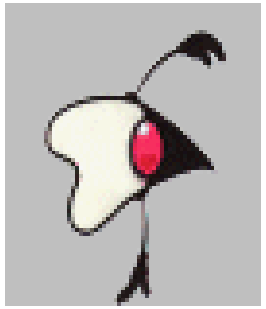
Introduction à Java



Conception de Java

- 1990 : projet Oak
  - SUN Microsystems Inc. (Gosling James)
- naissance de JAVA
  - réorientation du projet suite aux développements des techniques de communication
  - pause café ==> JAVA
- 1994 : introduction dans internet
- 1995 : JAVA anime le réseau
  - mai 1995 : navigateur Hot Java
- 1996 : premières applications JAVA
  - kit de développement JAVA (JDK)
  - Netscape
  - Microsoft
  - Symantec

Qu'est-ce que JAVA ?

- **JAVA est interprété**
  - **le pseudo-code est interprété par les navigateurs récents**
  - **Netscape Navigator (2 ->)**
  - **Microsoft Navigator (3.0)**
- **JAVA est indépendant de toute plate-forme**
  - **systèmes 32 bits usuels**
- **JAVA fonctionne comme un système d'exploitation virtuel**
  - **pour rester indépendantes de l'OS, les applets JAVA sont exécutées dans le cadre d'un programme de navigation compatible**
  - **le navigateur prend en charge les fonctions de débogueur et fonctionne comme un OS virtuel**

Qu'est-ce que JAVA ? (2)

- **JAVA est orienté objet**
  - **se rapproche des LOO**
    - **Eiffel**
    - **C++**
    - **Smalltalk**
- **JAVA est simple**
  - **abandon des pointeurs et de l'héritage multiple**
  - **l'adressage mémoire direct est interdit**
  - **aucun préprocesseur**
- **JAVA est rigoureusement Typé**
  - **la conversion automatique de type n'est pas implémentée**
  - **initialisation obligatoire ==> contrôle**

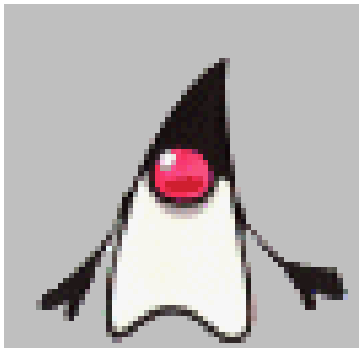
Qu'est-ce que JAVA ? (3)

- **JAVA organise la mémoire**
  - **ramasse miettes**
    - **Smalltalk**
    - **Eiffel**
- **JAVA est sûr**
  - **gestionnaire d'exception**
  - **définition des droits d'accès aux disques durs**
- **JAVA est rapide**
  - **compilation en pseudo-code de l'applet**
  - **les bib. de classes nécessaires ne sont liées qu'à l'exécution ==> faible taille de l'applet**
  - **compilation au vol possible**

Qu'est-ce que JAVA ? (4)

- **JAVA autorise le multitâche**
  - **threads : unité d'exécution isolée**
  - **permet de simuler le multitâche**
- **JAVA est dynamique**
  - **bib. de classes indépendantes ==> maintenance aisée**
- **JAVA est un langage ouvert**
  - **permet d'intégrer du code étranger**
- **JAVA et les distributeurs de logiciels**
  - **Oracle**
  - **Microsoft**

Le langage Java



La grammaire de Java

Types et Structures de contrôles

Les Bases

Identificateurs, Variables, Opérateurs

Commentaires (1)

- **Trois manières de créer des commentaires en JAVA.**
  - **/* ... */**

```
/* ceci est un commentaire
    sur plusieurs lignes
... qui se termine ici */
```

  - **//...**

```
int compteur ; // ceci est une variable entière
```

  - **/** et */.**

    Ce commentaire est à réserver dans les déclarations en vue d'une documentation automatique

Commentaires (2)

- **bonnes habitudes**
  - **Il faut absolument commenter les programmes.**
    - **un programme est écrit une seule fois**
    - **relu des dizaines de fois**
  - **un Commentaire est une reformulation de la spécification du programme**

- **mauvais commentaire**

```
int zoom=2 ; // zoom à 2
```

aucun renseignement sur le rôle de zoom et pourquoi 2.

- **commentaire CORRECT**

```
int zoom=2 ; // valeur par défaut du zoom au démarrage
```

Les identificateurs (1)

- **but**
  - ◆**nommer:**
    - **les variables,**
    - **les classes**
    - **les méthodes**
    - **les packages, ...**

    des programmes JAVA.
- **règle de nommage**

  identificateur =

  "dans {a..z, A..Z, $, _}"

  < " dans {a..z,A..Z,$,_,0..9,unicode character over 00C0}" > .

  (premier caractère puis le reste)

Les identificateurs (2)

- **exemples d'identificateur valides:**

```
$valeur_system
dateDeNaissance
ISO9000
```

- **exemples d'identificateur non-valides:**

```
ça          // Ç comme premier caractère
9neuf       // 9 comme premier caractère
note#       // # pas au dessus de 0X00C0
long        // OK mais c'est un mot réservé!
```

Les identificateurs (3)

Liste des mots réservés

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

Les identificateurs (4)

- **bonnes habitudes**
  - **ne pas utiliser le $ et le _ si vous devez utiliser des librairies en C;**
  - **ne pas utiliser le $ en première position;**
  - **séparer les noms composés en capitalisant la première lettre des noms à partir du deuxième mot.**

Conseillé	Acceptable	Déconseillé
monDeMethode	nom_de_methode	$init
ceceEstUneVariable	ceci_aussi	_type

Littéraux

- **Définition**

  Les littéraux définissent explicitement les valeurs sur lesquelles travaillent les programmes JAVA.

- **trois catégories de littéraux:**
  - **les booléens,**
  - **les nombres,**
  - **les caractères.**

Les littéraux : Booléens

- **mots clés**

```
false
true
```

- **permettent d'initialiser des variables booléennes**

- **(Attention: ne peut être assimilée à un 0 ou 1 comme dans le cas des langages C ou C++)**
- **exemple :**

```
boolean resultatAtteint = false ;
boolean continuer = true ;
```

Les littéraux : Entiers

- **trois formats:**
  - **en décimal,**
  - **en octal,**
  - **hexadécimal**
- **décimal**
  - **un entier ne commence jamais par un zéro.**

```
int nbrDeMois = 12;
```

- **octal**
  - **entier précédé d'un zéro.**

```
int nbrDeDoigts = 012; // =10 en décimal
```

- **hexadécimal**
  - **précédé d'un 0x ou 0X.**

```
int dixHuit= 0x12;
```

- **entiers déclarés littéralement = *int* sur 32 bits.**
- **entier *long* en lui faisant succéder un L (64 bits).**

Les littéraux : Flottants

- **Flottants**
  - **une mantisse**
  - **éventuellement exposant en puissance de 10.**
  - **obligatoirement un point décimal ou un exposant**
- **exemple de flottants sans partie exposant:**

```
2.  .5     2.5    2.0    .001
```

- **exemple de flottants avec partie exposant:**

```
2E0 2E3    2E-3   .2E3   2.5E-3
```

- **correspondance**
  - **Les nombres flottants déclarés = *float* sur 32 bits.**
  - **flottant *double* précision : se termine par un D (64 bits)**
  - **flottant *float* : se termine par un F (32 bits)**

```
3.14F  3.14159D
```

les littéraux : Caractères

- écriture : caractère encadré par deux apostrophes (quotation simple):

  `'x' 'a' '4'`

- le car. est choisi dans l'ensemble des caractères Unicode (16bits)

caractère	abréviation	séquence
continuation	<nouvelle ligne>	\
Nouvelle ligne	N	\n
tabulation	HT	\t
retour arrière	BS	\b
retour chariot	CR	\r
saut de page	FF	\f
backslash	\	\\
apostrophe	‘	\’
guillemet	"	\"
caractère octal	0377	\377
caractère hexadécimal	0xFF	\xFF
caractère Unicode	0xFFFF	\uFFFF

Les littéraux : domaine de valeurs

- **les 8 types élémentaires sont les suivants :**
  - **byte** octet -128 à +127
  - **boolean** codé sur un bit true ou false
  - **short** 2 octets -32768 à 32767
  - **int** 4 octets -2147483648 à 2147483647
  - **long** 8 octets -2^{63} à $(2^{63}-1)$
  - **float** 4 octets $\pm$1.4E-45 à $\pm$3.40282347E38
  - **double** 8 octets $\pm$4.9E-324 à $\pm$1.79769313486231570E308
  - **char** 2 octets
- **l'implémentation est indépendante**
  - **de la machine**
  - **et de l'O.S.**

les littéraux : Chaînes de car.

- **chaînes de caractères**
  - **suite de caractères entourée de guillemets,**
  - **type *String*.**
  - **constitue une classe**
  - **≠ un vecteur de caractères.**

chaîne de caractères	résultat de l'impression
""	
"\""	"
"texte sur 1 ligne"	texte sur 1 ligne
"texte sur \ndeux lignes"	texte sur deux lignes
"\ttabulé"	tabulé

Déclaration des variables (1)

- **L'utilisation des variables doit être précédée obligatoirement par une déclaration**

- **Le compilateur pourra vérifier**
  - **la compatibilité de type dans les expressions,**
  - **la visibilité de la variable**

- **La déclaration des variables permet d'augmenter la qualité des programmes**
  - **détecter des erreurs à la compilation**
  - **détecter au moment de l'exécution.**

Déclaration des variables (2)

- **syntaxe :**
  - **préfixer le nom de la variable par son type**
  - **exemple :**

```
int i;
int i,j,k;
```

- **modifier la définition de la variable**

  ***final* = une constante.**
  - **exemple :**

```
final int nbreDeRoues=4;
final float pi=3.14159;
```

Type simple versus composé

- types simples
  - les booléens,
  - les entiers
  - ...

- types composés,
  construits à partir d'autres types
  - vecteurs,
  - matrices,
  - classes,
  - interfaces.

Variable de type booléen

- variable dont le contenue sera vrai ou faux.
- affectée par les littéraux
  - `false`
  - `true`
- peut recevoir le résultat d'une expression logique

```
boolean voitureArretee = true;
```

Variable de type entier

- **quatre types d'entier:**
  - **byte**
  - **short**
  - **int**
  - **long**

Type	Nbre de bits	Exemple
`byte`	8	`byte nbrEnfants;`
`short`	16	`short volumeSon;`
`int`	32	`int i,j,k;`
`long`	64	`long detteSecu;`

- **affectée par**
  - **les littéraux entiers**
  - **le résultat d'une expression entière**

Variable de type flottant

- **deux types de flottant:**
  - `float`
  - `double`

Type	Nbre de bits	Exemple
`float`	32	`float ageMoyen;`
`double`	64	`double pi;`

- **affectée par**
  - **les littéraux flottants**
  - **résultat d'une expression flottante**

Variable de type caractère

- variable dont la valeur est un élément de l'ensemble Unicode de caractères
- assignée par
  - les littéraux caractères
  - le résultat d'une expression caractère.
- exemple

```
char separateur='\t';
```

- Le caractère en JAVA est représenté sur 16 bits.
  - Il ne sert qu'à conserver la valeur d'un seul caractère.
- Les chaînes sont représentées par la classe *String*.

Vecteurs et matrices

- **déclaration :post-fixer le type ou la variable par [ ] :**

```
int i[ ]; // vecteur d'entiers
int[ ] j; // j à le même type que i
char motsCroises[ ][ ]; // une matrice de caractères
```

- **pas contraints au moment de la déclaration**

- **L'allocation de l'espace nécessaire au vecteur se fera**
  - **au moyen d'une méthode *new***
  - **ou au moment de la déclaration grâce à l'initialisation**

```
int [] f = {1, 1, 3, 6 };
double e[][] = { {0.0, 1.0} , {1.0, 0.0}};
```

Les tableaux : ça a la couleur du C

- **Déclaration**
  - **int [ ] TableauEntier .....**
  - **int TableauEntier [ ] ....**
- **initialisation**
  - **int NbreDeJoursEnFevrier [ ] = { 28, 28, 28, 29 } ;**
  - **String LesNomsDesMois [ ] = new String [12];**
- **Les tableaux sont des objets et donc, ils doivent être alloués dynamiquement.**
  - **int Tableaux[ ] = new int [4];**
- **accès au tableau**
  - **indice entier débutant à 0**

Les Tableaux : mais ce n'est pas du C

- **il n'y a qu'un seul moyen de réserver la place pour un tableau, c'est de l'allouer EXPLICITEMENT**
- **les tableaux gèrent leur longueur**
  - **accessible par la méthode length()**
  - **lèvent une exception si un débordement intervient**
- **les tableaux sont indexés par des *int* et non par des *long***
- **les chaînes de car. ne sont pas des tableaux**

Les chaînes de caractères

- **En java :**
  - **les chaînes ne peuvent engendrer d'écrasement mémoire**
  - **il n'y a pas à se préoccuper de leur longueur**
  - **leur implémentation est portable et sûre**
  - **elles fournissent des méthodes de conversion vers d'autres formats**
  - **elles sont fournies avec le langage**
  - **elles gèrent tous les caractères indifféremment**

String ou StringBuffer

- **En Java, il existe deux classes différentes : *String* et *StringBuffer***
  - **la première représente les chaînes CONSTANTES**
  - **la seconde les chaînes modifiables**

- **Avantages de la classe *String***
  - **plus économe en ressources**
  - **constantes donc un autre programme ne peut la modifier**

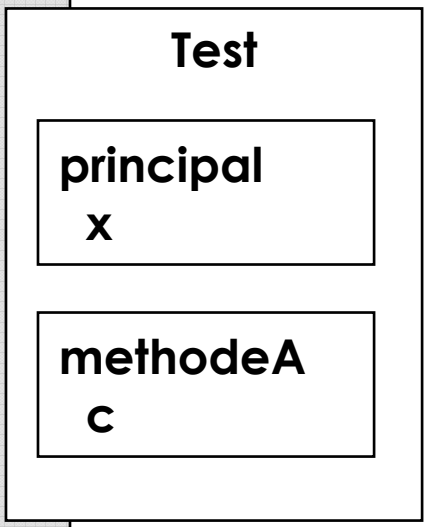
Chaînes : quelques méthodes

- **créer une chaîne : constructeurs**
  - **chaîne vide**
  - **à partir d'une autre chaîne**
  - **à partir d'une série d'octets**
  - **à partir d'une chaîne non constantes**
- **extraire un car. particulier**
- **comparer deux chaînes**
- **passer les car. de maj. en min. (et réciproquement)**
- **enlever les blancs**
- **remplacer les caractères**
- **ajouter une autre chaîne à la suite de la chaîne courante**

Visibilité des variables

- La variable est visible à l'intérieur du bloc où elle est définie.
  - un bloc est définit comme l'ensemble des instructions comprises entre deux accolades { }

```
class Test { // debut de test
   public static void principal(String args[])
   { // debut de principal
         float x
         ...
   } // fin de principal
   public void methodeA ()
   { // debut de methodeA
         char c
         ...
   } // fin de methodeA
} // fin de principal
```

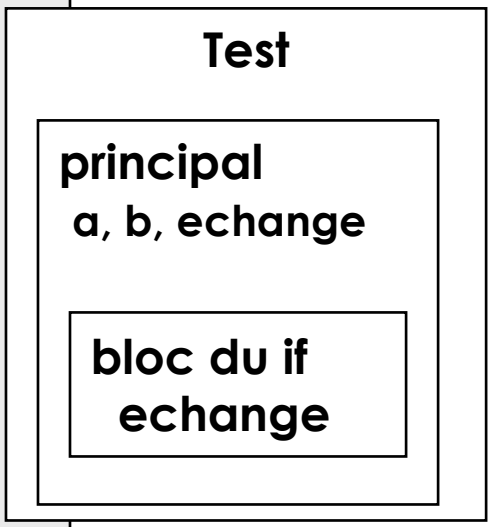


visibilité d'une variable

redéfinition des variables

- **si dans un bloc on redéfinit une variable existant dans un bloc supérieur,**
  cette nouvelle variable masque la variable supérieur à l'intérieur de ce bloc
  (donc aussi pour ses sous-blocs).

```
class Test { // debut de test
  public static void principal(String args[])
  { // debut de principal
       float echange; int a,b;
       ...
       if a < b then
       { int echange;
              echange=a;a=b;b=echange
       };
  } // fin de principal
       ...
  } // fin de Test
```



visibilité d'une variable dans un sous-bloc

Opérateurs

- regroupés par type d'opération:
  - numérique
  - de comparaison
  - logique
  - sur les chaînes de caractères
  - de manipulations binaires
- classification selon le nombre d'opérandes :
  - unaire
  - binaire
  - ternaire
- évaluées de la gauche vers la droite,
- une table de précédence indique la priorité entre les opérations.

```
x=z+w-y/(3*y^2)
```

  - on a = + / ( ) par ordre de lecture
  - La table de précédence => () / + - =.
  - Dans la (), on a * ^ par ordre de lecture,
  - l'ordre d'exécution est aussi *^.

