**http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/CoursJava**

***Cours java : table des matières***

|  |  |
| --- | --- |
| * [introduction](http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/CoursJava/Introduction.html) * [types primitifs](http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/CoursJava/TypeDeDonneePrimitifs.html) * [instructions](http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/CoursJava/Instructions.html) * [tableaux](http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/CoursJava/tableaux.html)   + la classe [Arrays](http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/CoursJava/Arrays.html) * [chaînes de caractères](http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/CoursJava/ChainesDeCaracteres.html) * [classes](http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/CoursJava/classes1.html)   + [La classe «Class»](http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/CoursJava/classeClass.html)   + [Classes «enveloppes»](http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/CoursJava/enveloppes.html)   + [Classes *BigInteger* et *BigDecimal*](http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/CoursJava/lesBig.html)   + exemple 1 [Temps](http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/CoursJava/Temps.html)   + exemple 2 [Date](http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/CoursJava/Date.html)   + exemple 3 [Chaine de caractères](http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/CoursJava/Chaine.html) * classes : [héritage](http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/CoursJava/classes2.html) * classes : [interne, locale et anonyme](http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/CoursJava/classes3.html) * [exceptions](http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/CoursJava/exceptions.html) * [paquetage](http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/CoursJava/paquetage.html) * [Interface](http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/CoursJava/interface.html) * [Flux](http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/Flux/Flux.html)   + [Flux d'octets en lecture](http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/Flux/FluxOctetsLectureHierarchie.html) (image cliquable)   + [Flux d'octets en écriture](http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/Flux/FluxOctetsEcritureHierarchie.html) (image cliquable)   + [Flux de caractères en lecture](http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/Flux/FluxCaract%E8resLectureHierarchie.html) (image cliquable)   + [Flux de caractères en écriture](http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/Flux/FluxCaract%25E8resEcritureHierarchie.html) (image cliquable)   + la classe [Scanner](http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/Flux/Scanner.html)   + la classe [RandomAccessFile](http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/Flux/RandomAccessFile.html) * la classe [File](http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/Flux/File.html) * [Généricité](http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/Conteneurs/genericite.html) * [Internationalisation/localisation](http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/CoursJava/internationalisation.html) des applications | * Conteneurs   + [Hiérarchie Collection](http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/Conteneurs/hierarchie.html) (image cliquable)   + [Hiérarchie Map](http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/Conteneurs/hierarchieMap.html) (image cliquable)   + Les interfaces :     - [Iterable](http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/Conteneurs/Iterable.html)     - [Collection](http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/Conteneurs/Collection.html)     - [List](http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/Conteneurs/List.html)     - [Set](http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/Conteneurs/Set.html)     - [SortedSet](http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/Conteneurs/SortedSet.html)     - [Iterator](http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/Conteneurs/Iterator.html)     - [ListIterator](http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/Conteneurs/ListIterator.html)   + Les classes abstraites :     - [AbstractCollection](http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/Conteneurs/AbstractCollection.html)     - [AbstractList](http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/Conteneurs/AbstractList.html)     - [AbstractSet](http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/Conteneurs/AbstractSet.html)     - [AbstractSequentialList](http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/Conteneurs/AbstractSequentialList.html)   + la classe [ArrayList](http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/Conteneurs/ArrayList.html)   + la classe [LinkedList](http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/Conteneurs/LinkedList.html)   + Arbres ( [application de démonstration](http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/Applis/baobab.jnlp) )     - [généralités](http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/CoursJava/arbres.html)     - [arbres binaires ordonnés](http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/CoursJava/arbresBinaires.html)     - rééquilibrage des arbres binaires ordonnés       * [Arbres AVL](http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/CoursJava/arbresAVL.html)       * [Arbres rouge-noir](http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/CoursJava/arbresRougeNoir.html)       * [Arbres 2-3-4](http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/CoursJava/arbres234.html)     - [TreeSet](http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/Conteneurs/TreeSet.html) et [TreeMap](http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/Conteneurs/TreeMap.html)   + Hachage     - [Table de hachage](http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/Conteneurs/TableDeHachage.html)     - [HashMap](http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/Conteneurs/HashMap.html)     - [HashSet](http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/Conteneurs/HashSet.html)   + Comparaison des performances [Hash vs Tree](http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/Conteneurs/testHashTree.html)   + la classe [Collections](http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/Conteneurs/Collections.html). |

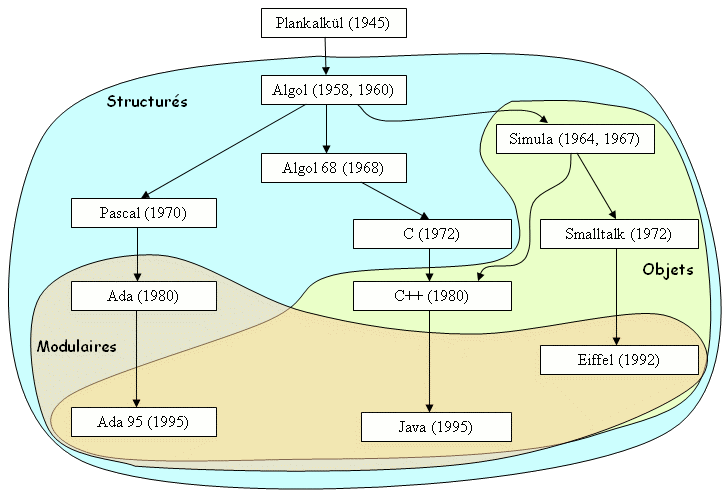
|  |  |  |
| --- | --- | --- |
| [classe *System*](http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/CoursJava/classeSystem.html). | [*Runtime*](http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/CoursJava/classeRuntimeProcess.html#Runtime), [*Process*](http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/CoursJava/classeRuntimeProcess.html#Process), [*ProcessBuilder*](http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/CoursJava/classeRuntimeProcess.html#ProcessBuilder) et [*Desktop*](http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/CoursJava/classeRuntimeProcess.html#Desktop). | [Les fichiers de *log*](http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/CoursJava/fichierDeLogs.html) |
| Classes [Date](http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/CoursJava/DateDateFormat.html) et [DateFormat](http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/CoursJava/DateDateFormat.html#DateFormat). | [Création de documentation avec javadoc](http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/CoursJava/javaDoc/javaDoc.html). | [Expressions régulières](http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/CoursJava/expressionsRegulieres.html). |
| [Quelques algorithmes de tri](http://imss-www.upmf-grenoble.fr/prevert/Prog/Tris/tableDesMatieres.html). | [Complexité des algorithmes](http://imss-www.upmf-grenoble.fr/prevert/Prog/Complexite/). (Initiation) |  |

***Cours java : Introduction***

*http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/CoursJava/3dnum_001.GIF*Historique

1. Le langage **Oak**, a été développé pour le projet Green chez Sun. Ce projet démarre en 1991 avec Patrick Naughton, Mike Sheridan et James Gosling. L’idée était de rechercher quelle était la «vague» suivante en informatique… et ils pensèrent que c’était la communication entre calculateurs et des systèmes implantés sur des appareils grand public : tv interactive, four mico-ondes, grille pain, réfrigérateur ; etc…  
   http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/CoursJava/dukeani.gifLe groupe ne se considérait pas comme un groupe de recherche, mais plutôt comme un groupe de production : «learning by doing»  
   C’est à ce moment que Joe Palrang inventa "Duke".  
   Projet abandonné en septembre1992
2. 1995, Java : Le projet Oak appliqué à WWW, se renomma Java. Combiné avec un navigateur spécial *Hotjava*, il apporte le concept d’*applet*.

http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/CoursJava/3dnum_002.GIFGénéalogie de JAVA

 Cette généalogie ne prend pas en compte les langages des lignées

* [LISP](http://fr.wikipedia.org/wiki/LISP)
* [PROLOG](http://fr.wikipedia.org/wiki/Prolog)
* [FORTRAN](http://fr.wikipedia.org/wiki/Fortran)
* [COBOL](http://fr.wikipedia.org/wiki/Cobol)
* [SNOBOL](http://fr.wikipedia.org/wiki/SNOBOL)

***Types de donnée primitifs***

|  |
| --- |
| **Sommaire**   1. [Types primitifs](http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/CoursJava/TypeDeDonneePrimitifs.html#type)    1. [booléen](http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/CoursJava/TypeDeDonneePrimitifs.html#boolean)    2. [char](http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/CoursJava/TypeDeDonneePrimitifs.html#char)    3. [entier](http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/CoursJava/TypeDeDonneePrimitifs.html#entier)    4. [flottant](http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/CoursJava/TypeDeDonneePrimitifs.html#flottant)    5. [conversions](http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/CoursJava/TypeDeDonneePrimitifs.html#conversions) 2. [Variables et constantes](http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/CoursJava/TypeDeDonneePrimitifs.html#variables) 3. [Expressions](http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/CoursJava/TypeDeDonneePrimitifs.html#expressions) |

1 Types primitifs

Ce sont des types de donnée non objet qu'on retrouve dans de nombreux autres langages de programmation. A chacun de ces types correspond un type objet appelé «[type enveloppe](http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/CoursJava/enveloppes.html)».

*1.1 Booléen* : **boolean**

C’est l’ensemble des deux valeurs **true** et **false**

*1.2 Caractère* : **char**  ([visualisation](http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/Applis/repCaractere.jnlp), [BabelMap](http://www.babelstone.co.uk/Software/BabelMap.html))

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

Compléments d’information sur [Unicode](http://fr.wikipedia.org/wiki/Unicode), et aussi [ici](http://hapax.qc.ca/pdf/Propri%E9t%E9s%20et%20formats%20Unicode.pdf).

Un caractère est représenté sur 2 octets.

*1.3 Types entiers*

Les différents types entiers se distinguent par la taille de leur représentation.

|  |
| --- |
|  |

* **byte** :   (1 octet ) entiers compris entre -128 et +127 (-27 et 27-1)
* **short** :  (2 octets ) entiers compris entre -32768 et +32767 (-215 et 215-1)
* **int** :  (4 octets ) entiers compris entre -2147483648 et +2147483647 (-231 et 231-1)
* **long** : (8 octets ) entiers compris entre -9223372036854775808 et +9223372036854775807 (-263 et 263-1)

|  |  |
| --- | --- |
| Le codage des entiers se fait en binaire pour les entiers positifs et en binaire en complément à deux pour les entiers négatifs :   * 9      est codé sur un octet en  00001001 * -9     est codé sur un octet en  11110111 * -1     est codé sur un octet en  11111111 * -128 est codé sur un octet en  10000000 * 127  est codé sur un octet en  01111111 | Le codage des nombres négatifs en complément à deux s'obtient de la façon suivante :   1. on représente la valeur absolue du nombre en binaire. 2. on inverse tous les bits de la représentation. 3. on ajoute 1. |

Les littéraux peuvent être notée dans trois bases différentes :

* décimal : notation habituelle 123, 928 …
* octal : le nombre commence par 0 , 0123, 0763 …
* hexadécimal : le nombre commence par 0x : 0x213, 0xA9F … les chiffres de 10 à 15 sont notés A, B, C, D, E, F.

Les littéraux sont par défaut de type *int*. Un littéral peut être rendu de type *long* en le suffixant avec **l** ou **L**.

* 12345 est de type *int*
* 12345L est de type *long*

*1.4 Types flottants*

|  |
| --- |
|  |

Ils se distinguent par la taille de leur représentation, et par conséquent par la précision et l’étendue des valeurs.

La représentation des réels est une représentation en virgule flottante : signe, exposant et significande ( [IEEE754](http://grouper.ieee.org/groups/754/) ).

* le **signe** : 0 pour les nombres positifs, 1 pour les négatifs
* l’**exposant** : un nombre entier dont la valeur  est obtenue en soustrayant la valeur maximum/2 au nombre stocké dans la partie exposant.
* le **significande** : un nombre entier.

La valeur du nombre en représentation flottante est obtenue de la façon suivante :

* 0<exposant<max : on a alors un nombre **normalisé** f = signe\*2exposant-max/2-1 \*1.significande
* exposant = 0
  + significande !=0  nombre **dénormalisé**f = signe\*2-max/2 \*0.significande
  + significande = 0 valeur 0
* exposant = max
  + significande =0 +/- infini
  + significande !=0 NaN (Not a Number)

|  |  |
| --- | --- |
| **float** : ( 4 octets )  dans ce cas max vaut 255 : ensemble des nombres [-3.40282347E38 .. -1.40239846E-45], 0, [1.40239846E-45 .. 3.40282347E38] | float |
| **double** : ( 8 octets ) dans ce cas max vaut 2047 : ensemble des nombres [-1.79769313486231570E308 .. -4.94065645841246544E-324], 0, [4.94065645841246544E-324 .. 1.79769313486231570E308] | double |

Les littéraux sont représentés de la façon suivante :

**entier.**{entier}{E{+, -}entier} ou **.entier**{E{+, -}entier}

Ce qui est entre accolades est facultatif. Par défaut un littéral flottant est de type **double**. Un littéral peut être forcée dans le type **float** en la suffixant avec **f** ou **F** ou dans le type **double**en la suffixant avec **d** ou **D**.

A la suite d’opérations en virgule flottante, on peut obtenir POSITIVE\_INFINITY ou NEGATIVE\_INFINITY, ou NaN ( Not a Number).

