.1**d…**)
  - Réel simple précision (4.55**f,** 0.46**F…)**

Les booléens

- **Variables logiques** contenant soit vrai (**true**) soit faux (**false**)
- **Notation**

  boolean x;

  x= **true;**

  x= **false;**

  x= **(5==5);** // l 'expression (5==5) est évaluée et la valeur est affectée à x qui vaut alors vrai

Les caractères

- **char** : contient une seule lettre
- le type char désigne des caractères en représentation **Unicode**
  - Codage sur **2 octets** contrairement à ASCII/ANSI codé sur un octet. Le codage ASCII/ANSI est un sous-ensemble d'Unicode
  - Notation **hexadécimale** des caractères Unicode de ' \u0000 ' à ' \uFFFF '.
- **Notation**

```
char a,b,c;
a='a';
b= '\u0022' //b contient le caractère guillemet : "
c=97; // x contient le caractère de rang 97 : 'a'
```

Exemple et remarque

```
int x = 0, y = 0;
float z = 3.1415F;
double w = 3.1415;
long t = 99L;
boolean test = true;
char c = 'a';
```

- **Remarque importante :**
  - Java exige que toutes les variables soient définies et initialisées. Le compilateur sait déterminer si une variable est susceptible d'être utilisée avant initialisation et produit une erreur de compilation.

Opérateurs

- Un **opérateur** est un symbole d'opération qui permet d'agir sur des variables ou de faire des "calculs"
- Une **opérande** est une entité (variable, constante ou expression) utilisée par un opérateur
- Une **expression** est une combinaison d'opérateur(s) et d'opérande(s), elle est évaluée durant l'exécution de l'algorithme, et possède une valeur (son interprétation) et un type
- **Exemple**
  - Dans l'expression **a + b**, a et b sont des opérandes et + l'opérateur
  - Dans l'expression **c= a* b** : c, a, b et a*b sont des opérandes et = et * sont des opérateurs
  - Si par exemple a et b sont des entiers, l'expression a + b, a*b et c sont aussi des entier

Opérateurs

Types opérateurs

- Un opérateur est **unaire** (non) ou **binaire** (+)
- Un opérateur est associé à ***un type*** et ne peut être utilisé qu'avec des données de *ce* type

Arithmétiques
Addition : + (ou concaténation)
Soustraction : -
Multiplication : *
Division : /
Division entière : DIV
Reste de DIV : MOD

Comparaisons
Inférieur : <
Inférieur ou égale : <=
Supérieur : >
Supérieur ou égale : >=
Différent : !=
Egale : =
Logiques
Conjonction : ET
Disjonction : OU
Disjonction exclusive : OUX
Négation : NON
Décalage à droit : >>
Décalage à gauche : <<

Opérateurs

- **les opérateurs d'incrémentation ++ et de décrémentation et - -**
  - ajoute ou retranche 1 à une variable
  - int n = 12; n ++; // n = 13 ; 12++ est illégale
  - n++; ⇔ n = n+1; et n--; ⇔ n = n-1;
  - Utilisation suffixée : ++n, --n, la incrémentation ou la décrémentation s'effectue en premier puis l'expression est évaluée

Les structures de contrôles

- **Les structures de contrôle classiques existent en Java :**
  - **if, else**
  - **switch, case, default, break**
  - **for**
  - **while**
  - **do, while**

Les structures if/ else

- **Effectuer une ou plusieurs instructions seulement si une certaine condition est vraie**

  if (*condition***)** *instruction*;

  et plus généralement : **if (***condition***)**

  { *bloc d'instructions*}

  condition *doit être un booléen ou renvoyer une valeur booléenne*

- **Effectuer une ou plusieurs instructions si une certaine condition est vérifiée sinon effectuer d'autres instructions**

  if (*condition***)** *instruction1* ; **else** *instruction2*;

  et plus généralement **if (***condition***)** { *1^er^ bloc d'instructions*}

  else {*2^ème^ bloc d'instruction*}

Les structures if / else

Max.java

```
import java.io.*;

public class Max
{
  public static void main(String args[])
  {
            Console console = System.console();
            int nb1 = Integer.parseInt(console.readLine("Entrer un entier:"));
            int nb2 = Integer.parseInt(console.readLine("Entrer un autre entier:"));
     if (nb1 > nb2)
       System.out.println("l'entier le plus grand est "+ nb1);
     else
       System.out.println("l'entier le plus grand est "+ nb2);
   }
}
```

Les structures while

- **Boucles indéterminées**
  - On veut répéter une ou plusieurs instructions un nombre indéterminés de fois : on répète l'instruction ou le bloc d'instruction tant que une certaine **condition reste vraie**
  - nous avons en Java une première boucle while (tant que)
    - **while** (*condition*) **{***bloc d'instructions***}**
    - les instructions dans le bloc sont répétées tant que la condition reste vraie.
    - On ne rentre jamais dans la boucle si la condition est fausse dès le départ

Les structures do … while

- **Boucles indéterminées**
  - un autre type de boucle avec le while:
    - **do** {*bloc d'instructions*} **while** (*condition*)
    - les instructions dans le bloc sont répétées **tant que la condition reste vraie.**
    - On rentre toujours **au moins une fois** dans la boucle : la condition est testée en fin de boucle.

Les structures avec while

Facto1.java

```
import java.io.*;

public class Facto1
{
  public static void main(String args[])
  {
     int n, result,i;
     n = Integer.parseInt(System.console().readLine("Entrer n:"));
     result = 1; i = n;
     while (i > 1)
      {
        result = result * i;
        i--;
      }
       System.out.println("la factorielle de "+n+" vaut "+result);
  }
}
```

Les structures for

- **Boucles déterminées**
  - On veut répéter une ou plusieurs instructions un nombre **déterminés** de fois : on répète l'instruction ou le bloc d'instructions pour un certain nombre de pas.
  - **La boucle for**

```
for (int i = 1; i <= 10; i++)
    System.out.println(i); //affichage des nombres de 1 à 10
```

  - une boucle for est en fait équivalente à une boucle while

for (*instruction1*; *expression1*; *expression2*) {*bloc*}

… est équivalent à …

instruction 1; **while** (*expression1*) {*bloc*; *expression2*}}

Les structures de contrôles for

Facto2.java

```
import java.io.*;

public class Facto2
{
  public static void main(String args[])
  {
     int n, result,i;
     n = Integer.parseInt(System.console().readLine("Entrer n:"));
     result = 1;
     for(i =n; i > 1; i--)
      {
         result = result * i;
      }
      System.out.println("la factorielle de "+n+" vaut "+result);
  }
}
```

Les structures switch

- **Sélection multiples**
  - l'utilisation de **if / else** peut s'avérer lourde quand on doit traiter plusieurs sélections et de multiples alternatives, pour cela existe en Java le **switch / case**

```
switch (expr) {
        case cas1 : ... ; break ;
        case cas2 : ... ; break ;
            ...
        default : ...    }
```

  - La valeur sur laquelle on teste doit être un **char** ou un **entier** (à l'exclusion d'un **long**).
  - L'exécution des instructions correspondant à une alternative commence au niveau du **case** correspondant et se termine à la rencontre d'une instruction **break** ou arrivée à la fin du **switch**

Les structures switch

Alternative.java

```
import java.io.*;

public class Alternative
{
  public static void main(String args[])
  {
    int nb = Integer.parseInt(System.console().readLine("Entrer n:"));
    switch(nb)
    {
      case 1:
        System.out.println("Un"); break;
      case 2:
        System.out.println("Deux"); break;
      default:
       System.out.println("Autre nombre");
     }
   }
  }
```

Les tableaux

- **Les tableaux permettent de stocker plusieurs valeurs de même type dans une variable.**
  - Les valeurs contenues dans la variable sont repérées par un indice
  - En langage java, les tableaux sont des objets
  - Comme en C/C++, les indices d'un tableau commencent à ' 0 '. Donc un tableau de taille 100 aura ses indices qui iront de 0 à 99.

Tableaux à une dimension

- **Déclaration :**

  type [] table

  type table[]

- **Accès**

  table[n]

  - Si n < 0 ou n >= taille, Java lance l'exception *ArrayIndexOutOfBoundsException*

- **Nombre d'éléments d'un tableau :**

  table.length

Tableaux à une dimension

- **Allocation:**
  - Allocation statique : (éléments du tableau connus)

    ```
    type table[] = {val0, val1, val2, val3, val4};
    ```

  - Allocation dynamique : (nombre d'éléments du tableau inconnu)

    ```
    table = new type[taille];
    ```

- **Copiage**
  - Allouer un nouveau tableau (variable tampon)
    - `type tampon[] = new type[6];`
  - Copier le contenu de l'ancien tableau dans le nouveau tableau
    - `System.arraycopy(table,0,tampon,0,table.length) ;`
  - Associer la mémoire du tampon à la variable originale
    - `table = tampon;`

Tableaux à plusieurs dimensions

- **Déclaration (2 dimensions) :**

  `type [] [] table , type [] table [] ou`

  `type table[][]`

- **Accès à la première dimension**
  - *Le n*e élément : `table[n]`
  - Le nombre d'éléments : `table.length`
- **Accès à la deuxième dimension**
  - Le *m*e élément : `table[n][m]`
  - Le nombre d'éléments de la ligne n : `table[n].length`
- **Généralisation pour plusieurs dimensions**

Tableaux à plusieurs dimensions

- **Allocation:**
  - Allocation statique : allouer le contenu de chaque dimension
    - `type table[][] = { { val00, val01, val02 } , { val10 },{ val20, val21 } } ;`
  - Allocation dynamique :
    - allouer l'espace pour ***toutes les dimensions*** d'un coup

      `type table[][] = new type[5][2];`
    - ou ***une dimension à la fois***

      ```
      type table[][];
      table = new type[5][];
      table[0] = new type[4];
      table[1] = new type[5];
      ```

Les tableaux :Exemple

Tab1.java

```
public class Tab1
{
  public static void main (String args[])
  {
    int tab[] ;
    tab = new int[4];
    tab[0]=5;
    tab[1]=3;
    tab[2]=7;
    tab[3]=tab[0]+tab[1];
  }
}
```

Pour déclarer une variable tableau on indique le *type* des éléments du tableau et le *nom de la variable tableau* suivi de *[ ]*

on utilise new <type> [taille]; pour initialiser le tableau

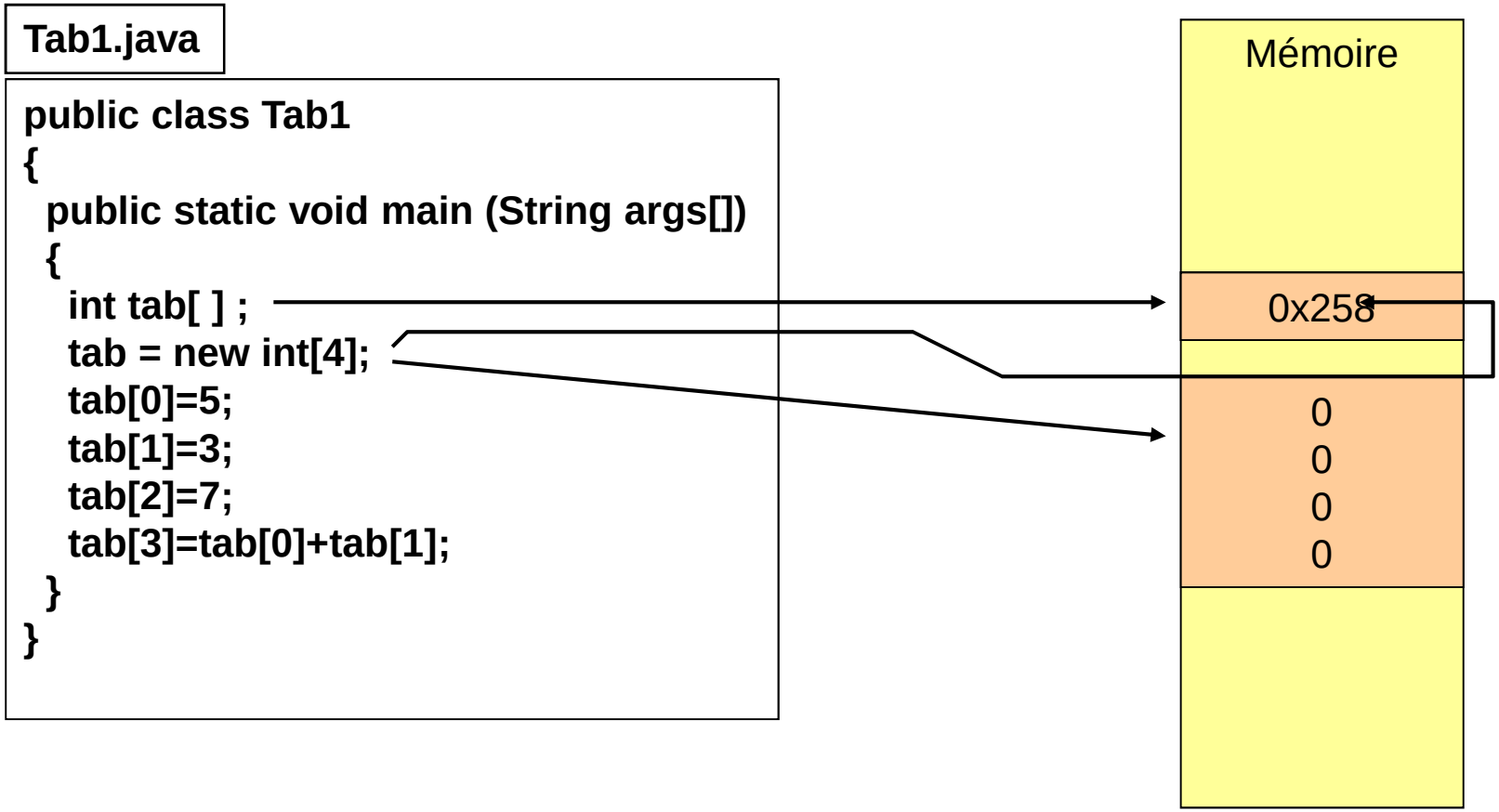
On peut ensuite affecter des valeurs au différentes cases du tableau : <nom_tableau>[indice]

Les indices vont toujours de 0 à (taille-1)

Les tableaux :Exemple

Tab1.java

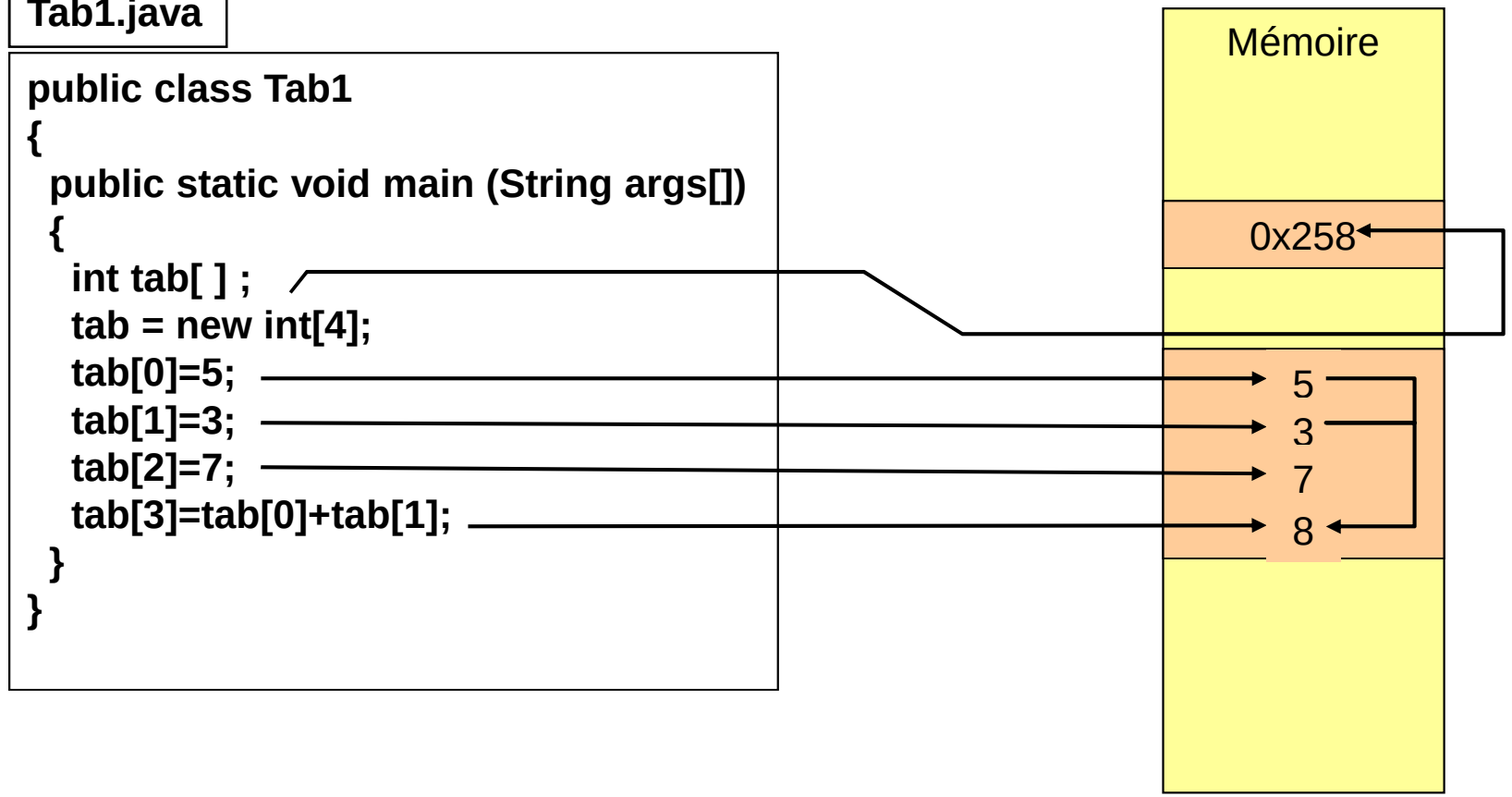
```
public class Tab1
{
  public static void main (String args[])
  {
    int tab[ ] ;
    tab = new int[4];
    tab[0]=5;
    tab[1]=3;
    tab[2]=7;
    tab[3]=tab[0]+tab[1];
  }
}
```