Précédence des opérateurs

- **Les opérateurs de même niveau depuis la gauche.**

.	[]	()		
++	--	!	~	instanceof
*	/	%		
+	-			
<<	>>	>>>		
<	>	<=	>=	
==	!=			
&				
^				
&&				
||				
?:				
=	op=			
,				

Affectation

- L'affectation assigne l'expression de droite, à l'expression de gauche (une variable).
  - opérateur binaire qui modifie son opérande gauche.
- exemple
  - `j=2; // expression d'affectation d'un littéral`
  - `i=j*3; // expression d'affectation d'une expression`
  - `i= (j=2)*3;// expression valide combinant les deux`
  - Cette dernière expression a pour résultat i=6!
- effets de bord: ~, ++ , --
  - dans une même expression rend confus la lecture.
  - décomposer les expressions afin qu'elles ne contiennent qu'un seul opérateur avec effet de bord.

Les structures de contrôle

dérivées des langages C et C++

Les structures de contrôle

- **le bloc d'instruction**

- **exécution conditionnelle**
  - **SI ... ALORS ... SINON**
  - **choix parmi ...**
- **exécution répétitive**
  - **TANT QUE ... FAIRE ...**
  - **FAIRE ... TANT QUE ...**
  - **POUR ... FAIRE ...**

  - **IDENTIFIER UNE INSTRUCTION**
  - **RUPTURE**
  - **CONTINUATION**

Le bloc d'instructions ou instruction composée

- groupe d'instructions délimité par une paire d'accolades

- définit la portée des variables.
- peuvent être imbriqués à l'intérieur d'un autre bloc
  - il n'est pas possible de déclarer des variables homonymes dans deux blocs imbriqués (contrairement à C++)

- exemples :

```
{   }              // bloc vide

{ int i;           // bloc regroupant plusieurs
  i = 6;           //     déclarations et instructions
  int j = 10;      // i et j ne sont définie que jusqu'à
}                  //     la fin du bloc
```

Exécution conditionelle

- **Il existe deux formes d'exécution conditionnelle**
  - ***alternative***
  - ***choix parmi N***

Alternative (1)

- **Si ... alors ... sinon**
  - **structure de base de la programmation**
  - **L'expression de test doit être booléenne (vraie ou fausse)**

- **Deux formes principales**
  - **if (condition) instruction 1;**

```
if (i==1) s = " i est égal à 1 ";
```

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

```
if (i==1)     s = " i est égal à 1 ";
else          s = " i est différent de 1 ";
```

Alternative (2)

- **Formes dérivées (NB : leur utilisation est préférable)**
  - **if (condition) { bloc };**

```
if ( vosVentes >= cible ) {
  performance = "Satisfaisant";
  prime = 200;
}
```

  - **if (condition) { bloc1 } else { bloc2 };**

```
if ( vosVentes >= cible ) {
  performance = "Satisfaisant";
  bonus = 100 + 0.01 * (vosVentes – cible);
}
else {
  s = "Insatisfaisant";
  bonus = 0;
}
```

Alternatives en cascade

- **if (c1) instr1 else if (c2) instr2 ... else instrN;**

```
if (i==1)
  s=" i est égal à 1 ";
else if (i==2)
  s=" i est égal à 2 ";
else if (i==3)
  s=" i est égal à 3 ";
else s=" i est différent de 1,2 et 3 ";
```

- **L'utilisation de {} est nécessaire pour marquer le début est la fin des instructions**

```
i=0; j=3;
if (i==0)
if (j==2) s=" i est égal à 2 ";
else i=j;
System.out.println(i);
```

- **Que va imprimer le programme précédant 0 ou 3 ?**

Choix parmi N instruction switch

- **se comporte comme un aiguillage limité *à char, byte, short ,int***
  - **évalue l'expression qui lui est liée,**
  - **compare aux *case*.**
  - **commence à exécuter le code du switch OK**
  - **exécute donc tous les instructions des cas suivants**
- **afin d'isoler chaque cas, il faut le terminer avec un *break*.**
- **Si aucun cas , clause *default***
- ***case* et *defautl* sont considérés comme des étiquettes.**

```
switch ( variable ) {
 case valeur1 : instr1;
 case valeur2 : instr2;
 ...
 case valeurN : instrN;
 default      : instrDefaut;
};
```

Choix parmi N (...)

- forme générale de switch avec break

```
switch ( variable ) {
  case valeur1: instr1; break;
  case valeur2: instr2; break;
...
  case valeurN: instrN; break;
  default: instrDefaut;
};
```

Choix parmi N (...)

```
// conversion de caractères minuscules en majuscules
char conversion(char c) {
  switch( c ) {
       case 'ä' :
       case 'à' :
       case 'â' :
              return 'A';
       case 'ë' :
       case 'é' :
       case 'è' :
       case 'ê' :
              return 'E';
       default: return (char)(c -32);
  }
}
```

Structures de contrôle Itératives

- **Java définit les structures itératives suivantes:**
  - **while** — **tant que … faire …**
  - **do … while** — **faire … tant que …**
  - **for** — **itération universelle**

Tant que faire (while)

- **la boucle doit être exécutée tant qu'une condition est remplie**
  sans que cette condition porte sur une forme d'indice

- **syntaxe :**
  - **while (*condition*) instruction;**
  - **while (*condition*) { bloc }**
  - **L'expression *condition* doit être du type booléen**

- **utile**
  - **lorsque le nombre maximal d'itération n'est pas connu à l'avance**
  - **lorsque la condition de sortie est indépendante de celui-ci**

Tant que faire (2)

- **exemple**

```
int i=100, somme_i=0, j=0;
// boucle 1: expression sans effet de bord
    while ( j<=i ) {
      somme_i+=j;
      ++j;
    }
    System.out.println("boucle 1:"+somme_i);

// boucle 2: expression avec effet de bord
    somme_i=0; j=0;
    while ( ++j<=i ) somme_i+=j;
    System.out.println("boucle 2:"+somme_i);
```

Faire … tant que …

- **L'expression e doit être du type booléen.**

```
do S1 while(e);
do {B1} while(e);
```

- **exemple :**

```
class TestDo{
  /* le résultat n'est pas erroné si l'on execute
     une addition de trop si i= 0 ! */
  public static void main (String args[]) {
    int i=100, somme_i=0, j=0;
    // boucle 1: expression sans effet de bord
    do {   somme_i+=j;
           ++j;
    } while (j<=i);
    System.out.println("boucle 1:"+somme_i);
```

Faire ... tant que ...

```
    // boucle 2: expression avec effet de bord
        somme_i=0; j=0;
        do {
              somme_i+=j;
        } while ( ++j<=i );

        System.out.println("boucle 2:"+somme_i);
    }
}
```

Pour … faire …

- structure employée pour exécuter un bloc d'instructions un nombre de fois en principe connu à l'avance

- syntaxe :

```
for (e1;e2;e3) S1;.
for (e1;e2;e3) B1;
```

- basé sur un itérateur qui est contrôlé dans l'instruction
  - expression e1 initialise l'itérateur + le déclare
  - expression e2 teste si la condition d'achèvement
  - expression e3 modifie l'itérateur

  - Les expressions *e1*, *e2* et *e3* sont optionnelles.
  - Le bloc B1 doit
    - modifier l'itérateur
    - peut utiliser un *break* pour quitter la boucle.

Pour ... initialisation

- la partie initialisation consiste en l'initialisation d'une ou plusieurs variables
- exemple:

```
for ( i=1 ; ...
for( i=1, j=5, k=12 ; ...
```

avec ces syntaxes, la ou les variables doivent avoir été déclarées préalablement

- il est possible d'utiliser des variables de types différents

```
int i;
long j;
char k;
for( i =1, j=5, k='a' ; ...
```

Pour ... initialisation

- **déclaration et initialisation des itérateurs**

```
for(int i=5, j=2, k=7 ; ...
```

  - **MAIS dans ce cas tous les itérateurs doivent être de même type**

- **il est possible d'utiliser une variable préalablement déclarée et initialisée**

```
int i =5;
for( ; ...    // i sera testée et modifiée
```

Pour ... test

- la partie test
  - se compose d'une expression logique qui est évaluée et prend la valeur true ou false
  - si la valeur est true, le bloc d'instructions suivant est exécuté dans le cas contraire, la boucle se termine et l'exécution se poursuit à la ligne suivant le bloc
- la boucle for ne peut comporter qu'une seule expression logique
  - mais celle-ci peut comprendre plusieurs expressions combinée à l'aide des opérateurs logiques

```
for(i=0, j=2, k='a'; i<=10 || j <= 12 || k <= 'x'; ...
```

Pour ... test

- **boucle infinie**
  - **il suffit d'utiliser true comme expression logique**
  - ```
    for(byte i=0; true ; i++)
       System.out.println(i);
    ```
  - **cet exemple ne provoque pas d'erreurs à l'exécution !!!**
    ```

Pour ... incrémentation

- **modification des itérateurs**
  - **il est possible d'effectuer toutes les opérations que vous souhaitez**

```
for(int i=0; i <=10; i++, System.out.println(i))  {
          ...
  }
```

Pour … faire …

- **La structure *for* est équivalent au *while***

```
for( e1; e2; e3){ S1 }
     est équivalent à
e1; while(e2) {S1; e3};
```

Identifier une instruction

- Attribution d'un label à une instruction ou à un bloc d'instruction afin de l'identifier

  (label = identificateur suivi de :)

- Ceci est utilisée en conjonction avec
  - Les instructions de rupture
  - Les instructions de continuation d'itération
- Exemple :

```
class LabelInst{
  public static void main (String[ ]  args) {
   boucle1:
     for (int i=0;i<10;i++)
       for (int j=0; j<10;j++){
            System.out.println(i+"/"+j);
       }
  }
}
```

Rupture

- utilisée dans l'instruction *switch*, *while*, *do* et *for*
  - quitter la boucle dans laquelle elle est comprise
  - remonter de plusieurs niveaux
- Syntaxe :

```
   break etiquette ;// permet de remonter de n niveaux
Ou break;           // permet de quitter le niveau courant
```

- Exemple :

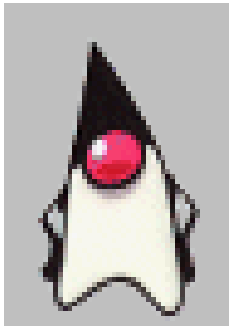
```
class TestBreak{
  public static void main (String[ ] args) {
    boucle1:
      for (int i=0;i<10;i++)
            for (int j=0; j<10;j++){
               System.out.println(i+"/"+j);
               if (j==1) break; // niveau courant
               if (i==2) break boucle1; // niveau de la boucle i
            }
  }
}
```

Continuation

- *continue* est utilisée dans *while, do* et *for.*
  - interruption du déroulement normal de la boucle
  - Le contrôle est repris après la dernière instruction de la boucle
- Il est possible d'associer une étiquette
- Exemple :