Les opérations sur les nombre flottants n’ont pas les même propriétés que les opérations habituellement utilisées sur les nombres réels ( commutativité, associativité, égalité ...) Voir :

* [What Every Computer Scientist Should Know About Floating-Point Arithmetic](http://www.physics.ohio-state.edu/~dws/grouplinks/floating_point_math.pdf)
* [How Java’s Floating-Point Hurts Everyone Everywhere](http://www.cs.berkeley.edu/~wkahan/JAVAhurt.pdf)
* [LES CALCULS FLOTTANTS SONT-ILS FIABLES ? ou UN ORDINATEUR "SAIT-IL" BIEN CALCULER ?](http://www.lactamme.polytechnique.fr/Mosaic/descripteurs/FloatingPointNumbers.01.Fra.html)

Pour se rendre compte de certains des problèmes évoqués dans les articles précédents, essayer les programmes suivants :

avec des valeur initiales pour *a* et *b* qui sont des puissances de 4 + 0.1, pour d’autres valeurs, ou en changeant les *double* en *float* ...

|  |  |
| --- | --- |
| **float** A=3.1F,B=2.3F,C=1.5F;  **float** X,Y;  X=(A\*B)\*C;  Y=A\*(B\*C);  System.out.println(X);  System.out.println(Y);  System.out.println();  System.out.println(4096.1F-4095.1F);  System.out.println(4096.1-4095.1); | les sorties console sont les suivantes :  10.695  10.694999  1.0  1.0000000000004547 |
| A essayer avec des valeur initiales pour *a* et *b* qui sont des puissances de 4 + 0.1, pour d’autres valeurs, ou en changeant les *double* en *float* ...  **public** **void** xxx(){  **double** a = 64.1;  **double** b = 63.1;  *// pour d’autres valeurs de a et b*  *// la suite des  x est (1, 1, 1, ...*  **double** x = 1.0;  **for** (**int**i=1; i <= 14; i++){  System.out.println(""+i+" "+x);  x = a\*x - b;*// b = a-1  donc x = b\*(x-1)+x*  *// x initial = 1, donc x = x = 1*  }  } | 1 1.0  2 0.9999999999999929  3 0.9999999999995381  4 0.9999999999703917  5 0.9999999981021048  6 0.9999998783449158  7 0.9999922019090874  8 0.9995001423725043  9 0.967959126077524  10 -1.05382001843072  11 -130.64986318140916  12 -8437.756229928327  13 -540923.2743384057  14 -3.4673244985091805E7 |

*1.5 Conversions*

Les conversions vers un type dont la taille de la représentation est plus grande ou égale se font automatiquement sans avoir besoin d’être spécifié.

|  |  |
| --- | --- |
| Dans l'image ci-contre, les flèches pleines indiquent les conversions automatiques sans perte d'information, les flèches en pointillés, les conversions automatiques avec une possible perte d'information : | conversions automatiques |

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

Exemples :

**double** d = 12345.6;

**byte** b = (**byte**)d; *// b vaut 57*

* Les conversions d’un type entier vers un autre type entier se font par troncature : on perd les bits les plus à gauche.
* Les conversions d’un flottant vers un entier commencent par une conversion de la valeur du flottant en *int* ou *long*, puis il y a une conversion du type précédent.

2 Variables, constantes

Une variable doit être déclarée. Une déclaration est un type suivi du nom de la variable. Le nom peut être suivi d’une initialisation ( = *valeur*).

Une déclaration de constante se fait comme une déclarationde variable précédée du mot **final**.

Un nom de variable ou de constante est contitué de lettres, de chiffres de '\_', et de symboles monétaires ('$', '£', '€', etc... ) et ne peut pas commencer par un chiffre. Plus généralement la méthode *Character.isJavaIdentifierStart(Char c)* retourne *true* si *c* peut commencer un identificateur, et la méthode *Character.isJavaIdentifierPart(Char c)* retourne *true* si *c* peut faire partie du reste de l'identificateur.

L'identificateur ne peut pas être un mot réservé Java :

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

|  |  |  |
| --- | --- | --- |
| On représente les variables ou constantes de type primitifs de la façon suivante : | **int** i = 12;  **double** d = 3.14; | http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/CoursJava/typePrimitif1.gif |

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

3 Expressions   ([visualisation](http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/CoursJava/expressions.jnlp))

Les expressions sont construites à partir de littéraux, variables, constantes, opérateurs et parenthèses. Toute les expressions retourne une valeur. Les expressions ( sauf pour les opérateurs && et || ) sont évaluées en évaluant d’abord l’opérande de gauche, puis l’opérande de droite, puis en réalisant l’opération.

**int** i = 0 ;

t[++i]=++i ; *// affectation de 2 à t[1]*

Certaines instructions sont des instructions expressions et peuvent se trouver partout où on peut trouver une instruction :

* affectation
* pré et post incrément ou décrément

| **opérateur** | **sémantique** | **priorité** | **associativité** |
| --- | --- | --- | --- |
| [] . () *appel de méthode* | sélection dans un tableau, sélection d'un attribut ou méthode de classe, et paramètre d'un appel de méthode | 0 | gauche vers droite |
| ++ -- | Pré ou post incrémentation ou décrémentation | 1 | droite |
| + - unaire |  | 1 | droite |
| ! | négation booléenne | 1 | droite |
| ~ | Négation bit à bit | 1 | droite |
| () *cast* | Transtypage. | 1 | droite |
| \* | Multiplication | 2 | gauche |
|  | Division : division réelle si un des deux opérandes est un réel, et quotient de la division si les deux opérandes sont des entiers. Une division par 0 lève une exception. | 2 | gauche |
| % | Reste de la division. (opérandes entiers ou réels) | 2 | gauche |
| + - binaires | addition et soustraction. | 3 | gauche |
| << >> | Décalage à gauche signé, complété par des 0, décalage à droite signé, le remplissage dépend du signe. | 4 | gauche |
|  |
| 32>> 3 = 4 |
| -32 >> 3 = -4 |
|  |
| >>> | Décalage à droite, le remplissage se fait avec de 0. | 4 | gauche |
|  |
| 32 >> 3 = 4 |
| -32 >>> 3 = 536870908 |
|  |
| < <= > >= | Opérateurs de comparaison. | 5 | gauche |
| == != | Opérateurs de comparaison. | 6 | gauche |
| & | **Et** bit à bit pour les opérandes entiers. | 7 | gauche |
| **Et** évaluant les 2 opérandes pour des opérandes booléens. |
| ^ | **Ou** exclusif bit à bit pour des opérandes entiers. | 8 | gauche |
| **Ou** évaluant les 2 opérandes pour des opérandes booléens. |
| | | **Ou** inclusif bit à bit pour des opérandes entiers. | 9 | gauche |
| **Ou** inclusif évaluant les 2 opérandes pour des opérandes booléens. |
| && | **Et** logique. le premier opérande est évalué : s’il vaut *false* la valeur de l’expression est *false*, sinon c’est la valeur du second opérande. | 10 | gauche |
| || | **Ou** logique. le premier opérande est évalué : s’il vaut *true* la valeur de l’expression est *true*, sinon c’est la valeur du second opérande. | 11 | gauche |
| ? : | Opérateur ternaire a ? b : c | 12 | droite vers gauche |
|  |
| *a* est évalué |
| *a* vaut true la valeur de l’expression est la valeur de *b* |
| *a* vaut false la valeur de l’expression est la valeur de *c* |
| *b* est une expression quelconque, mais *c* ne peut être une affectation que si elle est entre parenthèses. |
| Opérateurs d’affectation : la partie gauche doit être une variable. |
| =  +=  -= &= | Les opérateurs composés ont le sens suivant : | 13 | droite vers gauche |
| \*=  /=  %= | *a op= b* est équivalent à *a = (T)(a op ( b ))* où *T* est le type de *a*. L’expression *b* est évaluée avant de d’évaluer *a op b* ; |
| <<=  >>=  >>>= | *String s; s += ’a’+’b’;* est équivalent à *s = s+(’a’+’b’)*. |
| &=  |=  ^= | a pour valeur *true* si l’opérande de gauche a pour type l’opérande de droite et *false* sinon. |
| **instanceof** |  | 14 | gauche |
| **cast** |  | 1 | gauche |
| **new** |  | 1 | gauche |

Les opérations sur les nombres réels ne lèvent pas d’exceptions, mais produisent des valeurs NaN, POSITIVE\_INFINITY ou NEGATIVE\_INFINITY

**float** a = 1000f;

System.out.println(a/0); *// affichage de Infinity*

a = a\*a\*a\*a;

System.out.println(a); *// affichage de 1.0E12*

a = a\*a\*a\*a;

System.out.println(a); *// affichage de Infinity*

System.out.println(a/a); *// affichage de NaN*

***Instructions***

|  |
| --- |
| **Sommaire**   1. [Instruction bloc](http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/CoursJava/Instructions.html#bloc) 2. [Instructions conditionnelles](http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/CoursJava/Instructions.html#if) 3. [Instruction switch](http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/CoursJava/Instructions.html#switch) 4. [Itérations](http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/CoursJava/Instructions.html#itérations)    1. [while](http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/CoursJava/Instructions.html#while)    2. [do](http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/CoursJava/Instructions.html#do)    3. [for](http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/CoursJava/Instructions.html#for) 5. [Exemples](http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/CoursJava/Instructions.html#exemples) 6. [for du jdk1.5](http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/CoursJava/Instructions.html#newFor) 7. [Etiquettes](http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/CoursJava/Instructions.html#etiq) |

1 Instruction bloc

Les accolades jouent avec les instructions le même rôle que les parenthèses avec les expressions : elles permettent de faire d’une séquence d’instructions séparées par des ; une seule instruction. On attachera un soin particulier à l’indentation des instructions mises entre accolades :

|  |  |
| --- | --- |
| {  *I1* ;  *I2* ;  . . .  *In* ;  } | La liste d’instructions peut être vide. L'exécution d'une instruction bloc, c'est l'exécution en séquence des instructions composant l'instruction bloc. Une variable déclarée dans un bloc a une **portée** qui va jusqu'à la fin du bloc : elle est connue depuis sa déclaration, jusqu'à l'accolade fermante. |

2 Instructions if

Il existe 2 formes de l’instruction conditionnelle :

|  |  |
| --- | --- |
| **if**( *expression booléenne* ) instruction I1 ; | L’exécution de l’instruction commence par l’évaluation de l’expression booléenne : si celle-ci a valeur **true**, alors il y a exécution de l’instruction *I1*. |
| **if**( *expression booléenne* ) instruction I1 ;  **else** instructionI2 ; | L’exécution de l’instruction commence par l’évaluation de l’expression booléenne : si celle-ci a valeur **true**, alors il y a exécution de l’instruction *I1* sinon il y a exécution de l’instruction *I2*. |

Les instructions *I1* ou *I2* peuvent être l’instruction vide.

3 Instruction switch

L’instruction *switch* se construit de la façon suivante :

**switch**( *expression entière ou caractère* ){

**case** *v1* : *instructions I1;*

**case** *v2* : *instructions I2;*

...

**default** : *instructions In;*

}

L’exécution de l’instruction commence par l’évaluation de l’expression entière, soit *v* cette valeur : Puis on recherche la première valeur *vi* égale à *v* : on commence alors l’exécution des instructions qui suivent la valeur *vi*. Si cette valeur n’est pas trouvée, ce sont les instructions *In* qui sont exécutées.

En général après l’exécution de *I1*, on ne désire pas exécuter *I2*, etc … Alors la structure du *switch* sera plutôt :

**switch**( *expression entière ou caractère* ){

**case** *v1* : *instructions I1;* **break**;

**case** *v2* : *instructions I2;* **break**;

...

**default** : *instructions In;*

}

Les instructions *I1*, *I2* ou *In* peuvent être l’instruction vide.

4 Itérations

Les instructions d'itération sont des instructions qui permettent de répéter l'exécution d'une (ou plusieurs) instruction(s) zéro, une ou plusieurs fois.

*4.1 while*

Le *tantque* se construit de la façon suivante :

**while**( *expression booléenne* ) Instruction ;

On évalue d’abord l’expression booléenne :

1. si celle-ci a la valeur **true**, on exécute l’instruction puis on re-exécute le *tantque*
2. si celle-ci a la valeur **false** l’exécution du *tantque* est terminée

L’instruction  peut être l’instruction vide.

*4.2 do*

Le *do* se construit de la façon suivante :

**do**{

*instructions*

}**while**( *expression booléenne*);

On exécute d’abord les instructions, puis on  évalue l’expression booléenne :

1. si celle-ci a la valeur **true**, on re-exécute le *do*
2. si celle-ci a la valeur **false** l’exécution du *do* est terminée

La liste d’instructions  peut être l’instruction vide.

*4.3 for*

Le *for* se construit de la façon suivante :

**for**(*initialisations* ; *expression booléenne* ; *mises à jour*) *instructions* ;

1. On exécute d’abord les initialisations
2. Puis on  évalue l’expression booléenne :
   * si celle-ci a la valeur **true**, on exécute les instructions de mise à jour, puis on re-exécute le *for* à partir de 2.
   * si celle-ci a la valeur **false**, l’exécution du *for* est terminée

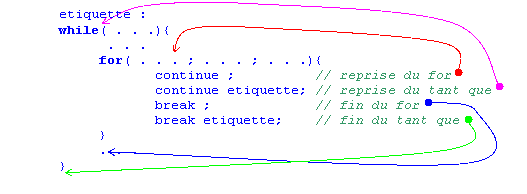
Les initialisations et les mises à jour sont séparées par des virgules.

La liste d’instructions peut être l’instruction vide.

Les instructions dépendant des itérations *for*, *while*,ou *do* peuvent contenir des instructions :

* **break** : qui stoppe l’itération.
* **continue** : qui permet d’aller directement à l’évaluation de la condition de continuation.

Les instructions Java peuvent être étiquetées, alors les **break**, ou **continue** inclus dans une instruction étiquetée peuvent faire référence à cette étiquette :



5 Exemple :

Nous nous proposons de calculer la somme des *n* premiers entiers, pour n>=0 :

|  |  |
| --- | --- |
| while | **int** n = 10;  **int** somme = 0;  **int** i;  i = 1;  **while**( i<=n){  somme = somme + i;  i = i+1;  }  System.out.println(" la somme des "+n+" premiers entiers est : "+somme); |
| for | **int** n = 10;  **int** somme = 0;  **int** i;  **for**( i=1; i<=n; ++i)  somme = somme + i;  System.out.println(" la somme des "+n+" premiers entiers est : "+somme); |  |
| do | **int** n = 10;  **int** somme = 0;  **int** i=0;  **do**{  somme = somme + i;  ++i;  }**while**(i<=n);  System.out.println(" la somme des "+n+" premiers entiers est : "+somme); |  |

6 for : le nouveau for du JDK 1.5 :

**for**(*Paramètre* *formel* : expression) *instruction*;

L'expression doit être un tableau ou  une instance de classe implémentant l'interface *Iterable* : L'interface *java.util.collection* étend depuis le JDK1.5 l'interface *Iterable*.

Exemples :

1. **int** [] t = {1, 2, 3, 4, 5, 6};
2. **for**( **int** a : t) System.out.println(a);

L'instruction suivante ne modifie pas le tableau t.