Les tableaux :Exemple

Tab1.java

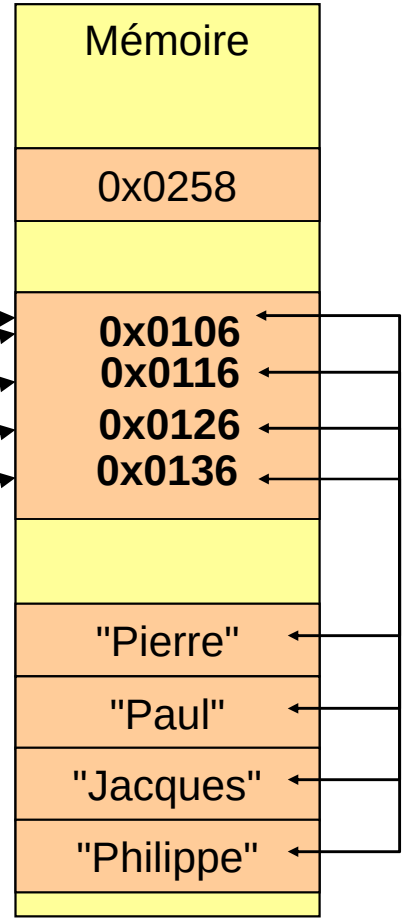
```
public class Tab1
{
  public static void main (String args[])
  {
    int tab[ ] ;
    tab = new int[4];
    tab[0]=5;
    tab[1]=3;
    tab[2]=7;
    tab[3]=tab[0]+tab[1];
  }
}
```



Les tableaux :Exemple

Tab2.java

```
public class Tab2
{
  public static void main (String args[])
  {
    String tab[ ] ;
    tab = new String[4];
    tab[0]=new String("Pierre");
    tab[1]=new String("Paul");
    tab[2]=new String("Jacques");
    tab[3]=new String("Philippe");
  }
}
```



La classe String

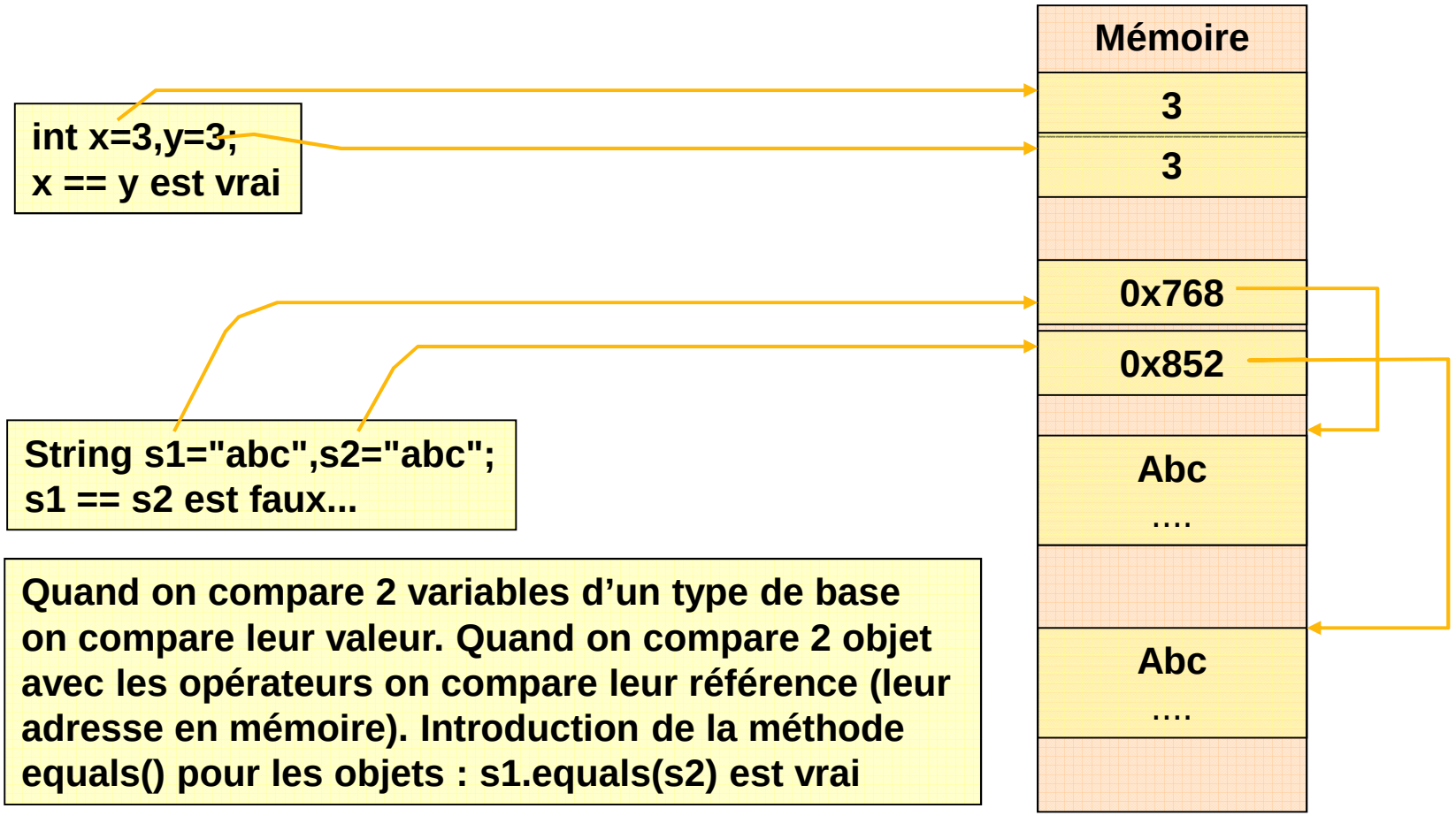
- Attention ce n'est pas un type de base. Il s'agit d'une classe défini dans l'API Java (Dans le package java.lang)

```
String s="aaa"; // s contient la chaîne "aaa" mais
String s=new String("aaa"); // identique à la ligne précédente
```

- **La concaténation**
  - l'opérateur **+** entre 2 String les concatène :

```
String str1 = "Bonjour ! ";
String str2 = null;
str2 = "Comment vas-tu ?";
String str3 = str1 + str2;   / * Concaténation de chaînes : str3 contient
" Bonjour ! Comment vas-tu ?*/
```

Différences entre objets et types de base



Quand on compare 2 variables d'un type de base on compare leur valeur. Quand on compare 2 objet avec les opérateurs on compare leur référence (leur adresse en mémoire). Introduction de la méthode equals() pour les objets : s1.equals(s2) est vrai

La classe String

- **Longueur d'un objet String :**
  - méthode int length() : renvoie la longueur de la chaîne

    String str1 = "bonjour";

    int n = str1.length(); // n vaut 7
- **Sous-chaînes**
  - méthode **String substring(int debut, int fin)**
  - extraction de la sous-chaine depuis la position debut jusqu'à la position fin non-comprise.

    String str2 = str1.substring(0,3); // str2 contient la valeur "bon"
  - le premier caractère d'une chaîne occupe la position 0
  - le deuxième paramètre de substring indique la position du premier caractère que l 'on ne souhaite pas copier

La classe String

- **Récupération d'un caractère dans une chaîne**
  - méthode **char charAt(int pos)** : renvoie le caractère situé à la position pos dans la chaîne de caractère à laquelle on envoie se message

```
String str1 = "bonjour";
char unJ = str1.charAt(3); // unJ contient le caractère 'j'
```

- **Modification des objets String**
  - Les String sont inaltérables en Java : on ne peut modifier individuellement les caractères d'une chaîne.
  - Par contre il est possible de modifier le contenu de la variable contenant la chaîne (la variable ne référence plus la même chaîne).

```
str1 = str1.substring(0,3) + " soir"; /* str1= "bonsoir" */
```

La classe String

- **Les chaînes de caractères sont des objets :**
  - pour tester si 2 chaînes sont égales il faut utiliser la méthode **boolean equals(String str)** et non **==**
  - pour tester si 2 chaînes sont égales à la casse près il faut utiliser la méthode **boolean equalsIgnoreCase(String str)**

```
String str1 = "BonJour";
String str2 = "bonjour"; String str3 =  "bonjour";
boolean a, b, c, d;
a = str1.equals("BonJour");
b = (str2 = = str3);
c = str1.equalsIgnoreCase(str2
d = "bonjour".equals(str2);
```

La classe String

- **Quelques autres méthodes utiles**
  - **boolean startsWith(String str)** : pour tester si une chaine de caractère commence par la chaine de caractère str
  - **boolean endsWith(String str)** : pour tester si une chaîne de caractère se termine par la chaine de caractère str

```
String str1 =   "bonjour ";
boolean a = str1.startsWith("bon");
boolean b = str1.endsWith("jour");
```

La classe Math

- **Les fonctions mathématiques les plus connues sont regroupées dans la classe Math qui appartient au package java.lang**
  - les fonctions **trigonométriques**
  - les fonctions **d'arrondi**, de valeur **absolue**, ...
  - la **racine carrée**, la **puissance**, **l'exponentiel**, le **logarithme**, etc.
- **Ce sont des méthodes de classe (static)**

  double calcul = **Math.sqrt** (**Math.pow**(5,2) + **Math.pow**(7,2));

  double **sqrt**(double x) : racine carrée de x

  double **pow**(double x, double y) : x puissance y

*La programmation par **objet** en java*

Concept d'objet

- Un **objet** est un élément visible ou tangible
- Il possède des **attributs** (données) et un modèle comportemental composée par des **méthodes** (traitements)
- Tout objet est une instance d'une classe
- Une application OO en exécution s'apparente à **une usine à objet communicants**
- **Exemples d'objets du monde réel**
  - **chien**
    - **état** : nom, couleur, race, poids....
    - **comportement** : manger,aboyer,renifler...

L'approche objet

- **Programmation dirigé par les données et non par les traitements**
  - Se concentre d'abord sur les **entités** à manipuler avant de se concentrer sur la **façon** de les manipuler

- **Notion d'encapsulation**
  - les données et les méthodes sont regroupés dans une même **entité** (la classe).

- **Cycle de vie d'un objet**
  - construction (en mémoire)
  - Utilisation (changements d'état par affectations, comportements par exécution de méthodes)
  - destruction

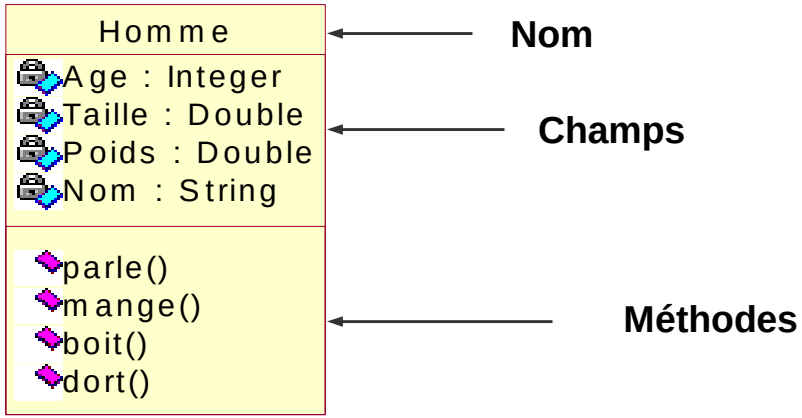
Concept de classe

- La **classification** d'un univers qu'on cherche à modéliser est sa distribution systématique en diverses catégories, d'après des critères précis
- En informatique, la **classe est un modèle** décrivant les caractéristiques communes et le comportement d'un ensemble d'objets
- Mais **l'état** de chaque objet est **indépendant** des autres
  - Les objets sont des représentations dynamiques (appelées **instances**) du modèle défini au travers de la classe
  - Une classe permet d'instancier plusieurs objets, et chaque objet est instance d'une seule classe

La classe : définition

- **Classe :** description d'une famille d'objets ayant une même structure et un même comportement. Elle est caractérisée par :
  - **Un nom** ou identificateur de classe
  - **Une composante statique** : des **champs** (ou **attributs**) nommés ayant une valeur. Ils caractérisent l'état des objets pendant l'exécution du programme
  - **Une composante dynamique** : des **méthodes** représentant le comportement des objets de cette classe. Elles **manipulent** les champs des objets et caractérisent les **actions** pouvant être effectuées par les objets.

Représentation graphique de classe



Une classe représentée avec la notation UML (Unified Modeling Language)

Syntaxe de définition d'une classe

```
class Nom_de_classe
{
        //déclaration des attributs
        //déclaration des méthodes
}       // Pas de point virgule
```

- **Le corps des méthodes** est écrit dans la classe

- **L'encapsulation de données**
  - Les membres **privés** ne sont utilisables que dans la classe
  - Les membres **publics** sont utilisables partout

- Toutes les classes de Java héritent de **java.lang.Object**. Il héritent ainsi des méthodes telles que getClass(), clone(), toString()

Syntaxe de définition d'une classe

- Exemple : Une classe définissant un point

```
Class Point
{
        double x; // abscisse du point
        double y; // ordonnée du point
        // translate de point de (dx,dy)
        void translate (double dx, double dy) {
                x = x+dx;
                y = y+dy;
        }
        // calcule la distance du point à l'origine
        double distance() {
                return Math.sqrt(x*x+y*y);
        }
}
```

Nom de la Classe

Attributs

Méthodes

L'instanciation

- **Instanciation : concrétisation d'une classe en un objet**
  - Dans nos programmes Java nous allons définir des classes et ***instancier*** ces classes en des objets qui vont interagir. Le fonctionnement du programme résultera de **l'interaction entre ces objets *instanciés***
- **Instance**
  - Représentant physique d'une classe
  - Son état est défini par les valeurs des variables de sa classe.
  - Son comportement est défini par les méthodes de sa classe
- **Exemple :**
  - si nous avons une classe voiture, alors votre voiture est une instance particulière de la classe voiture.
  - Classe = concept, description
  - Objet = représentant ***concret*** d'une classe

Les constructeurs

- **L'appel de new pour créer un nouvel objet déclenche, dans l'ordre :**
  - L'allocation mémoire nécessaire au stockage de ce nouvel objet et l'initialisation par défaut de ces attributs,
  - L'initialisation explicite des attributs, s'il y a lieu,
  - L'exécution d'un constructeur.
- **Un constructeur est une méthode d'initialisation.**

```
public class Application
{
  public static void main(String args[])
  {
    Personne jean = new Personne()
    jean.setNom("Jean") ;
  } }
```

Le constructeur est ici celui par défaut (pas de constructeur défini dans la classe Personne)

Les constructeurs

- **Le constructeur est une méthode :**
  - de même nom que la classe,
  - sans type de retour.
- Toute classe possède **au moins un constructeur**. Si le programmeur ne l'écrit pas, il en existe un par **défaut**, sans paramètres, de code vide.
- La destruction d'objets est faite par la machine virtuelle grâce au processus **Garbage Collector**

Les constructeurs

Personne.java

```
public class Personne
{
  public String nom;
  public String prenom;
  public int age;
  public Personne(String unNom, String unPrenom, int unAge)
  {
            nom=unNom;
            prenom=unPrenom;
             age = unAge;
  }
 }
```

Les constructeurs

- Pour une même classe, il peut y avoir **plusieurs constructeurs**, de signatures différentes (**surcharge**).
- L'appel de ces constructeurs est réalisé avec le **new** auquel on fait passer les paramètres.
  - **p1 = new Personne("Pierre", "Richard", 56);**
- **Déclenchement du "bon" constructeur**
  - Il se fait en fonction des **paramètres** passés lors de l'appel (nombre et types).
- **Attention**
  - Si le programmeur crée un constructeur (même si c'est un constructeur avec paramètres), le constructeur par défaut n'est plus disponible. Attention aux erreurs de compilation !