```
class TestContinue{
  public static void main (String args[]) {
    boucle1:
      for (int i=0;i<10;i++)
              for (int j=0; j<10;j++){
                if (j>1) continue; // quitte la boucle locale
                if (i>2) continue boucle1; // quitte la boucle for i ...
                System.out.println(i+"/"+j);
              }
      System.out.println("en dehors des boucles");
  }
}
```

Objets et Programmation

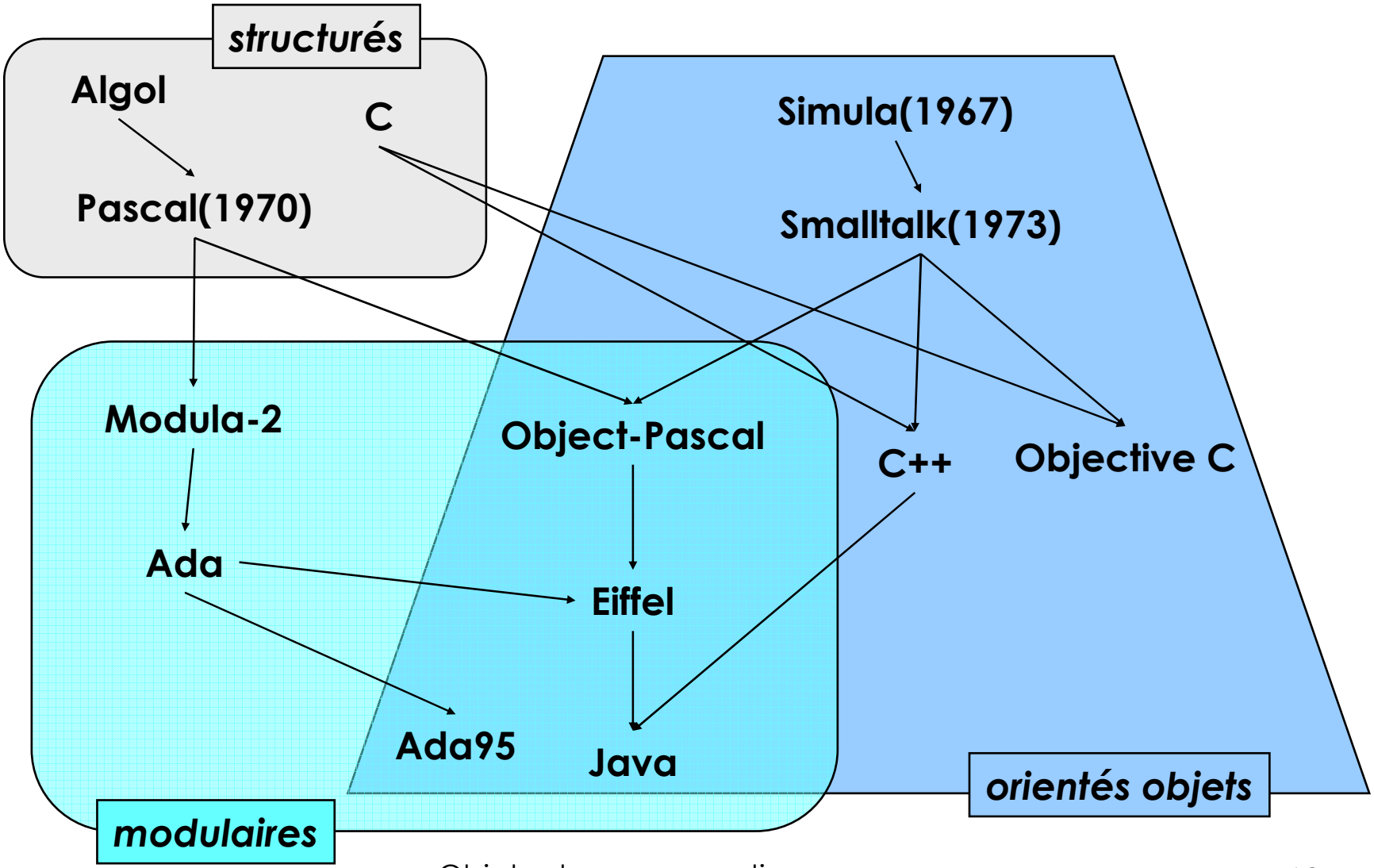


rappels

Objets et Programmation

- **Objets et Programmation**
  - **origine des langages orientés-objet**
  - **modularité, encapsulation**
  - **objets, classes, messages**
  - **exemples en Java**
  - **héritage, liaison dynamique**

Origine des langages orientés objet



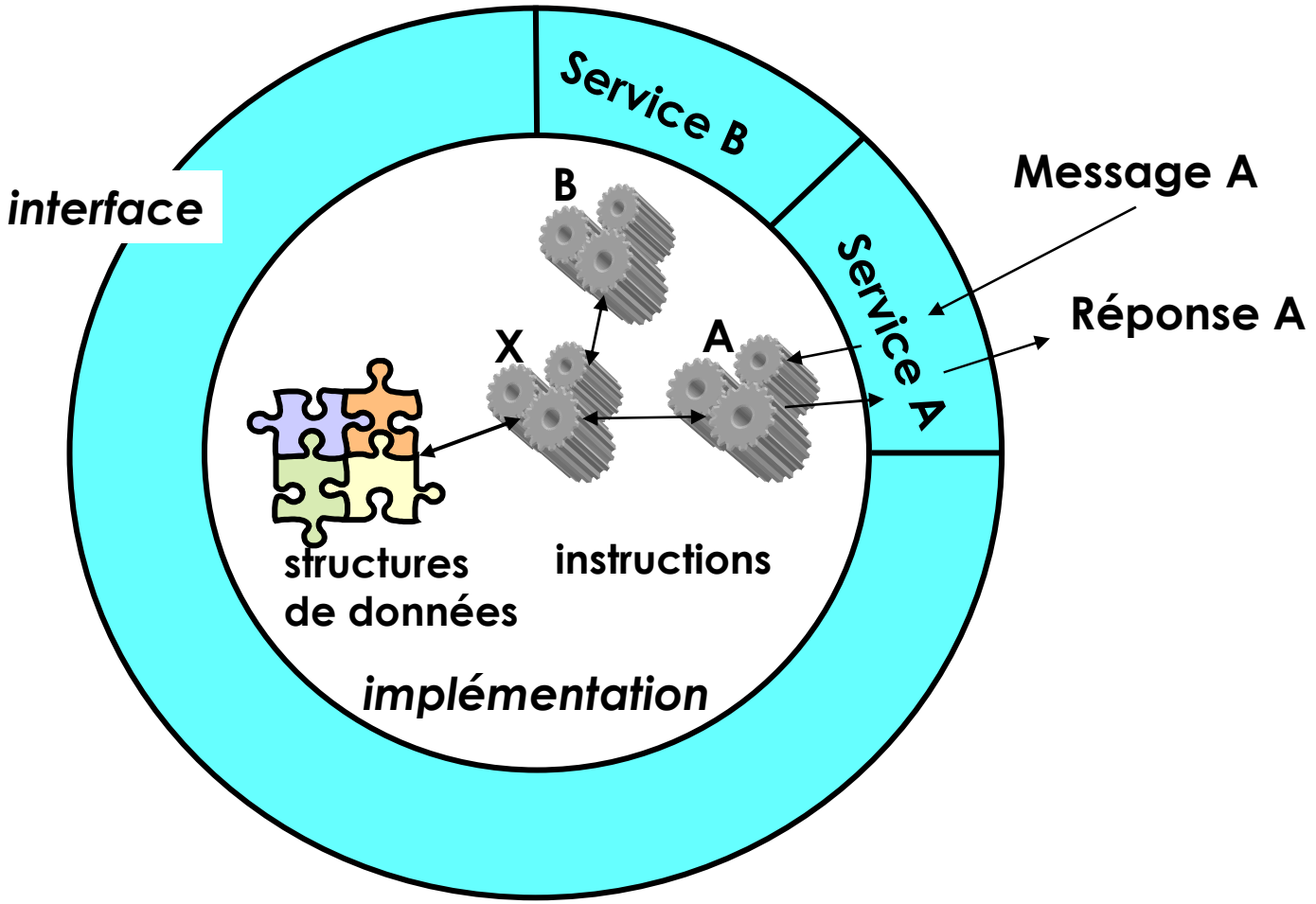
Modularité et Encapsulation

- **Modularité**
  - technique de *décomposition* de systèmes
  - *réduire la complexité* d'un système par un assemblage de sous-systèmes plus simples
  - réussite de la technique dépend du degré d'*indépendance* entre les sous-systèmes(ou degré de *couplage*)
  - on appelle *module*, un sous-système dont le couplage avec les autres est relativement faible par rapport au couplage de ses propres parties

Encapsulation

- **Encapsulation**
  - ◆**technique pour *favoriser la modularité* (l'indépendance) des sous-systèmes**
  - ◆**séparer l'interface d'un module de son implémentation**
    - ***interface* (partie publique): liste des services offerts (quoi)**
    - ***implémentation* (partie privée): réalisation des services (comment)**
      - **structure de données**
      - **instructions et algorithmes**
  - ◆**protection des données par des règles (dans les instructions):les modules communiquent par *messages*, pas par accès aux données**

Un Module


Service B
interface
B
Message A
Service A
Réponse A
X
A
structures
de données
instructions
implémentation

Types abstraits de données

- **Type "classique"**
  ensembles de valeurs possibles
- **Type abstrait de données**
  ensemble d'opérations applicables
- **Exemple :TAD rectangle**
  -- Opération de construction:

```
rect: entier X, entier Y, entier L, entier H --> rectangle;
translation: rectangle R, entier DX, entier DY--> rectangle;
agrandissement: rectangle R, entier DL, entier DH --> rectangle.
```

  -- Opération d'accès

```
gauche: rectangle R --> entier;
haut: rectangle R --> entier;
largeur: rectangle R --> entier;
hauteur: rectangle R --> entier;
bas: rectangle R --> entier;
droite: rectangle R --> entier.
```

  -- Sémantique => équations

```
gauche(translation(R, DX, DY)) <==> gauche(R) + DX
```

Modules + Types abstraits => Langage à objet (1)

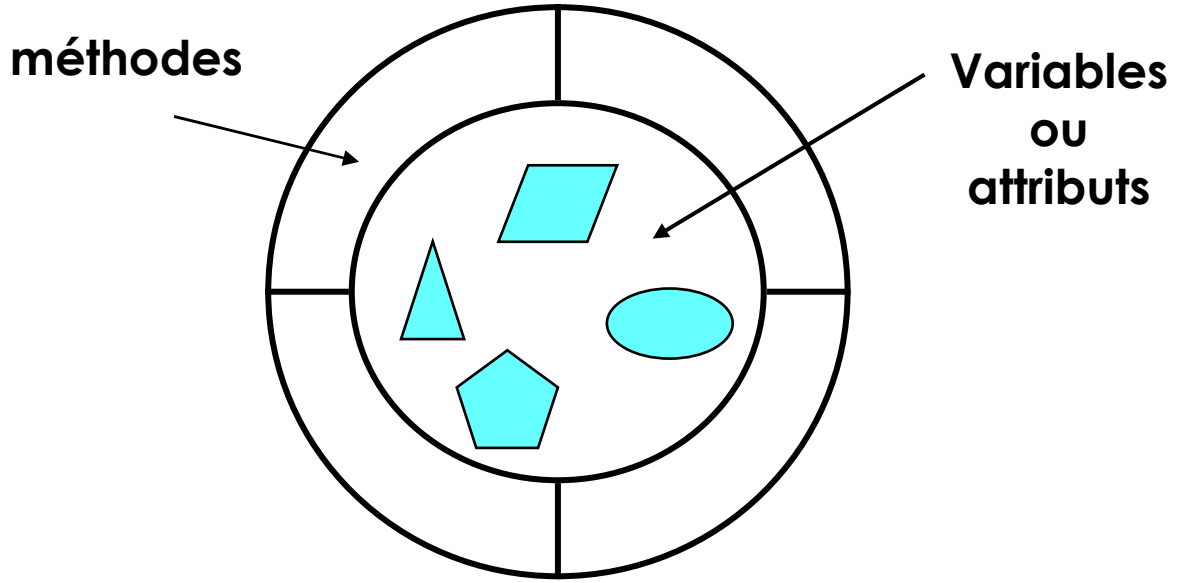
- **Un objet est un module**
  - **privé: variables d'instance, code des méthodes**
  - **public: nom et paramètres des méthodes**
  - **communication: invocation de méthodes, retour de résultats**
- **Classe = générateur d'objet**
  - **génère des objets de même structure**
  - **localise la définition de structure et action des objets**
  - **définit la visibilité: private, public, etc.**

Modules + Types abstraits => Langage à objet (2)

- **Classe --> type abstrait**
  - **partie publique --> spécif. (partielle) TA**
- **Classe --> structuration du logiciel**
  - **liens "client/serveur"**
    - **statiques (variables)**
    - **dynamiques (méthodes)**
  - **découplage spécification / réalisation**
  - **différentes versions de la réalisation**

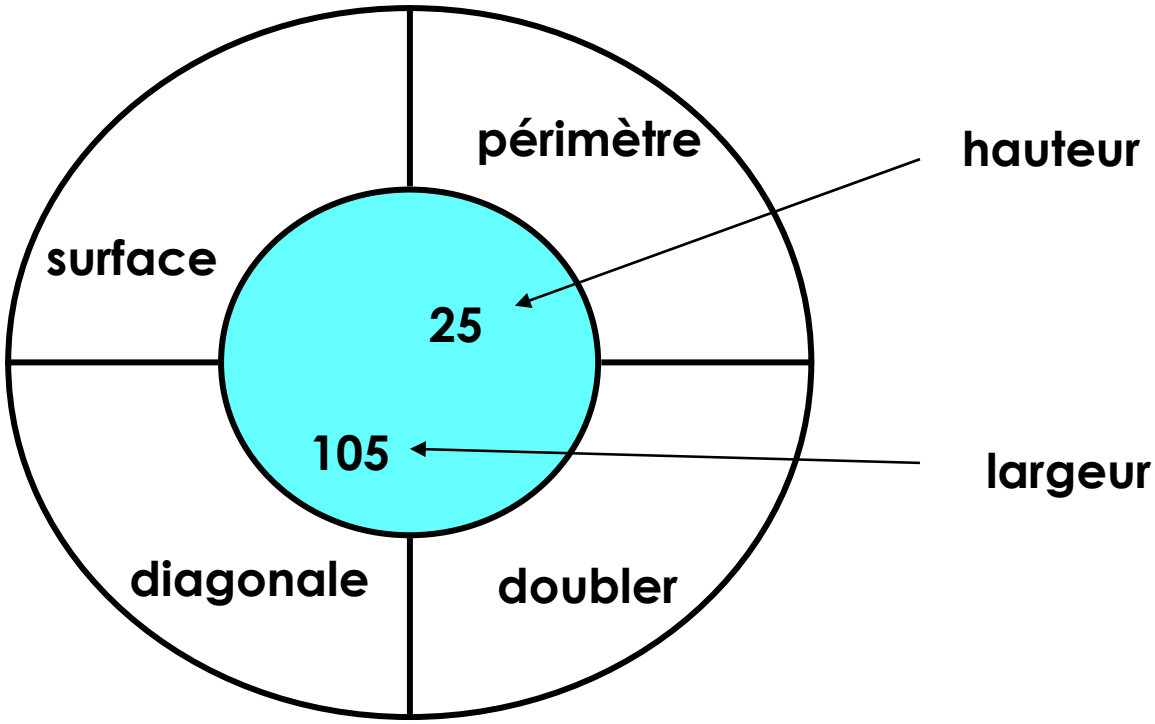
Propriétés des Objets

- **Un objet est une entité contenant**
  - **Des données (état)**
  - **Des procédures associées (comportement)**
- **Un objet possède une identité unique et invariable**



Propriétés des Objets (2)

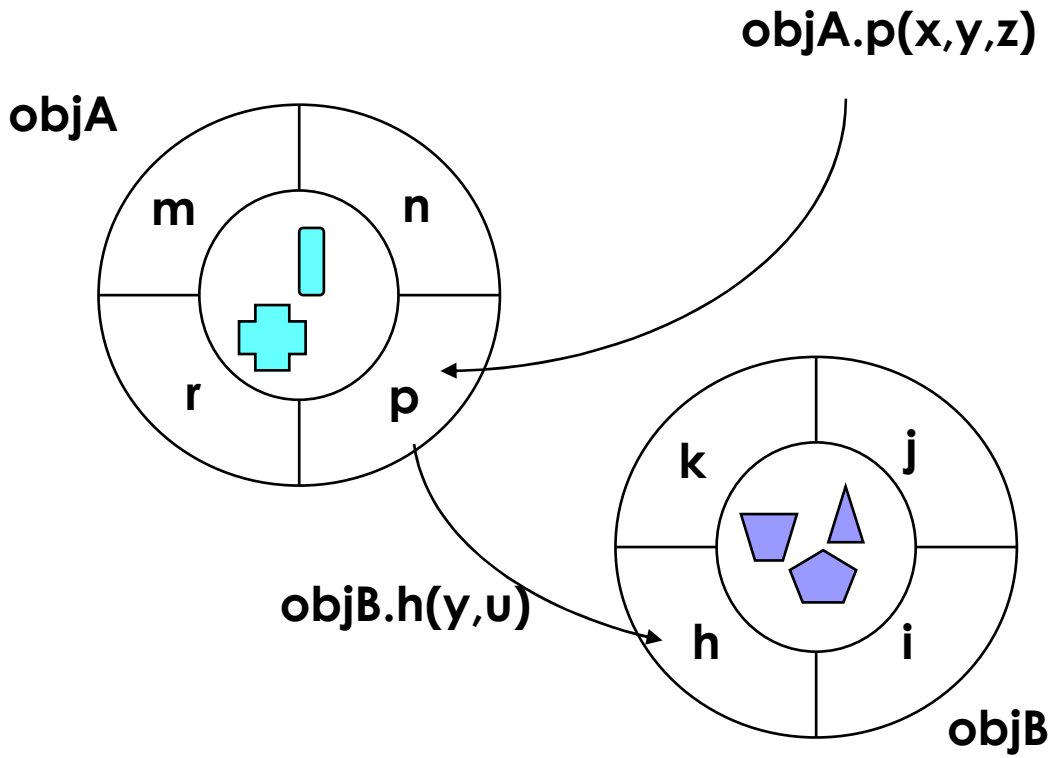
- **Exemple : un Objet rectangle**



Messages (1)

- **Pour utiliser un objet on lui envoie des messages**
- **Un message déclenche l'exécution d'une méthode**
- **Une méthode peut envoyer des messages à d'autres objets**
- **Un système est constitué d'objets qui communiquent entre eux**

Messages (2)



Messages (3)

- **Exemples en Java :**

```
rect1.doubler();

d = rect1.diagonale();

System.out.println("Hello");

i = "Les sanglots longs".size();