**for**( **int** a : t)++a;

1. Liste l = **new** ArrayList();
2. **for**( **int** a : l) System.out.println(a);

7 Les étiquettes

En Java toute instruction (pas les déclarations) peut être étiquettée. Une étiquette est un identificateur Java suivi de :

Exemple

**int** a = 0;

etiq1 : **if**(**true**);

etiq2 : **switch**(1){

*// ...*

}

etiq3 : a=1;

etiq4 : a++;

etiq5 : {

*// ...*

}

etiq6 : **for**(; ; ) **break**;

etiq7 : **while**( **true** ) **break**;

L'utilisation à tort et à travers des étiquettes permet d'écrire des programmes illisibles. La seule utilisation qui peut être intéressante est de pouvoir arrêter une itération englobant une autre itération depuis l'intérieur de cette dernière :

etiq1 : **while**(**true**){

**while**(...){

**if**(...) **break** etiq1; *// Arrête l'itération étiquettée par etiq1*

}

}

***Tableaux***

|  |
| --- |
| **Sommaire**   1. [Définition](http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/CoursJava/tableaux.html#definition) 2. [Tableau à plusieurs dimensions](http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/CoursJava/tableaux.html#dims) 3. [Tableau en paramètre](http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/CoursJava/tableaux.html#param) 4. [Utilitaires](http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/CoursJava/tableaux.html#util) |

1 Définition

Un tableau est une structure de données contenant un groupe d'éléments tous du même type. Le type des éléments peut être un type primitif ou une classe. Lors de la définition d’un tableau les [] spécifiant qu’il s’agit d’un tableau peuvent être placés avant ou après le nom du tableau :

*// ti est un tableau à une dimension de* ***int***

**int** ti [];

*// tc est un tableau à une dimension de* ***char***

**char**[] tc;

Un tableau peut être initialisé :

**int** ti1 [] = { 1, 2, 3 , 4};

**char**[] tc = {’a’, ’b’, ’c’};

|  |  |
| --- | --- |
| Pour allouer l’espace nécessaire au tableau il faut utiliser **new** :  *// ti est un tableau à une dimension de 10* ***int***  ti = **new** **int** [10];  *// tc est un tableau à une dimension de 15* ***char***  tc = **new** **char** [15];  L'utilisation d'un tableau pour lequel l'espace n'a pas été alloué provoque la levée d'une exception NullPointerException  Les éléments d’un tableau sont indicés à partir de 0. Chaque élément peut être accédé individuellement en donnant le nom du tableau suivi de l’indice entre [ ]  ti [0] ;*// premier élément du tableau ti*  ti [9] ;*// dernier élément du tableau ti*  L’accès à un élément du tableau en dehors des bornes provoque la levée d’une exception ArrayIndexOutOfBoundsException  Un tableau possède un attribut ***length*** qui permet de connaître le nombre d’éléments d’un tableau.   * ti.length vaut 10. * tc.length vaut 15. | un un |

2 Tableau à plusieurs dimensions

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

|  |  |
| --- | --- |
| Toutes les lignes d’un tableau à 2 dimensions n’ont pas forcément le même nombre d’éléments :  **int** t22[][] ;  t22 = **new** **int**[5][];  **for**( **int** i = 0; i< t22.length; ++i){  t22[i]= **new** **int** [i+1];  }  **for**( **int** i = 0; i< t22.length; ++i){  **for**( **int** j = 0; j<t22[i].length; ++j){  *//accès à t22[i][j]*  }  } | tableau5.gif |

3 Tableaux en paramètre :

La spécification d’un paramètre tableau se fait en écrivant autant de couples de [] que de dimensions du tableau, mais sans donner la taille de chaque dimension. Cette taille peut être obtenue dans la méthode à l’aide de l’attribut *length*.

Le contenu des éléments peut être modifié, mais pas le tableau lui-même.

4 Utilitaires pour les tableaux

*System.arraycopy* : la classe *System* a une méthode de copie rapide de tableaux :

**public** **static** **void** arraycopy(Object src, **int** srcPos, Object dest, **int** destPos, **int** l)

Copie un tableau depuis *src*, et partir de la position *srcPos*, dans la destination *dest* à partir de *destPos* et sur une longueur *l*. La copie n’est pas une copie profonde : seules les références sont copiées.

* Si *src* et *dest* sont le même tableau, tout se passe comme si les éléments à copier étaient d’abord copiés dans un tableau auxiliaire.
* Si *src* ou *dest*vaut *null*, une *NullPointerException* est levée
* Une *ArrayStoreException* est levée si :
  + *src* ou *dest* n’est pas un tableau
  + *src* et *dest* sont des tableaux dont les éléments ne sont pas de même type.
* Une *IndexOutOfBoundsException* est levée si :
  + *srcPos* ou *destPos* ne sont pas « corrects »
  + *destPos+l* ou *srcPos+l* sortent du tableau

La classe *java.lang.reflect.Array* contient des méthodes statiques permettant :

* de récupérer un élément d’un tableau, en le convertissant dans un autre type
* d’affecter un tableau avec un élément
* de créer, à l’exécution, un tableau

|  |  |
| --- | --- |
| **static** Object get(Object t, **int** index) | retourne l’objet d’indice *index* dans le tableau *t*. Cet élément est convertit en un des types enveloppants s’il est de type primitif (*Integer* pour *int*, *Long* pour *long*, etc...). |
| **static** xxx getXxx(Object t, **int** index) | retourne l’objet d’indice *index* dans le tableau *t*. Cet élément est convertit en le type primitif *xxx* si c’est possible |
| **static** **void** set(Object t, **int** index, Object valeur) | affecte une nouvelle valeur à l’élément d’indice *index*du tableau *t*. La valeur est convertie en le bon type avant l’affectation |
| **static** **void** setXxx(Object t, **int** index, xxx z) | affecte une nouvelle valeur à l’élément d’indice *index*du tableau *t*. |

Les méthodes précédentes peuvent lever les exceptions suivantes :

* *NullPointerException* si l’objet au rang *index* vaut *null*.
* *IllegalArgumentException* si le premier argument n’est pas un tableau.
* *ArrayIndexOutOfBoundsException* si *index* est en dehors des bornes du tableau

|  |  |
| --- | --- |
| **static** Object newInstance(Class componentType, **int** l) | Crée un nouveau tableau dont les éléments sont de type *componentType* et la longueur est *l*. Cette méthode peut lever les exceptions :   * *NullPointerException* si le *componentType* vaut *null*. * *nullNegativeArraySizeException* si *l* est négative. |
| **static** Object newInstance(Class componentType,  **int**[] dimensions) | Crée un nouveau tableau dont les éléments sont de type *componentType* :   * Si *componentType* est une class ou une interface (pas tableau) le nouveau tableau a dimensions.length dimensions. Le nombre d’éléments dans la ièmedimension est égal à *dimensions[i].* * Si *componentType* représente une classe tableau, le nombre de dimensions du nouveau tableau est égal à la somme du nombre de dimensions de *componentType* plus *dimensions.length* . |

Exemple :

**int** dims[]={2, 3};

Integer ii = **new** Integer(1);

Object t = java.lang.reflect.Array.newInstance(ii.getClass(), dims);

Permet de créer un tableau d’*Integer* à 2 dimensions : 2 lignes et 3 colonnes.

***tableaux : Arrays***

|  |
| --- |
| **Sommaire**   1. [Tri par partition](http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/CoursJava/Arrays.html#partition) 2. [Tri par fusion](http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/CoursJava/Arrays.html#fusion) |

Cette  classe contient des méthodes statiques permettant de manipuler de tableaux  :

* Recherche, recherche dichotomique
* Tris
* Remplissage
* Egalité
* *toString()*

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

La méthode *static boolean equals(X[] a, X[] a2)*  où *X* peut être un type primitif ou un *Object* retourne *true* si les deux tableaux sont égaux et *false* sinon. Deux tableaux sont égaux s’ils contiennent le même nombre d’éléments, et si les éléments de même rang sont égaux. Si des éléments du tableau peuvent être des tableaux, il faut utiliser la méthode *deepEquals(X[] a, X[] a2)*.

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

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

**public** **static** **int** binarySearch(**int**[] a, **int** cle) {

**int** debut = 0;

**int** fin = a.length-1;

**while** (debut <= fin) {

**int** milieu =( debut + fin)/2;

**if** (a[milieu]< cle)  debut = milieu + 1;

**else** **if** (a[milieu]> cle) fin = milieu - 1;

**else** **return** milieu; *// trouvé*

}

**return** -(debut + 1); *// pas trouvé.*

}

La méthode *static void sort(Object[] a)* est programmée en utilisant :

* Un tri rapide si *X* est un type primitif. adapté de Jon  L. Bentley and M. Douglas McIlroy's "[Engineering a Sort Function](http://www.cs.ubc.ca/local/reading/proceedings/spe91-95/spe/vol23/issue11/spe862jb.pdf)", Software-Practice and Experience, Vol. 23(11) P. 1249-1265 (Novembre 1993). Cet algorithme n’est pas stable.
* Un tri par fusion si *X* est un *Object*. Cet algorithme est stable.

Les objets du tableau à  trier doivent être Comparable

1 Tri  rapide

**public** **static** **void** sort(**int**[] a) {

sort(a, 0, a.length);

}

**private** **static** **void** sort(**int** [] x, **int** off,  **int** len) {

*// Tri par insertion si len <7*

**if** (len < 7) {

**for** (int i=off; i<len+off; i++)

**for** (int j=i; j>off && x[j-1]>x[j]; j--)

echanger(x, j, j-1);

**return**;

}

*// Choix du pivot*

**int** m = off + len/2; *// pour les petits tableaux,*

*// c’est l’élément du milieu*

**if** (len > 7) {

**int** l = off;

**int** n = off + len - 1;

**if** (len > 40) { *// Pour les grands tableaux le médian de 9*

**int** s = len/8;

l = med3(x, l,  l+s, l+2\*s);

m = med3(x, m-s,   m,   m+s);

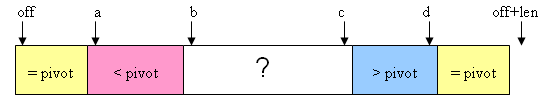
n = med3(x, n-2\*s, n-s, n);

}

m = med3(x, l, m, n); *// tableaux moyen médian de 3*

}

**int** pivot = x[m];



*// établir l’invariant: v\* (<v)\* (>v)\* v\**

**int** a = off, b = a, c = off + len - 1, d = c;

**while**(**true**) {

**while** (b <= c && x[b] <= pivot) {

**if** (x[b] == pivot) echanger(x, a++, b);

b++;

}

**while** (c >= b && x[c] >= pivot) {

**if** (x[c] == pivot) echanger(x, c, d--);

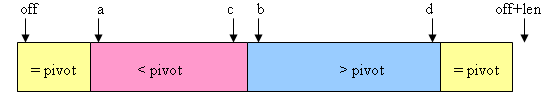
c--;

}

**if** (b > c) **break**;

echanger(x, b++, c--);

}

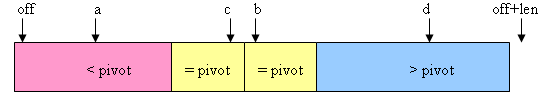


*// mettre les valeurs égales au pivot au milieu*

**int** s, n = off + len;

s = Math.min(a-off, b-a  );  echanger(x, off, b-s, s);

s = Math.min(d-c,   n-d-1);  echanger(x, b,   n-s, s);



*// Appels récursifs sur les éléments non pivot.*

**if** ((s = b-a) > 1) sort(x, off, s);

**if** ((s = d-c) > 1) sort(x, n-s, s);

}

**private** **static** **void** echanger(**int** [] x, **int** a, **int** b) {

**int** t = x[a];

x[a] = x[b];

x[b] = t;

}

**private** **static** **void** echanger(**int** x[], **int** a, **int** b, **int** n) {

**for** (**int** i=0; i<n; i++, a++, b++)

echanger(x, a, b);

}

**private** **static** **int** med3(**int** x[], **int** a, **int** b, **int** c) {

**return** (x[a] < x[b]

? (x[b] < x[c] ? b : x[a] < x[c] ? c : a)

: (x[b] > x[c] ? b : x[a] > x[c] ? c : a));

}

2 Tri  par fusion

**public** **static** **void** sort(Object[] a) {

Object [] aux =(Object[])a.clone();

mergeSort(aux, a, 0, a.length);

}

**private** **static** **void** mergeSort(Object src[], Object dest[], **int** deb, **int** fin) {<

**int** longueur = fin - deb;

*// tri par insertion pour les petits tableaux*

**if** (longueur < 7) {

**for** (**int** i=deb; i<fin; i++)

**for** (**int** j=i; j>deb && ((Comparable)dest[j-1]).compareTo ((Comparable)dest[j]) >0; j--)

swap(dest, j, j-1);

**return**;

}

*// trier les 2 moitiés de dest vers src*

**int** milieu = (deb + fin)/2;

mergeSort(dest, src, deb, milieu);

mergeSort(dest, src, milieu, fin);

*// Si src est ordonné, copier src dans dest*

**if** (((Comparable)src[milieu-1]).compareTo((Comparable) src[milieu]) <= 0) {

System.arraycopy(src, deb, dest, fin, longueur);

**return**;

}

*/// fusionner les 2 moitiés de src vers dest*

**for**(**int** i = deb, p = deb, q = milieu; i < fin; i++) {

**if** (q>=fin || p<milieu && ((Comparable)src[p]).compareTo(src[q])<=0)

dest[i] = src[p++];

**else** est[i] = src[q++];

}

}

***Chaîne de caractères***

|  |
| --- |
| **Sommaire**   1. [String](http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/CoursJava/ChainesDeCaracteres.html#String) 2. [StringBuffer, StringBuilder](http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/CoursJava/ChainesDeCaracteres.html#StringBuffer) 3. Quelques fonctions «avancées»    1. [Conversion](http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/CoursJava/ChainesDeCaracteres.html#conversion) vers les types primitifs.    2. [Formatage](http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/CoursJava/ChainesDeCaracteres.html#formatage).    3. [matches, split, replaceFirst, replaceAll](http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/CoursJava/ChainesDeCaracteres.html#matches).    4. [StringTokenizer](http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/CoursJava/ChainesDeCaracteres.html#StringTokenizer). |

Trois classes chaîne de caractères : *String*, *StringBuffer* et *StringBuilder*.

* La classe *String* est une classes dont les objets sont immuables : ils ne peuvent pas changer de valeur.
* Les classes *StringBuffer* et *StringBuilder* produisent des objets qui peuvent changer de valeur. Ces deux classes ont exactement les mêmes méthodes.
  + Les méthodes de la classe *StringBuffer* sont synchronisées, et peuvent être utilisées par plusieurs *thread* sur une même chaîne de caractères.
  + Les méthodes de la classe *StringBuilder* ne sont pas synchronisées, elles sont donc plus rapides que celle de la classe *StringBuffer*, mais ne sont pas «thread safe».

1 String

Les chaînes de caractères littérales de Java (par exemple "abc"), sont représentées comme des instances de cette classe.

La classe *String* est une classe spéciale :

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

*1.1 Constructeurs*

<td

String(**char**[] value)</td

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

La concaténation de chaîne peut également se faire à l’aide de la méthode *concat(String s).* La méthode *length()* renvoie la longueur ( nombre de caractères) de la chaîne.