Les constructeurs

Personne.java

```
public class Personne
{
  public String nom;
  public String prenom;
  public int age;
  public Personne(String unNom,
                  String unPrenom,
                  int unAge)
  {
    nom=unNom;
    prenom=unPrenom;
    age = unAge;
  }
 }
```

Définition d'un Constructeur. Le constructeur par défaut (Personne())n'existe plus. Le code précédent occasionnera une erreur

```
public class Application
{
  public static void main(String
args[])
  {
    Personne jean = new Personne()
    jean.setNom("Jean") ;
  } }
```

Va donner une erreur à la compilation

Les constructeurs

Personne.java

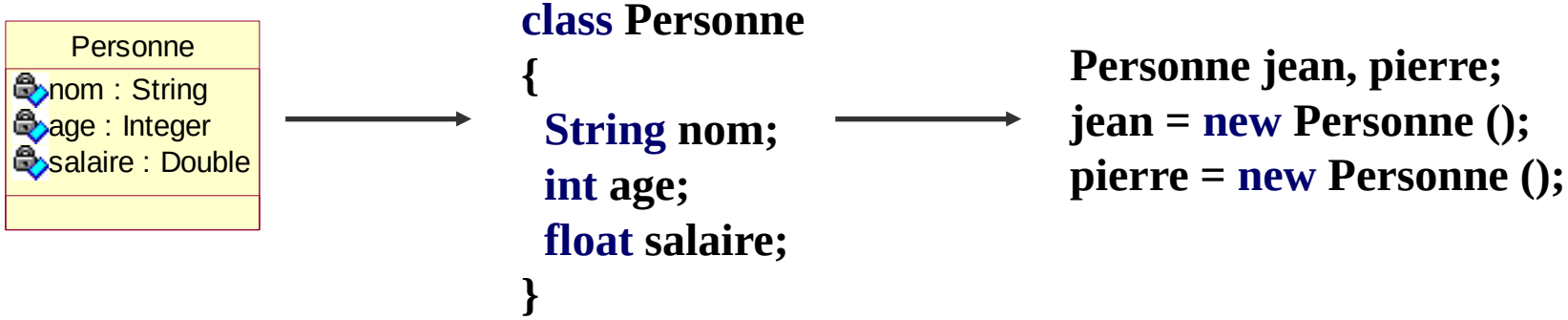
```
public class Personne
{
  public String nom;
  public String prenom;
  public int age;
  public void Personne()
   {
     nom=null; prenom=null;
     age = 0;
   }
  public String Personne(String unNom,
              String unPrenom, int unAge)
  {
    nom=unNom;
    prenom=unPrenom; age = unAge;
  } }
```

Redéfinition d'un Constructeur sans paramètres

On définit plusieurs constructeurs qui se différencient uniquement par leurs paramètres (on parle de leur signature)

Classe et objet en Java

Du modèle à ... **... la classe Java et de la classe à ...** **... des instances de cette classe**



L'opérateur d'instanciation en Java est new :

MaClasse monObjet = new MaClasse();

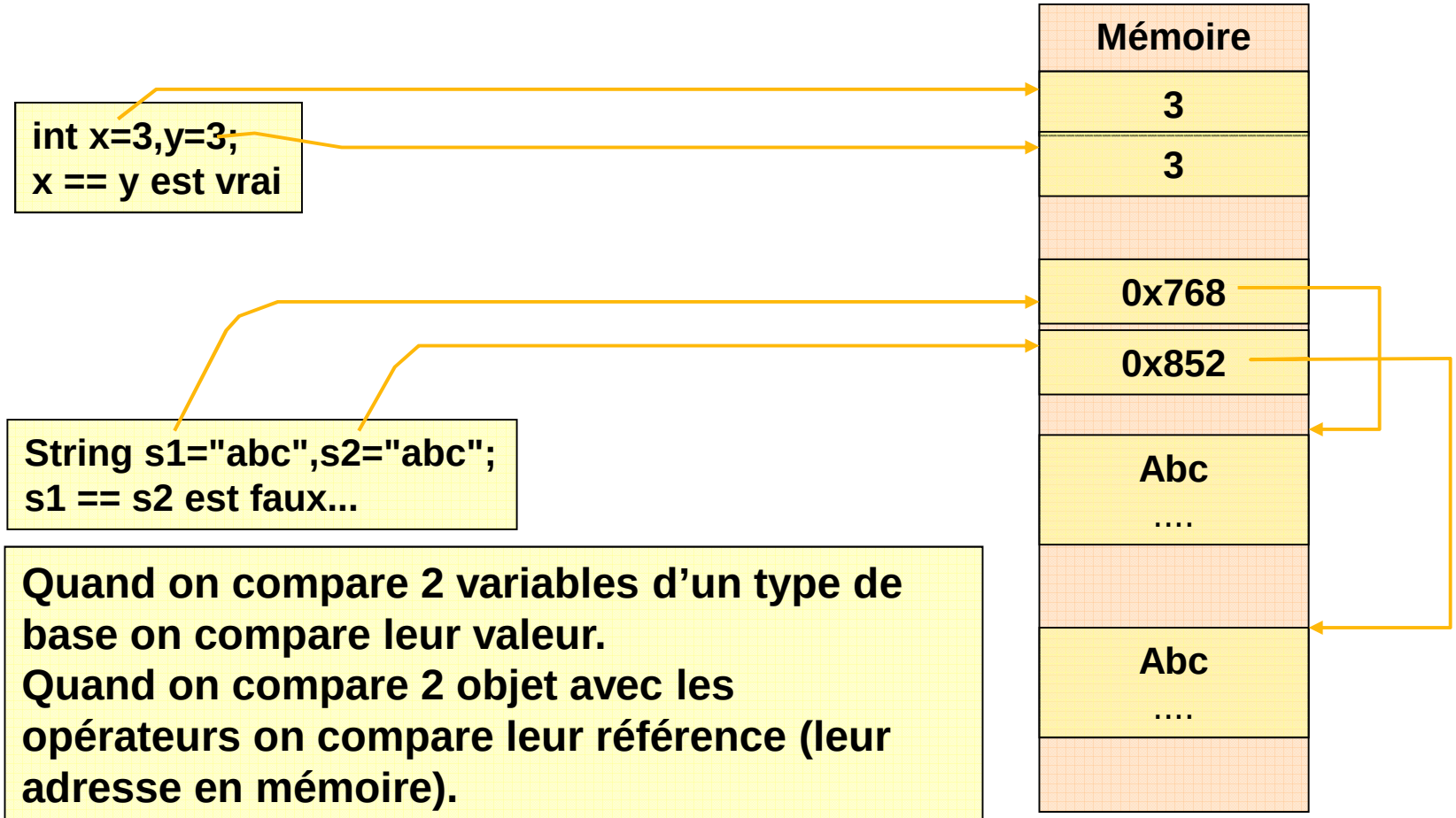
En fait, new va réserver l'espace mémoire nécessaire pour créer l'objet « monObjet » de la classe « MaClasse »

Le new ressemble beaucoup au malloc du C

Référence

- Lorsqu'une **variable** est d'un type **objet** ou **tableau**, ce n'est pas l'objet ou le tableau lui-même qui est stocké dans la variable mais une **référence** vers cet objet ou ce tableau *(on retrouve la notion d'adresse mémoire ou du pointeur en C)*.
- Lorsqu'une **variable** est d'un **type de base**, **la variable contient la valeur.**
- **La référence** est, en quelque sorte, un **pointeur** pour lequel le langage assure une manipulation transparente
- Cependant, c'est au programmeur de prévoir **l'allocation mémoire** nécessaire pour stocker effectivement l'objet (utilisation du new).

Différences entre objets et types de base



Quand on compare 2 variables d'un type de base on compare leur valeur.
Quand on compare 2 objet avec les opérateurs on compare leur référence (leur adresse en mémoire).
(méthode equals() pour les objets : s1.equals(s2) est vrai)

Accès aux attributs d'un objet

Personne.java

```
public class Personne
{
  public String nom;
  public String prenom;
  public int age;
  public void setNom(String unNom)
   {     nom = unNom;   }
  public String getNom()
  {    return (nom);  }
}
```

Application.java

```
public class Application
{
  public static void main(String args[])
  {
    Personne jean = new Personne()
    jean.nom = "Jean" ;
    jean.prenom = "Pierre" ;
  }
}
```

Remarque :

Contrairement aux variables, les attributs d'une classe, s'ils ne sont pas initialisés, se voient affecter automatiquement une **valeur par défaut**.

Cette valeur vaut : 0 pour les variables numériques, false pour les booléens, et null pour les références.

Accès aux méthodes d'un objet

Personne.java

```
public class Personne
{
  public String nom;
  public String prenom;
  public int age;
  public void setNom(String unNom)
   {
          nom = unNom;
   }
  public String getNom()
  {
          return (nom);
  }
}
```

Application.java

```
public class Application
{
  public static void main(String args[])
  {
          Personne jean = new Personne()
          jean.setNom("Jean") ;
  }
}
```

Méthodes et de paramètres

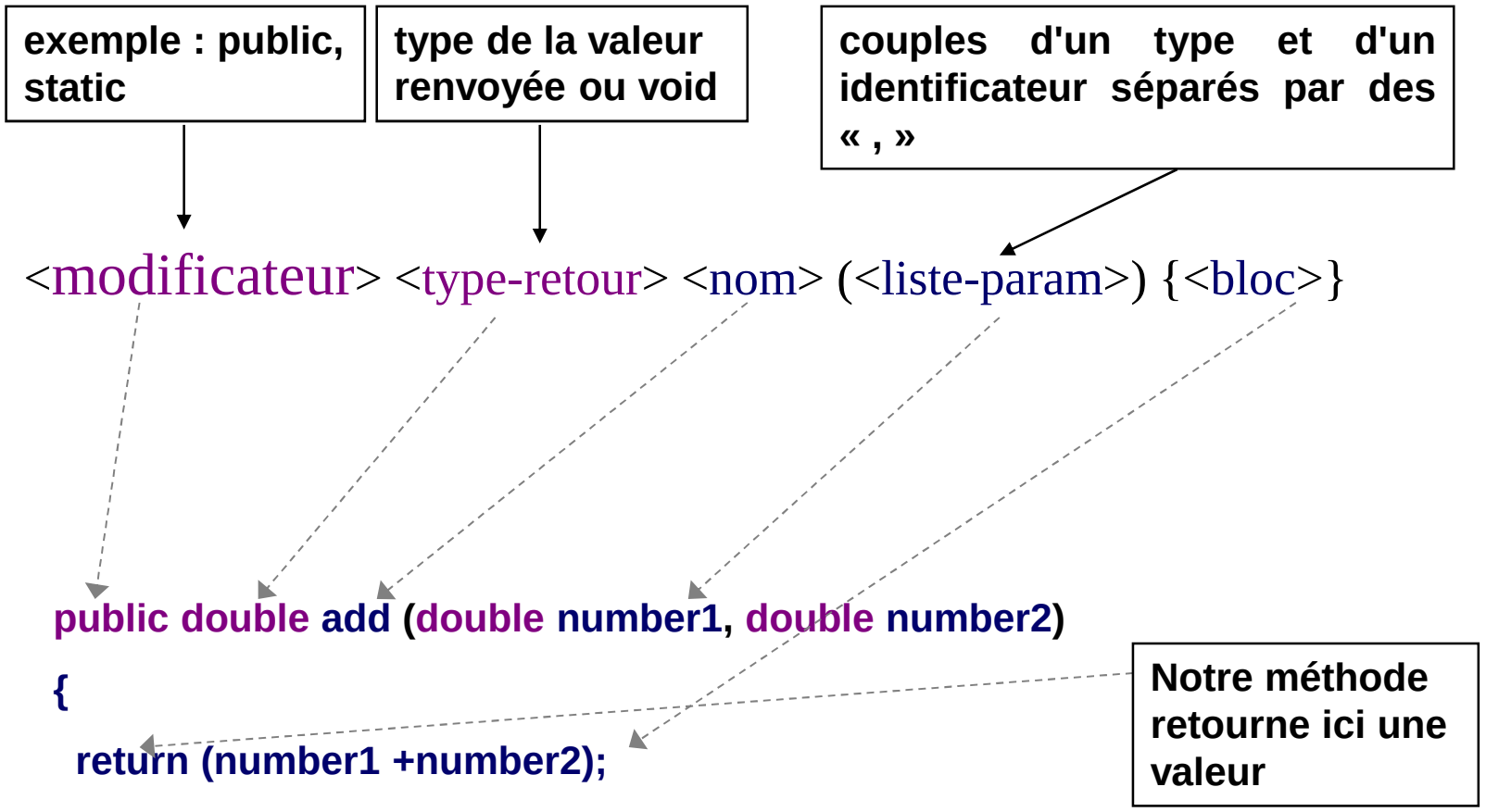
La notion de méthodes dans les langages objets

- Proches de la notion de **procédure** ou de **fonction** dans les langages procéduraux.
- La méthode c'est avant tout le regroupement d'un ensemble d'instructions suffisamment **générique** pour pouvoir être **réutilisées**
- Comme pour les procédures ou les fonctions (au sens mathématiques) on peut **passer des paramètres** aux méthodes et ces dernières peuvent **renvoyer des valeurs** (grâce au mot clé **return**).

Mode de passage des paramètres

- Java n'implémente qu'un seul mode de passage des paramètres à une méthode : **le passage par valeur**.
- **Conséquences :**
  - l'argument passé à une méthode ne peut être modifié,
  - si l'argument est une instance, c'est sa référence qui est passée par valeur. Ainsi, le contenu de l'objet peut être modifié, mais pas la référence elle-même.

Syntaxe des méthodes



Portée des variables

- Les variables sont **connues et ne sont connues qu'à l'intérieur** du bloc dans lequel elles sont déclarées
- En cas de **conflit** de nom entre des variables **locales** et des variables **globales**, c'est toujours la variable la plus locale qui est considérée comme étant la variable désignée par cette partie du programme

```
public class Bidule
{
  int i, j, k;
  public void truc(int z)
  {
     int j,r;
     r = z;
     i=k+r ;
  }
}
```

Destruction d'objets

- Java utilise **le ramasse-miettes** (ou Garbage Collector - GC en anglais) qui s'occupe de collecter les objets qui ne sont plus référencés. Il fonctionne en permanence dans un thread de faible priorité
- Le ramasse-miettes peut être "**désactivé**" en lançant l'interpréteur java avec l'option **-noasyncgc .**
- Inversement, le ramasse-miettes peut être **lancé** par une application avec l'appel **System.gc();**
- Il est possible d'indiquer ce qu'il faut faire juste avant de détruire un objet en utilisant la méthode **finalize()** de l'objet (pour fermer une base de données, fermer un fichier, couper une connexion réseau,…)

L’encapsulation

- les données et les procédures qui les manipulent sont regroupées dans une même **entité**, l'objet.
- Les détails d'implémentation sont **cachés**, le monde extérieur n'ayant accès aux données que par l'intermédiaire d'un ensemble d'opérations constituant l'interface de l'objet.
- Le programmeur n'a pas à se soucier de la représentation physique des entités utilisées et peut raisonner en termes **d'abstractions**.

Contrôle d'accès

- **Chaque attribut et chaque méthode d'une classe peut être :**
  - **Visible** depuis les instances de **toutes** les classes d'une application. En d'autres termes, son nom peut être utilisé dans l'écriture d'une méthode de ces classes. Il est alors **public** : **(public)**
  - **Visible uniquement** depuis les instances de sa classe. En d'autres termes, son nom peut être utilisé uniquement dans l'écriture d'une méthode de sa classe. Il est alors **privé** : **(private)**
  - Visible **aussi** depuis les instances de ses classes dérivés . Il est alors **protégé** : **(protected)**

Contrôle d'accès

- **En toute rigueur, il faudrait toujours que :**
  - les attributs ne soient pas visibles,
    - Les attributs ne devraient pouvoir être lus ou modifiés que par l'intermédiaire de méthodes prenant en charge les vérifications et effets de bord éventuels.
  - les méthodes "**utilitaires**" ne soient pas visibles,
  - seules les **fonctionnalités** de l'objet, destinées à être utilisées par d'autres objets soient visibles.