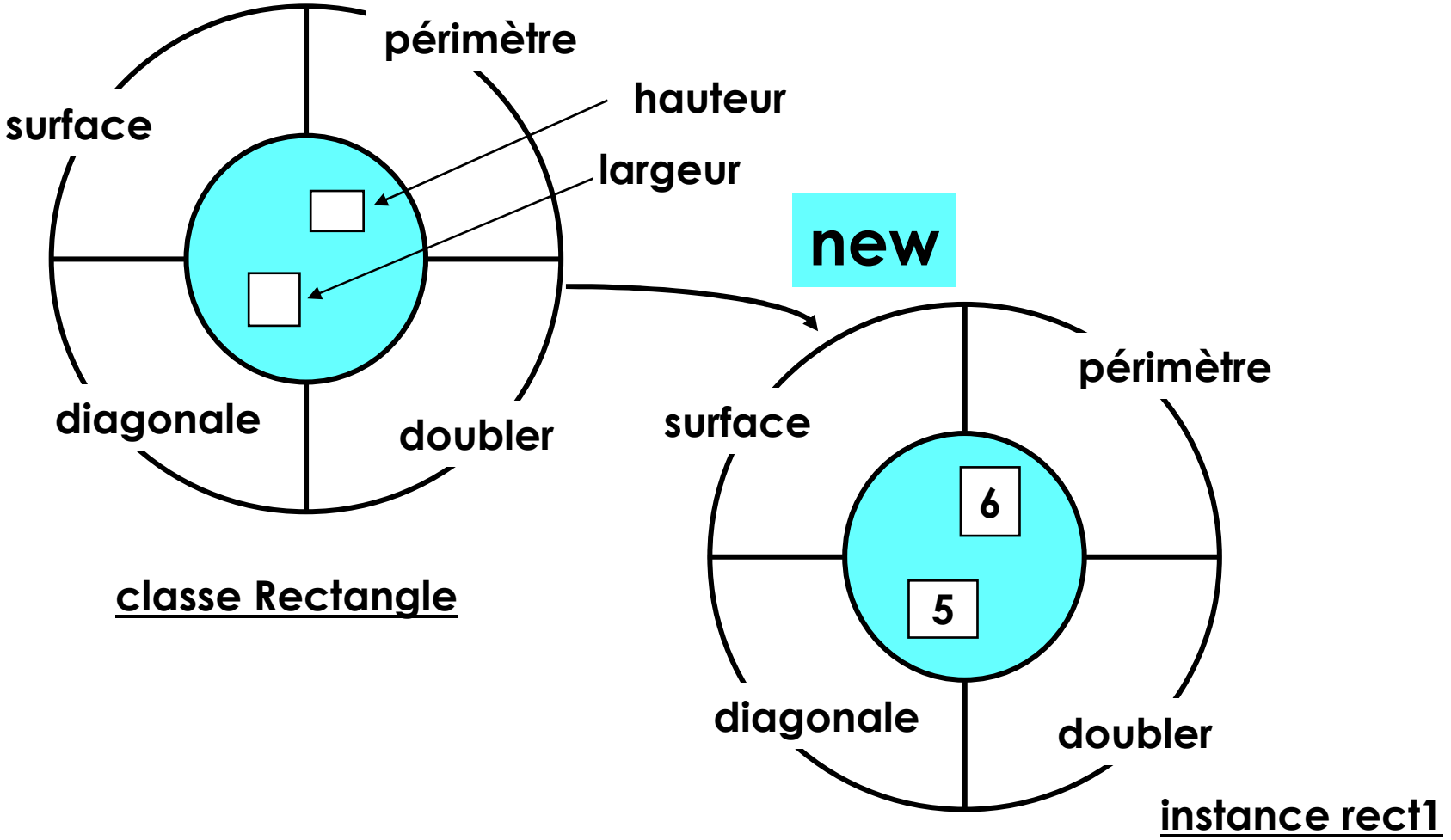
uneListe.insertAt(12, "oui" );

z = Math.cos(2.45);
```

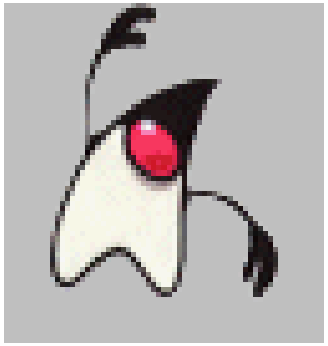
Classes

- **Une classe est un moule pour fabriquer des objets de même structure et de même comportement**
- **Un objet est une instance d'une classe**
- **Une classe C définit un type C**
- **Une variable de type C peut faire référence à un objet de la classe C**

Classe et objet



Les Objets en Java



Déclaration simplifiée d'une classe

```
class nouveauType {
  variables d'instances
  méthodes d'instances
  constructeurs
}
```

Exemple :

```
class Rectangle {
     …
}
```

Attributs d'instance

- **Les variables d'instance**
  - **sont propres à chaque instance de la classe**
  - **constituent la mémoire de chaque objet**

- **Exemple :**
  - **`largeur` et `hauteur` peuvent être déclarées comme variables d'instances de la classe Rectangle**

```
class Rectangle {
  double largeur;
  double hauteur;
  ...
 }
```

Initialisation des attributs

- initialisation des attributs
  - en indiquant les valeurs qui leur seront données lors de la création de l'objet
- Ces valeurs leur sont données avant l'appel du constructeur
- les attributs non initialisés explicitement ont des valeurs par défauts, égale à zéro.
- exemple :

```
class Rectangle {
   double largeur = 5 ;
   double hauteur = 8;
   …
}
```

Méthodes d'instance

- **Une méthode d'instance d'une classe détermine le comportement des objets de ce type**
  - **une définition de méthode consiste en :**
    - **des déclarations de portées, s'il y en a**
    - **le type de retour de la méthode**
    - **le nom de la méthode**
    - **l'information requise de l'extérieur pour accomplir les instructions**
    - **les étapes pour accomplir l'action**

les méthodes sont
- spécifier ET
- implémenter

  - **Exemple :**

```
class Rectangle {
  double surface() { return largeur*hauteur; }
  void doubler() { largeur*=2; hauteur*=2; }
  ...
}
```

Signature d'une méthode

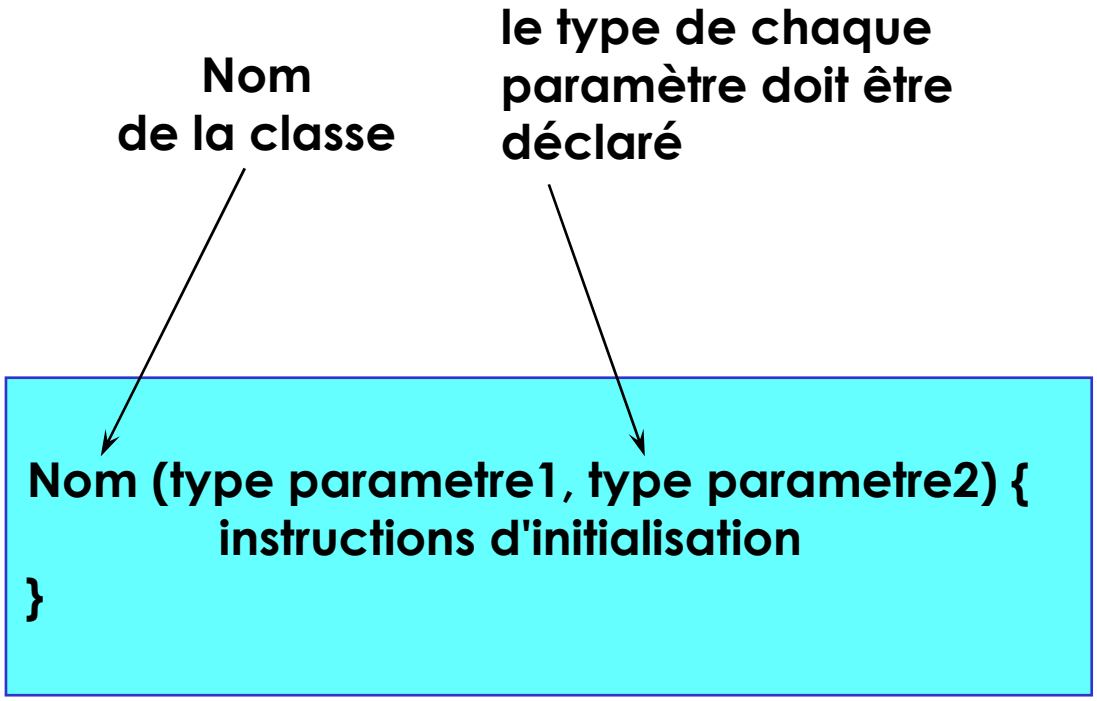
- **Définition**
  - **Le type du résultat**
  - **Le nom de la méthode**
  - **Le nom et le type des paramètres**

  - **Forment la SIGNATURE de la méthode**

Constructeurs

- le constructeur est appelé au moment de le création de l'objet afin d'initialiser une instance
- Plusieurs constructeurs peuvent exister, ils différent les uns des autres par leur(s) paramètre(s)
- l'utilisation du new entraîne
  - la création physique de l'objet
  - l'appel d'un constructeur

Anatomie d'un constructeur



Typologie des constructeurs

- **constructeur spécialisé**
- **constructeur par défaut**
  - **fournit par Java SSI aucun constructeur n'existe**
  - **peut être défini par l'utilisateur**
- **constructeur de copie**
  - **Permet de cloner un objet**
  - **à définir explicitement**
- **Les pièges du C++ sont évités :**
  - **seuls les constructeurs spécifiés sont utilisables**

Constructeurs : exemple

```
class Rectangle {
    …
    // constructeur spécialisé
    Rectangle (double initL, double initH) {
            largeur = initL;
            hauteur = intH;
    };
    // constructeur par défaut
    Rectangle() {
            largeur = 40;
            hauteur = 30;
    }
    // constructeur de copie
    Rectangle( Rectangle autre ) {
            largeur  = autre.largeur;
            hauteur = autre.hauteur;
    }
    …
}
```

le mot-clé : this

- dans un constructeur il est possible d'appeler un autre constructeur de la classe en se servant du mot-clé: "this"
- exemple :

```
class Vehicule {
  int imma;
  char carburant;
  public Vehicule() { carburant = 'E' ; }

  public Vehicule(int i) {
    this(); // appel du constructeur par défaut
    imma = i;}
}
```

Destructeur

- il n'y a pas de destructeur en Java, contrairement à C++
- En Java, on ne libère pas explicitement les objets
  Le système les récupère
- Il est possible de définir une méthode finalize() pour une classe
- cette méthode est exécutée
  - soit à la libération de l'objet
  - soit par l'appel à System.runFinalization()

Création d'instances et affectation

- **Création d'une instance**
  - **Pour créer une instance on invoque un constructeur à l'aide de l'instruction new**
  - **Exemple :**

```
new Rectangle(50,40);
```

- **Affectation**
  - **On peut affecter un objet à une variable**
    - **Il suffit de déclarer une variable du type correspondant**
    - **Puis on procède à l'affectation**
  - **Exemple:**

```
Rectangle rect1;
rect1 = new Rectangle(50,40);
Rectangle rect2 = new Rectangle (60,40);
```

Exemple de définition d'une classe

exemple de classe

```
class Rectangle {
   //attributs
   double largeur, hauteur;

   // constructeur
   Rectangle( double initL, initH) {
      largeur = initH;
      hauteur = initL;
   }

  // méthodes
   void doubler(){ largeur*=2; hauteur*=2; }
   double surface() { return largeur*hauteur; }
   double perimetre() { return 2*(hauteur*largeur); }
}
```

Invocation des méthodes

- **Invocation d'une méthode**
  - **s'effectue en nommant l'instance et en la faisant suivre du nom de la méthode et de la liste, éventuellement vide, d'expressions**
  - **Si la méthode possède des paramètres, les expressions données lors de l'invocation (arguments) sont évaluées et affectées aux paramètres correspondants de la méthode**

- **syntaxe:**

  instance.méthode(expression1, ...)

  - **Exemple : pour obtenir la surface du rectangle rect1**

    ```
    surf = rect1.surface();
    ```

le mot-clé: "this"

- le mot-clé "this" sert également à désigner le receveur d'un message

- exemple :

```
class Rectangle {
  ...
  boolean estEgal( Rectangle aux ) {
    return ( this.largeur == aux.largeur )
         && ( this.hauteur == aux.hauteur );
  }
  ...
}
```

Variable de classe

- **Une variable de classe**
  - ◆**permet de caractériser une classe c'est à dire un ensemble d'objets:**
    - **Exemple : le taux de TVA des Produits courants**
  - ◆**est accessible en le préfixant du nom de la classe**
    - **Exemple : Produit.tauxTVA**
  - ◆**est définie en se servant du modificateur `static`**
  - ◆**et doit toujours être initialisée**
  - ◆**Exemple :**

```
class Produit {
  static float tauxTVA = 18.6 ;
  ...
```

Méthodes de classe

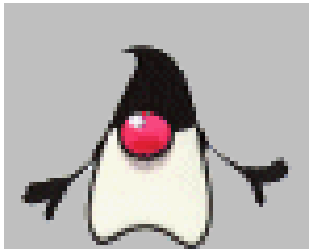
- **Les méthodes de classe**
  - sont destinées à agir sur la classe plutôt que sur ses instances
  - elles permettent de manipuler des variables de classe

- **L'appel d'une méthode de classe**
  - se fait en préfixant le nom de la méthode par le nom de la classe

- **Exemple :**

```
class Produit {
  ...
  static float getTauxTVA() {
    return tauxTVA; }
}
```

Héritage



L'héritage

- L'héritage est une relation entre classes définie par le mot-clé extends

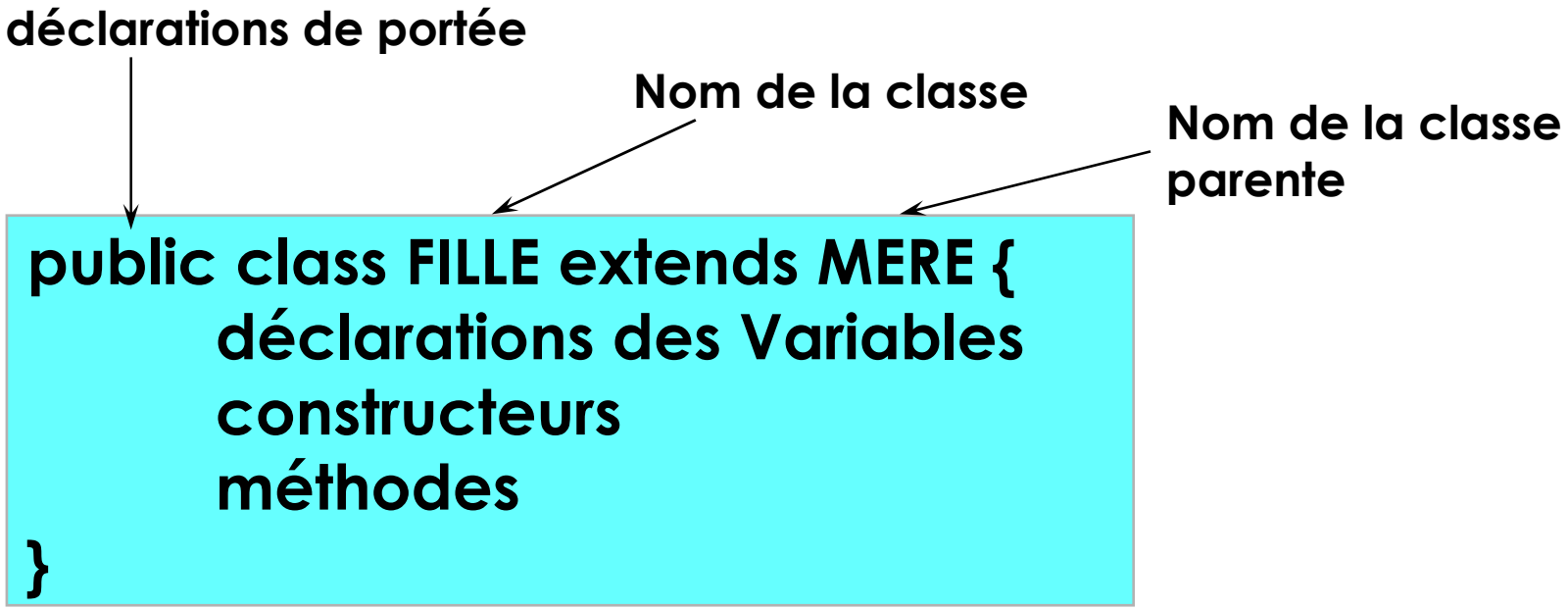
- une classe Fille hérite d'une classe Mère signifie :
  - qu'elle en reprend les attributs et les méthodes
  - qu'un objet de la classe Fille est aussi un objet de la classe Mere

Attributs et méthodes

- **la classe qui hérite (sous-classe), reprend les attributs et les méthodes de la surclasse**

- **mais, elle peut :**
  - **les enrichir de nouveaux attributs et de nouvelles méthodes**
  - **redéfinir les méthodes**

Anatomie d'une définition d'une classe



Exemple d'héritage