*1.2 Comparaisons*

|  |  |
| --- | --- |
| **int** compareTo(Object o) | Compare une chaîne de caractère à un autre objet. Renvoie une valeur <0 ==0 ou > 0 |
| **int** compareTo(String anotherString) | Compare une chaîne de caractère à un autre objet. Renvoie une valeur <0 ==0 ou > 0. La comparaison est une comparaison lexicographique. |
| **int** compareToIgnoreCase(String str) | Compare une chaîne de caractère à un autre objet. Renvoie une valeur <0 ==0 ou > 0. La comparaison est une comparaison lexicographique, ignorant la casse. |
| **boolean** equals(Object anObject) | Compare la chaîne a un objet et retourne *true* en cas d’égalité et *false* sinon |
| **boolean** equalsIgnoreCase(Object anObject) | Compare la chaîne a un objet et retourne *true* en cas d’égalité et *false* sinon ( on ignore la casse) |
| **boolean** regionMatches(**boolean** ignoreCase,  **int** toffset, String other,  **int** ooffset, **int** len) | Teste l’égalité de deux parties de chaînes |
| **boolean** regionMatches(**int** toffset,  String other, **int** ooffset, **int** len) | Teste l’égalité de deux parties de chaînes |

*1.3 Caractère et sous-chaîne*

|  |  |
| --- | --- |
| **char** charAt(**int** i) | Retourne le caractère à l’indice spécifié en paramètre. |
| String substring(**int** d) | Sous-chaîne depuis *d* jusqu’à la fin |
| String substring(**int** d, **int** f) | Sous-chaîne depuis *d* jusqu’au caractère d’indice *f* non inclus. |
| **boolean**  startsWith(String prefix) | Renvoie *true* si *prefix* est un préfixe de la chaîne |
| **boolean** startsWith(String prefix, **int** i) | Renvoie *true* si *prefix* est un préfixe de la chaîne à partir de *i*. |
| **boolean** endsWith(String suffix) | Retourne *true* si *suffix* est un suffixe de la chaîne |

*1.4 Conversions*

|  |  |
| --- | --- |
| String toLowerCase() | Retourne une chaîne égale à la chaîne convertie en minuscules. |
| String toLowerCase(Locale locale) | Retourne une chaîne égale à la chaîne convertie en minuscules. |
| String toString() | Retourne une chaîne égale à la chaîne |
| String toUpperCase() | Retourne une chaîne égale à la chaîne convertie en majuscules. |
| String toUpperCase(Locale locale) | Retourne une chaîne égale à la chaîne convertie en majuscules. |
| String trim() | Retourne une chaîne égale à la chaîne sans les espaces de début et de fin. |
| String replace(**char** ac, **char** nc) | Retourne une chaîne ou tous les *ac* ont été remplacé par des *nc*. S’il n’y a pas de remplacement, la chaîne elle-même est retournée. |

|  |  |
| --- | --- |
| **static** String copyValueOf(**char**[] data) | Construit une chaîne de caractères à partir d’un tableau de caractères |
| **static** String copyValueOf(**char**[] data,  **int** offset, **int** count) | Construit une chaîne de caractèresà partir d’une partie de tableau de caractères |
| **byte** []getBytes(String enc) | Convertit la chaîne en tableau de byte. |
| **byte** []getBytes() | Convertit la chaîne en tableau de byte. |
| **void** getChars(int srcBegin,  **int** srcEnd, **char**[] dst, **int** dstBegin) | Copie les caractères de la chaîne dans le tableau en paramètres. |
| **char** []toCharArray() | Convertit la chaîne de caractères en un tableau de caractères. |

|  |  |
| --- | --- |
| **static** String valueOf(**boolean** b) | Retourne la représentation en chaîne du booléen |
| **static** String valueOf(**char** c) | Retourne la représentation en chaîne du caractère |
| **static** StringvalueOf(**char**[] data) | Retourne la représentation en chaîne du tableau de caractères |
| **static** String valueOf(**char**[] data,  **int** offset, **int** count) | Retourne la représentation en chaîne du tableau de caractères (partie) |
| **static** String valueOf(**double** d) | Retourne la représentation en chaîne du double |
| **static** String valueOf(**float** f) | Retourne la représentation en chaîne du float |
| **static** String valueOf(**int** i) | Retourne la représentation en chaîne du int |
| **static** String valueOf(**long** l) | Retourne la représentation en chaîne du long |
| **static** String valueOf(Object obj) | Retourne la représentation en chaîne de l’objet |

*1.5 Recherche*

|  |  |
| --- | --- |
| **int** indexOf(**int** ch) | Retourne l’indice de la première occurrence du caractère |
| **int** indexOf(**int** ch, **int** fromIndex) | Retourne l’indice de la première occurrence du caractère à partir de *fromIndex* |
| **int** indexOf(String str) | Retourne l’indice de la première occurrence de la chaîne |
| **int** indexOf(String str, **int** fromIndex) | Retourne l’indice de la première occurrence de la chaîne à partir de *fromIndex* |
| **int** lastIndexOf(**int** ch) | Retourne l’indice de la dernière occurrence du caractère |
| **int** lastIndexOf(**int** ch, **int** fromIndex) | Retourne l’indice de la dernière occurrence du caractère à partir de *fromIndex* |
| **int** lastIndexOf(String str) | Retourne l’indice de la dernière occurrence de la chaîne |
| **int** lastIndexOf(String str, **int** fromIndex) | Retourne l’indice de la dernière occurrence de la chaîne à partir de *fromIndex* |

*1.6 Remarque*

La classe *String* maintient un pool de chaînes de caractères. Dans ce pool sont représentées :

* les chaînes littérales "..."
* les chaînes pour lesquelles on a fait un appel à le méthode native *intern()*

C'est pour celà que la comparaison avec l'opérateur == fonctionne pour les chaines de caractères du pool !

String s1 = "abcde";

String s2 = "abcde";

String s3 = **new** String (s2);

String s4 = s3.intern();

System.out.println(s1==s2);*//* ***true*** *les deux sont dans le pool*

System.out.println(s1==s3);*//* ***false***

System.out.println(s1==s4);*//* ***true*** *les deux sont dans le pool*

2 StringBuffer, StringBuilder

Les *StringBuffer* sont utilisés pour compiler l’opérateur + de la classe String. Par exemple :

x = "a" + 4 + "c"

est compilé en :

x = **new**StringBuffer().append("a").append(4).append("c").toString()

Ces deux classes ont deux attributs :

* **char** [] value : le tableau dans lequel sont représentés les caractères de la chaîne. La taille de ce tableau est la capacité de la chaîne de caractères.
* **int** count : le nombre de caractères présents dans le tableau value.

*2.1 Constructeurs*

|  |  |
| --- | --- |
| StringBuffer() | Construit une chaîne vide de capacité initiale de 16 caractères. |
| StringBuffer (**int** l) | Construit une chaîne vide de capacité initiale de *l* caractères. |
| StringBuffer (String s) | Construit une chaîne de caractères à partir de la chaîne *s* |

*2.2 Méthodes*

|  |  |
| --- | --- |
| **int** capacity() | la capacité de la chaîne de caractères |
| **int** length() | la longueur de la chaîne de caractères |
| **void** ensureCapacity (**int** n) | Cette méthode redimensionne le tableau, si nécessaire, de façon que la capacité de la chaîne soit au moins égale à *n* |
| **void** setLength (**int** n) | la longueur de la chaîne est *n* ; les caractères éventuellement ajoutés ont une valeur indéterminée. |

*2.3 Concaténation*

|  |  |
| --- | --- |
| StringBuffer append(**boolean** b) | Concatène la représentation en chaîne du booléen. |
| StringBuffer append(**char** c) | Concatène la représentation en chaîne du caractère. |
| StringBuffer append(**char**[] str) | Concatène la représentation en chaîne du du tableau de caractères. |
| StringBuffer append(**char**[] str,  **int** offset, **int** len) | Concatène la représentation en chaîne du tableau de caractères. |
| StringBuffer append(**double** d) | Concatène la représentation en chaîne du double. |
| StringBuffer append(**float** f) | Concatène la représentation en chaîne du float. |
| StringBuffer append(**int** i) | Concatène la représentation en chaîne du int. |
| StringBuffer append(**long** l) | Concatène la représentation en chaîne du long. |
| StringBuffer append(Object obj) | Concatène la représentation en chaîne de l’objet |
| StringBuffer append(String str) | Concatène la représentation en chaîne de la chaîne. |

*2.4 Caractère et sous-chaîne.*

|  |  |
| --- | --- |
| **char** charAt(**int** i) | Retourne le caractère à l’indice spécifié en paramètre. |
| String substring(**int** i) | Sous-chaîne depuis *i* jusqu’à la fin |
| String substring(**int** i, **int** l) | Sous-chaîne depuis *i* et de longueur *l*. |

*2.5 Conversions.*

|  |  |
| --- | --- |
| String toString() | Retourne une chaîne égale à la chaîne |

*2.6 Modifications.*

|  |  |
| --- | --- |
| StringBuffer delete(**int** start, **int** end) | Enlève tous les caractères compris entre *start* et *end*. |
| StringBuffer deleteCharAt(**int** index) | Enlève le caractère à l’indice *index* |

3 Fonctions avancées sur les chaînes de caractères.

*3.1 Conversions vers les types primitifs.*

Les classes *Byte*, *Short*, *Integer* et *Long* possèdent une méthode *X.parseX(String s)* et une méthode *X.parseX(String s, int b)* qui permettent de convertir une chaîne de caractères en un entier en considérent cet entier ecrit dans la base *b*. Si cette conversion n'est pas possible il y levée d'une exception NumberFormatException.

exemple :

**byte** b = Byte.parseByte("-3f", 16);

System.out.println(b);

affiche -63 à la console.

Les classes *Float* et *Double* possèdent une méthode *X.parseX(String s)* qui permet de convertir une chaîne de caractères en un flottant. Si cette conversion n'est pas possible il y levée d'une exception NumberFormatException.

La classe *Boolean* possède une méthode *Boolean.parseBoolean(String s)* qui permet de convertir une chaîne de caractères en un booléen. Le résultat est *true* si et seulement si la chaîne de caractères *s* est non *null* et si la valeur de la chaîne est *"true"*.

*3.2 Formatage.*

La classe *String* est munie de deux méthodes *format* qui permettent de construire des chaînes de caractères formatées.

public static String format(String format, Object... args)

public static String format(Locale l, String format, Object... args)

Exemple :

|  |  |
| --- | --- |
| String s = String.format("|%05d| |%4.2f| |%-4c|",  123, 3.14159, 'a');  System.out.println(s); | affiche à la console :  |00123| |3,14| |a | |

Le paramètre format est une chaîne de caractères contenant des spécificateurs de format de structure générale : **%[argument\_index$][flags][width][.precision]conversion**. Ce qui est entre [] est facultatif.

Nous allons préciser succintement le sens des différents constituants d'un spécificateur de format, pour plus de précision voir [ici](http://java.sun.com/j2se/1.5.0/docs/api/java/util/Formatter.html#syntax).

|  |  |
| --- | --- |
| **Conversion** | **Description** |
| **'b' 'B'** | Si l'argument correspondant est *null*, alors le résultat est *false*, si l'argument correspondant est un booléen, le résultat est la valeur du booléen, sinon c'est *true* |
| **'h' 'H'** | Si l'argument est *null* alors le résultat est *null*, sinon le résultat est obtenu par l'appel de *Integer.toHexString(arg.hashCode())*. Pour un entier, on a sa valeur en héxadécimal. |
| **'s' 'S'** | Si l'argument est *null* alors le résultat est *null*, sinonsi l'argument implémente *Formattable*,alors *arg.formatTo* est appelé, sinonle résultat est obtenu en appelant *toString()*. |
| **'c' 'C'** | Le résultat est le caractère unicode. |
| **'d'** | Le résultat est un entier décimal. |
| **'o'** | Le résultat est un entier octal. |
| **'x' 'X'** | Le résultat est un entier héxadécimal. |
| **'e' 'E'** | Le résultat est un nombre flottant en notation scientifique. |
| **'f'** | Le résultat est un nombre flottant en notation décimale (avec un point ou une virgule suivant la locale). |
| **'g' 'G'** | Le résultat est un nombre flottant en notation décimale, ou scienttifique suivant sa valeur. |
| **'a' 'A'** | Le résultat est un nombre flottant en héxadécimal, avec son exposant et son significande. |
| **'t' 'T'** | date et time. |
| **'%'** | le résultat est %. |
| **'n'** | Le résultat est le séparateur de lignes. |

La partie **width** est le nombre de caractères minimum utilisés pour écrire l'argument correspondant.

La partie **precision** est en général le nombre de caractères maximum utilisés pour écrire l'argument correspondant. Pour les nombres flottants ecrits en décimal, il s'agit du nombre de chiffres après la virgule. Pour les nombre entiers, les dates, le % et le séparateur delignes une exception est levée.

|  |  |
| --- | --- |
| **Flag** | **Description** |
| **'-'** | le résultat est justifié à gauche. |
| **'#'** |  |
| **'+'** | Le résultat a toujours son signe. |
| **' '** | Le résultat a toujours un espace devant les valeurs positives. |
| **'0'** | Le résultat est étendu à gauche avec des 0. |
| **','** |  |
| **'('** | Les nombres négatifs sont entre parenthèses. |

*3.3 matches, split, replaceAll, replaceFirst.*

la classe *String* contient des méthodes ayant un paramètre qui est une expression régulière.

|  |  |
| --- | --- |
| **boolean** matches(String regex) | Retourne *true* si la chaîne de caractères s'apparie avec l'[expression régulière](http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/CoursJava/expressionsRegulieres.html) et *false* sinon. Un appel *s.matches(regexp)* a le résultat est le même qu'un appel de *Pattern.matches(regex, s)* |
| String replaceAll(String regex,  String replacement) | Retourne la chaîne dans laquelle on a remplacé tous les appariements de *regex* par *replacement* |
| String replaceFirst(String regex,  String replacement) | Retourne la chaîne dans laquelle on a remplacé le premier appariement de *regex* par *replacement*. |
| String[]split(String regex) | Les différents appariements de *regex* découpent la chaîne de caractères en plusieurs chaînes. Ces chaînes sont retournées dans un tableau de chaînes de caractères. |

Exemple : le code suivant permet de découper un texte en mots et range ces mots dans un tableau.

String s = "....";

String[] t = s.split("[.,;:?!' ]+");

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

*3.4 StringTokenizer.*

La classe *StringTokenizer* permet de découper une chaîne de caractères en jetons (*token*). Le résultat du découpage est donné comme une énumération.