Contrôle d'accès

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

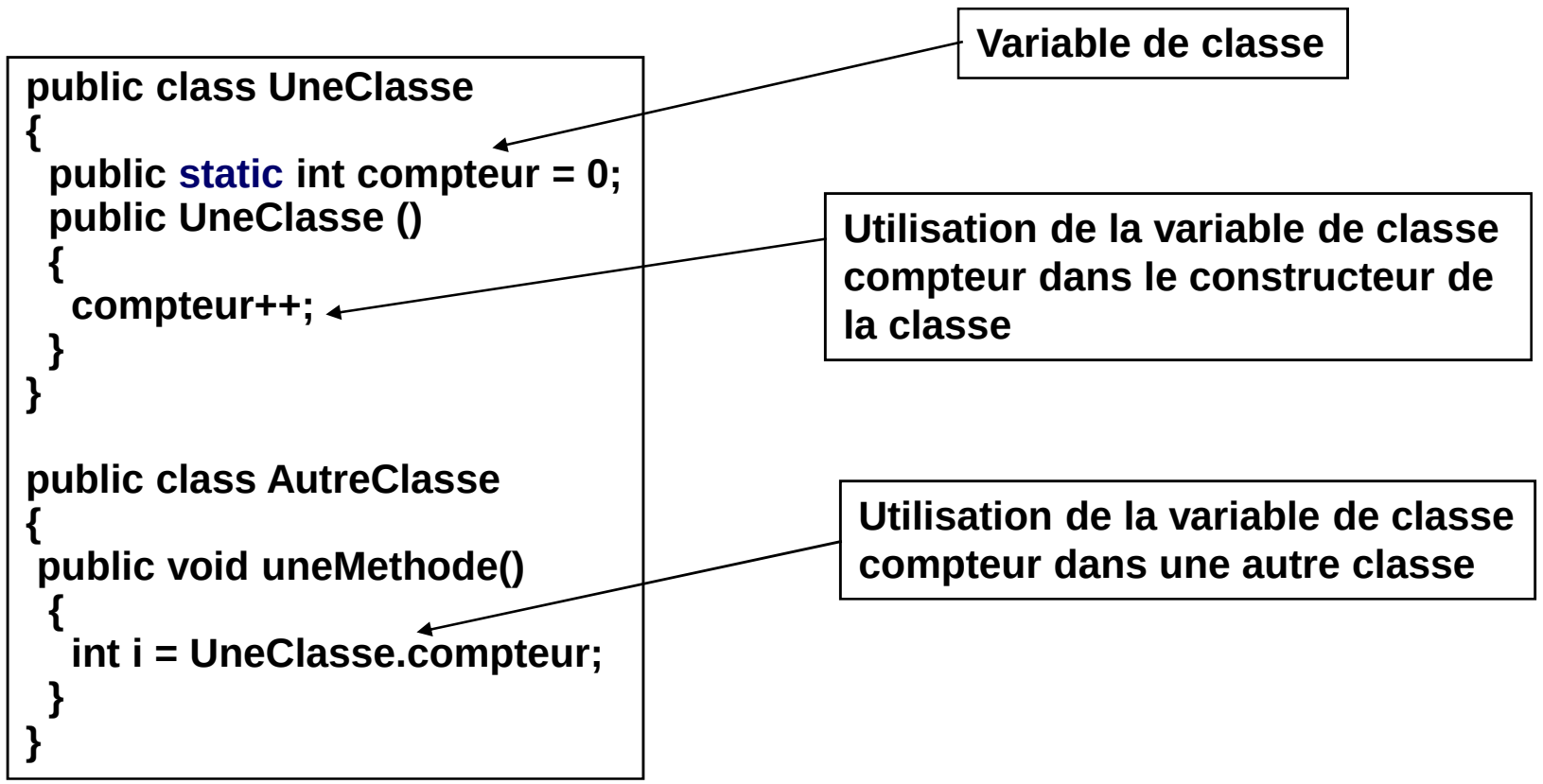
```
public class ProgPpal
   {
      public static void main(String args[])
      {
        Parallelogramme p1 = new Parallelogramme();
        p1.longueur = 5;       // Invalide car l'attribut est privé
        p1.profondeur = 4;     // OK
        p1.affiche( );         // OK
      } }
```

Variables de classe

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

Variables de classe

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

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



Méthodes de classe

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

Méthodes de classe

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

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

Autres exemples de méthodes de classe courantes

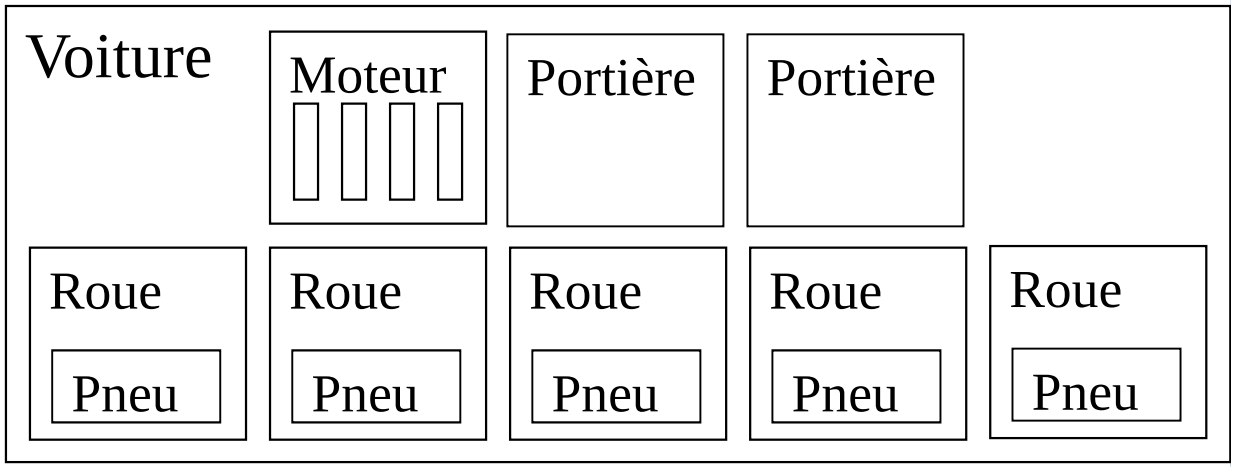
```
Math.sin(x);

String String.valueOf (i);
```

Composition d'objets

- **Un objet peut être composé à partir d'autres objets**

  Exemple : Une voiture composée de

  - 5 roues (roue de secours) chacune composée
  - d'un pneu
  - d'un moteur composé
    - de plusieurs cylindres
  - de portières
  - etc…



Chaque composant est un attribut de l'objet composé

Composition d'objets

Syntaxe de composition d'objets

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

 class  Roue {
      private float diametre;
      Pneu pneu ;}


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

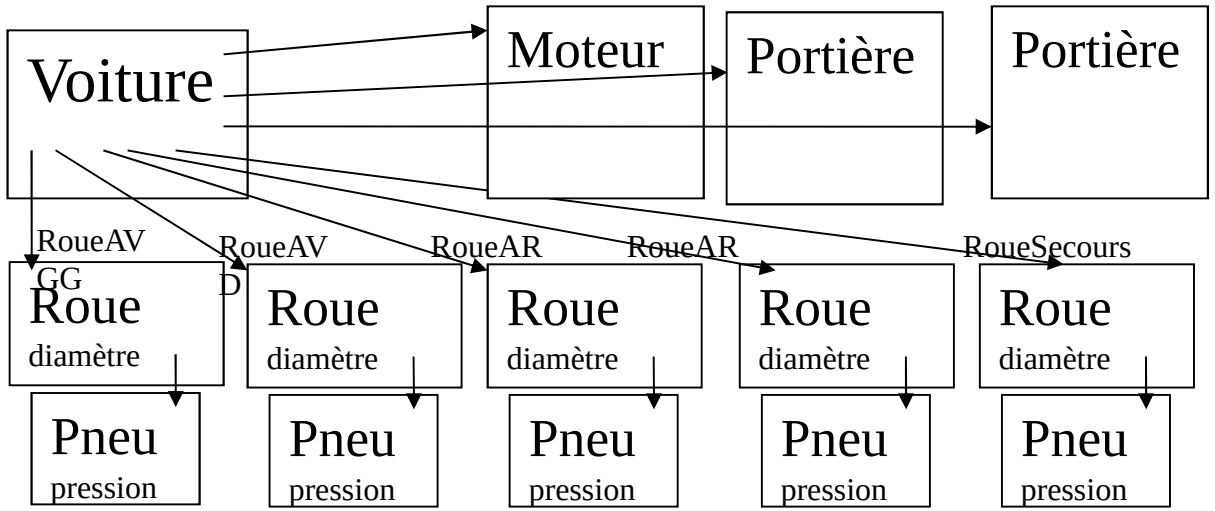
Composition d'objets

- **Généralement, le constructeur d'un objet composé doit appeler le constructeur de ses composants**

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

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

Composition d'objets

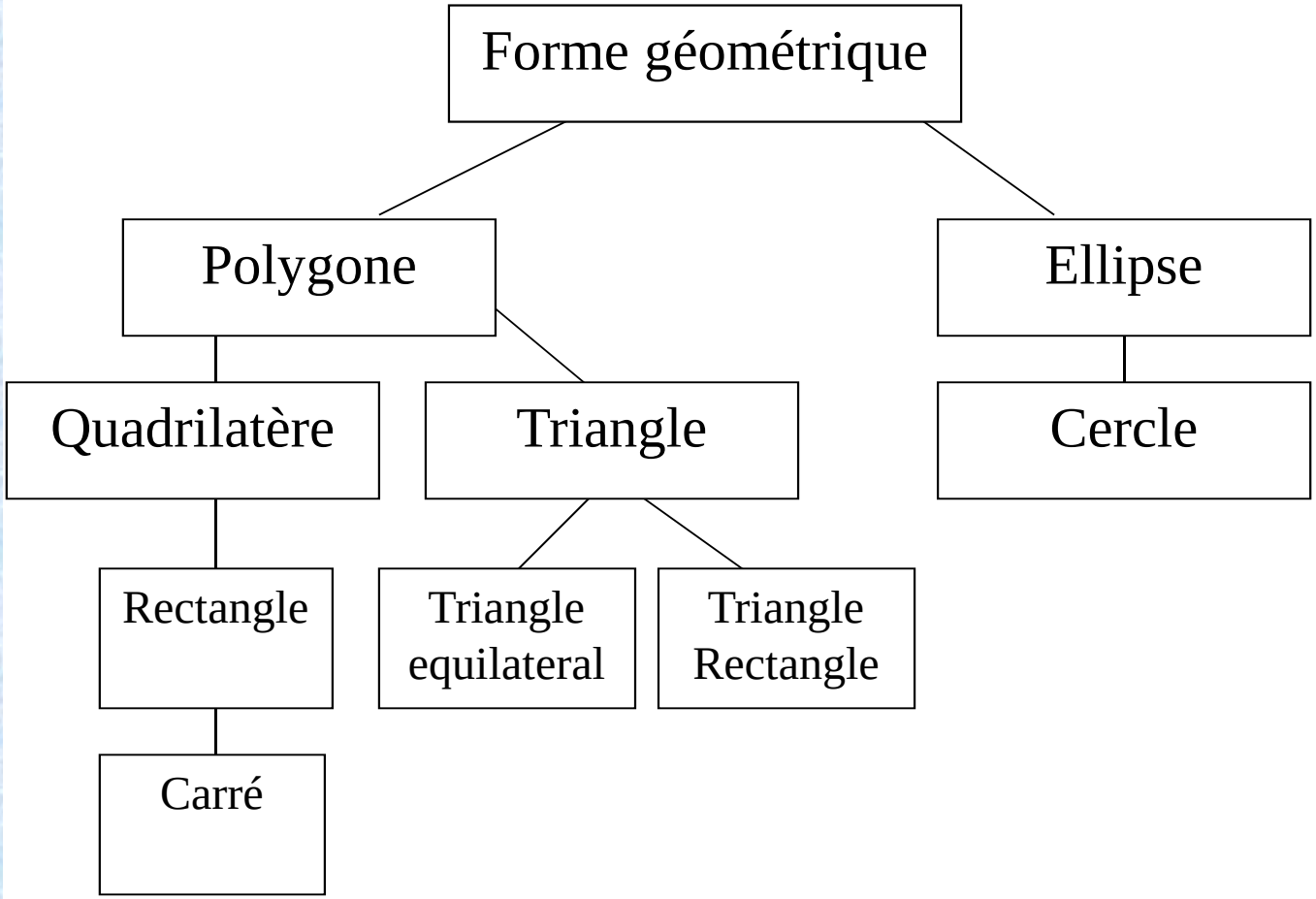
- **L'instanciation d'un objet composé instancie ainsi tous les objets qui le composent**



L'héritage : Concept

- La modélisation du monde réel nécessite une classification des objets qui le composent
- Classification = distribution systématique en catégories selon des critères précis
- Classification = hiérarchie de classes
- Exemples :
  - classification des éléments chimiques
  - classification des êtres vivants

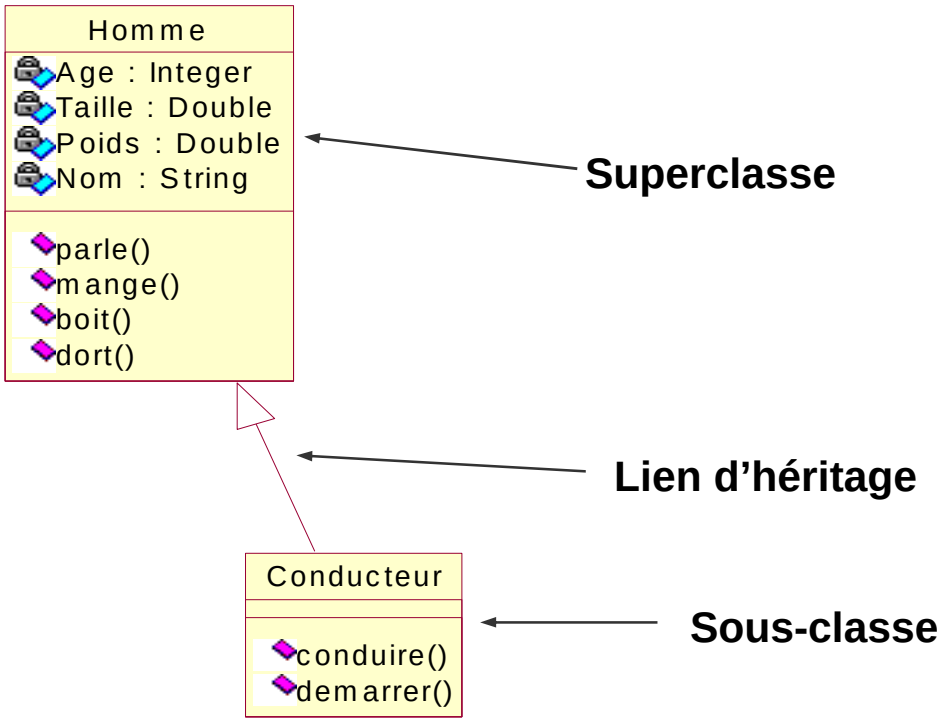
L'héritage (2) : exemple



L'héritage

- **Définition** : mécanisme permettant le partage et la réutilisation de propriétés entre les objets. La relation d'héritage est une relation de généralisation / spécialisation.
  - **La classe parente est la superclasse.**
  - **La classe qui hérite est la sous-classe.**
- **Représentation graphique (UML)**

L'héritage (3) : représentation graphique



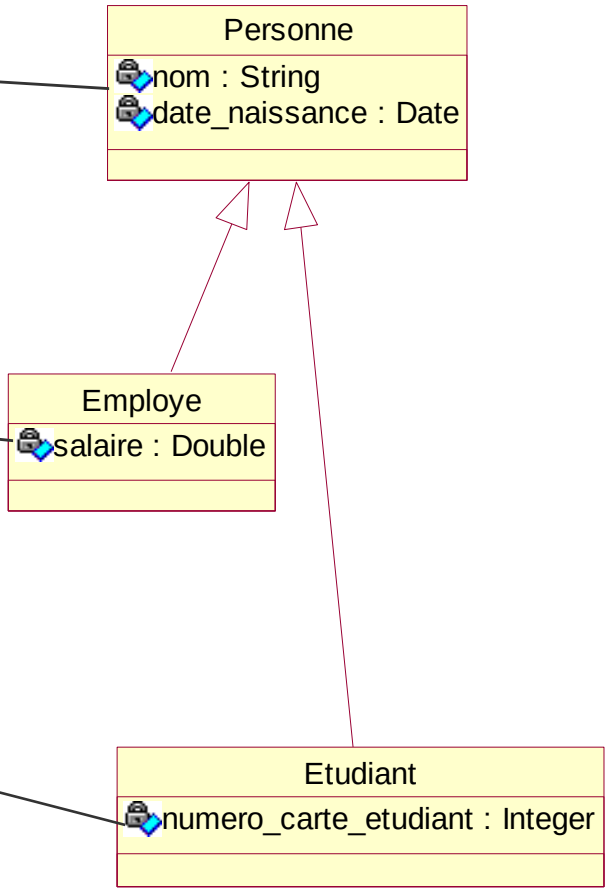
Représentation avec UML d'un héritage (simple)

L'héritage avec Java (1)

- Java implémente le mécanisme d'héritage simple qui permet de "factoriser" de l'information grâce à une relation de généralisation / spécialisation entre deux classes.
- Pour le programmeur, il s'agit d'indiquer, dans la sous-classe, le nom de la superclasse dont elle hérite.
- Par défaut toutes classes Java hérite de la classe *Object*
- L'héritage multiple n'existe pas en Java.
- Mot réservé d'héritage : ***extends***

L'héritage avec Java (2)

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

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

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



L'héritage en Java (3)

- **Constructeurs et héritage**
  - par défaut le constructeur d'une sous-classe appelle le constructeur "par défaut" (celui qui ne reçoit pas de paramètres) de la superclasse. Attention donc dans ce cas que le constructeur sans paramètre existe toujours dans la superclasse...
  - Pour forcer l'appel d'un constructeur précis, on utilisera le mot réservé **super.** Cet appel devra être la **première instruction** du constructeur.