```
class Vehicule {
        int nbRoues;
        int vitesse;
        void Accelerer () { vitesse += 10; }
}

class Camion extends Vehicule {
        int poidsChargement ;
}
```

```
// deux appels valables
Vehicule unVehicule = new Vehicule () ;
unVehicule.Accelerer () ;
unVehicule = new Camion () ;
unVehicule.Accelerer () ;
```

Surcharge des méthodes

- le mot-clé "super" permet à une méthode d'une classe fille d'appeler la méthode surchargée de la classe mère
- exemple :

```
class Camion extends Vehicule {
    ....
    public void Voir() {
        super.Voir() ;
        system.out.println( chargeUtile ); // différentielle
    }
....
}
```

Constructeur de la classe Fille

- lorsqu'on construit une "Fille",
  - on appelle le constructeur de "Fille"
  - mais le constructeur de "Mère" est également invoqué implicitement ou explicitement
- implicitement si le constructeur de "Mère" ne prend pas de paramètres
- explicitement
  - dans ce cas on se sert du mot-clé "super"

    ex: super(param1, param2)
  - l'appel du constructeur de la classe "Mère" DOIT ETRE la première instruction du constructeur de la classe "Fille"

Constructeur de la classe Fille(2)

```
class Vehicule {
 private int imma;

  public Vehicule(int im) {
     imma = im ;
  }

  public Vehicule() {
     imma = 2;
  }
}
```

```
class Camion extends Vehicule {
 private int puissance;

 public Camion(int im, int puis) {
            super(im) ;
            puissance = puis;
    }
  public Camion() {
            puissance = 1600;
     }
}
```

Final

- Java a prévu un mécanisme pour interdire la redéfinition des méthodes

- le mot-clé "final" indique au compilateur qu'il est interdit de redéfinir une méthode

- exemple :

```
class Auto {
   int puissance;
   final int getPuissance() { return puissance ;}
   final void setPuissance(int p) { puissance = p; }
}
```

Final et Attribut Constant

- un attribut peut être "final"
  ce qui permet de définir une constante

- il doit être initialisé dans sa déclaration
  - exemple : final int tailleMax =12 ;
  - il ne peut être modifié ni par une méthode d'une sous-classe, ni par une méthode la classe elle-même

- si un attribut est "final static" , il est constant et accessible simplement en donnant le nom de la classe
  - class Mathematique {
    final static public float PI = 3.1415 ; ....
    }

Polymorphisme

Introduction

- **La création des classes permet de structurer l'environnement**

  mais il n'existe pas une seule façon de structurer l'environnement

  - **Pour un Enfant : un VELO est un JOUET**
  - **Pour un Adulte : un VELO est un véhicule**
  - **Pour un Industriel : un VELO est un PRODUIT**

- **Pour le concepteur, il ne s'agit pas de choisir la structure, la mieux adaptée à son problème mais de les faire COEXISTER**

Introduction (2)

- **Problème 1**
  - **« il est impossible d'additionner des pommes et des oranges »**
  - **Pourtant : 3 pommes + 5 oranges = 8 fruits**

- **Problème 2**
  - **« un objet peut appartenir à une structure ou à un autre suivant l'angle sous lequel on aborde le problème »**
  - **Un objet peut être à la fois un animal et un moyen de transport**
  - **Le bon sens nous dit que oui, mais la relation n'est pas la même dans les deux cas**

Sur-casting des Objets (1)

- **Arithmétique sur les fruits**
  - **Pour additionner des pommes et des oranges, il suffit de dire que nous manipulons des fruits :**

    3 fruits (qui se trouvent être des pommes)

    + 5 fruits (qui se trouvent être des oranges)

    = 8 fruits

  - **les pommes et les oranges doivent d'abord être transformées en fruit (SUR-CASTING)**

Sur-casting des Objets (2)

- **En java, nous pouvons écrire :**

  3(fruits)pommes

  + 5(fruits)oranges

  = 8 fruits

- **Autre façon de voir:**

  3 pommes

  + 5 oranges

  = ?

- **Laissons à l'interpréteur se poser la question :**
  - **« Peut-on effectuer l'opération ainsi ? »**

Sur Casting sur les Objets (3)

- **Puisque la réponse est NON, l'interpréteur devra effectuer automatiquement le sur-casting nécessaire pour produire le résultat**

- **Les deux approches sont-elles identiques ?**
  - **La première approche oblige à effectuer le sur-casting AVANT que l'opération soit posée !**

Sur-casting sur les Objets (4)

- **Première manifestation du Polymorphisme**
  - **La deuxième approche permet d'effectuer le sur-casting à la demande (JAVA)**

    « un objet peut-être considérer comme appartenant à sa classe ou à une autre classe PARENTE selon le besoin et cela de façon dynamique »

  - **Le lien entre une instance et une classe n'est pas unique et statique; il est établit au moment où l'objet est utilisé**

Retour sur l'initialisation (1)

- **Pour qu'un objet puisse être considéré comme une instance de la classe Pomme ou une instance de la classe Fruit, une condition est nécessaire :**
  - **« il doit être réellement une instance de chacune de ces classes »**

- **Il faut donc que, lors de l'initialisation, un objet de chaque classe soit créé**

Retour sur l'initialisation (2)

```
import java.io.*;
public class Polymorphisme {
   public static void main(String []argv) {
        Pomme pomme = new Pomme(85);
        Orange orange = new Orange(105);
        pese(orange);
   }
   static void pese(Fruit f) {
        int p = f.poids;
        System.out.println("Ce fruit pèse "+ p +" gr.");
   }
}
abstract class Fruit {
   int poids;
   Fruit() {
        System.out.println("création d'un fruit.");
   }
}
```

Retour sur l'initialisation (3)

```
class Pomme extends Fruit {
  Pomme(int p) {
       poids = p;
       System.out.println("Création d'une pomme.");
  }
}

class Orange extends Fruit {
  Orange(int p) {
       poids = p;
       System.out.println("Création d'une orange.");
  }
}
```

Retour sur l'initialisation (4)

- **Exécution du programme :**

```
Création d'un Fruit.
Création d'une Pomme.
Création d'un Fruit.
Création d'une Orange.
Ce fruit pèse 107 gr.
```

- **Remarques**
  - Avant de créer une pomme, le programme crée un fruit.
  - Dernière ligne du main : `pese(orange)`
    - La méthode est appelée avec un argument de type Orange, or
      il n'existe qu'une seule version de la méthode `pese` et elle prend un argument de type Fruit

Retour sur l'initialisation (5)

- **Remarques**
  - **Le message d'erreur habituel :**

    ```
    "incompatible type for method. Can't convert
      Orange to Fruit "
    ```

    n'apparaît pas !

  - **L'objet passé à la méthode est de type *Orange* MAIS également de type *Fruit***

  - **L'objet pointé par la *référence* orange peut être affecté sans problème au paramètre f qui est de type Fruit**

  - **Le fait d'établir le lien entre l'objet et la classe parente (fruit) est un sur-casting**

Le Sur-casting (1)

- L'objet pointé par la référence orange n'est en rien modifié par le sur-casting
- A l'appel de la méthode `pese`, l'objet pointé par la référence orange est passé à la méthode
- Le référence f pointe alors sur cet objet, f étant déclaré de type Fruit, Java vérifie si l'objet correspond à ce type.
- La référence orange continue de pointer vers l'objet orange qui est toujours de type Orange et Fruit
- Avantage : il n'existe qu'une seule méthode pese

Le Sur-casting (2)

- Version sans sur-casting

```
public class Polymorphisme2 {
        public static void main(String [] argv) {
                Pomme pomme = new Pomme(85);
                Orange orange = new Orange(105);
                pese( orange );
        }
        static void pese(Pomme f) {
          int p = f.poids;
          System.out.println("Ce fruit pèse "+ p +" gr.");
        }
        static void pese(Orange f) {
          int p = f.poids;
          System.out.println("Ce fruit pèse "+ p +" gr.");
        }
    }
```

- Critique : nous sommes obligés de créer et de maintenir DEUX méthodes ➔ Programmation Inefficace

Le Sur-casting (3)

- **Cette deuxième version se justifie SEULEMENT si nous voulons différencier les traitements selon le type du fruit**

```
public class Polymorphisme3 {
        public static void main(String [] argv) {
                Pomme pomme = new Pomme(85);
                Orange orange = new Orange(105);
                pese( orange );
        }
        static void pese(Pomme f) {
          int p = f.poids;
          System.out.println("La pomme pèse "+ p +" gr.");
        }
        static void pese(Orange f) {
          int p = f.poids;
          System.out.println("L'orange pèse "+ p +" gr.");
        }
  }
```

Sur-casting Explicite

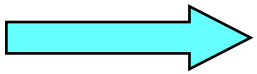
- Utilisation de l'opérateur de sur-casting
  - Méthode : `static String getPoids(Fruit f) { ..... }`
  - Utilisation : `String p = getPoids( (Fruit)pomme );`
    - L'utilisation de l'opérateur de sur-casting est inutile

- Affectation d'un objet à une référence de type différent
  - `Fruit pomme = new Pomme(85);`
    - La référence pomme est déclaré de type `Fruit`
    - Un objet de type `Pomme` est créé et affecté à la référence pomme
    - Ni la référence ni l'objet ne sont modifiés !!

Sur-casting implicite

- Le sur-casting est implicite lorsque aucune référence n'est là pour indiquer qu'il a lieu

- Exemple:
  - Les tableaux ne peuvent contenir que des références d'objet
  - L'affectation d'une référence d'objet à un tableau provoque un sur-casting vers le type Object



```
Fruit p = new Pomme(85);
Fruit o = new Orange(115);
Fruit[] sac = new Fruit[10];// tableau d'objets
sac[0] = p; // sur-casting
sac[1] = o;
```

Sous-casting

- La méthode `getPoids` possède un argument `f` référence de type Fruit.

- Est-il possible d'effectuer un traitement général puis ensuite un traitement particulier dépendant de l'objet pointé (pomme ou orange) ?

- Par exemple :

```
System.out.println( getNom( f ) );
```

sachant que deux méthodes de la classe Fruit `getNom` existent

```
static String getNom(Pomme f) {return "Pomme";}
static String getNom(Orange f) {return "Fruit";}
```

Le compilateur refuse le sous-casting !!!

Liaison précoce (early binding)

- **Définition de la liaison précoce**
  - **Lorsqu'un compilateur rencontre un appel de méthode, il doit être à même de savoir exactement de quelle méthode il s'agit**
- **Le lien entre l'appel et la méthode est établi au moment de la compilation**
- **Java utilise cette technique pour les appels de méthodes déclarées final**

Liaison tardive (late binding)

- Le compilateur établit le lien entre l'appel et la méthode au moment de l'exécution du programme)
- Ce lien est établi avec la méthode la plus spécifique de la méthode
- Dans la version finale de l'exemple, nous utilisons une méthode `getNom` dans chaque classe, au lieu d'une méthode statique
- La méthode appropriée sera utilisée en fonction de l'objet référencé par la référence de type Fruit

Résumé : Polymorphisme

- **Java supporte le polymorphisme par construction**
  - **les méthodes sont à liaisons tardives**
  - **les objets sont en fait des référence d'objets**

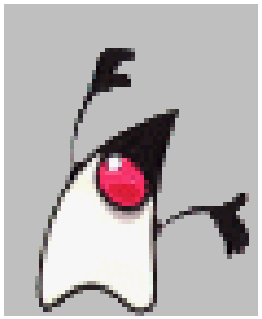
- **exemple :**

```
Vehicule garage [ ] = new Vehicule [2];
garage[0] = new Vehicule ();
garage[1] = new Camion ();

for (int i=0; i<2;i++)
      { garage[i].voir() ; }
```

Java sous vide

les Packages



Paquetage

- notion introduite par le langage ADA dans les années 1980
- représente un ensemble de classes travaillant conjointement sur le même domaine
- exemples :
  - liste et cellule de liste
  - arbre et noeud
  - dictionnaire et association
  - entrées-sorties (flux)

création et utilisation des paquetages

- un paquetage permet
  - regrouper syntaxiquement des classes qui vont ensemble conceptuellement
  - définir un niveau de protection pour les attributs et les méthodes plus fin que "tout ou rien"
- rappel :
  - les membres qui ont été déclarés amicaux (friendly) ne sont visibles qu'aux seuls autres membres du même paquetage

Construction de paquetage

- chaque classe appartient à un paquetage
  - par défaut : unnamed
- la classe est ajoutée au paquetage au moment de sa compilation
- la classe indique le paquetage auquel elle appartient par le biais du mot clé :

  package

Création d'un paquetage (2)

- **étape 1 : déclarer le paquetage**

  package monPaquetage ; // toujours 1ère ligne du fichier