Constructeurs

|  |  |
| --- | --- |
| StringTokenizer(String s) | Construit un *StringTokenizer* sur la chaîne *s* utilisant les délimiteurs par défaut :  " \t\n\r\f" |
| StringTokenizer(String s, String delim) | Construit un *StringTokenizer* sur la chaîne *s* utilisant les délimiteurs contenus dans *delim*. |
| StringTokenizer(String s, String delim,  **boolean** retourneDelim) | Construit un *StringTokenizer* sur la chaîne *s* utilisant les délimiteurs contenus dans *delim*, les délimiteurs faisant partie du jeton retourné. |

Méthodes

|  |  |
| --- | --- |
| **public** **boolean** hasMoreTokens() | Retourne *true* s'il reste des jetons dans la chaîne et *false* sinon. |
| **public** String nextToken() | Retourne le jeton suivant s'il existe et lève une exception NoSuchElementException sinon. |
| **public** String nextToken(String delim) | Retourne le jeton suivant, en utilisant les délimiteurs *delim*, s'il existe et lève une exception NoSuchElementException sinon. |
| **public** **boolean** hasMoreElements() | Retourne *true* s'il reste des jetons dans la chaîne et *false* sinon. |
| **public** **Object** nextElement() | Retourne le jeton suivant s'il existe et lève une exception NoSuchElementException sinon. |
| **public** **int** countTokens() | Retourne le nombre de jetons restant, en utilisant l'ensemble de délimiteurs courant. |

Exemple :  le code  :

StringTokenizer s = **new** StringTokenizer("Ceci est un, petit, test");

**while** (s.hasMoreTokens()) {

System.out.println(s.nextToken());

}

Affiche le résultat suivant :

Ceci

est

un

petit

test

|  |  |
| --- | --- |
|  |  |

***Classe***

|  |
| --- |
| **Sommaire**   1. [Définition](http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/CoursJava/classes1.html#def) 2. [Création d'instance](http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/CoursJava/classes1.html#creer) 3. [Attribut et méthode d'instance](http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/CoursJava/classes1.html#attrMeth)    1. [Attribut](http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/CoursJava/classes1.html#attr)    2. [Méthode](http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/CoursJava/classes1.html#meth) 4. [Constructeurs, *finalize*](http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/CoursJava/classes1.html#const) 5. [Attribut et méthode de classe](http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/CoursJava/classes1.html#attrC)    1. [Bloc statique](http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/CoursJava/classes1.html#bstat) 6. [classe *Object*](http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/CoursJava/classes1.html#object) |

Une classe est la description de données appelées **attributs**, et d’opérations appelées **méthodes**.

Une classe est un modèle de définition pour des objets ayant le même ensemble d’attributs, et le même ensemble d’opérations.

A partir d’une classe on peut créer un ou plusieurs objets par **instanciation** ; chaque objet est une **instance** d’une seule classe.

Les caractéristiques de la programmation objet sont :

* Encapsulation des données (attributs) et des comportements( méthodes) : les attributs et les méthodes sont définies dans le même environnement (capsule).
* Masquage de l’information : l’utilisateur de la classe peut ne pas avoir accès directement aux attributs.
* Héritage : on peut définir une classe à partir d’une autre classe.
* Polymorphisme : une même méthode peut avoir un comportement différent en fonction de l’instance à la quelle est appliquée.

1 Définition d’une classe :

**class** X **extends** Y{

*// liste des attributs*

*// liste des méthodes*

}

Si *Y* est la classe de base de la hiérarchie des objets Java *Object* , alors la partie *extends* est facultative.

exemple :

|  |  |  |
| --- | --- | --- |
| **class** A **extends**  Object{  **int** a ;  **void** f(){  ...  }  } | est équivalent à | **class** A {  **int** a ;  **void** f(){  ...  }  } |

Une classe a une visibilité :

* **public** le mot *class* est alors précédé de *public*, tout utilisateur qui importe le paquetage peut utiliser la classe. Dans ce cas elle doit être définie dans un fichier qui a pour nom le nom de la classe.
* **privé** le mot *class* est alors précédé de *private*,  seules des classes définies dans le même fichier peuvent utiliser cette classe.
* **paquetage** le mot *class* n’est pas précédé de mot particulier, toutes les classes du paquetage peuvent utiliser la classe.

2 Création d’instances

Pour créer une instance de la classe *A*, on écrira :

A a ;   *// définition de la variable référence*

a = **new** A() ; *// création de l’instance*

La création d’une instance par l’opérateur **new** se déroule en trois temps :

* Réservation de l’espace mémoire suffisamment grand pour représenter l’objet.
* Appel du constructeur de l’objet. Initialisation des attributs, et d’une référence à l’objet représentant la classe de l’instance en train d’être créée.
* Renvoi d’une référence sur l’objet nouvellement créé

|  |  |
| --- | --- |
| **class** Point{  **int** x ;  **int** y ;  . . .  }  Point a, b ;  a = **new** Point(…) ;  b = a; | représentation de l'instance de Point |

La destruction des instances se fait automatiquement par un « *thread* » le *garbage collector* qui cherche tous les objets qui ne sont plus référencés et les supprime. Le *garbage* *collector* peut être appelé directement par *System.gc()*.

3 Attributs et méthodes

*3.1 Attribut :*

Un attribut se définit en donnant son type, puis son nom, ( [] pour un tableau ), et éventuellement une partie initialisation.

Un attribut a une visibilité :

* **public** sa définition est précédée de *public*, et il peut être utilisé par tout utilisateur de la classe.
* **privé**sa définition est précédée de *private*, et il ne peut être utilisé qu’à l’intérieur de la classe
* **protégé**sa définition est précédée de *protected*, et il ne peut être utilisé qu’à l’intérieur de la classe, ou des classes dérivées.
* **paquetage** l’attribut peut être utilisé dans toute classe du même paquetage.

A l’intérieur de la définition d’une méthode, l’accès à un attribut se fait soit directement, soit en préfixant l'attribut par **this** qui est une référence à l’objet pour lequel est appelé la méthode.

A l’extérieur de la définition de la classe, l’accès se fait en écrivant le nom de l’instance, un point le nom de l’attribut ou de la méthode.

|  |  |
| --- | --- |
| **class** Point{  **private** **int** x ;  **public** **int** y ;  **void** f(){  x = 0;  **this**.y = 1;  }  } | En dehors de la définition de la classe :  Point p = **new** Point();  p.x = 0; *// Erreur de compilation : la visibilité de x est privée !*  p.y = 1; |

*3.2 Méthode :*

Une méthode est définie par :

* Son type de retour : type de la valeur retournée par la méthode. Si la méthode ne retourne pas de valeur le type spécifié est alors **void**. Le type retourné peut être tableau.
* Son nom
* Ses paramètres : les paramètres sont spécifiés par leur type et leur nom ( éventuellement [] ) et sont séparés par des virgules. Tous les paramètres sont des paramètres transmis par valeur. La plupart des paramètres ( sauf ceux de type primitif ) étant des références, la transmission de paramètre est en fait une transmission par référence : la valeur référencée peut être modifié, mais pas la référence elle même. On peut avoir un paramètre *ellipse*, et dans ce cas ce paramètre doit être le dernier de la liste des paramètres. Ce paramètre est en fait un tableau. L'appel de la méthode peut alors se faire avec un nombre variable de paramètres effectifs.  
  Exemple :
* **void** m(X ... e){
* *e est un tableau à une dimension d'éléments de type X*

}

Un appel de la méthode *m* se fait alors de la façon suivante :

m(**null**);

m();

m(x1);

m(x1, x2);

m(x1, x2, x3)

...

*ce dernier appel est équivalent à :*

X[] t={x1, x2, x3};

m(t);

*ou*

m(**new** X[]{x1, x2, x3});

}

Une méthode est caractérisée par sa ***signature*** :

1. son nom.
2. la liste des types des paramètres, dans l'ordre.

Dans une classe deux méthodes différentes ne peuvent pas avoir la même signature. Deux méthodes ayant le même nom, mais des liste de types de paramètres différentes sont des surcharges l'une de l'autre.

 Dans le corps d’une méthode, on peut utiliser la référence **this**, qui référence l’objet pour lequel la méthode est appelée, cet usage étant facultatif.  
**this** ne peut pas être modifié.

Une méthode a une visibilité :

* **public** sa définition est précédée de *public*, et elle peut être utilisée par tout utilisateur de la classe.
* **privé**sa définition est précédée de *private*, et elle ne peut être utilisée qu’à l’intérieur de la classe
* **protégé**sa définition est précédée de *protected*, et elle ne peut être utilisée qu’à l’intérieur de la classe, ou des classes dérivées.
* **paquetage** la méthode peut être utilisée dans toute classe du même paquetage.

A l’extérieur de la définition de la classe, l’accès se fait en écrivant le nom de l’instance, un point le nom de l’attribut ou de la méthode.

4 Constructeurs, finalize

Certaines méthodes sont des **constructeurs** : ils ont pour nom le nom de la classe,  ils ne retournent pas de valeur, et sont appelés au moment de la création d’une instance de la classe.

Si aucun constructeur n’est défini, Java fournit un constructeur sans paramètre appelé constructeur implicite ou par défaut, qui appelle le constructeur implicite de la super classe, les attributs ne sont pas initialisés.

Une classe peut être munie de la méthode **finalize()**. Cette méthode est appelée par le ramasse miettes ( *garbage collector* ) qui permet de récupérer l’espace occupé par les différents instances d’un programme qui ne sont plus utilisées.

5 Attributs et méthodes de classe.

Les attributs ou les méthodes d’une classe peuvent être **static**, et on parle de méthode ou d’attribut de classes, alors que pour les méthodes et attributs non statiques, on parle d’attribut et de méthode d’instances.

Pour les attributs, il n’y a qu’une seule représentation des attributs **static** et non pas une représentation par instance.

Un attribut, ou une méthode *static* peut être utilisé indépendamment de l’existence d’instances, en le préfixant par le nom de la classe. Une méthode *static* ne peut donc accéder qu’à des attributs ou méthodes *static*.

Exemples :

* La classe *Math* ne contient que des attributs statiques ( *PI*, *E* ) et des méthodes statiques( *sqrt*, *exp*, *sin*, *cos*, *log*, …)
* La classe *BigInteger* contient deux attributs *static* (*ONE* et *ZERO*) et une méthode statique *BigInteger valueOf(long)*
* La classe *Font* contient plusieurs attributs *static* parmi lesquels  *BOLD*, *ITALIC*, et *PLAIN*   et des méthodes statiques parmi lesquelles *Font decode(String str)* qui fabrique une fonte à partie de son nom.
* La classe *Color* contient des couleurs prédéfinies *WHITE*, *RED*, *BLACK*, *BLUE*, ...

*5.1 Bloc Static.*

Une classe ( non imbriquée ) peut être munie de un ou plusieurs blocs *static* qui sont exécutés au chargement de la classe.

Un bloc *static* est défini par :

**static**{

. . .

}

Un bloc statique peut servir à l’initialisation des attributs *static*, ou au chargement de librairies.

6 La classe *Object*

La classe *Object* est la classe racine de toute la hiérarchie des objets Java, y compris les tableaux.

|  |  |
| --- | --- |
| **protected**  Object clone() | Création d’une copie de l’objet.   * Si l’objet n’implémente pas l’interface *Cloneable*, alors une exception  *CloneNotSupportedException* est levée. Tous les tableaux implémentent l’interface *Cloneable*. * Sinon la méthode crée une nouvelle instance de la classe et initialise tous ses attributs avec les valeur des attributs de l’objet : si l’attribut est une référence c’est la référence qui est copiée : il n’y a pas **copie en profondeur**. |
| **public** **boolean** equals(Object obj) | Retourne *true* si un objet est égal à un autre. La méthode doit être :   * Réflexive * Transitive * Symétrique |
| **protected  void** finalize() | Appelée par le ramasse-miettes, quand il n’y a plus de référence à l’objet. |
| **public** Class getClass() | Retourne la classe de l’objet. |
| **public** **int hashCode()** | Retourne une valeur de *hash-code* pour l’objet. |
| **public** **void** notify() | Réveille un seul *thread* en attente sur le moniteur de l’objet. |
| **public** **void** notifyAll() | Réveille tous les *threads* en attente sur le moniteur de l’objet. |
| **public** String toString() | Retourne une représentation en chaîne de l’objet : la définition pour *Object* est getClass().getName() + '@' + Integer.toHexString(hashCode()). |
| **public** **void** wait() | Met le *thread* courant en attente jusqu’à ce qu’il soit réveillé par un *notify()* ou un *notifyAll()* |
| **public** **void** wait(long timeout) |  |
| **public** **void** wait(long timeout, int nanos) |  |

[*haut de la page*](http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/CoursJava/classes1.html#haut)

***Héritage***

|  |
| --- |
| **Sommaire**   1. [Définitions](http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/CoursJava/classes2.html#def) 2. [Constructeurs et héritage](http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/CoursJava/classes2.html#constH) 3. [*this* et *super*](http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/CoursJava/classes2.html#thisSuper) 4. [Opérateur *instanceof*](http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/CoursJava/classes2.html#instanceof) 5. [Redéfinition des méthodes](http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/CoursJava/classes2.html#redef) 6. [Classe abstraite](http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/CoursJava/classes2.html#abstrait) 7. [Exemples](http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/CoursJava/classes2.html#exemples)    1. [Employés](http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/CoursJava/classes2.html#employe)    2. [Formes](http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/CoursJava/classes2.html#formes) 8. [Construction d'une instance](http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/CoursJava/classes2.html#cons) |

1 Définitions.

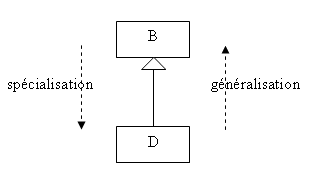
Une classe Java dérive toujours d’un autre classe, *Object* quand rien n’est spécifié. Pour spécifier de quelle classe hérite une classe on utilise le mot-clé *extends* :

**class** D **extends** B {

. . .

}

La classe *D* dérive de la classe *B*. On dit que le classe *B* est la super classe, la classe de base, ou la classe mère de la classe dérivée *D*, et que *D* dérive de *B*, ou que *D*est une sous-classe de *B*.



La visibilité *protected* rend l'accès possible :

* depuis les classes dérivée.
* depuis les classes du paquetage.

Exemple :

|  |  |  |
| --- | --- | --- |
| **class** B{  **private** **int** a;  **protected** **int** b;  **int** c;  **public** **int** d;  }  **class** D **extends** B {  **int** x;  **void** f(){  b = …;  }  } | On peut alors représenter un objet de la classe D de la façon suivante : | représentation mémoire |
|  |  |

Une référence à une classe de base peut être affectée d’une référence à une classe dérivée, l’inverse doit faire l’objet d’une opération de conversion de type ou *cast* :

B b = **new** B();

D d = **new** D();

b = d;

d = b; interdit

d = (D) b ; *// OK avec cast*

Cette dernière instruction peut lever une exception *ClassCastException* si le cast est impossible.