L'héritage en Java (4)

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

Appel explicite à ce constructeur avec le mot clé super

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

L'héritage en Java (5)

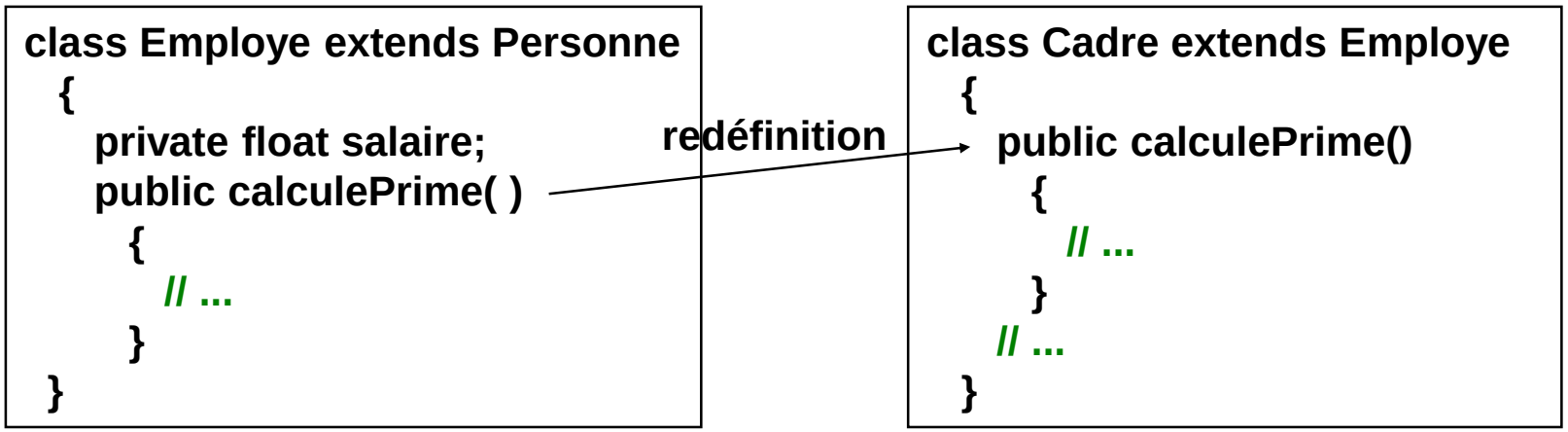
```
public class Personne
{
  public String nom, prenom;
  public int anNaissance;
  public Personne()
  {
    nom=""; prenom="";
  }
  public Personne(String nom,
                  String prenom,
                  int anNaissance)
  {
    this.nom=nom;
    this.prenom=prenom;
    this.anNaissance=anNaissance;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

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

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

Redéfinition de méthodes

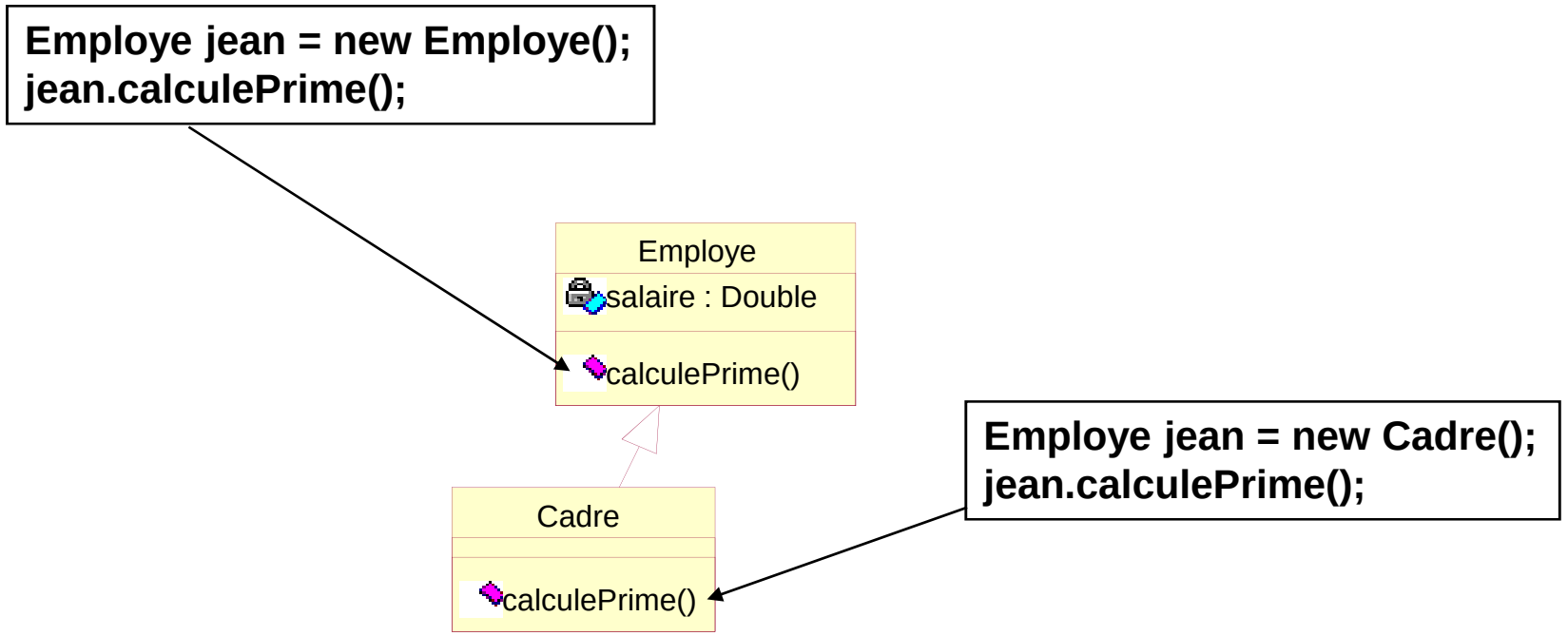
- Une sous-classe peut redéfinir des méthodes existant dans une de ses superclasses (directe ou indirectes), à des fins de spécialisation.
  - Le terme anglophone est "***overriding***". On parle aussi de masquage.
  - La méthode redéfinie doit avoir la même signature.



Recherche dynamique des méthodes

- **Le polymorphisme**
  - Capacité pour une entité de prendre plusieurs formes.
  - En Java, toute variable désignant un objet est potentiellement polymorphe, à cause de l'héritage.
  - Polymorphisme dit « d'héritage »
- **le mécanisme de "lookup" dynamique :**
  - déclenchement de la méthode la plus spécifique d'un objet, c'est-à-dire celle correspondant au type réel de l'objet, déterminé à l'exécution uniquement (et non le type de la référence, seul type connu à la compilation, qui peut être plus générique).
  - Cette dynamicité permet d'écrire du code plus générique.

Recherche dynamique des méthodes



Surcharge de méthodes

- Dans une même classe, plusieurs méthodes peuvent posséder le même nom, pourvu qu'elles diffèrent en nombre et/ou type de paramètres.
  - On parle de **surdéfinition** ou **surcharge**, on encore d'**overloading** en anglais.
  - Le choix de la méthode à utiliser est en fonction des paramètres passés à l'appel.
    - Ce choix est réalisé de façon statique (c'est-à-dire à la compilation).
  - Très souvent les constructeurs sont surchargés (plusieurs constructeurs prenant des paramètres différents et initialisant de manières différentes les objets)

Opérateur InstanceOf

- L'opérateur **instanceof** confère aux instances une capacité d'introspection : il permet de savoir si une instance est instance d'une classe donnée.
  - Renvoie une valeur booléenne

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

//...

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

Forçage de type / transtypage

- Lorsqu'une référence du type d'une classe désigne une instance d'une sous-classe, il est nécessaire de forcer le type de la référence pour accéder aux attributs spécifiques à la sous-classe.
- Si ce n'est pas fait, le compilateur ne peut déterminer le type réel de l'instance, ce qui provoque une erreur de compilation.
- On utilise également le terme de transtypage
- Similaire au « cast » en C

Forçage de type / transtypage

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

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

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

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

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

L'autoréférence : this

- Le mot réservé **this**, utilisé dans une méthode, désigne la référence de l'instance à laquelle le message a été envoyée (donc celle sur laquelle la méthode est « exécutée »).
- **Il est utilisé principalement :**
  - lorsqu'une référence à l'instance courante doit être passée en paramètre à une méthode,
  - pour lever une ambiguïté,
  - dans un constructeur, pour appeler un autre constructeur de la même classe.

L'autoréférence : this

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

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

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

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

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

Appelle le constructeur MaClasse(int a, int b)

Appelle le constructeur MaClasse(int c)

Référence à la superclasse

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

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

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

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

Classes abstraites

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

Classes abstraites

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

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

Récursivité

- Algorithme qui comporte
  - **Au moins un appel à lui même**
  - **au moins une condition d'arrêt**
- Exemple : factorielle
  - **n! = n * (n-1) !**
  - **0!=1**

Écriture d'un algorithme récursif

```
public static long factorielle(int n){
  if (n<=0) return 1;
  else return n*factorielle(n-1);}
```

CONDITION D'ARRET

APPEL RECURSIF

```
factorielle(4)= 4*factorielle(3)          4*6
  factorielle(3)= 3*factorielle(2)        3*2
    factorielle(2)= 2*factorielle(1)      2*1
      factorielle(1)= 1*factorielle(0)    1*1
```

Empilage du contexte local

- Chaque procédure stocke dans une zone mémoire
  - **les paramètres**
  - **les variables locales**
- Cette zone s'appelle **la pile** car les données sont
  - **empilées lors de l 'appel d'une procédure**
  - **désempilées à la fin de la procédure**
- A chaque appel récursif, l 'ordinateur empile donc
  - **les variables locales**
  - **les paramètres qui doivent changer à chaque appel**

Transformer une boucle en une procédure récursive

- **Procédure itérative :**

```
static void compter() {
for (i=1;i<10;i++) System.out.println(i);}
```

- **Procédure récursive équivalente**

```
static void compter_rec(int i) {
    System.out.println(i)
    if(i<10) compter_rec(i+1);}
```

Transformer deux boucles imbriquées en une procédure récursive

- **Procedure itérative :**

```
static void compter() {
    for (a=1;a<10;a++)
          for (b=1;b<5;b++)
                System.out.println(a*b);}
```

- **1ère procédure récursive:**

```
static void compter_rec(int a) {
    if(a<10) {
          for (b=1;b<5;b++)
                System.out.println(a*b);
                compter_rec(a+1);}  }
```

- **2ème procédure récursive:**

```
static void compter_rec(int a,int b) {
    if(a<10) if (b<5){
                System.out.println(a*b);
                compter_rec(a,b+1);}
             else compter_rec(a+1,1);}
```

Exemple d ’algorithme récursif

- **Inverser une chaîne de caractères**

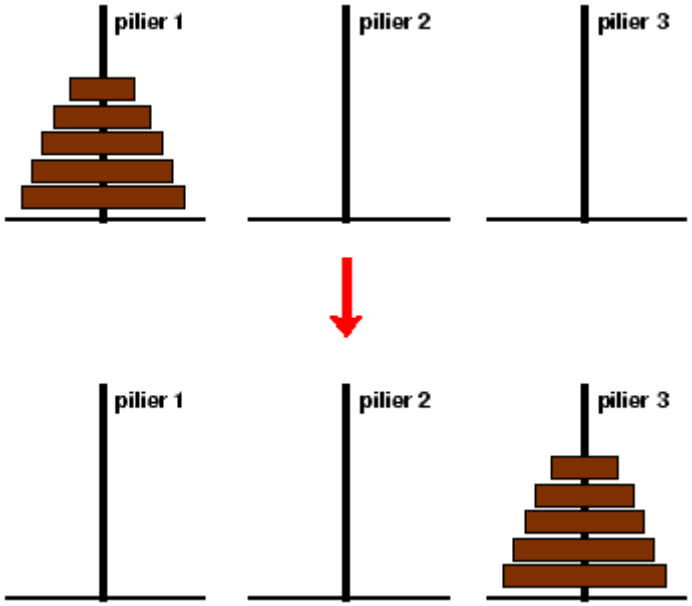
```
public static String inverse(String s) {
int l= s.length();
if (l<=1) return s;
else return inverse(s.substring(1,l))+ s.charAt(0);}
```

CONDITION D’ARRET

APPEL RECURSIF

```
inverse(“abcd”)-> inverse(“bcd”)+”a”
    inverse(“bcd”)-> inverse(“cd”)+”b”
          inverse(“cd”)-> inverse(“d”)+”c”
                dc
          dcb
    dcba
```

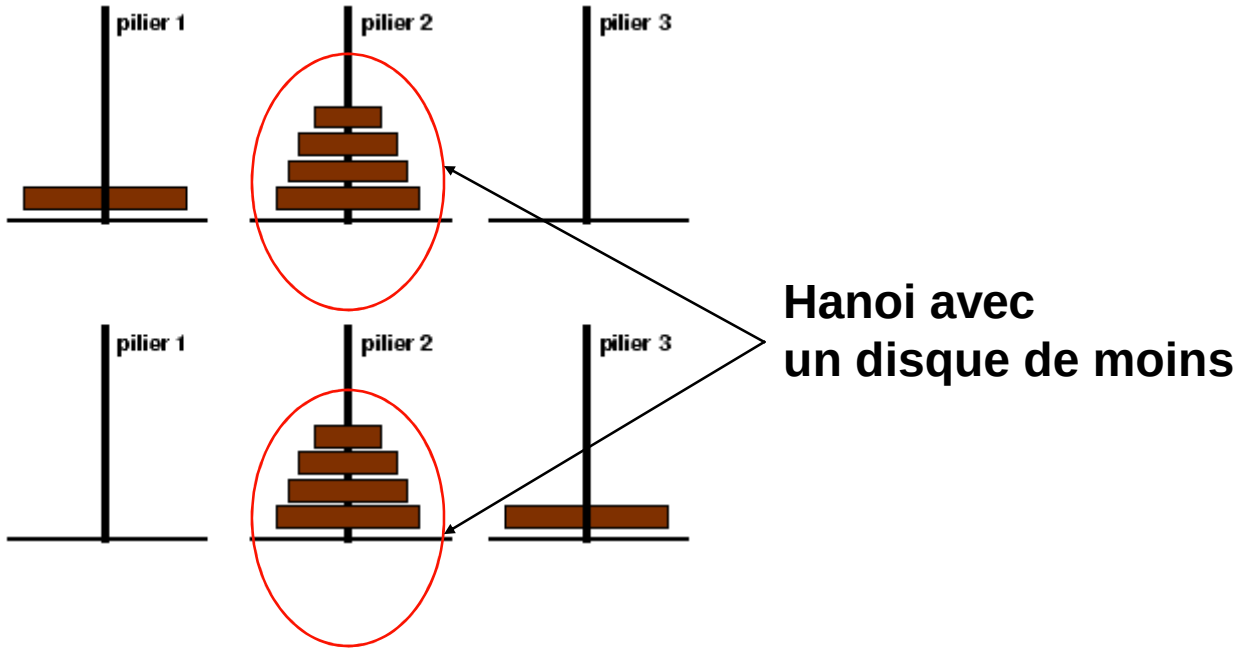
Tours de Hanoi



- déplacer les disques d'un pilier à un autre
- un disque d'un certain diamètre ne peut pas être placé au dessus d'un disque de diamètre inférieur.

Tours de Hanoi

si on a "n" disques à déplacer :
- on déplace "n-1" disques vers le pilier intermédiaire
- on déplace le disque "n" du pilier initial vers le pilier final
- on déplace les "n-1" disques du pilier intermédiaire vers le pilier final



Efficacité de la récursivité ?

- **La récursivité est légèrement moins rapide qu ’un algorithme itératif équivalent**
  - Temps nécessaire à l ’empilage et au dépilage des données
- **La récursivité utilise plus de ressources mémoire**
  - pour empiler les contextes
- **La récursivité est plus « élégante »**
- **Les algorithmes récursifs sont souvent plus faciles à écrire**

les exceptions

(Traitement des erreurs en Java)

Prévoir les erreurs d'utilisation

- **Certains cas d'erreurs peuvent être prévus à l'avance par le programmeur.**
  Exemples:
  - erreurs d'entrée-sortie (I/O fichiers)
  - erreurs de saisie de données par l'utilisateur

- **Le programmeur peut :**
  - «Laisser planter» le programme à l'endroit où l'erreur est détectée
  - Manifester explicitement le problème à la couche supérieure
  - Tenter une correction

Notion d'exception

En Java, les erreurs se produisent lors d'une exécution sous la forme d'exceptions.

Une exception :

- est un objet, instance d'une classe d'exception
- provoque la sortie d'une méthode
- correspond à un type d'erreur
- contient des informations sur cette erreur

Les exceptions

- **Exception**

situation particulière imposant une rupture dans le cours d'un programme : erreur, impossibilité…

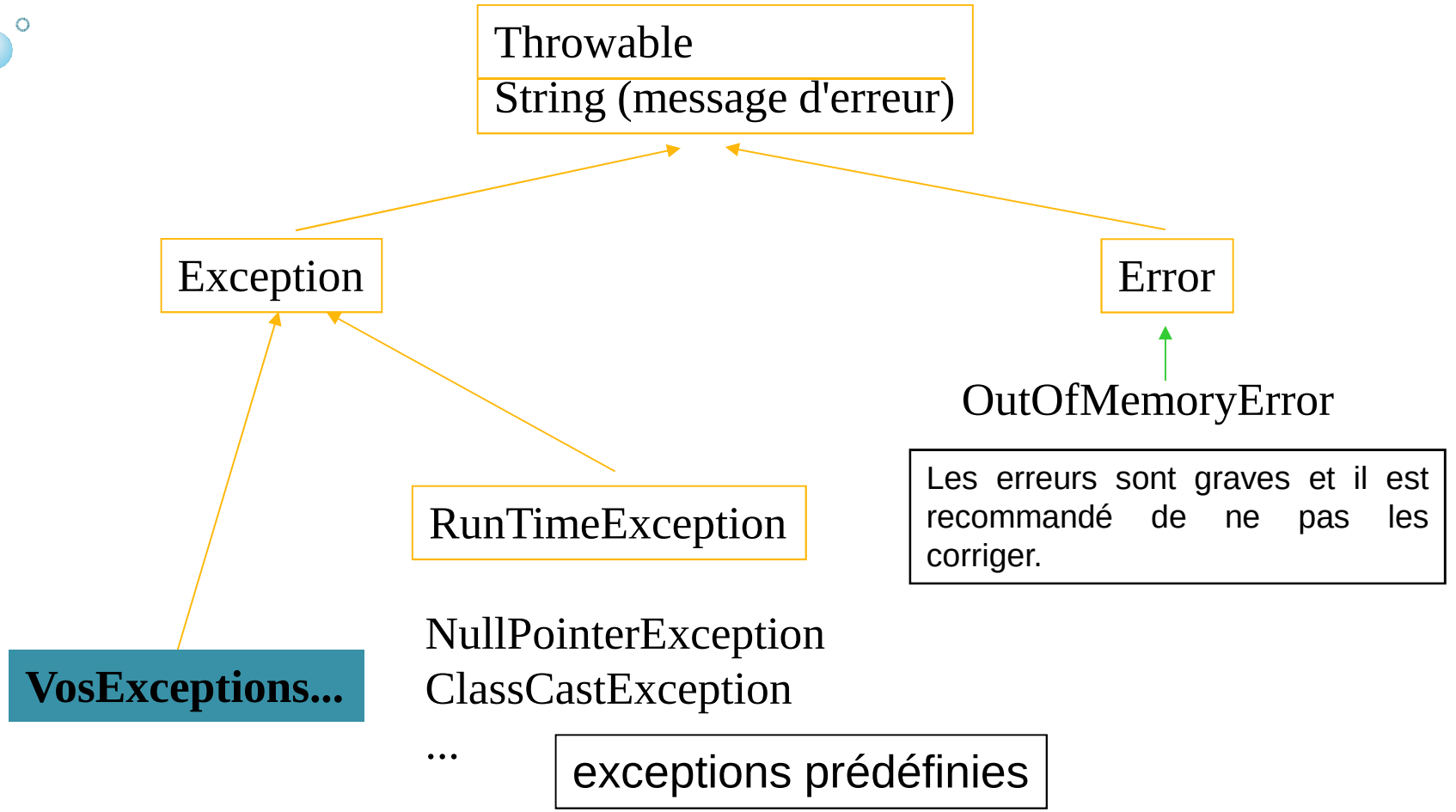
- Un objet JAVA Exception est une « bulle » logicielle produite dans cette situation qui va remonter la pile d'exécution pour trouver une portion de code apte à la traiter

- Si cette portion n'existe pas, le programme s'arrête en affichant la pile d'exécution. Sinon, la portion de code sert à pallier au problème (poursuite éventuelle, ou sortie)

Terminologie

- Une exception est un signal qui indique que quelque chose d'exceptionnel est survenu en cours d'exécution.
- **Deux solutions alors :**
  - laisser le programme se terminer avec une erreur,
  - essayer, malgré l'exception, de continuer l'exécution normale.
- Lever une exception consiste à signaler quelque chose d'exceptionnel.
- Capturer l'exception consiste à essayer de la traiter.

Arbre des exceptions



Nature des exceptions

- En Java, les exceptions sont des objets ayant 3 caractéristiques:
  - Un type d'exception (défini par la classe de l'objet exception)
  - Une chaîne de caractères (option), (hérité de la classe Throwable).
  - Un « instantané » de la pile d'exécution au moment de la création.
- Les exceptions construites par l'utilisateur étendent la classe *Exception*
- *RunTimeException, Error* sont des exceptions et des erreurs prédéfinies et/ou gérées par Java

Quelques exceptions prédéfinies

- Division par zéro pour les entiers **: ArithmeticException**
- Référence nulle **: NullPointerException**
- Tentative de forçage de type illégale : **ClassCastException**
- Tentative de création d'un tableau de taille négative : **NegativeArraySizeException**
- Dépassement de limite d'un tableau : **ArrayIndexOutOfBoundsException**

Capture d'une exception

- Les sections **`try`** et **`catch`** servent à capturer une exception dans une méthode (attraper la bulle…)
- **exemple :**