```
import java.io.* ;
public class Banque { .... }
class Client { ... }
class Compte { ... }
```

- **étape 2 : après compilation**
  - **mettre les fichiers** .class **compilés dans un répertoire qui a le nom du paquetage**
  - **le groupe de fichiers : Banque.class, Client.class, ... sont placés dans un répertoire appelé** monPaquetage
  - **ce répertoire est soit un sous répertoire du répertoire de travail, soit accésible par le biais de l'instruction CLASSPATH**

Utilisation d'un paquetage

- **l'instruction d'importation signale à Java les classes qui sont utilisées**

- **syntaxes :**
  - **import monPaquetage.Banque;**
  - **import monPaquetage.*;**

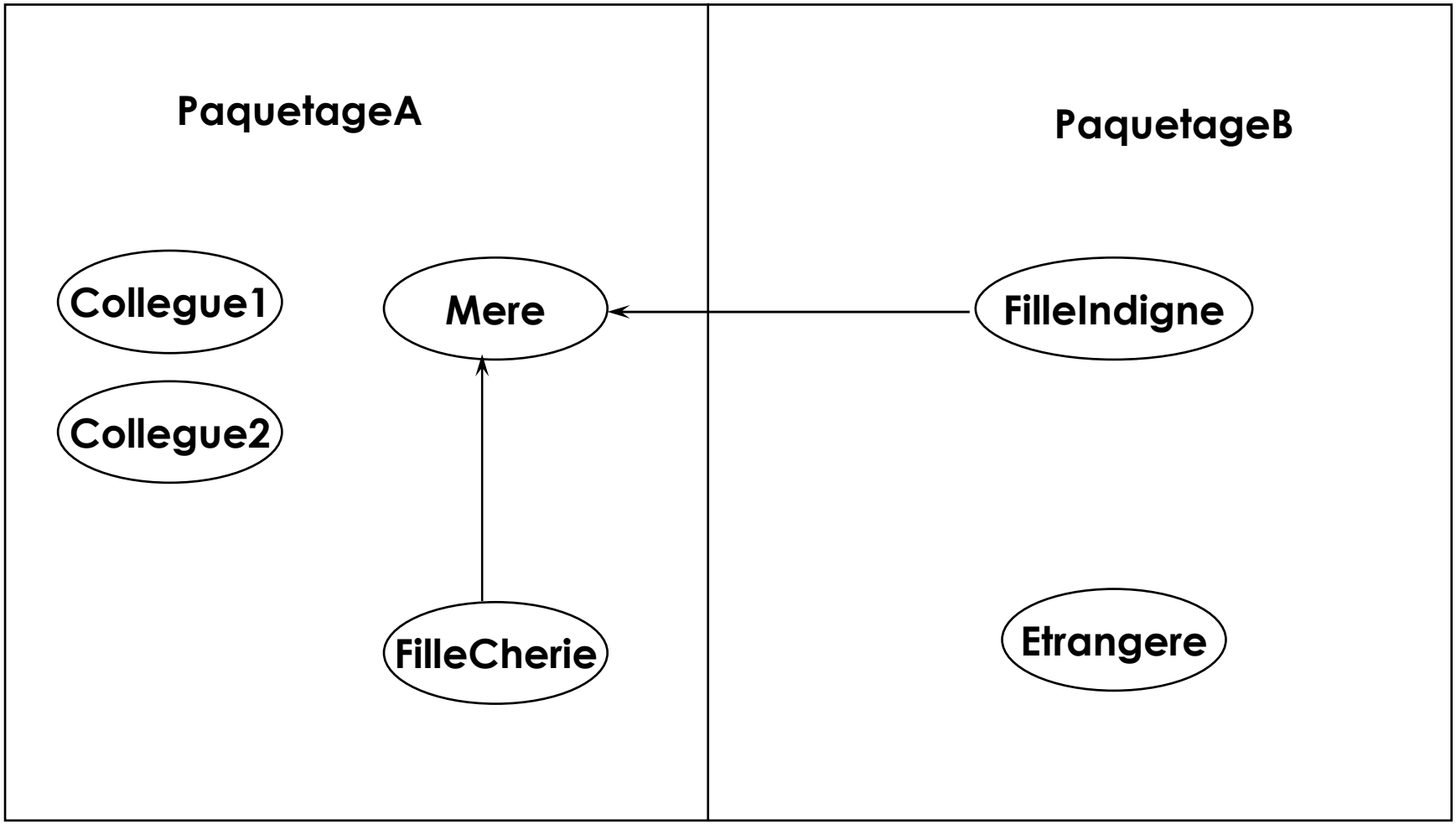
- **Remarque :**
  - **lors d'un conflit de nom entre 2 paquetages, le nom complet de la classe ou de la méthode doit être utilisé**

Paquetages de la bibliothèque Java

- java.lang
  - les classes fondamentales
- java.applet
- java.awt
  - gestion de l'interface
- java.io
  - flux
- java.net
  - E/S Internet de bas niveau
- java.util

Encapsulation des objets

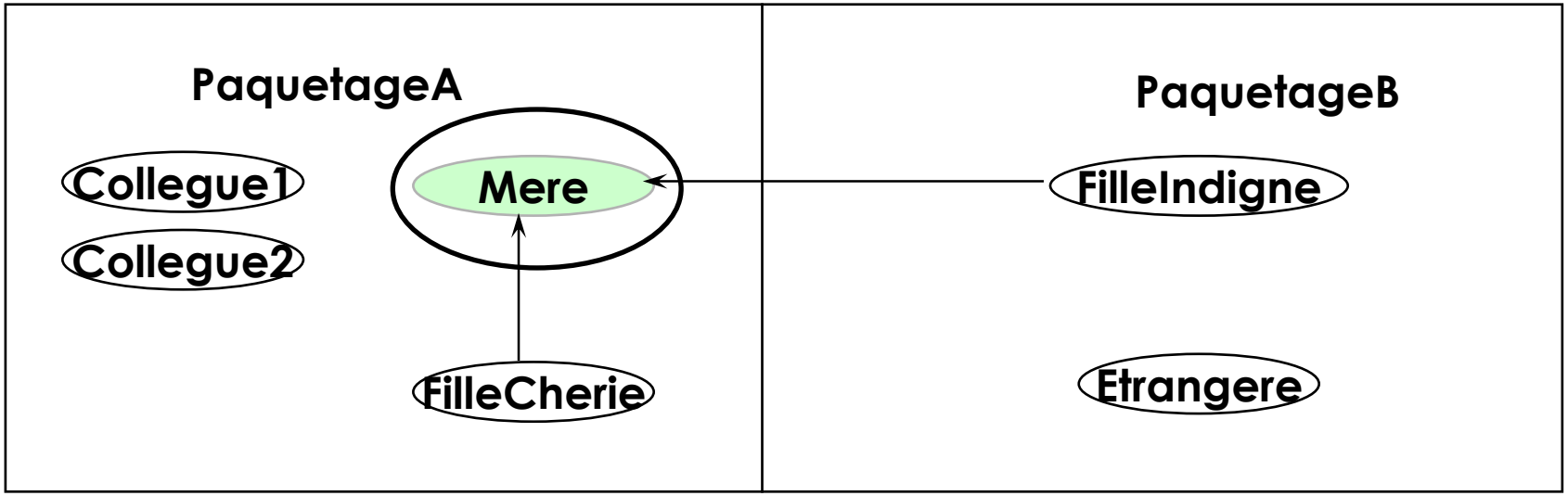
Relations possibles entre classes



Le respect de la vie privée

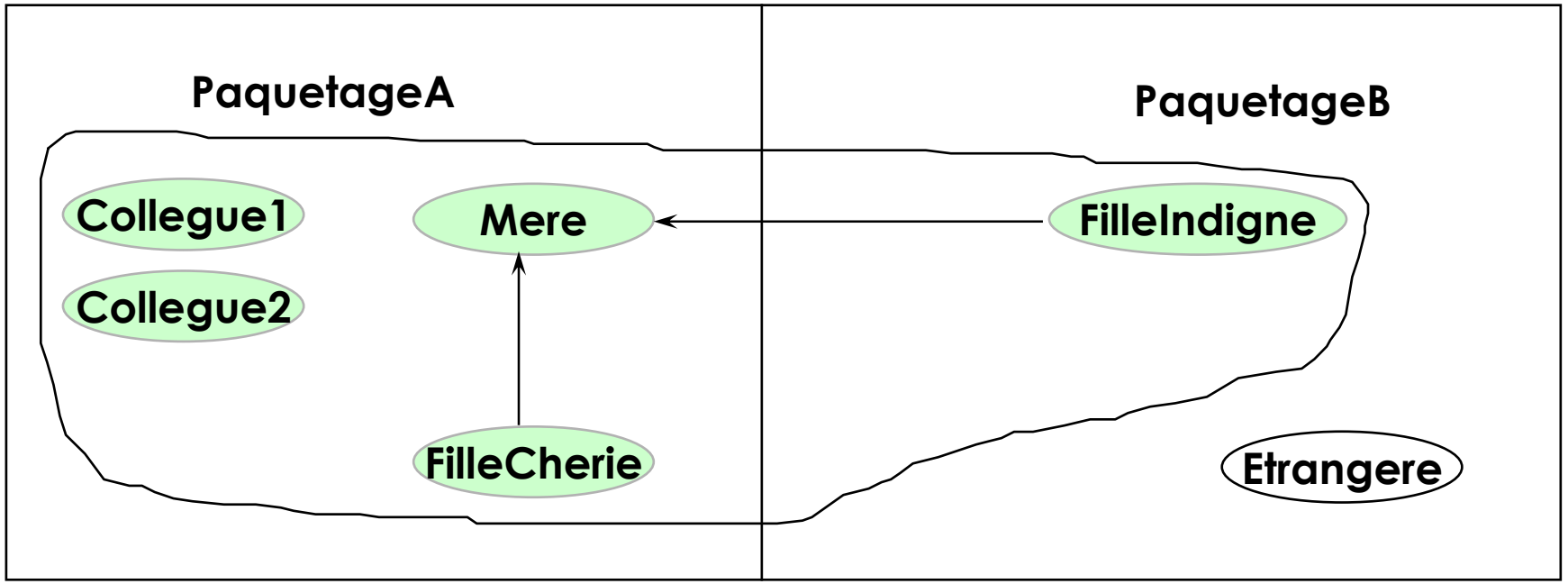
- protection "private"
  - elle empêche les objets d'autres classes d'accéder aux attributs ou aux méthodes de la classe considérée
  - exemple :

```
private void MethodeTresProtegee() { ... }
private int AttributTresProtegee ;
```



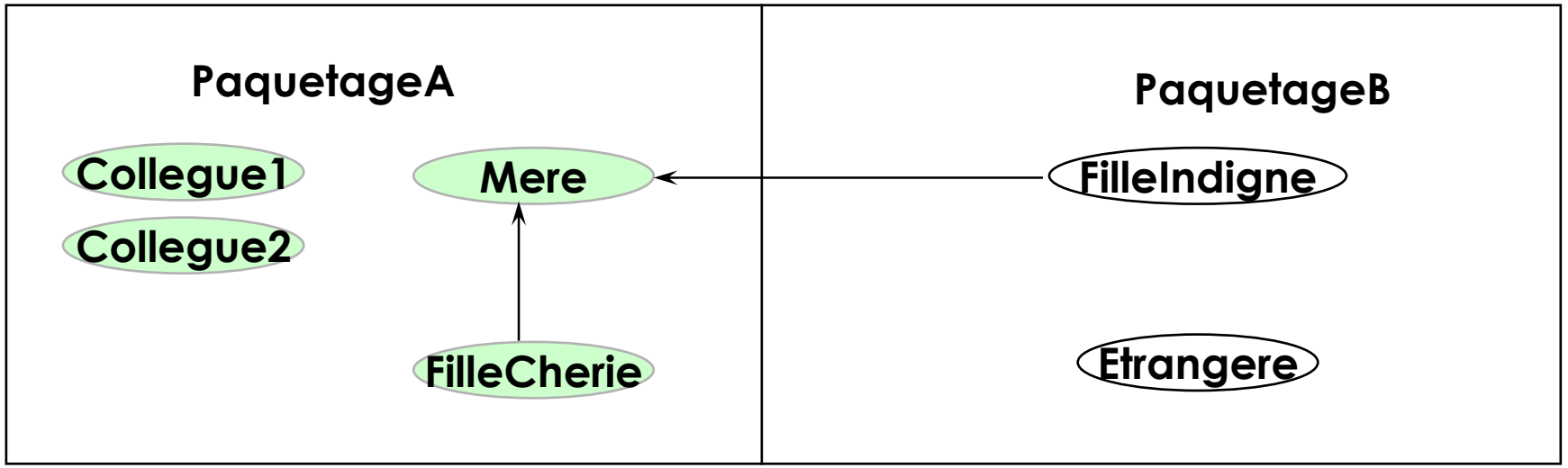
Les proches

- protection : "protected"
  - restreint l'accès aux sous-classes et aux classes du même paquetage



Les copains d'abord

- protection : "friendly"
  - le mot clé n'existe pas
  - protection par défaut
  - autorise l'accès aux classes du même paquetage



N'importe qui

- **protection : "public"**
  - **un attribut ou une méthode ainsi qualifiée est accessible à tout le monde**

Séparation de l'interface

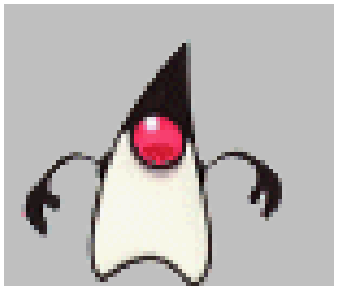
- **différents cas d'utilisation des mécanisme de protection**
  - **l'attribut ou la méthode appartient à l'interface de la classe : il ou elle doit être public**
  - **l'attribut ou la méthode appartient au corps de la classe : il ou elle doit être protégé(e)**
    la protection maximale est préférable

Résumé des accès possibles

Accès à partir de	private	protected	friendly	public
Mere	O	O	O	O
FilleCherie	N	O	O	O
Collegue	N	O	O	O
FilleIndigne	N	R	N	O
Etrangere	N	N	N	O

- O : accès possible
- N : accès impossible
- R : accès réservé aux sous-classes

Classes Abstraites et Interfaces



Méthode Abstraite

- **méthode abstraite :**

  méthode dont on donne la signature, mais sans en décrire l'implémentation

- **syntaxe :**

```
abstract void MethodeAbstraite(type par1, ...) ;
```

- **classe abstraite :**
  - **possède une méthode abstraite**
  - **doit être déclarée abstraite :**

```
abstract class Abstraite {
   abstract void MethodeAbstraite(type par1, ... ) ;
}
```

Intérêt des classes abstraites

- **sert à définir des concepts incomplets**
- **factorise des attributs et/ou des comportements communs à un ensemble de sous-classes**
- **sert de racine à une arborescence de classes et permet donc son évolution**

Classes Abstraites et Polymorphisme

- une classe ABSTRAITES ne peut être instanciée
- permet la gestion du polymorphisme :

```
abstract class Vehicule{
  abstract void Stopper() ;
}
class Voiture extends Vehicule {
  void stopper() { ..... }
}
class GardienDeLaPaix {
  void ArreterVehicule (Vehicule aux ) {
          aux.stopper();
}
```

Interface

- **Une interface est une classe :**
  - dont toutes les méthodes dont abstraites
  - dont tous les attributs sont final
  - les mots-clés abstract et final sont inutiles dans la déclaration d'une interface
  - exemple :

```
interface Vehicule {
  int nbRoues = 2;
  void methode() ;
}
```

- **une interface est une spécification formelle de classe**

  on précise ce que les sous classes doivent offrir comme service en leur laissant la liberté de l'implémentation

Implémentation d'une interface

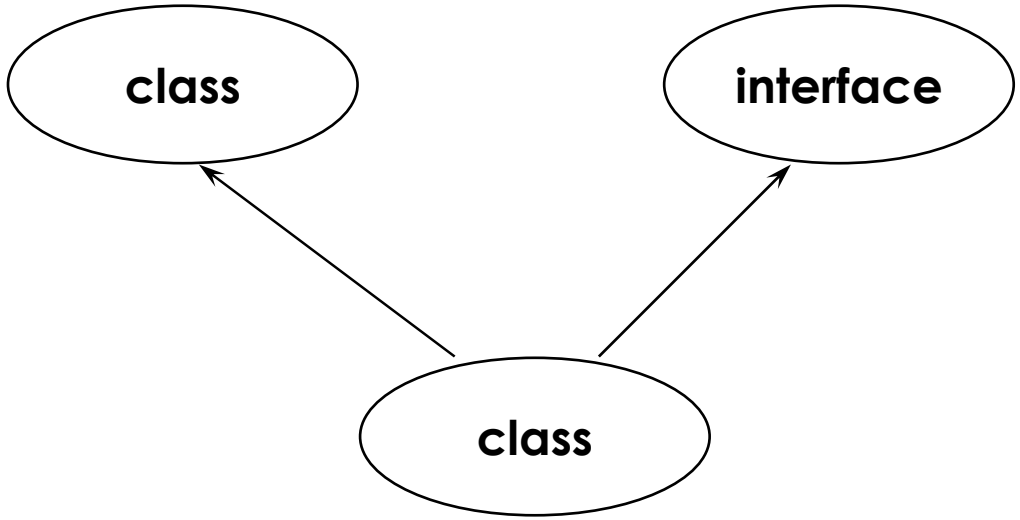
- **lorsqu'on hérite d'une interface, on l'implémente**

- **syntaxe :**