2 Constructeurs et héritage.

* Si, dans une classe, un constructeur est défini sans commencer par un appel de constructeur, Java insère un appel du constructeur par défaut de la super classe.
* On peut commencer le code d’un constructeur par un appel d’un constructeur de la classe de base : super(…), ou d'un autre constructeur de la classe this(…) alors Java n’ajoute rien.

|  |  |
| --- | --- |
| **class** X{  **int** x;    **public** X (**int** x){  **this**.x = x;  }  } | **class** Y **extends** X{  }  provoque l'erreur de compilation : le super constructeur implicite *X()* n'est pas défini pour le constructeur par défaut. |

3 **this** et **super**.

Chaque instance est munie de deux références particulières :

* **this** réfère l’instance elle-même.
* **super** réfère la partie héritée de l’instance.

4 Opérateur **instanceof**.

L’opérateur ***instanceof*** permet de savoir à quelle classe appartient une instance :

**class** B{ …}

**class** D **extends** B{…}

**class** C {…}

B b = **new** B();

D d = **new** D();

C c = **new** C();

b **instanceof** B *// true*

b **instanceof** D *// false*

d **instanceof** B *// true*

d **instanceof** D *// true*

b = d;

b **instanceof** B *// true*

b **instanceof** D *// true*

c **instanceof** B *// erreur de compilation Erreur No. 365 :*

*// impossible de comparer C avec B en ligne …, colonne ..*

Exemple  la méthode *equals(Object o)*, pour une classe *X*, est définie, en général, de la façon suivante :

**public** **boolean** equals(Object o){

**if**(**this**==o) **return** **true**;

**if**(!(o **instanceof** X) **return** **false**;

X x = (X)o;

...

}

5 Redéfinition des méthodes

Une méthode définie dans une classe peut être redéfinie dans une classe dérivée.

|  |  |
| --- | --- |
| **Sans redéfinition de la méthode *m*** | **Avec redéfinition de la méthode *m*** |
| **class** B {  **void** m(){  System.out.println(  "méthode m de B");  }  }  **class** D **extends** B{  } | **class** B {  **void** m(){  System.out.println(  "méthode m de B");  }  }  **class** D **extends** B{  **void** m(){  System.out.println(  "méthode m de D");  }  } |
| |  |  | | --- | --- | | B b=**new** B();  D d=**new** D();  b.m();  b = d;  b.m(); | http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/CoursJava/classes2_fichiers/image009.gif | | |  |  | | --- | --- | | B b=**new** B();  D d=**new** D();  b.m();  b=d; b.m(); | http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/CoursJava/classes2_fichiers/image010.gif | |

Remarques :

1. Ne pas confondre la redéfinition avec la surcharge :
   * **Redéfinition** : même type de retour et mêmes paramètres.
   * **Surcharge** : type de retour et paramètres différents, les paramètres au moins doivent être différents.
2. Une classe définie avec le modificateur d’accès *final* ne peut pas être dérivée.
3. Une méthode définie avec le modificateur d’accès *final* ne peut pas être redéfinie dans les classes dérivées. Ceci peut se faire pour des raisons :
   * d’efficacité : le compilateur peur générer du code *inline*, sans faire appel à la méthode.
   * de sécurité : le mécanisme de polymorphisme dynamique fait qu’on ne sait pas quelle méthode va être appelée.

6 Classe abstraite

Une classe définie avec le modificateur *abstract* est une classe abstraite qui ne peut pas produire d’instance. Sa définition peux contenir des méthodes abstraites. En revanche une classe qui contient une méthode abstraite doit être abstraite. Une méthode abstraite est une méthode définie uniquement par son prototype. Le rôle des classes abstraites est la factorisation de fonctionnalités communes à plusieurs classes dérivées.

Exemple : Supposons que nous voulions manipuler deux formes géométriques, rectangle (donné par son point haut, gauche, sa largeur et sa longueur) et cercle (donné par son centre et son rayon). Nous désirons pouvoir déplacer ces formes géométriques, calculer leur surface et leur périmètre .

|  |  |
| --- | --- |
| **sans classe abstraite** | **avec classe abstraite : on peut factoiser *deplaceDe*** |
| http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/CoursJava/classe22.gif | http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/CoursJava/classe23.gif |
| **class** Rectangle{  **int** x, y, largeur, longueur;  Rectangle(**int** x, **int** y, **int** la, **int** lo){  **this**.x = x; **this**.y = y;  largeur = la; longueur = lo;  }  **void** deplaceDe( **int** dx, **int** dy){  x+=dx; y+=dy;  }  **double** perimetre(){  **return** 2\*(largeur+longueur);  }  **double** surface(){  **return** largeur\*longueur;  }  }    **class** Cercle{  **int** x, y, rayon;  Cercle(**int** x, **int** y, **int** r){  **this**.x = x; **this**.y = y;  rayon = r;  }  **void** deplaceDe( **int** dx, **int** dy){  x+=dx; y+=dy;  }  **double** perimetre(){  **return** 2\*Math.PI\*rayon;  }  **double** surface(){  **return** Math.PI\*rayon\*rayon;  }  } | **abstract class** Forme{  **int** x, y;  Forme(**int** x, **int** y){  **this**.x = x; **this**.y = y;  }  **void** deplaceDe( **int** dx, **int** dy){  x+=dx; y+=dy;  }  **abstract double** perimetre();  **abstract double** surface();  }    **class** Rectangle **extends** Forme{  **int** largeur, longueur;  Rectangle(**int** x, **int** y, **int** la, **int** lo){  **super**(x, y);  largeur = la; longueur = lo;  }  **double** perimetre(){  **return** 2\*(largeur+longueur);  }  **double** surface(){  **return** largeur\*longueur;  }  }    **class** Cercle **extends** Forme{  **int** rayon;  Cercle(**int** x, **int** y, **int** r){  **super**(x, y);  rayon = r;  }  **double** perimetre(){  **return** 2\*Math.PI\*rayon;  }  **double** surface(){  **return**Math.PI\*rayon\*rayon;  }  } |

7 Exemples

*7.1 Employé*

Supposons que nous définissions la classe suivante :

**public** **class** Employe{

**protected** String nom ;

**protected** **double** salaire ;

**public** Employe ( String nom, double salaire){

**this**.nom = nom ; **this**.salaire = salaire ;

}

**public** String getNom(){ **return** nom ;}

**public** **double** getSalaire() { **return** salaire ;}

**public**String toString() { **return** getNom()+” ”+getSalaire(); }

}

Un chef «est un» employé qui a une prime en plus du salaire :

**public** **class** Chef **extends** Employe{

**private** **double** prime;

**public** Chef ( String nom, double salaire, double prime){

**super(** nom, salaire) ;

**this**.prime = prime;

}

**public** **double** getSalaire() {**return** salaire + prime;}

}

La méthode *getSalaire* pourrait être (mieux ?) programmée de la façon suivante : cela permettrait de changer les accès de nom et salaire en *private*.

**public** **double** getSalaire() { **return** **super**.getSalaire() + prime;}

Une entreprise est un tableau d’employés :

Employe entreprise[] = **new** Employe[5] ;

entreprise[0]= **new** Chef("Cronos", 1000, 500) ;

entreprise[1]= **new** Chef("Zeus ", 1000, 600) ;

entreprise[2]= **new** Employe("Ares", 620) ;

entreprise[3]= **new** Employe ("Apollon", 700) ;

entreprise[4]= **new** Employe ("Aphrodite ", 100) ;

L’affichage des noms et salaires des différents employés se fera de la façon suivante :

**for** ( **int** i = 0 ; i< 5 ; ++i){

**if** (entreprise[i] **instanceof** Chef) System.out.print("Chef : ");

System.out.println( entreprise[i].toString());

}

Quand on fait un appel de *toString* pour une instance de la classe *Chef*, le code de la méthode *toString* héritée de *Employe* s’exécute avec des appels de *getNom* et de *getSalaire* qui se font pour un *Chef* et ce sont les méthodes *getNom* et *getSalaire* de la classe *Chef* qui sont appelées.

Le calcul de la somme des salaires des employés qui ne sont pas des chefs :

**double** sommeSalaires = 0;

**for** ( **int** i = 0 ; I < 5 ; ++i)

**if**(!(entreprise[i] **instanceof** Chef)) sommeSalaires += entreprise[i].getSalaire();

*7.2 Des formes.*

|  |  |
| --- | --- |
| Nous nous proposons de définir une hiérarchie de classes permettant  la visualisation de formes sur un écran. Les classes *FormeA,* *Forme* et *BiPoints* sont des classes abstraites. | diagramme des classes Formes |

**abstract** **public** **class** FormeA {

**protected** FormeA(){}

**abstract** **public** **void** affiche(Graphics g);

**abstract** **public** **void** cache(Graphics g);

**abstract** **public** **void** deplaceDe(**int** dx, **int** dy, Graphics g);

}

**abstract** **public** **class** Forme **extends** FormeA{

**protected** **int** x, y;

**protected**Color couleur;

**protected** Forme(int x, int y, Color c){

**this**.x = x; **this**.y = y;

couleur = c;

}

**public** **void** deplaceDe(**int** dx, **int** dy, Graphics g){

cache(g);

x+=dx; y += dy;

affiche(g);

}

}

La classe *Point* est une classe concrète pour laquelle nous définissons les fonctionnalités *cache*, *affiche* et *deplaceDe* :

**public** **class** Point **extends** Forme {

**public** Point(**int** **int** y, Color c) {

**super**(x, y, c);

}

**public** **void** affiche(Graphics g) {

g.setColor(couleur);

g.drawRect(x, y, 1, 1);

}

**publicvoid** cache( Graphics g) {

g.setColor(Color.white);

g.drawRect(x, y, 1, 1);

}

}

La classe *Cercle* est une classe concrète pour laquelle nous définissons les fonctionnalités *cache*, *affiche* :

**public** **class** Cercle **extends** Forme {

**int** rayon;

**public** Cercle(**int** x, **int** y, **int** r, Color c) {

**super**( x, y, c);

rayon = r;

}

**public** **void** affiche(Graphics g) {

g.setColor(couleur);

g.drawOval(x-rayon, y-rayon, 2\*rayon, 2\*rayon);

}

**public** **void** cache( Graphics g) {

g.setColor(Color.white);

g.drawOval(x-rayon, y-rayon, 2\*rayon, 2\*rayon);

}

}

La classe abstraite *BiPoint* met en facteur le constructeur et la méthode *deplaceDe* de *Rectangle* et *Segment* :

**abstract** **public** class BiPoint **extends** Forme {

**int** x1, y1;

**public** BiPoint( **int** x, **int** y, **int** x1, **int** y1, Color c) {

**super**(x, y, c);

**this**.x1 = x1; **this**.y1 = y1;

}

**public** **void** deplaceDe(**int** dx, **int** dy, Graphics g) {

cache(g);

x+=dx; y += dy;x1+=dx; y1+=dy;

affiche(g);

}

}

Les classes *Rectangle* et *Segment* se différentient par le tracé :

**public** **class** Rectangle **extends** BiPoint {

**public** Rectangle(int x, int y, int x1, int y1, Color c){

**super**(x, y, x1, y1, c);<

}

**public** **void** affiche(Graphics g) {

g.setColor(couleur);

g.drawRect(x, y, x1-x, y1-y);

}

**public** **void** cache(Graphics g) {

g.setColor(Color.white);

g.drawRect(x, y, x1-x, y1-y);

}

}

**publicclass** Segment **extends** BiPoint{

**public** Segment (int x, int y, int x1, int y1, Color c){

**super**(x, y, x1, y1, c);

}

**public** **void** affiche(Graphics g) {

g.setColor(couleur);

g.drawLine(x, y, x1-x, y1-y);

}

**public** **void** cache(Graphics g) {

g.setColor(Color.white);

g. drawLine (x, y, x1-x, y1-y);

}

}

8 Processus de construction d’une instance

1. Chargement de la classe si elle n’est pas déjà chargée, avec :
   * Chargement de la classe de base ( si elle n’est pas chargée )
   * Construction des attributs statiques et des blocs statiques de la classe dans l’ordre de leurs définitions( avec éventuellement chargement des classes et exécution de leurs statiques)
2. Allocation de la mémoire.
3. Validation du polymorphisme : pour toute méthode appelée par la suite il sera tenu compte du polymorphisme.
4. Appel du constructeur de la super classe
5. Initialisation des attributs ( définis avec une initialisation )
6. Exécution du corps du constructeur.

Exemple soient les 3 classes suivantes :

**public** **class** Base {

**static**{

System.out.println("bloc statique Base ");

}

**public** Base() {

System.out.println("constructeur de Base ");

}

}

**public** **class** Derivee extends Base {<

**static** {

System.out.println("bloc statique 1 de Derivee ");

}

**static** UneAutre x1 = **new** UneAutre(1);

UneAutre x2 = **new** UneAutre(2);

**static** {

System.out.println("bloc statique 2 de Derivee ");

}

**public** Derivee() {

System.out.println("constructeur de Derivee ");

}

}

**public** **class** UneAutre {

**static** {

System.out.println("statique de UneAutre ");

}

**public** UneAutre(**int** i) {

System.out.println("constructeur de UneAutre "+ i);

}

}

La création d’une instance de la classe *Derivee* produit :

|  |  |
| --- | --- |
| bloc statique de Base | Exécution de 1.a |
| bloc statique 1 de Derivee | Exécution de 1.b |
| statique de UneAutre | Exécution de 1.b |
| constructeur de UneAutre 1 | Exécution de 1.b |
| bloc statique 2 de Derivee | Exécution de 1.b |
| constructeur de Base | Exécution de 4 |
| constructeur de UneAutre 2 | Exécution de 5 |
| constructeur de Derivee | Exécution de 6 |

***Héritage***

|  |
| --- |
| **Sommaire**   1. [Définitions](http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/CoursJava/classes2.html#def) 2. [Constructeurs et héritage](http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/CoursJava/classes2.html#constH) 3. [*this* et *super*](http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/CoursJava/classes2.html#thisSuper) 4. [Opérateur *instanceof*](http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/CoursJava/classes2.html#instanceof) 5. [Redéfinition des méthodes](http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/CoursJava/classes2.html#redef) 6. [Classe abstraite](http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/CoursJava/classes2.html#abstrait) 7. [Exemples](http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/CoursJava/classes2.html#exemples)    1. [Employés](http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/CoursJava/classes2.html#employe)    2. [Formes](http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/CoursJava/classes2.html#formes) 8. [Construction d'une instance](http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/CoursJava/classes2.html#cons) |

1 Définitions.

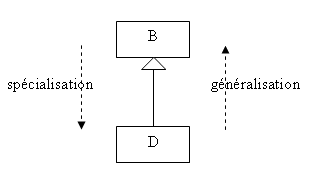
Une classe Java dérive toujours d’un autre classe, *Object* quand rien n’est spécifié. Pour spécifier de quelle classe hérite une classe on utilise le mot-clé *extends* :

**class** D **extends** B {

. . .