```
public void XXX(……) {
        try{ ……..}
        catch {
         ……
         ……
        }
}
```

try / catch / finally

```
try
  {
    ...
  }
catch (<une-exception>)
  {
    ...
  }
catch (<une_autre_exception>)
  {
    ...
  }

...

finally
  {
    ...
  }
```

- Autant de blocs catch que l'on veut.
- Bloc finally facultatif.

Traitement des exceptions

- Le bloc **try** est exécuté jusqu'à ce qu'il se termine avec succès ou bien qu'une exception soit levée.
- Dans ce dernier cas, les clauses **catch** sont examinées l'une après l'autre dans le but d'en trouver une qui traite cette classe d'exceptions (ou une superclasse).
- Les clauses **catch** doivent donc traiter les exceptions de la plus spécifique à la plus générale.
- Si une clause **catch** convenant à cette exception a été trouvée et le bloc exécuté, l'exécution du programme reprend son cours.

Traitement des exceptions

- Si elles ne sont pas immédiatement capturées par un bloc catch, les exceptions se propagent en remontant la pile d'appels des méthodes, jusqu'à être traitées.
- Si une exception n'est jamais capturée, elle se propage jusqu'à la méthode **main(),** ce qui pousse l'interpréteur Java à afficher un message d'erreur et à s'arrêter.
- L'interpréteur Java affiche un message identifiant :
  - l'exception,
  - la méthode qui l'a causée,
  - la ligne correspondante dans le fichier.

Bloc finally

- Un bloc **finally** permet au programmeur de définir un ensemble d'instructions qui est toujours exécuté, que l'exception soit levée ou non, capturée ou non.

- La seule instruction qui peut faire qu'un bloc finally ne soit pas exécuté est System.exit().

Interception vs propagation

Si une méthode peut émettre une exception

(ou appelle une autre méthode qui peut en émettre une) il faut :

- soit **propager** l'exception (la méthode doit l'avoir déclarée);
- soit **intercepter** et traiter l'exception.

Exemple

Exemple de propagation

```
public int ajouter(int a, String str) throws
    numberFormatException{
        int b = Integer.parseInt(str);
        a = a + b;
        return a; }
```

Exemple d'interception

```
public int ajouter(int a, String str) {
    try { int b = Integer.parseInt(str);
        a = a + b;          }
    catch (NumberFormatException e) {
        System.out.println(e.getMessage());
    }
    return a;}
```

Exemple d'interception

```
public int ajouter(int a, String str) {
    try {
        int b = Integer.parseInt(str);
        a = a + b;
    } catch (NumberFormatException e) {
        System.out.println(e.getMessage());
    }
    return a;
}
```

Les objets `Exception`

- La classe **Exception** hérite de La classe **Throwable.**
- La classe **Throwable** définit un message de type String qui est hérité par toutes les classes d'exception.
- Ce champ est utilisé pour stocker le message décrivant l'exception.
- Il est positionné en passant un argument au constructeur.
- Ce message peut être récupéré par la méthode **`getMessage().`**

Exemple

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

Levée d'exceptions

- Le programmeur peut lever ses propres exceptions à l'aide du mot réservé **throw**.
- **throw** prend en paramètre un objet instance de **Throwable** ou d'une de ses sous-classes.
- Les objets exception sont souvent instanciés dans l'instruction même qui assure leur lancement.

```
throw new MonException("Mon exception s'est produite !!!");
```

Emission d'une exception

- L'exception elle-même est levée par l'instruction **throw**.

- Une méthode susceptible de lever une exception est identifiée par le mot-clé **throws** suivi du type de l'exception

exemple :

```
public void ouvrirFichier(String name)  throws MonException
        {if (name==null) throw new MonException();
                    else
                     {...}
        }
```

throws

- Pour "laisser remonter" à la méthode appelante une exception qu'il ne veut pas traiter, le programmeur rajoute le mot réservé **throws** à la déclaration de la méthode dans laquelle l'exception est susceptible de se manifester.

```
public void uneMethode() throws IOException
  {
    // ne traite pas l'exception IOException
    // mais est succeptible de la générer
  }
```

throws

- Les programmeurs qui utilisent une méthode connaissent ainsi les exceptions qu'elle peut lever.
- La classe de l'exception indiquée peut tout à fait être une super-classe de l'exception effectivement générée.
- Une même méthode peut tout à fait "laisser remonter" plusieurs types d'exceptions (séparés par des ,).
- Une méthode doit traiter ou "laisser remonter" toutes les exceptions qui peuvent être générées dans les méthodes qu'elle appelle.

Conclusion

- Grâce aux exceptions, Java possède un mécanisme sophistiqué de gestion des erreurs permettant d 'écrire du code « robuste »
- Le programme peut déclencher des exceptions au moment opportun.
- Le programme peut capturer et traiter les exceptions grâce au bloc d'instruction `catch ... try ... finally`
- Le programmeur peut définir ses propres classes d 'exceptions

La gestion en Java des

Entrées/Sorties

Les entrées / sorties

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

La gestion des fichiers

- La gestion de fichiers proprement dite se fait par l'intermédiaire de la classe File.

- Cette classe possède des méthodes qui permettent d'interroger ou d'agir sur le système de gestion de fichiers du système d'exploitation.

- Un objet de la classe File peut représenter un fichier ou un répertoire.

La gestion des fichiers

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

La gestion des fichiers

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

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

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

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

La gestion des fichiers

```
import java.io.*;
public class Listeur
{
  public static void main(String[] args)
  { litrep(new File( "c:\\"));}

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

Le nom complet du fichier est rep\fichier

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

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

Notion de flux

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

Notion de flux

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

Notion de flux

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

Notion de flux

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

Flux d'octets et flux de caractères

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

Lecture de fichier

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

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

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

Ecriture dans un fichier

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

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

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

Les flux d'octets

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

La classe InputStream

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

La classe InputStream

- **Les méthodes principales qui peuvent être utilisées sur un InputStream sont :**
  - **public abstract int read () throws IOException** qui retourne l'octet lu ou -1 si la fin de la source de données est atteinte. C'est cette méthode qui doit être définie dans les sous-classes concrètes et qui est utilisée par les autres méthodes définies dans la classe **InputStream**.
  - **int read (byte[ ] b)** qui emplit un tableau d'octets et retourne le nombre d'octets lus
  - **int read (byte [] b, int off, int len)** qui emplit un tableau d 'octets à partir d 'une position donnée et sur une longueur donnée,

La classe InputStream

- **void close ()** qui permet de fermer un flux,
  - Il faut fermer les flux dès qu'on a fini de les utiliser. En effet, un flux ouvert consomme des ressources du système d'exploitation qui sont en nombre limité.
- **int available ()** qui retourne le nombre d'octets prêts à être lus dans le flux,
  - Attention : Cette fonction permet d'être sûr qu'on ne fait pas une tentative de lecture bloquante. Au moment de la lecture effective, il se peut qu'il y ait plus d'octets de disponibles.
- **long skip (long n)** qui permet d'ignorer un certain nombre d'octets en provenance du flot. Cette fonction renvoie le nombre d'octets effectivement ignorés.

La classe OutputStream

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

La classe OutputStream

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

Empilement de flux filtrés

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

Empilement de flux filtrés

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

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

Empilement de flux filtrés

Lecture bufferisée de nombres depuis un fichier

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

Lecture de nombre dans un fichier au format zip

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

Constructeurs de flux

- **Classe FileInputStream**
  - FileInputStream (String name)
  - FileInputStream (File f)
- **Classe FileOutputStream**
  - FileOutputStream (String name)
  - FileOutputStream (String name, boolean append)
  - FileOutputStream (File f)

Constructeurs de flux

- **Classe BufferedInputStream**
  - BufferedInputStream (InputStream in)
  - BufferedInputStream (InputStream in, int n)
- **Classe BufferedOutputStream**
  - BufferedOutputStream (OutputStream out)
  - BufferedOutputStream (OutputStream out, int n)

Flux de fichiers à accès direct (1)

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

Flux de fichiers à accès direct (2)

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

Les flux de caractères (1)

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

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

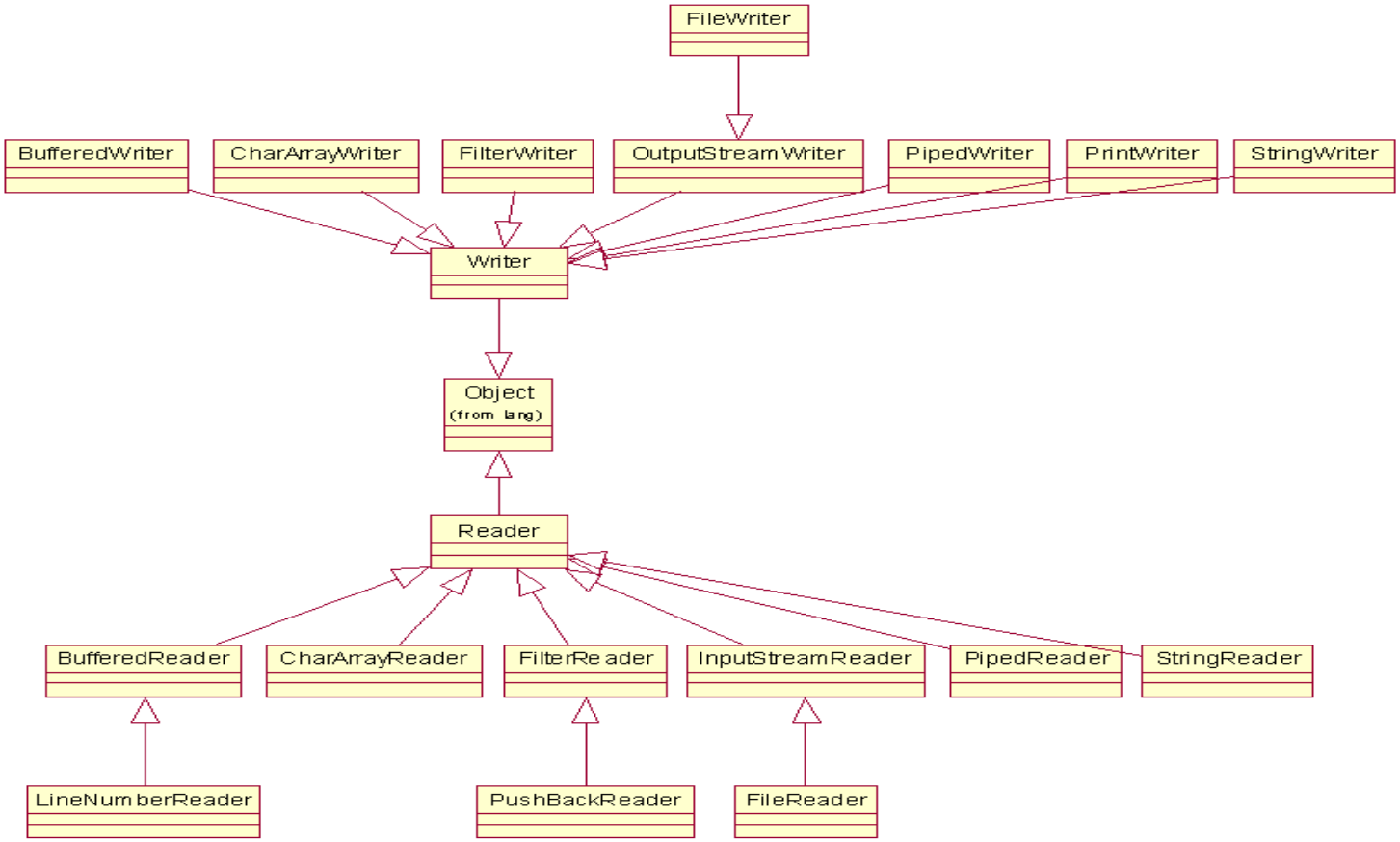
Conversion des caractères d ’un fichier
avec un codage explicitement indiqué

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

Les flux de caractères (2)

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

La hiérarchie des flux de caractères



Les flux de données prédéfinis (1)

Il existe 3 flux prédéfinis :

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

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

Les flux de données prédéfinis (2)

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

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

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

La sérialisation

La sérialisation consiste à prendre un objet en mémoire et à en sauvegarder l'état sur un flux de données (vers un fichier, par exemple).

Ce concept permet aussi de reconstruire, ultérieurement, l'objet en mémoire à l'identique de ce qu'il pouvait être initialement.

La sérialisation peut donc être considérée comme une forme de persistance des données.

2 classes **ObjectInputStream** et **ObjectOutputStream** proposent, respectivement, les méthodes **readObjet** et **writeObject**

Par défaut, les classes ne permettent pas de sauvegarder l'état d'un objet sur un flux de données. Il faut implémenter l'interface **java.io.Serializable**.