```
class Camion implements Vehicule { ..... }
```

Héritage multiple

- **Java interdit l'héritage multiple**
- **mais une classe peut hériter d'une classe et implémenter une interface**



Exemple

```
interface IPersistent {
   void Store) ;
   void Restore() ; }

class Employe {
   String nom;
   Employe( String s) { nom = new String(s) ;}
 }

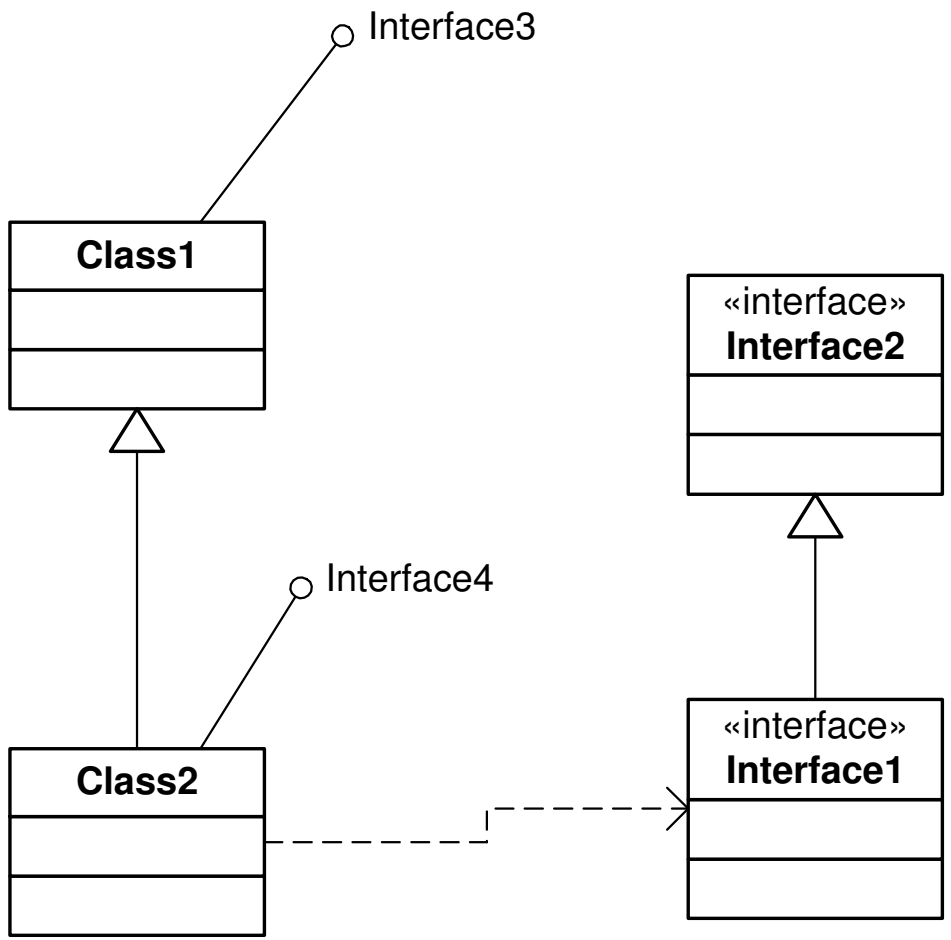
class PersistentEmploye extends Employe
                              implements Ipersistant {
..... }
```

Le polymorphisme et les interfaces

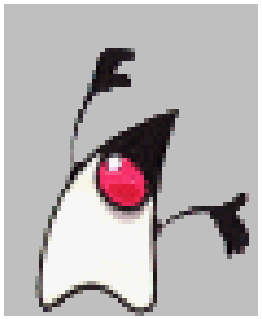
- **de même**
  - **qu'un objet est à la fois une instance de sa classe et de toutes ses classes parentes**
  - **un objet est reconnu comme une instance d'une quelconque de ses interfaces**

- **résumé un objet est une instance de**
  - **sa classe**
  - **toutes les classes parentes de sa classe**
  - **toutes les interfaces qu'il implémente**
  - **toutes les interfaces parentes des interfaces qu'il implémente**
  - **toutes les interfaces qu'implémentent les classes parentes de sa classe**
  - **toutes les interfaces parentes de précédentes**

le polymorphisme et les interfaces



Les Exceptions



Principe

- Un moyen de structurer le traitement des cas exceptionnels

- Principe:
  - une instruction génère une exception
    - une exception est un objet
  - l'exception se propage
    - vers les blocs englobants
    - vers les méthodes appelantes
  - l'exception est capturée

Définir une exception

- **Les exceptions sont des OBJETS**
  - **définies dans des classes**
  - **la classe mère de toutes les exceptions est *Throwable***
  - **seuls les objets définis par une sous-classe de *Throwable* peuvent être:**
    - **levés**
    - **et capturés**
  - **exemple :**

```
class GrandMere extends Exception {    };
class Mere extends Exception {    };
class Fille extends Exception {   };
```

Définir une exception (exemple)

```
class NbNegatifException extends Exception {
  private int errNb; // valeur de l 'entier erroné

  public NbNegatifException(int errNb) {
      this.errNb= errNb;  // memorise le contexte
  }

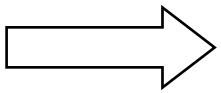
  public int getErrNb() { return errNb; }

  public String toString () {
      return new String(" nombre erroné : " + errNb);
  }
}
```

Lever une exception

- les exceptions peuvent être levées par :
  - le système,
  - une librairie que vous utilisez
  - vous-même (instruction *throw*)

- exemple :



```
if (nb < 0) {
  throw new NbNegatifException( nb ); }

  // suite du programme ssi le nombre est positif
```

Capturer une exception

- les exceptions sont traitées dans les blocs *catch*
- un bloc *catch* ne peut se situer qu'à la suite d 'un bloc *try*
- un bloc *try* est toujours suivi d 'un ou de plusieurs blocs *catch*
- exemple :

```
try { … }
catch (NbNegatifException e) { … traiter e … }
catch ( AutreException e) { … }
```

Capturer une exception : exemples

- Conversion String -> Entier

```
String msg;
int parInt;
msg = getParameter("LONGUEUR");
  // essaye de "lire" le paramètre comme un entier
try parInt = Integer.parseInt(msg);
 // en cas d'erreur met à -1
catch (NumberFormatException e) parInt = -1;
```

- Accès contrôlé à un tableau

```
try nbre = tab[i];
catch (ArrayIndexOutOfBounds b) {
  System.out.println(b.getMessage());
  nbre = 0;
}
```

Capturer une exception (2)

- **l 'ordre des *catch* est important,**

  lorsqu'une exception remonte, le premier bloc *catch* susceptible de la traiter est celui qui est exécuté au détriment des autres

- **exemple :**

```
class GrandMere extends Exception { … };
class Mere extends GrandMere { … };
class Fille extends Mere { …. }

class Loup extends Exception { …. }
```

Capturer une exception (3)

```
try {
   // sequence de code pouvant lever Grandmere, Mere, Fille
}
catch (Mere  m) {
   // m est l 'exception créée par un throw
   System.out.println(m.toString() ); }
catch (GrandMere gm ) { ….. }
catch (Fille f) {  … }
```

Capturer une exception (4)

- **Si une exception de type GrandMere est levée, elle ne pourra pas être capturée par le premier bloc *catch*, car GrandMere est une sur-classe de Mere.**

  L 'exception sera traitée dans le second bloc

- **Si une exception de type Mere est levée, elle sera traitée par le premier bloc. Elle ne sera pas traitée dans le second bloc, car elle a déjà été traitée**

- **Si une exception de type Fille est levée, elle sera traitée dans le PREMIER BLOC, le dernier bloc *catch* ne sert à rien**

Capturer une exception (5)

- l 'ordre exact des blocs doit être le suivant :

```
try {

   // sequence de code pouvant lever Grandmere,Mere, Fille

}
catch (Fille f) {
   … }
catch (Mere  m) {
   …  }
catch (GrandMere gm ) {
   …  }
```

- depuis la classe la plus spécialisée (Fille) vers la classe la plus générale (Mere)

Finaliser le traitement (1)

- **Java permet de permettre l'exécution d'une portion de code quoi qu 'il arrive.**
  - **ce code est décrit dans le bloc *finally***
  - **ce code s 'exécute après le bloc *try*, et après un éventuel bloc *catch***

- **cela couvre les cas suivants :**
  - **le bloc try s 'exécute normalement,**
  - **le bloc try s 'exécute et lève une exception attrapée dans le bloc catch**
  - **le bloc try s 'exécute et lève une exception qui n 'est attrapée dans aucun des blocs catch qui le suit**

Finaliser le traitement (2)

- **intérêt du bloc *finally***
  - **factoriser du code qui aurait dû être dupliqué sinon**
  - **permet d 'effectuer des traitements après un bloc *try* même si une exception a été levé et non capturée dans les blocs *catch* suivant le bloc *try***
- **exemple :**

```
try {
      … ouvrir une fichier }
catch ( Mere m) {
      … traite l 'exception m }
finally {
   … fermer le fichier }
```

Mécanisme des exceptions

- une exception est un objet qui est instancié lors d 'un incident : on dit que l 'exception est *levée*
- le traitement du code de la méthode est interrompu et l 'exception est *propagée* à travers la pile d 'exécution de méthode appelée en méthode appelante
- si aucune méthode ne *capture* l 'exception, celle-ci remonte jusqu 'à la méthode du fond de la pile d 'exécution : l 'exécution se termine par une indication d 'erreur

Déclarer une exception

- une méthode doit comporter dans sa signature l 'ensemble des exceptions dont le soulévement est à sa portée et qui ne sont pas traitées dans ses blocs *catch*
- c 'est à dire
  - des exceptions qui sont levées dans la méthode,
  - des exceptions qui sont levées dans les méthodes appelés par la méthode courante
- cette déclaration se fait par l 'emploi de *throws*

Déclarer une exception (2)

- exemple :

```
void MéthodeQuelconque() throws Loup {
   try {
      // ouvrir le fichier
      // effectuer des traitements qui
      // lévent une exception Mere ou Loup
   }
   catch(Mere m) {
      …
   }
   finally {
      fermer le fichier    }
```

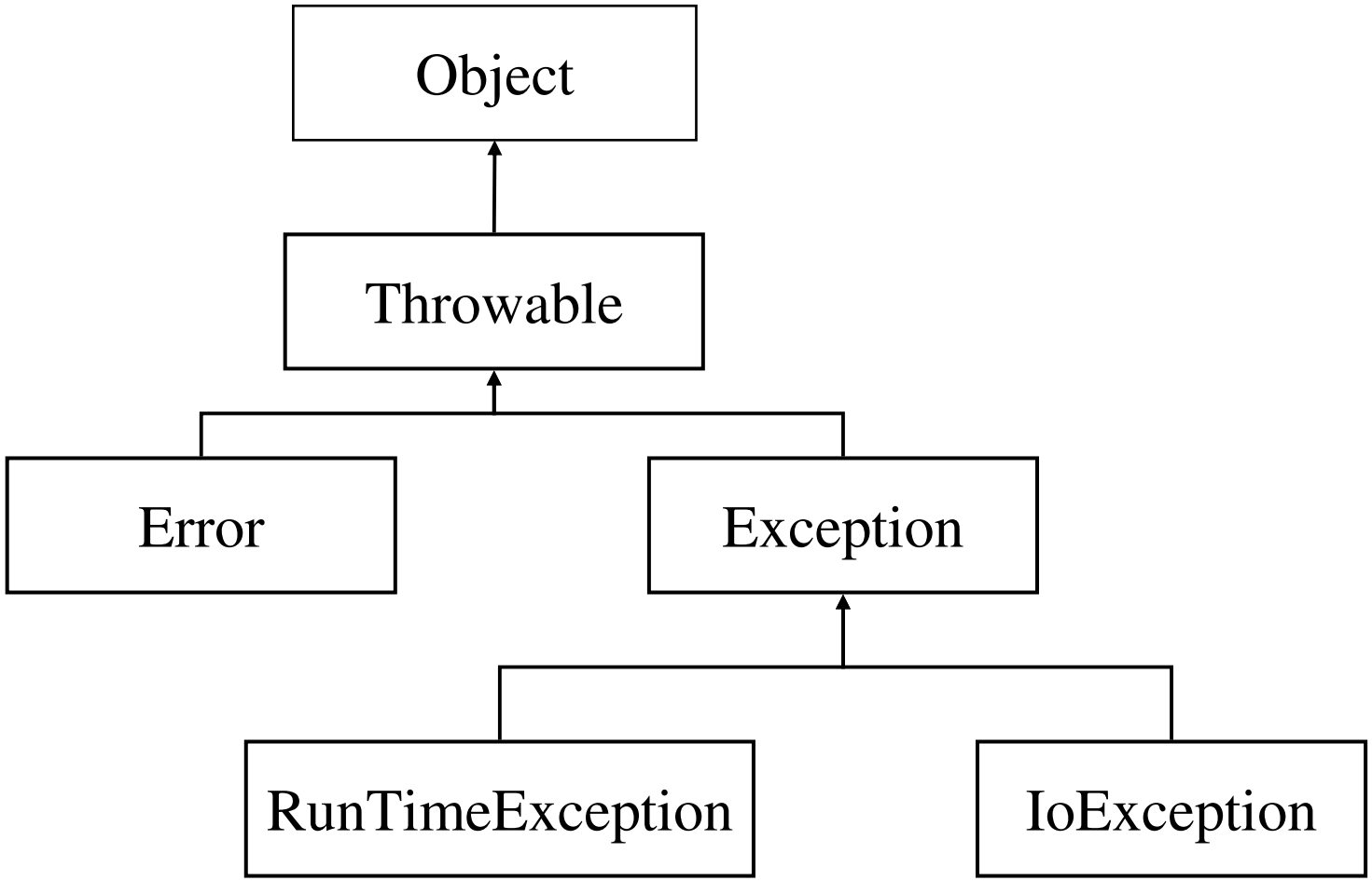
Déclarer une exception (3)

- **nous avons deux contraintes contradictoires**
  - **nous voulons que dans les méthodes soient déclarées toutes les exceptions à portée de la méthode et non traitées par celle-ci**
  - **nous ne voulons pas déclarer systématiquement les exceptions de base**
- **la solution est basée sur l 'utilisation de l 'héritage**

les différents types d 'exception

- **toutes les exceptions sous-classes directes ou indirectes de la classe *RuntimeException* n 'ont pas à être déclarées**
- **il y a trois types d'exception :**
  - **les *Error* : vous pouvez les traiter , mais c 'est tout**
  - **les *RuntimeException* : vous pouvez les traiter et les lever**
  - **les autres exceptions : vous pouvez les définir, les lever et les capturer, mais vous devrez les déclarer si vous ne les traitez pas**

Hiérarchie des classes d 'exception



Le bon usage des exceptions

- Les exceptions doivent rester … exceptionnelles
- Ce n'est pas une nouvelle structure de contrôle en plus desif, while, for, switch
- Pour éclaircir les traitements d'erreur du genre