}

La classe *D* dérive de la classe *B*. On dit que le classe *B* est la super classe, la classe de base, ou la classe mère de la classe dérivée *D*, et que *D* dérive de *B*, ou que *D*est une sous-classe de *B*.



La visibilité *protected* rend l'accès possible :

* depuis les classes dérivée.
* depuis les classes du paquetage.

Exemple :

|  |  |  |
| --- | --- | --- |
| **class** B{  **private** **int** a;  **protected** **int** b;  **int** c;  **public** **int** d;  }  **class** D **extends** B {  **int** x;  **void** f(){  b = …;  }  } | On peut alors représenter un objet de la classe D de la façon suivante : | représentation mémoire |
|  |  |

Une référence à une classe de base peut être affectée d’une référence à une classe dérivée, l’inverse doit faire l’objet d’une opération de conversion de type ou *cast* :

B b = **new** B();

D d = **new** D();

b = d;

d = b; interdit

d = (D) b ; *// OK avec cast*

Cette dernière instruction peut lever une exception *ClassCastException* si le cast est impossible.

2 Constructeurs et héritage.

* Si, dans une classe, un constructeur est défini sans commencer par un appel de constructeur, Java insère un appel du constructeur par défaut de la super classe.
* On peut commencer le code d’un constructeur par un appel d’un constructeur de la classe de base : super(…), ou d'un autre constructeur de la classe this(…) alors Java n’ajoute rien.

|  |  |
| --- | --- |
| **class** X{  **int** x;    **public** X (**int** x){  **this**.x = x;  }  } | **class** Y **extends** X{  }  provoque l'erreur de compilation : le super constructeur implicite *X()* n'est pas défini pour le constructeur par défaut. |

3 **this** et **super**.

Chaque instance est munie de deux références particulières :

* **this** réfère l’instance elle-même.
* **super** réfère la partie héritée de l’instance.

4 Opérateur **instanceof**.

L’opérateur ***instanceof*** permet de savoir à quelle classe appartient une instance :

**class** B{ …}

**class** D **extends** B{…}

**class** C {…}

B b = **new** B();

D d = **new** D();

C c = **new** C();

b **instanceof** B *// true*

b **instanceof** D *// false*

d **instanceof** B *// true*

d **instanceof** D *// true*

b = d;

b **instanceof** B *// true*

b **instanceof** D *// true*

c **instanceof** B *// erreur de compilation Erreur No. 365 :*

*// impossible de comparer C avec B en ligne …, colonne ..*

Exemple  la méthode *equals(Object o)*, pour une classe *X*, est définie, en général, de la façon suivante :

**public** **boolean** equals(Object o){

**if**(**this**==o) **return** **true**;

**if**(!(o **instanceof** X) **return** **false**;

X x = (X)o;

...

}

5 Redéfinition des méthodes

Une méthode définie dans une classe peut être redéfinie dans une classe dérivée.

|  |  |
| --- | --- |
| **Sans redéfinition de la méthode *m*** | **Avec redéfinition de la méthode *m*** |
| **class** B {  **void** m(){  System.out.println(  "méthode m de B");  }  }  **class** D **extends** B{  } | **class** B {  **void** m(){  System.out.println(  "méthode m de B");  }  }  **class** D **extends** B{  **void** m(){  System.out.println(  "méthode m de D");  }  } |
| |  |  | | --- | --- | | B b=**new** B();  D d=**new** D();  b.m();  b = d;  b.m(); | http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/CoursJava/classes2_fichiers/image009.gif | | |  |  | | --- | --- | | B b=**new** B();  D d=**new** D();  b.m();  b=d; b.m(); | http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/CoursJava/classes2_fichiers/image010.gif | |

Remarques :

1. Ne pas confondre la redéfinition avec la surcharge :
   * **Redéfinition** : même type de retour et mêmes paramètres.
   * **Surcharge** : type de retour et paramètres différents, les paramètres au moins doivent être différents.
2. Une classe définie avec le modificateur d’accès *final* ne peut pas être dérivée.
3. Une méthode définie avec le modificateur d’accès *final* ne peut pas être redéfinie dans les classes dérivées. Ceci peut se faire pour des raisons :
   * d’efficacité : le compilateur peur générer du code *inline*, sans faire appel à la méthode.
   * de sécurité : le mécanisme de polymorphisme dynamique fait qu’on ne sait pas quelle méthode va être appelée.

6 Classe abstraite

Une classe définie avec le modificateur *abstract* est une classe abstraite qui ne peut pas produire d’instance. Sa définition peux contenir des méthodes abstraites. En revanche une classe qui contient une méthode abstraite doit être abstraite. Une méthode abstraite est une méthode définie uniquement par son prototype. Le rôle des classes abstraites est la factorisation de fonctionnalités communes à plusieurs classes dérivées.

Exemple : Supposons que nous voulions manipuler deux formes géométriques, rectangle (donné par son point haut, gauche, sa largeur et sa longueur) et cercle (donné par son centre et son rayon). Nous désirons pouvoir déplacer ces formes géométriques, calculer leur surface et leur périmètre .

|  |  |
| --- | --- |
| **sans classe abstraite** | **avec classe abstraite : on peut factoiser *deplaceDe*** |
| http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/CoursJava/classe22.gif | http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/CoursJava/classe23.gif |
| **class** Rectangle{  **int** x, y, largeur, longueur;  Rectangle(**int** x, **int** y, **int** la, **int** lo){  **this**.x = x; **this**.y = y;  largeur = la; longueur = lo;  }  **void** deplaceDe( **int** dx, **int** dy){  x+=dx; y+=dy;  }  **double** perimetre(){  **return** 2\*(largeur+longueur);  }  **double** surface(){  **return** largeur\*longueur;  }  }    **class** Cercle{  **int** x, y, rayon;  Cercle(**int** x, **int** y, **int** r){  **this**.x = x; **this**.y = y;  rayon = r;  }  **void** deplaceDe( **int** dx, **int** dy){  x+=dx; y+=dy;  }  **double** perimetre(){  **return** 2\*Math.PI\*rayon;  }  **double** surface(){  **return** Math.PI\*rayon\*rayon;  }  } | **abstract class** Forme{  **int** x, y;  Forme(**int** x, **int** y){  **this**.x = x; **this**.y = y;  }  **void** deplaceDe( **int** dx, **int** dy){  x+=dx; y+=dy;  }  **abstract double** perimetre();  **abstract double** surface();  }    **class** Rectangle **extends** Forme{  **int** largeur, longueur;  Rectangle(**int** x, **int** y, **int** la, **int** lo){  **super**(x, y);  largeur = la; longueur = lo;  }  **double** perimetre(){  **return** 2\*(largeur+longueur);  }  **double** surface(){  **return** largeur\*longueur;  }  }    **class** Cercle **extends** Forme{  **int** rayon;  Cercle(**int** x, **int** y, **int** r){  **super**(x, y);  rayon = r;  }  **double** perimetre(){  **return** 2\*Math.PI\*rayon;  }  **double** surface(){  **return**Math.PI\*rayon\*rayon;  }  } |

7 Exemples

*7.1 Employé*

Supposons que nous définissions la classe suivante :

**public** **class** Employe{

**protected** String nom ;

**protected** **double** salaire ;

**public** Employe ( String nom, double salaire){

**this**.nom = nom ; **this**.salaire = salaire ;

}

**public** String getNom(){ **return** nom ;}

**public** **double** getSalaire() { **return** salaire ;}

**public**String toString() { **return** getNom()+” ”+getSalaire(); }

}

Un chef «est un» employé qui a une prime en plus du salaire :

**public** **class** Chef **extends** Employe{

**private** **double** prime;

**public** Chef ( String nom, double salaire, double prime){

**super(** nom, salaire) ;

**this**.prime = prime;

}

**public** **double** getSalaire() {**return** salaire + prime;}

}

La méthode *getSalaire* pourrait être (mieux ?) programmée de la façon suivante : cela permettrait de changer les accès de nom et salaire en *private*.

**public** **double** getSalaire() { **return** **super**.getSalaire() + prime;}

Une entreprise est un tableau d’employés :

Employe entreprise[] = **new** Employe[5] ;

entreprise[0]= **new** Chef("Cronos", 1000, 500) ;

entreprise[1]= **new** Chef("Zeus ", 1000, 600) ;

entreprise[2]= **new** Employe("Ares", 620) ;

entreprise[3]= **new** Employe ("Apollon", 700) ;

entreprise[4]= **new** Employe ("Aphrodite ", 100) ;

L’affichage des noms et salaires des différents employés se fera de la façon suivante :

**for** ( **int** i = 0 ; i< 5 ; ++i){

**if** (entreprise[i] **instanceof** Chef) System.out.print("Chef : ");

System.out.println( entreprise[i].toString());

}

Quand on fait un appel de *toString* pour une instance de la classe *Chef*, le code de la méthode *toString* héritée de *Employe* s’exécute avec des appels de *getNom* et de *getSalaire* qui se font pour un *Chef* et ce sont les méthodes *getNom* et *getSalaire* de la classe *Chef* qui sont appelées.

Le calcul de la somme des salaires des employés qui ne sont pas des chefs :

**double** sommeSalaires = 0;

**for** ( **int** i = 0 ; I < 5 ; ++i)

**if**(!(entreprise[i] **instanceof** Chef)) sommeSalaires += entreprise[i].getSalaire();

*7.2 Des formes.*

|  |  |
| --- | --- |
| Nous nous proposons de définir une hiérarchie de classes permettant la  visualisation de formes sur un écran. Les classes *FormeA,* *Forme* et *BiPoints* sont des classes abstraites. | diagramme des classes Formes |

**abstract** **public** **class** FormeA {

**protected** FormeA(){}

**abstract** **public** **void** affiche(Graphics g);

**abstract** **public** **void** cache(Graphics g);

**abstract** **public** **void** deplaceDe(**int** dx, **int** dy, Graphics g);

}

**abstract** **public** **class** Forme **extends** FormeA{

**protected** **int** x, y;

**protected**Color couleur;

**protected** Forme(int x, int y, Color c){

**this**.x = x; **this**.y = y;

couleur = c;

}

**public** **void** deplaceDe(**int** dx, **int** dy, Graphics g){

cache(g);

x+=dx; y += dy;

affiche(g);

}

}

La classe *Point* est une classe concrète pour laquelle nous définissons les fonctionnalités *cache*, *affiche* et *deplaceDe* :

**public** **class** Point **extends** Forme {

**public** Point(**int** **int** y, Color c) {

**super**(x, y, c);

}

**public** **void** affiche(Graphics g) {

g.setColor(couleur);

g.drawRect(x, y, 1, 1);

}

**publicvoid** cache( Graphics g) {

g.setColor(Color.white);

g.drawRect(x, y, 1, 1);

}

}

La classe *Cercle* est une classe concrète pour laquelle nous définissons les fonctionnalités *cache*, *affiche* :

**public** **class** Cercle **extends** Forme {

**int** rayon;

**public** Cercle(**int** x, **int** y, **int** r, Color c) {

**super**( x, y, c);

rayon = r;

}

**public** **void** affiche(Graphics g) {

g.setColor(couleur);

g.drawOval(x-rayon, y-rayon, 2\*rayon, 2\*rayon);

}

**public** **void** cache( Graphics g) {

g.setColor(Color.white);

g.drawOval(x-rayon, y-rayon, 2\*rayon, 2\*rayon);

}

}

La classe abstraite *BiPoint* met en facteur le constructeur et la méthode *deplaceDe* de *Rectangle* et *Segment* :

**abstract** **public** class BiPoint **extends** Forme {

**int** x1, y1;

**public** BiPoint( **int** x, **int** y, **int** x1, **int** y1, Color c) {

**super**(x, y, c);

**this**.x1 = x1; **this**.y1 = y1;

}

**public** **void** deplaceDe(**int** dx, **int** dy, Graphics g) {

cache(g);

x+=dx; y += dy;x1+=dx; y1+=dy;

affiche(g);

}

}

Les classes *Rectangle* et *Segment* se différentient par le tracé :

**public** **class** Rectangle **extends** BiPoint {

**public** Rectangle(int x, int y, int x1, int y1, Color c){

**super**(x, y, x1, y1, c);<

}

**public** **void** affiche(Graphics g) {

g.setColor(couleur);

g.drawRect(x, y, x1-x, y1-y);

}

**public** **void** cache(Graphics g) {

g.setColor(Color.white);

g.drawRect(x, y, x1-x, y1-y);

}

}

**publicclass** Segment **extends** BiPoint{

**public** Segment (int x, int y, int x1, int y1, Color c){

**super**(x, y, x1, y1, c);

}

**public** **void** affiche(Graphics g) {

g.setColor(couleur);

g.drawLine(x, y, x1-x, y1-y);

}

**public** **void** cache(Graphics g) {

g.setColor(Color.white);

g. drawLine (x, y, x1-x, y1-y);

}

}

8 Processus de construction d’une instance

1. Chargement de la classe si elle n’est pas déjà chargée, avec :
   * Chargement de la classe de base ( si elle n’est pas chargée )
   * Construction des attributs statiques et des blocs statiques de la classe dans l’ordre de leurs définitions( avec éventuellement chargement des classes et exécution de leurs statiques)
2. Allocation de la mémoire.
3. Validation du polymorphisme : pour toute méthode appelée par la suite il sera tenu compte du polymorphisme.
4. Appel du constructeur de la super classe
5. Initialisation des attributs ( définis avec une initialisation )
6. Exécution du corps du constructeur.