Exemple de sérialisation

```
void sauvegarde(String s) {
try  {FileOutputStream f = new FileOutputStream(new File(s));
          ObjectOutputStream oos = new ObjectOutputStream(f);
          oos.writeObject(this);
        oos.close();}
      catch (Exception e)
       { System.out.println("Erreur "+e);}
}

static Object relecture(String s) {
try  {FileInputStream f = new FileInputStream(new File(s));
          ObjectInputStream oos = new ObjectInputStream(f);
          Object o=oos.readObject();
        oos.close();}
      catch (Exception e)
       { System.out.println("Erreur "+e);}
}
```

Structures collective en Java

Motivations

- **1, 2, … plusieurs**
  - monôme, binôme, … polynôme
  - point, segment, triangle, … polygone
- **Importance en conception**
  - Relation entre classes
    - «… est une collection de … »
    - « … a pour composant une collection de … »
  - Choisir la meilleure structure collective
    - plus ou moins facile à mettre en œuvre
    - permettant des traitements efficaces
    - selon la taille (prévue) de la collection

Définition d 'une collection

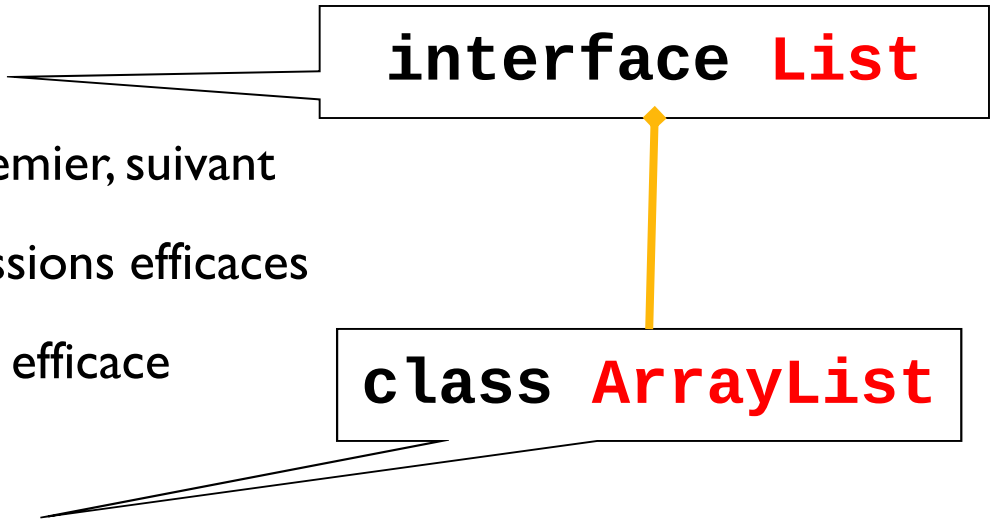
- **Une *collection* regroupe plusieurs données de même nature**
  - Exemples : promotion d'étudiants, sac de billes, ...
- **Une *structure collective* implante une collection**
  - plusieurs implantations possibles
    - ordonnées ou non, avec ou sans doublons, ...
    - accès, recherche, tris (algorithmes) plus ou moins efficaces
- **Objectifs**
  - adapter la structure collective aux besoins de la collection
  - ne pas re-programmer les traitements répétitifs classiques (affichage, saisie, recherche d'éléments, …)

Structures collectives classiques

- **Tableau**
  - accès par index
  - recherche efficace si le tableau est trié (dichotomie)
  - insertions et suppressions peu efficaces
  - défaut majeur : nombre d'éléments borné

`type[]` et `Array`

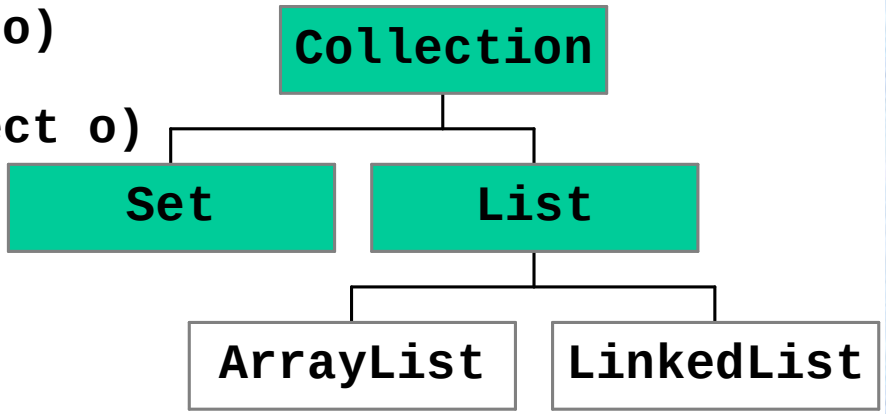
- **Liste**
  - accès séquentiel : premier, suivant
  - insertions et suppressions efficaces
  - recherche lente, non efficace



- **Tableau dynamique = tableau + liste**

Paquetage `java.util` de Java 2

- **Interface `Collection`**
- **Interfaces `Set` et `List`**
- **Méthodes**
  - `boolean add(Object o)`
  - `boolean remove(Object o)`
  - …
- **Plusieurs implantations**
  - tableau : **`ArrayList`**
  - liste chaînée : **`LinkedList`**



- Algorithmes génériques : tri, maximum, copie ...
  - ♦ méthodes statiques de **Collection**

Collection : méthodes communes

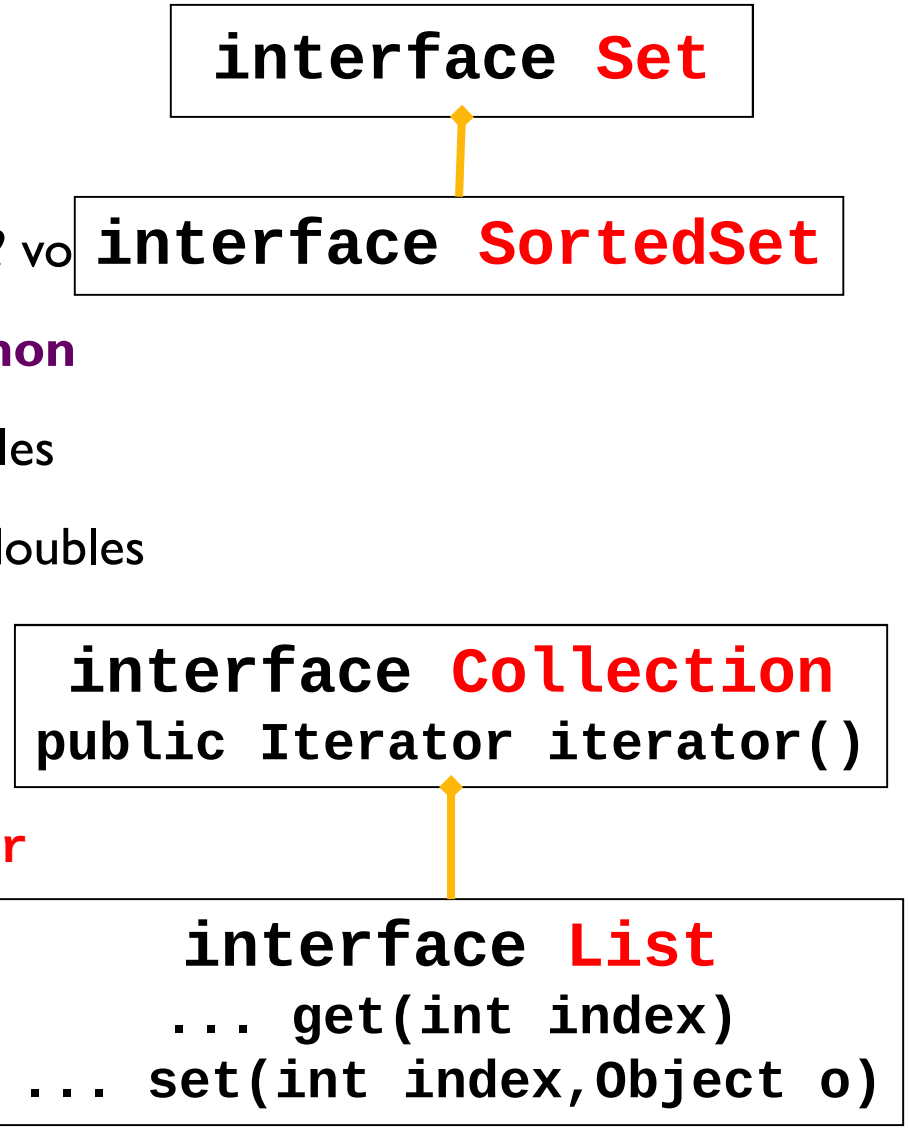
- **boolean add(Object)** : ajouter un élément
- **boolean addAll(Collection)** : ajouter plusieurs éléments
- **void clear()** : tout supprimer
- **boolean contains(Object)** : test d'appartenance
- **boolean containsAll(Collection)** : appartenance collective
- **boolean isEmpty()** : test de l'absence d'éléments
- **Iterator iterator()** : pour le parcours
- **boolean remove(Object)** : retrait d'un élément
- **boolean removeAll(Collection)** : retrait de plusieurs éléments
- **boolean retainAll(Collection)** : intersection
- **int size()** : nombre d'éléments
- **Object[] toArray()** : transformation en tableau
- **Object[] toArray(Object[] a)** : tableau de même type que a

Exemple : ajout d'éléments

```
import java.util.*;

public class MaCollection {
   static final int N = 25000;
   List listEntier = new ArrayList();

   public static void main(String args[]) {
      MaCollection c = new MaCollection();
      int i;
      for (i = 0; i < N; i++) {
         c.listEntier.add(new Integer(i));
      }
   }
}
```

Caractéristiques des collections

- **Ordonnées ou non**
  - Ordre sur les éléments ? vo
- **Doublons autorisés ou non**
  - ***liste*** (`List`) : avec doubles
  - ***ensemble*** (`Set`) : sans doubles
- **Besoins d'accès**
  - indexé
  - séquentiel, via `Iterator`



Fonctionnalités des Listes

- **Implantent l'interface `List`**
  - **`ArrayList`**
    - Liste implantée dans un tableau
    - accès immédiat à chaque élément
    - ajout et suppression lourdes
  - **`LinkedList`**
    - accès aux éléments lourd
    - ajout et suppression très efficaces
    - permettent d'implanter les structures FIFO (file) et LIFO (pile)
    - méthodes supplémentaires : **`addFirst()`, `addLast()`, `getFirst()`, `getLast()`, `removeFisrt()`, `removeLast()`**

Fonctionnalités des ensembles

- **Implantent l'interface `Set`**
- **Eléments non dupliqués**
  - **`HashSet`**
    - table de hashage
    - utiliser la méthode **`hashCode()`**
    - accès très performant aux éléments
  - **`TreeSet`**
    - arbre binaire de recherche
    - maintient l'ensemble trié en permanence
    - méthodes supplémentaires
      - first() (mini), last() (maxi), subSet(deb,fin), headSet(fin), tailSet(deb)

Recherche d'un élément

- **Méthode**
  - `public boolean contains(Object o)`
  - interface `Collection`, redéfinie selon les sous-classes
- **Utilise l'égalité entre objets**
  - égalité définie par `boolean equals(Object o)`
  - par défaut (classe `Object`) : égalité de références
  - à redéfinir dans chaque classe d'éléments
- **Cas spéciaux**
  - doublons : recherche du premier ou de toutes les occurrences ?
  - structures ordonnées : plus efficace, si les éléments sont comparables (voir tri)

Tri d'une structure collective

- **Algorithmes génériques**
  - Collections.sort(List l)
  - Arrays.sort(Object[] a,…)
- **Condition :** collection d'éléments dont la classe définit des règles de comparaison
  - en implémentant l'interface **java.lang.Comparable**
    - implements Comparable
  - en définissant la méthode de comparaison
    - public int compareTo(Object o)
    - **a.compareTo(b) == 0** si **a.equals(b)**
    - **a.compareTo(b) < 0** pour **a** strictement inférieur à **b**
    - **a.compareTo(b) > 0** pour **a** strictement supérieur à **b**

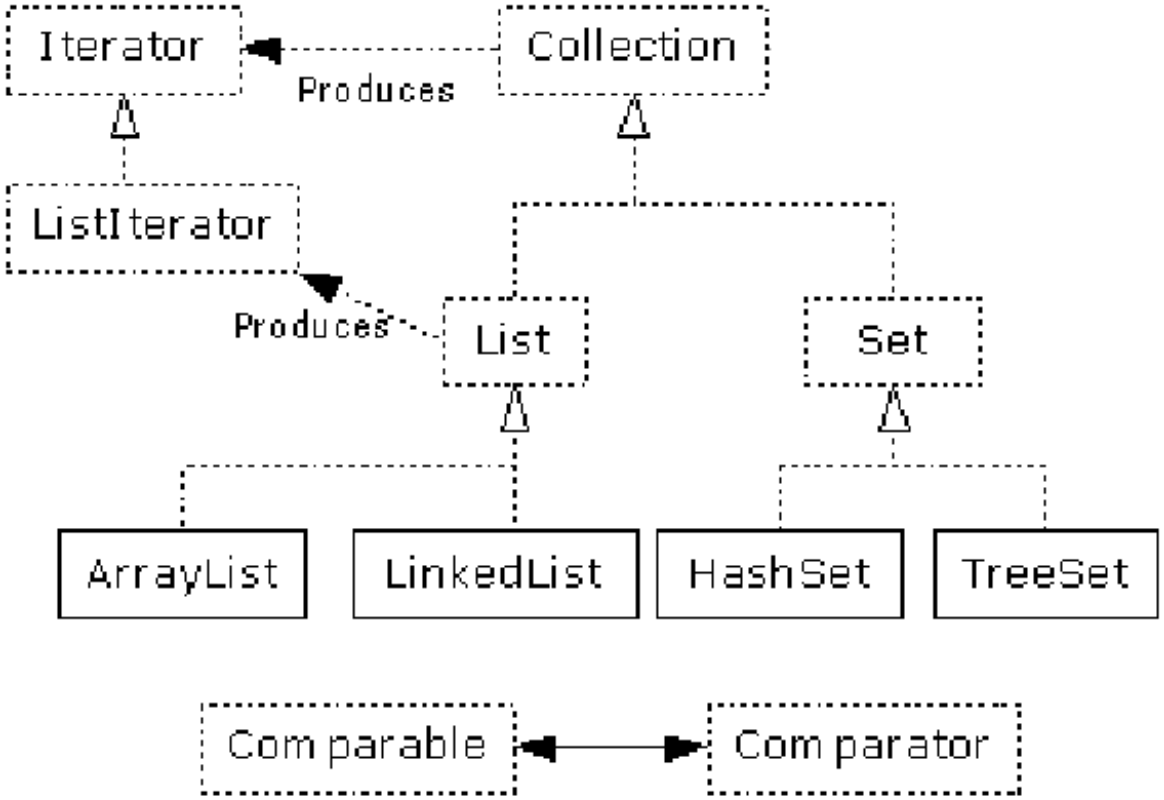
Généricité des algorithmes

- **N'utiliser que les méthodes communes**
- **Déclaration**
  - `Collection col = new ArrayList();`
- **Parcours des éléments de la collection : `Iterator`**
  - accès indexé pas toujours disponible (méthode **`get()`**)
  - utiliser la méthode **`Iterator iterator()`**
  - se déplacer avec les méthodes **`next()`** et **`hasNext()`**
  - exemple

```
Collection col = new TreeSet();
Iterator i = col.iterator();
while (i.hasNext())
      traiter ((transtypage)i.next());
```

Vue d'ensemble des collections

- **Hiérarchie simplifiée**



Paquetage en Java

Problématique

- **Il est recommandé d'écrire un fichier par classe ou par interface**
- **La réalisation d'une application Java peut donc générer des centaines de fichiers**
- **On pourrait vouloir utiliser le même nom de classe pour des concepts différents**
- **On voudrait pourvoir « masquer » l'existence de certaines classes**
- **⇒ Utilisation des packages**

Motivations

- **Regrouper plusieurs définitions de classes dans un groupe logique.**
- **Faciliter la recherche de l'emplacement physique des classes**
- **Rendre improbable la confusion entre des classes de même nom**
- **Structurer l'ensemble des classes selon une arborescence**
- **Permettrent de nuancer des niveaux de visibilité entre les classes selon qu'elles appartiennent ou non à un même paquetage.**

Utilisation de paquetages prédéfinis

- **Chaque classe et interface de l'API Java appartient à un `package` particulier qui contient un groupe de classes et d'interfaces reliées selon un thème commun (entrées/sorties, outils réseau, outils graphiques, ...).**
- **Par exemple, le paquetage java.io contient des classes permettant de travailler avec les entrées/sorties**
- **`import java.io.*;` = « si on rencontres un nom de classe inconnu, chercher dans le paquet java.io ».**
- **Si deux paquets importés contiennent deux classes qui portent le même nom, il faut utiliser le nom complet.**

Déclaration des paquets

- **Instruction** package **au début du fichier :**

```
package test.monpaquet;
public class Bonjour extends Object {
public void affiche(){
System.out.println("bonjour");}
}
```

- **Le fichier `Bonjour.java` peut se trouver n'importe où. Vous pouvez le compiler.**
- **Pour utiliser le paquet, placer `Bonjour.class` dans un répertoire `test/monPaquet/.`**

Utilisation des paquets

- **Pour accéder aux classes déclarées dans `monpaquet`, utiliser l'instruction : `import test.monpaquet.*;`**

```
import test.monpaquet.*;
class Principal extends Bonjour{
public static void main(String[] arg){
Bonjour b = new Bonjour();
b.affiche();
}
```

Définition des chemins

- **option -classpath de la commande javac. Indiquer à la suite de -classpath les chemins absolus ou relatifs des répertoires contenant les classes nécessaires à la compilation.**

```
javac -d $HOME/mon_paquetage MaClasse.java
```

- **variable d'environnement CLASSPATH doit contenir le chemin d'accès au répertoire racine du paquetage,**

Choix d ’un nom de paquetage

- **Choisir un nom en rapport clair avec l ’objectif des classes contenues dans le paquetage**
- **Pour s ’assurer qu ’un nom de paquetage est unique, il est recommandé d ’utiliser votre nom de domaine à l ’envers**

  Exemples :

  ```
  com.apple.quicktime.v2
  ```

- **Il est conseillé de choisir un nom commençant par un minuscule**

Notion de librairie

- Il arrive souvent que l'on veille réutiliser des classes entre plusieurs projets.
- Java offre la possibilité de stocker un groupe de classes dans une archive compressé. Ce type d'archive se nomme JAR (Java Application Archive).
- Les JAR permettent de facilement distribuer un groupe de classes compilées en un seul fichier. De plus, les classes étant compressé, cela permet de réduire considérablement la temps de chargement des classes dans le cadre d'une Applet

Création d'une librairie

- Une fois les classes sont codées et compilées, ouvrir un terminal de ligne de commande. Déplacer vous dans le répertoire contenant les classes compilés. Et tapez la ligne de commande suivante **:**

```
jar -cf ma_librairie.jar .
```

- **Explication :**
  - jar est une commande inclus dans le JDK, c'est elle qui permet la création de l'archive.
  - l'argument -c sert à indiquer que l'on veut créer une archive
  - l'argument f indique le nom du fichier de notre future archive (ma_librairie.jar dans notre cas)
  - le . à la fin de la ligne indique les fichier à être inclus, dans notre cas le répertoire courant

Utilisation d'une librairie

- Pour utiliser le JAR, il suffit de l'inclure dans le classpath lors du démarrage de l'application.

```
java -cp ma_librairie.jar MonApplication
```

- La notion de package fournit un mécanisme de réutilisation logicielle.