```
if (erreur) <trait-erreur>
else {...
          if (erreur) <trait-erreur>
          else {...
                    if (erreur) <trait-erreur>
                    else {...
                              (suite)
                    }
          }
  }
```

- Eviter de capter les erreurs graves non récupérables (NullPointerException, …)

Egalité d'objets

Egalité de deux objets

- **nécessité**
  - **d'avoir une méthode qui teste l'égalité de deux objets en se basant sur leur contenu**
- **exemple : classe Fraction**
  - **deux fractions sont égales si leur numérateur et leur dénominateur le sont**
  - **MAIS 1/2 = 2/4 = 3/6, etc.**
  - **il faut donc écrire une méthode equals() spécifique pour les fractions**

Introduction

- La classe `Object` propose la méthode `equals()` afin de comparer la valeur de deux instances
- Par défaut, l'implémentation retourne `true` si le paramètre est égal à `this`
- Mais deux instances différentes peuvent avoir également la même valeur
- classe Fraction

```
public boolean equals( Fraction f) {
  return ( this.numerateur * f.denominateur ==
           this.denominateur * f.numerateur );
}
```

Références d'objets

- dans le cas où les variables d'instances sont des références d'objet, la méthode equals doit en tenir compte et décider d'appliquer
  - soit le test d'identité
  - soit le test d'égalité sur ces variables

- dans le premier cas : égalité de surface
- dans le second cas : égalité profonde

Propriétés de l'égalité

- **l'opération equals() se doit d'être:**
  - **symétrique**
    - **a.equals(b) et b.equals(a) doivent toujours donner le même résultat**
  - **réflexive**
    - **a.equals(a) doit toujours donner `true`**
  - **transitivité**
    - **si a.equals(b) et b.equals(c) donnent `true` alors a.equals(c) doit donner true**

Spécialisation

- **De plus, Il faut que la comparaison fonctionne malgré l'héritage**

- **A titre d'illustration, nous allons choisir le modèle objet suivant :**
  - **une classe Personne**
  - **une classe Employé spécialisation de Personne**

exemple (2)

- Un employé est une sorte de Personne
  - Tous les traitements possibles avec une Personne doivent être validés avec un Employé
  - Il est possible de comparer si deux instances Personne ont les mêmes valeurs, il doit donc être possible de comparer une Personne et un Employé
  - Si une Personne possède les mêmes valeurs que l'Employé au niveau des attributs de Personne (nom, prénom,…), la méthode equals() peut retourner true.

Stratégies de comparaison

- **En fait, il existe deux stratégies de comparaison**

- **approche rigide**
  - **comparer deux instances, c'est comparer l'union des attributs de chaque instance. Si l'une d'elles ne possède pas un des attributs à comparer, l'égalité n'est pas résolue**

- **approche souple**
  - **comparer deux instances de types différents peut consister à comparer uniquement l'intersection des attributs de chaque instance**

Approche rigide

- Puisqu'il n'est pas possible de bénéficier de la méthode Personne.equals() dans une sous-classe, il est préférable de découper la méthode en deux pour différencier le test des attributs et la vérification de la classe

- exemple :

```
public class Personne {
  protected final boolean egalite (Personne p) {
    return (nom.equals(p.nom)) && (prenom.equals(p.prenom));
  }

  public boolean equals(Object x) {
    if( ( x!=null) && ( x.getClass() == Personne.class) )
            { return egalite( (Personne)x );}
    return false;
  }
}
```

Approche rigide (2)

- La méthode equals() version rigide de la classe Employe peut alors être rédigée ainsi:

```
public class Employe extends Personne {
  protected final boolean egalite( Employe e) {
    return ( (super.egalite(e) ) && (salaire == e.salaire)));
  }

  public boolean equals( Object x) {
    if( ( x!=null) && ( x.getClass() == Employe.class) ) {
            return egalite( (Employe)x );
    }
    return false;
  }
}
```

Approche souple (1)

- La méthode equals(), version souple de la classe Personne, peut être rédigée de la manière suivante :

```
public class Personne {
  public boolean equals(Object x) {
    if( x instanceof Personne) {
            Personne p = (Personne) x;
            return  (nom.equals(p.nom)) &&
                    (prenom.equals(p.prenom));
    }
    return false;
  }
  ...
}
```

Approche souple (2)

- La méthode equals() de la classe Employe doit tenir compte de l'héritage

```
public class Employe extend Personne {
  public boolean equals(Object x) {
    boolean rc = super.equals(x);

    if( (rc) && (x instanceof Employe)) {
            Employe e = (Employe) x;
            return  ( salaire == e.salaire);
    }
    return rc;
  }
  ...
}
```

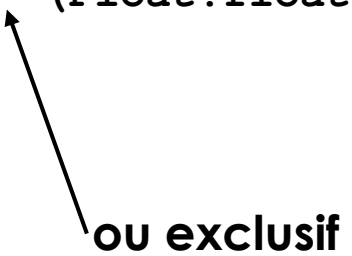
la valeur de Hash

- La classe Object propose la méthode hashCode() qui doit être redéfinie si la méthode equals l'est car il ne peut exister deux valeurs de hash pour deux instances égales.
- Pour la classe Personne

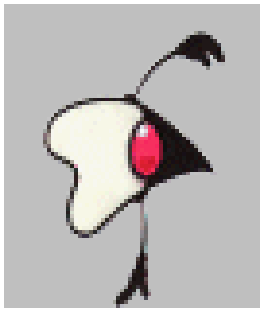
```
public int hashCode() {
  return ( nom.hashCode() ^ prenom.hashCode()) ; // exemple
}
```

- et pour la classe Employe

```
public int hashCode() {
  return (super.hashCode()) ^ (Float.floatToIntBits(salaire));
}
```



Gestion des collections



SDK 1.0 et SDK 1.1

Gestion des Collections

- **Dans les SDK 10 et 1.1 , le langage Java implémente dans le paquetage java.util, quelques classes *containers***

- **Ces classes sont :**
  - **Les listes : `Vector` et `Stack`**
  - **Les tables : `Dictionary`, `Hashtable` et `Properties`**

- **Par ailleurs, le langage Java permet de définir des itérateurs sur une classe container grâce à la classe `Enumeration`**

La classe Vector

- La classe *`Vector`*, que l'on trouve dans le package *`java.util`*, permet de représentation des collections polymorphes (listes).

- un objet de type `Vector` constitue un tableau dynamique d'éléments dont la taille évolue en fonction des besoins

- Les éléments placés dans des `Vector` sont obligatoirement des objets, d'où la nécessiter d'utiliser les classes enveloppes pour les types primitifs (`Integer`, `Double`, ...)

- quand un indice spécifié en argument est incorrect, une exception de type `ArrayIndexOutBoundsException` est levée

Méthodes de la classe Vector

Méthode	But
`addElement( Object )`	ajoute l'élément spécifié comme dernier élément
`setElementAt(Object,int)`	change l'élément de l'index spécifié pour être l'objet spécifié
`elementAt(int)`	retourne l'élément de l'index spécifié
`indexOf(Object)`	cherche l'élément spécifié, en partant de la première position, et retourne son index
`contains(Object)`	retourne `true` si l'objet spécifié est dans la collection (utilise la méthode `equals`)
`removeElement(Object)`	enlève l'élément spécifié du Vector
`removeAllElements()`	enlève tous les éléments du Vector
`size()`	retourne le nombre d'éléments du Vector
`elements()`	retourne un énumération des éléments

Itération sur une collection

- parcours d'un vecteur
  - La méthode *elements*( ) de la classe *Vector* retourne une énumération permettant d'appliquer une action sur chaque éléments de la collection
  - Dans l'exemple suivant, l'objet mesMalades est une instance de la classe Vector contenant des instances de la classe utilisateur Malade

```
Enumeration e = mesMalades.elements();
while( e.hasMoreElements() )
{
  s = (Malade) e.nextElement( ) ;
  s.voir();
}
```

Itération sur une collection

- **remarque sur l'exemple**

`Enumeration e = mesMalades.elements();`	**initialisation de l'itérateur**
`e.hasMoreElements()`	**test d'arrêt**
`e.nextElement()`	**accès à l'élément puis incrémentation**

La classe Hashtable

- définition
  - La classe *Hashtable*, que l'on trouve dans le package *java.util*, permet de représentation des tables polymorphes
  - Une table est un ensemble de couple clé – valeur
  - Tout objet peut être clé ou valeur; mais les opérations de recherche et d'ajout dans un dictionnaire nécessitent que la classe Clé surchargent les méthodes :
    - `hashCode`()
    - `equals`()

Méthodes de la classe Hashtable

Méthode	But
`put(Object c, Object v)`	ajoute le couple (c,v) dans la table (clé : c, valeur : v)
`remove(Object c)`	supprime l'élément de clé c
`get(Object c)`	retourne l'élément dont la clé est c ou null
`containsKey(Object c)`	retourne true si la clé est présente
`contains(Object)`	retourne true si la valeur est présente
`isEmpty()`	retourne true si la table est vide
`clear()`	enlève tous les éléments
`size()`	retourne le nombre d'éléments
`keys()`	retourne une énumération des clés
`elements()`	retourne une énumération des éléments

Exemple d'utilisation de Hashtable()

```
Hashtable dico = new Hashtable(); // appel au constructeur
dico.put("Zola", "Germinal");     // ajout d'une œuvre
dico.put("Flaubert", "Madame Bovary");

System.out.println( (String)dico.get("Zola") );
    // affiche Germinal

// afficher les auteurs
Enumeration e = dico.keys();
while( e.hasMoreElements() )  {
  a = (String) e.nextElement( ) ; // un auteur
  System.out.println(a);
}

// afficher les œuvres
Enumeration e = dico.elements;
while(  e.hasMoreElements() ) {
  l = (String) e.nextElement( ) ; // un livre
  System.out.println(l);
}
```

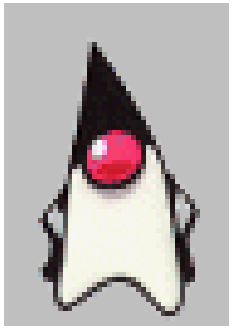
Dictionnaire sécurisé

- **Méthode**
  - **typer la clé et la valeur qui ne sont que des Objets dans la classe Hashtable afin de sécuriser le dictionnaire**

- **Exemple :**

```
public class EnsPersonne extends Hashtable {
    public boolean containsKey( String n) {
           return super.containsKey(n); }
    public void put( String c, Personne p) {
           super.put(c,n);   }
    public Personne get(String c) {
           return (Personne) super.get(c);}
    public voir remove(String n) {
           super.remove(n); }
}
```

Applet Java



Applets et applications

- **avantages des Applets**
  - **s'exécutent dans le navigateur**
  - **possèdent les mêmes caractéristiques que lui**
    - **packages graphiques, dessins et trait. d'images**
    - **élément d'interface utilisateur**
    - **gestion du réseau**
    - **gestion des événements**
- **inconvénients des Applets**
  - **aucun accès aux fichiers de l'utilisateur**
  - **ne communiquent qu'avec le serveur dont elles sont issues**
  - **ne peuvent exécuter de programme dans le système client**
  - **ne peuvent charger un prog. natif de la plate-forme locale (ex : les bib. partagées DLL)**

Création des applets

- créer une sous-classe d'Applet du package java.applet
- la classe Applet offre des fonctions utiles pour fonctionner avec :
  - le navigateur
  - AWT
    - gérer la souris
    - éléments de l'Interface Homme Machine (IHM)
    - touches de fonctions
    - dessiner à l'écran
- signature :

```
public class maClasse extends Applet
```

Création d'une applet (2)

- **quand JAVA détecte une applet dans une page WEB, il charge**
  - **la classe applet initiale à partir du réseau**
  - **toutes les classes annexes utilisée par elle**

Activités principales d'une applet

- une applet a de nombreuses activités différentes qui correspondent à divers événements importants de leur cycle de vie
- événements :
  - initialisation
  - dessin
  - actions souris
- A chaque activité correspond une méthode
  Lorsque l'événement intervient, Java appelle ces méthodes spécifiques
- ces méthodes d'activités ne possèdent aucune définition par défaut
  il faut donc PARFOIS les surcharger

Initialisation

- s'effectue au moment du chargement de l'applet
- cette opération comprend :
  - la création des objets nécessaires
  - la mise en Œuvre d'un état initial
  - le chargement d'images ou de polices
  - l'initialisation de paramètres
- syntaxe :

```
public void init () { ...  }
```

Démarrage

- après son initialisation, l'applet se lance
- idem, lorsqu'une applet a été interrompue (ex: l' utilisateur choisit un lien vers une autre page)
- une applet peut être lancée plusieurs fois pendant sa vie mais n'est initialisée qu'une fois
- syntaxe :

```
public void start() {  ..... }
```

Arrêt

- l'arrêt intervient quand l'utilisateur quitte la page dans laquelle l'applet s'exécute.
- si l'utilisateur quitte une page, l'applet tourne toujours et mobilise des ressources système (défaut)
- mais la méthode peut être surchargée
- syntaxe :

```
public void stop() { .... }
```

Destruction

- la destruction permet à l'applet de libérer ses ressources avant son interruption ou celle du navigateur.
- syntaxe :

```
public void destroy() { ... }
```

Dessin

- permet à l'applet de dessiner à l'écran
  - texte
  - ligne
  - arrière-plan en couleur
  - image
- syntaxe :

```
public void paint ( Graphics g) { ... }
```

paint reçoit comme argument une instance de la classe Graphics, donc il faut importer le package : java.awt.graphics

Applet simple

```
import java.awt.Graphics;
import java.awt.Font;
import java.awt.Color;

public class HelloApplet extends java.applet.Applet {
        Font f = new Font("TimesRoman",Font.BOLD,36);

        public void paint (Graphics g) {
                g.setFont(f);
                g.setColor(Color.red);
                g.drawString("Hello again!", 5,, 50);
        }
}
```

Inclusion d'une applet dans une page Web

```
<HTML>
 <HEAD>
        <TITLE> Cette page contient une applet </TITLE>
 </HEAD>
 <BODY>
        <P>Ma premiere applet Java dit :
        <BR>
        <APPLET CODE ="HelloApplet.class" WIDTH=200 HEIGHT=50>
            Il aurait dû y avoir une applet ici si votre navigateur
                supportait Java
        </APPLET>
 </BODY>
</HTML>
```

Transmission de paramètres aux applets

- une applet peut récupérer les entrées du fichier HTML qui contient l'étiquette <APPLET>
- pour gérer des paramètres dans une applet, il faut :
  - une étiquette paramètre spéciale dans le fichier HTML
  - coder dans l'applet l'analyse de ces paramètres
- dans le fichier HTML, les paramètres sont définis par l'étiquette <PARAM>.
  elles a deux attributs :
  - NAME pour le nom
  - VALUE pour la valeur

Transmission de paramètres aux applets (2)

```
<APPLET CODE="Ma seconde applet> WIDTH=200 HEIGHT=50>
<PARAM NAME=font VALUE="TimesRoman">
<PARAM NAME=taille VALUE="36">
```

- ces param. sont transmis à l'applet au moment du chargement
- la méthodes getParameter() dans init() les récupère.
  - elle reçoit un argument (une chaîne qui contient le nom du param. recherché)
  - et retourne sa valeur dans une chaîne

Transmission de paramètres aux applets (3)

```
import java.awt.*;

public class HelloApplet extends java.applet.Applet {
        Font f = new Font("TimesRoman",Font.BOLD,36);
        int taille;
        public void init() {
                String s = getParameter("taille");
                if (s == null) taille =12;
                else taille = Integer.parseInt(s);
                s = getParameter("font");
                if (s ==null) s = "Courrier";
                f = new Font(s,Font.BOLD,taille);
        }
        public void paint (Graphics g) {
                g.setFont(f);
                g.setColor(Color.red);
                g.drawString("Hello again!", 5, 50);
        }
}
```