Les interfaces de Java

Interfaces : comment classifier ?

- Java ne permet pas l'héritage multiple
- Or, il existe parfois différentes classifications possibles selon plusieurs critères

Exemple de classification

- **Selon la forme**

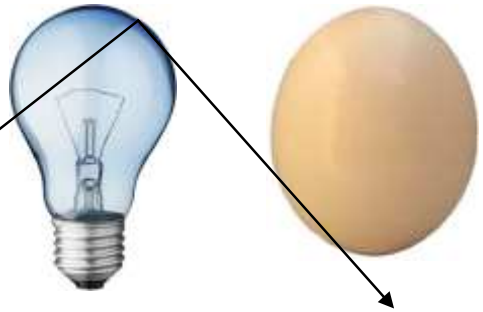
Solides convexes

Polyèdres

Parallélépipède

Cube

Solides de révolution



Cylindres

Sphères

Autre classification

- **Selon la matière**

matière

Matière organique

composée

minéral

carbone

Autres critères

- D'autres critères qui pourraient servir à réaliser une classification décrivent des comportements ou des capacités
  - « électrique »
  - « comestible »
  - « lumineux »

- Or ces « mécanismes » peuvent être commun à différentes classes non reliées entre elles par une relation d 'héritage

Notion d' « Interfaces »

- Pour définir qu'une certaine catégorie de classes doit implémenter un ensemble de méthodes, on peut regrouper les déclarations de ces méthodes dans une interface.
- Le but est de décrire le fait que de telles classes pourront ainsi être manipulées de manière identique.
- **Exemple** :
  - Tous les appareils électriques peuvent être allumés ou éteint
  - Tous les objets comestibles peuvent être mangés
  - Tous les objets lumineux éclairent

Définition d'Interface

- **Une interface est donc la description d'un ensemble des procédures (méthodes) que les classes Java peuvent mettre en oeuvre.**
- **Les classes désirant appartenir à la catégorie ainsi définie**
  - déclareront qu'elles implémentent cette interface,
  - fourniront le code spécifique des méthodes déclarées dans cette interface.
- **Cela peut être vu comme un contrat entre la classe et l'interface**
  - la classe s'engage à implémenter les méthodes définies dans l'interface

Codage d'une interface en Java

- Mot réservé : interface
- Dans un fichier *nom_interface*.java, on définit la liste de toutes les méthodes de l'interface

  interface nomInterface {
  type_retour methode1(*paramètres*);
  type_retour methode2(*paramètres*);
  … }

- Les méthodes d'une interface sont abstraites : elles seront écrites spécifiquement dans chaque classe implémentant l'interface

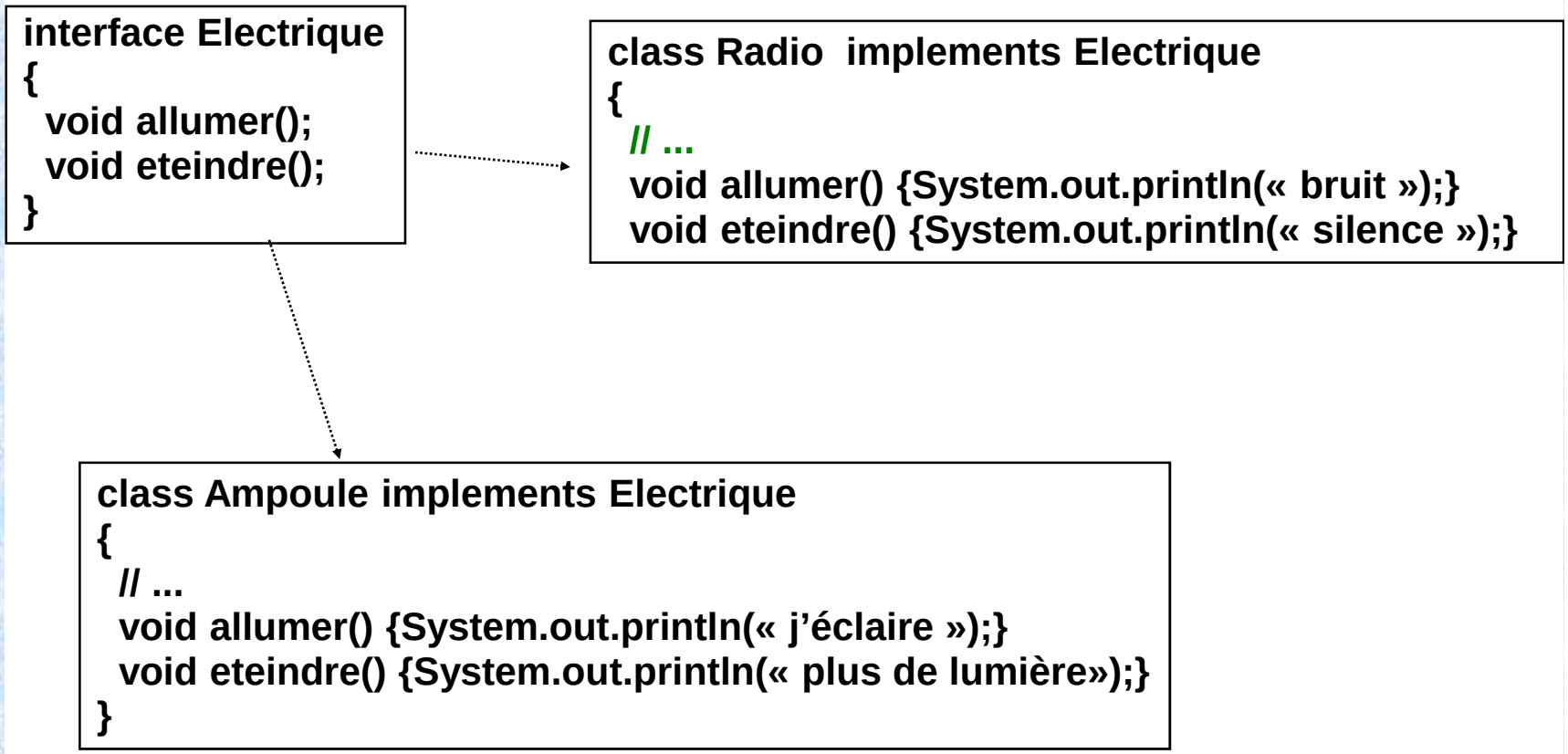
- Le modificateur abstract est facultatif.

Implémentation d'une interface dans une classe

- **Mot réservé : `implements`**
- **La classe doit expliciter le code de chaque méthode définie dans l'interface**

```
class MaClasse implements nomInterface {
                                …
  type_retour methode1(paramètres)
  {code spécifique à la méthode1 pour cette classe};

   …                                                    }
```

Exemple d'Interface (1)



Exemple d'Interface (2)

```
// …

Ampoule monAmpoule = new Ampoule();
Radio maRadio = new Radio();
Electrique c;
Boolean sombre;

// …

if(sombre == true)
   c = monAmpoule;
else
   c = maRadio;

c.allumer();
…
c.eteindre();

// …
```

Utilisation des interfaces

- Une variable peut être définie selon le type d'une interface
- Une classe peut implémenter plusieurs interfaces différentes
- L'opérateur instanceof peut être utilisé sur les interfaces

Exemple :

```
interface Electrique
…
interface Lumineux
…
class Ampoule implements Electrique, Lumineux
…
Electrique e;
Object o = new Ampoule();
if (o instanceof Electrique) {e=(Electrique)o;e.allumer();}
```

Conclusion sur les interfaces

- **Un moyen d'écrire du code générique**
- **Une solution au problème de l'héritage multiple**
- **Un outil de concevoir d'applications réutilisables**

Les threads

Qu'est-ce qu'un Thread ?

- **les threads sont différents des processus :**
  - ils partagent code, données et ressources : « processus légers »
  - mais peuvent disposer de leurs propres données.
  - ils peuvent s'exécuter en "parallèle"
- **Avantages :**
  - légèreté grâce au partage des données
  - meilleures performances au lancement et en exécution
  - partage les ressources système (pratique pour les I/O)
- **Utilité :**
  - puissance de la modélisation : un monde multithread
  - puissance d'exécution : paralèllisme
  - simplicité d'utilisation : c'est un objet Java (java.lang)

Création

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

Méthode 1 : Sous-classer Thread

```
class Proc1 extends Thread {
Proc1() {...} // Le constructeur
...
public void run() {
... // Ici ce que fait le processus : boucle infinie
}
}
...
Proc1 p1 = new Proc1(); // Création du processus p1
p1.start(); // Demarre le processus et execute p1.run()
```

Méthode 2 :
une classe qui implémente Runnable

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

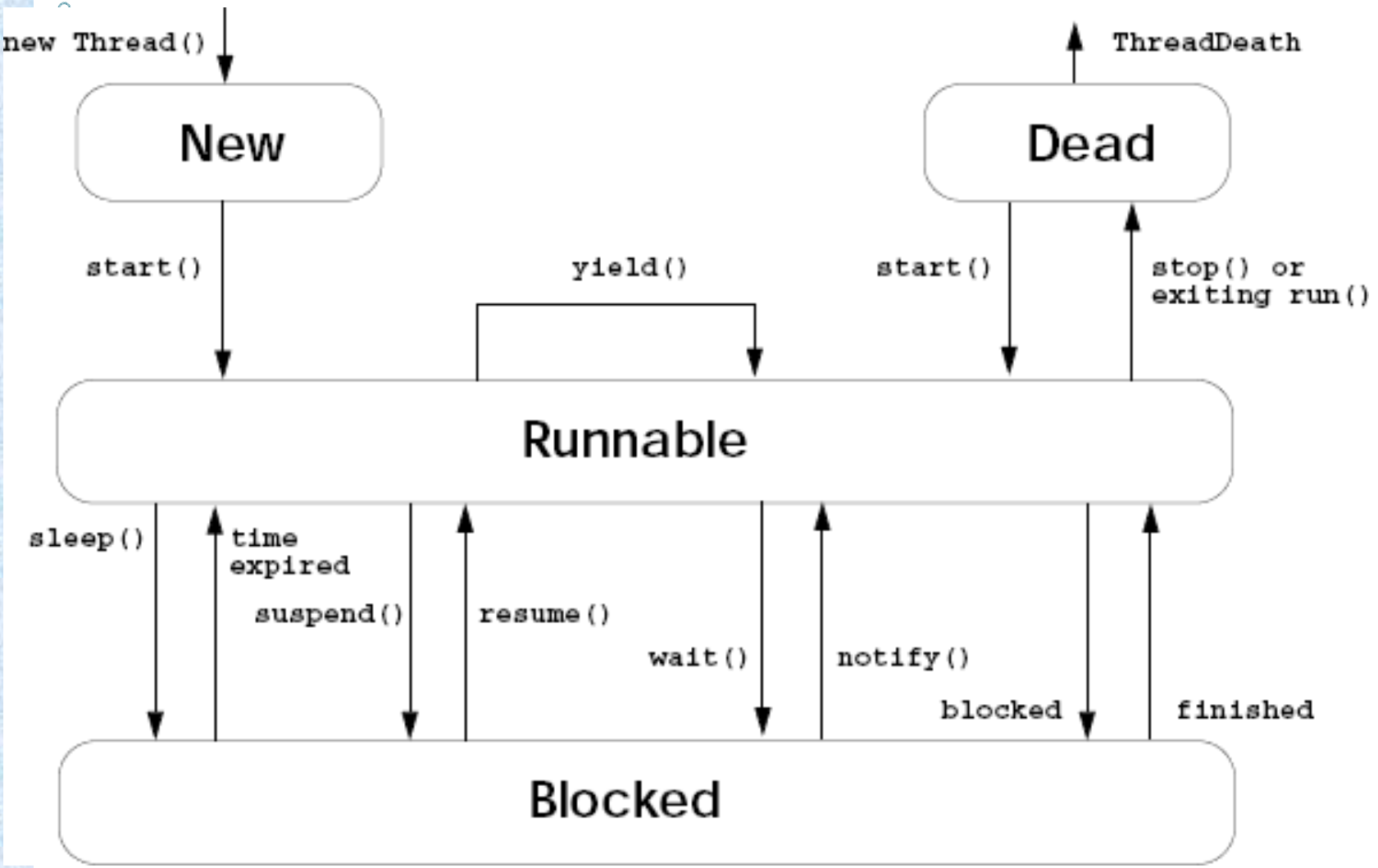
Quelle solution choisir ?

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

```
public class MyThreadApplet
extends Applet implements Runnable {}
```

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

Le cycle de vie



Les états d'un thread

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

Exemple

```
class ThreadCompteur extends Thread {
int no_fin;
ThreadCompteur (int fin) {no_fin = fin;} // Constructeur
// On redéfinit la méthode run()
public void run () {
for (int i=1; i<=no_fin ; i++) {
  System.out.println(this.getName()+":"+i);} }

public static void main (String args[]) {
// On instancie les threads
ThreadCompteur cp1 = new ThreadCompteur (100);
ThreadCompteur cp2 = new ThreadCompteur (50);
cp1.start();
cp2.start();
} }
```

Les états d'un Thread (suite)

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

Exemple d'utilisation de sleep

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

Les priorités

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

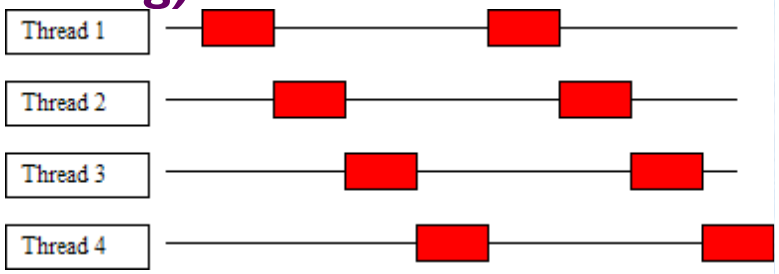
Les priorités (suite)

Les méthodes :

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

La gestion du CPU

- **Time-slicing (ou round-robin scheduling) :**
  - La JVM répartit de manière équitable le CPU entre tous les threads de même priorité. Ils s'exécutent en "parallèle".



- **Préemption (ou priority-based scheduling) :**
  - Le premier thread du groupe des threads à priorité égale monopolise le CPU. Il peut le céder :
    - involontairement : sur entrée/sortie
    - volontairement : appel à la méthode statique yield()

    Attention : ne permet pas à un thread de priorité inférieure de s'exécuter (seulement de priorité égale)
    - implicitement en passant à l'état endormi (wait(), sleep() ou suspend())

La concurrence d'accès

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

La synchronisation

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

Utiliser synchronized

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

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

- **Pour Contrôler l'accès à un objet :**

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

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

Daemons

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

Exemple de synchronisation

```
class Impression {
synchronized public void imprime(String t) {
for (int i=0; i<t.length(); i++) {
  System.out.print(t.charAt(i));
 } } }
class TPrint extends Thread {
static Impression mImp = new Impression();
String txt;
public TPrint(String t) {txt = t;}

public void run() {
for (int j=0; j<3; j++) {mImp.imprime(txt);}}

static public void main(String args[]) {
TPrint a = new TPrint("bonjour ");
TPrint b = new TPrint("au revoir ");
a.start();
b.start();
}}
```

Les « ThreadGroup »

Pour contrôler plusieurs threads

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

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

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

Les groupes de threads

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

Création d'un groupe de threads

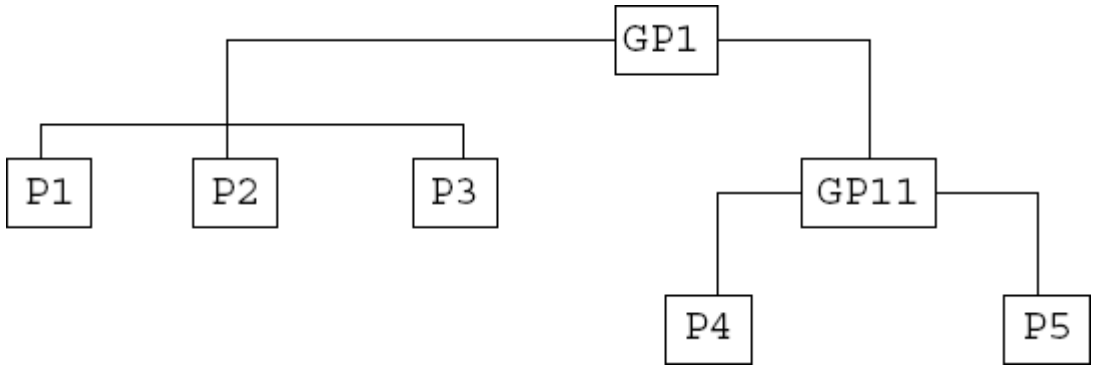
- **Pour créer un groupe de processus :**

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

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

Création de groupe de threads (suite)

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



Contrôler les ThreadGroup

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

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

  - Par exemple : appliquer la méthode `stop()` à un ThreadGroup revient à invoquer pour chaque Thread du groupe cette même méthode
  - ce sont des méthodes de manipulation récursive

Avantages / Inconvénients des threads

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