Exemple soient les 3 classes suivantes :

**public** **class** Base {

**static**{

System.out.println("bloc statique Base ");

}

**public** Base() {

System.out.println("constructeur de Base ");

}

}

**public** **class** Derivee extends Base {<

**static** {

System.out.println("bloc statique 1 de Derivee ");

}

**static** UneAutre x1 = **new** UneAutre(1);

UneAutre x2 = **new** UneAutre(2);

**static** {

System.out.println("bloc statique 2 de Derivee ");

}

**public** Derivee() {

System.out.println("constructeur de Derivee ");

}

}

**public** **class** UneAutre {

**static** {

System.out.println("statique de UneAutre ");

}

**public** UneAutre(**int** i) {

System.out.println("constructeur de UneAutre "+ i);

}

}

La création d’une instance de la classe *Derivee* produit :

|  |  |
| --- | --- |
| bloc statique de Base | Exécution de 1.a |
| bloc statique 1 de Derivee | Exécution de 1.b |
| statique de UneAutre | Exécution de 1.b |
| constructeur de UneAutre 1 | Exécution de 1.b |
| bloc statique 2 de Derivee | Exécution de 1.b |
| constructeur de Base | Exécution de 4 |
| constructeur de UneAutre 2 | Exécution de 5 |
| constructeur de Derivee | Exécution de 6 |

***Classes interne, locale et anonyme***

|  |
| --- |
| **Sommaire**   1. [Classe interne](http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/CoursJava/classes3.html#interne) 2. [Classe locale](http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/CoursJava/classes3.html#locale) 3. [Classe anonyme](http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/CoursJava/classes3.html#anonyme) |

1 Classe interne.

Le langage Java autorise la définition d’une classe à l’intérieur d’une autre classe :

**class** Englobante {

**int** a;

**class** Englobee{

. . .

Englobante.**this**.a = 12

}

. . .

}

Un des intérêts de cette notation est qu’une classe interne peut avoir accès aux méthodes est attributs de la classe englobante.

Comme les attributs et les méthodes les classes internes ont :

* un accès *public*, *protected*, *private* ou *paquetage*, l’accès porte sur la classe elle-même et non sur ses attributs.
* **class** A {
* **private** **class** B{
* **int** c;
* . . .
* }
* . . .
* B b = **new** B();
* }
* A a = **new**A();
* a.b.c = … ; *// b est non visible*
* a.b.getClass(); *// b est non visible*
* une qualification qui peut être *static* : alors une instance de la classe  interne a une existence  indépendante de l’existence d’une instance de la classe *Englobante*. On peut alors créer une instance de la classe englobée, sans qu'il existe d'instance de la classe englobante Dans ce cas, on ne peut évidemment pas accéder aux attributs ou méthodes autres que *static*, de la classe englobante.

Les créations d’instances se font de la façon suivante :

|  |  |
| --- | --- |
| Classe interne **static** | Class interne **non static** |
| **public** class A {  **int** a = 321;  **static** **class** B{  **int** b = 123;  }  } | **public** class A {  **int** a = 321;  **class** B{  **int** b = 123;  }  } |
| A a = **new** A();  A.B b = **new** A.B(); | A a = **new** A();  A.B b = a.**new** B(); |

Lors de la compilation, le compilateur génère un fichier constitué du nom de la classe englobante puis un ‘$’, puis le nom de la classe interne, puis “.class” : exemple : A$B.class

2 Classe locale.

Une classe interne définie dans un bloc est une classe interne dont la portée est limitée au bloc : c’est une classe *interne locale*. Une classe locale ne peut pas être static. Une classe locale peut accéder aux attributs de la classe englobante ainsi qu'aux paramètres et variable locales de la méthode où elle est définie, à condition que ceux-ci soit spécifiés ***final***.

Lors de la compilation, le compilateur génère un fichier constitué du nom de la classe englobante puis un ‘$’, puis un nombre, puis le nom de la classe interne, puis “.class” : exemple :

|  |  |
| --- | --- |
| **public** **class** A {  **public** A() {  **class** L{  …  }  }  …  } | Génère A$1L.class |

3 Classe anonyme.

Il est possible de définir une classe interne, sans lui donner de nom par dérivation d’un super classe, ou par implémentation d’une interface.

|  |  |
| --- | --- |
| **Dérivation de super classe** | **Implémentation d’interface** |
| SuperClasse c = **new** SuperClasse( ){  *// méthodes redéfinies*  }; | Interface c = **new** Interface(){  *// implementation des*  *// méthodes de Interface*  }; |

Lors de la compilation, le compilateur génère un fichier constitué du nom de la classe englobante puis un ‘$’, puis un nombre puis “.class”.

Les classes anonymes sont (surtout) utilisées pour implémenter les méthodes d’une interface *Listener*.

Exemple :

jTextArea.addKeyListener(**new** java.awt.event.KeyAdapter() {

**public** **void** keyReleased(java.awt.event.KeyEvent e) {

*...*

}

});

***Exceptions***

|  |
| --- |
| **Sommaire**   1. [Définitions](http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/CoursJava/exceptions.html#def) 2. [Traitement d'une exception](http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/CoursJava/exceptions.html#trait) 3. [Hierarchie des exceptions](http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/CoursJava/exceptions.html#hierarchie) 4. [Classe *Throwable*](http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/CoursJava/exceptions.html#throwable) |

1 Définitions

|  |  |
| --- | --- |
| **Levée d’exception** | **Surveillance et traitement d’exception** |
| * **throw** : une exception est levée lorsqu’une situation exceptionnelle se produit. | * **try** : un bloc d’instructions est susceptible de provoquer une situation exceptionnelle, c'est-à-dire une levée d’exception : il est surveillé. * **catch**: une exception levée dans un bloc surveillé peut être capturée et traitée. |

Exemple :

**for**( **int** i = 0; ;++i) System.out.println(t[i]);

    est une séquence d’instructions qui va lever une exception de type *ArrayIndexOutOfBoundsException*lorsque *i* va être égal à *t.length*.

On peut donc surveiller cette instruction et capturée l’exception qui est levée pour la traiter :

**try**{

**for**( **int** i = 0; ; ++i) System.out.println(t[i]);

}**catch**( IndexOutOfBoundsException iobe){

*// traitement de l’exception*

}

Les intructions surveillées par un bloc *try* peuvent lever plusieurs types d’exceptions et donc le bloc *try* peut être suivi de plusieurs *catch*, et éventuellement d’un *finally*. Lorsque une exception est levée, les *catch* sont parcourus en séquence et le premier qui capture l’exception du type de celle qui a été levée, est celui qui traite l’exception. S’il y a un bloc *finally*, il est exécuté après toutes les instructions du bloc *try*, ou après la dernière instruction qui a traité une exception levée dans le bloc *try*.

Une méthode qui :

* Lève une exception
* Ne traite pas une exception levée par des appels à d’autres méthodes doit donner la liste des exceptions qu’elle peut déclencher. Seules les exceptions des classes *Error*, *RuntimeError* et de leurs dérivées n’ont pas à être citées.

**void** methode(…) **throws** E1, E2{

*// du code qui peut lever*

*// des exceptions de la classe E1 ou E2*

*// sans les capturer*

}

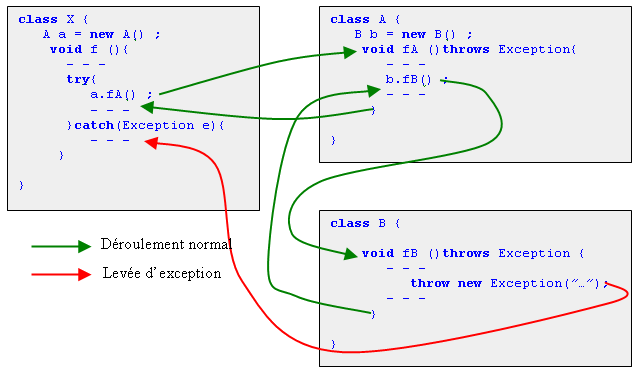
2 Traitement d’une exception

|  |  |
| --- | --- |
| **try**{  *I1 ;*  *I2 ;*  *I3 ;*  }**catch**(Exception1 e1){  *T11;*  }**catch**(Exception2 e2){  *T21;*  *T22;*  }**catch**(Exception3 e3){  . . .  }**finally**{  *F;*  }  *InstructionSuivantes ;* | Si *Exception2* dérive de *Exception1* le *catch* de *Exception2* doit se trouver avant le *catch* de *Exception1*.       La clause *finally* est **toujours** exécutée, qu'il y ait eu une levée d'exception ou non. Elle est aussi exécutée si un *catch* ou le bloc *try* se termine par un *return*, un *break* ou un *continue*. |

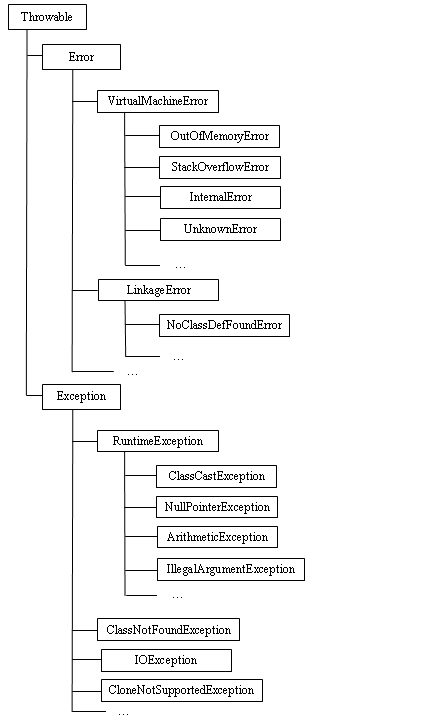
* *I2* ne déclenche pas d’exception : *I3* est exécuté, puis *F1*, puis *InstructionsSuivantes*.
* *I2* déclenche une exception de la classe  *Exception2* : alors il y a exécution de *T21* puis *T22*, puis *F1*, puis *InstructionsSuivantes*. *I3* n’est pas exécutée.

Pour intercepter toutes les exceptions, il suffit d’écrire *catch(Throwable t){ . . . }*.

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



3 Hiérarchie des exceptions



La liste des [*Error*](http://java.sun.com/javase/6/docs/api/java/lang/Error.html) existantes, la liste des [*Exceptions*](http://java.sun.com/javase/6/docs/api/java/lang/Exception.html) existantes, et la liste des exceptions [*Runtime*](http://java.sun.com/javase/6/docs/api/java/lang/RuntimeException.html) existantes.

4 La classe Throwable

La classe *Throwable* est la classe mère de toutes les exceptions et erreurs : seules des instances de *Throwable* ou de ses classes dérivées peuvent être levée par l’instruction *throw* ou être argument d’un *catch*.

Une instance de *Throwable* peut avoir une cause : une autre instance de *Throwable* qui est à l'origine de la création de cette instance («chaînage des exceptions»).

La classe *Throwable* a un constructeur par défaut et un constructeur qui a en argument une chaîne de caractères : le « message » de l’exception.

|  |  |
| --- | --- |
| Throwable() | Le constructeur par défaut. |
| Throwable(String message) | Le constructeur avec un paramètre, le message de l'exception. |
| Throwable(Throwable cause) | Le constructeur avec une cause en paramètre. |
| Throwable(String message, Throwable cause) | Le constructeur avec un message et une cause. |

|  |  |
| --- | --- |
| Throwable fillInStackTrace() |  |
| StackTraceElement[]getStackTrace() | Retourne un tableau représentant l'état de la pile, au moment où a été levée l'exception. Un élément de stack, contient les informations suivantes :   * le nom de la méthode * le numéro de ligne où a été levée l'exception * le nom de la classe * le nom du fichier |
| Throwable initCause(Throwable t) | Initialise la cause de cette instance de *Throwable* avec t. Cette méthode ne peut être appelée qu'une seule fois, sauf si le *Throwable* a déjà été créé avec une cause, auquel cas elle ne peut pas être appelée.  La méthode retourne une référence à l'objet pour lequel elle a été appelée. La méthode lève une exception :   * *IllegalArgumentException* si la cause du *Throwable* est *this*. * *IllegalStateException* si la cause a déjà été créée. |
| String getLocalizedMessage() | Si la méthode n’est pas redéfinie, identique à *getMessage().* |
| String getMessage() | Retourne le message |
| **void** printStackTrace() | Imprime le contenu de la pile dans le fichier d’erreur *err* |
| **void** printStackTrace (PrintStream s) | Imprime le contenu de la pile dans *s* |
| **void** printStackTrace(PrintWriter s) | Imprime le contenu de la pile dans *s* |
| String toString() |  |

Une *Error* est une exception qui indique des problèmes graves : une application ne résout pas ce genre de problème. Une méthode n’a pas à déclarer dans sa clause *throws* les *Error* qui ne seraient pas capturées.

Une *Exception* en revanche peut être capturée et traitée par l’application, ou apparaître dans la liste des exceptions levées par la méthode.

***Paquetages***

Java offre la possibilité de regrouper les classes en ensembles appelés *package*. Ceci permet de ranger les classes de façon hiérarchique.

Exemple : le paquetage java a la structure suivante :

|  |  |
| --- | --- |
| java  | applet  | awt  | beans  | io  | lang  | | ArithmeticException  | | ...  | | Boolean  | | Class  | | Object  | | reflect  | | | ...  | | | Method  | | | ...  | | String  | | Math  | | ...  | net  | rmi  | security  | sql  | text  | util  | ... | Le paquetage contenant les classes concernant les *applet*  L'*abstract Windowing Toolkit*  Le paquetage des classes gérant les entrées-sorties.  Le paquetage contenant les classes de base de Java.  La classe *Classe*  La classe *Object*  Le paquetage des classes pour le réflexion.  La classe *String* représentation des chaînes de caractères.  La classe *Math* contenant PI, E et les méthodes mathématiques usuelles.  Le paquetage des classes réseau.  Le paquetage des classes utilitaires. |

Sous Windows® les paquetages sont effectivement rangés dans une structure de répertoire.

|  |  |
| --- | --- |
| **package** monPaquetage;  **class** X{  ...  } | La classe *X* appartient au paquetage *monPaquetage*. |
| **package** monPaquetage.monSousPaquetage;  **class** Y{  ...  } | La classe *Y* appartient au paquetage *monPaquetage.monSousPaquetage*. |

Lorsqu’on utilise les classes X et Y en dehors de leur paquetage, on peut :

* Soit les désigner complètement *monPaquetage.X*, ou *monPaquetage.monSousPaquetage.Y*
* Soit importer les paquetages, puis désigner les classes par leur nom :
* **import** monPaquetage.\*;
* **import** monSousPaquetage.\*;

X ou Y

Il y a problème si deux paquetages différents contiennent deux classes de même nom :

Les paquetages *java.util*et *java.sql*contiennent tous les deux une classe *Date* :

**import** java.util.\*;

**import** java.sql.\*;

...

Date d; *// erreur de compilation*

La solution est alors de nommer complètement la classe :

**import** java.util.\*;

**import** java.sql.\*;

java.util.Date d; *// OK*

Une classe, un attribut ou une méthode, dont la définition n’est pas précédée de *public*, *private* ou *protected*, a une portée limitée au paquetage dans lequel il ou elle est défini.

***Interface***

|  |
| --- |
| **Sommaire**   1. [Définitions](http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/CoursJava/interface.html#definition) 2. [Interface *Comparable*](http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/CoursJava/interface.html#comparable) 3. [Interface *Cloneable*](http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/CoursJava/interface.html#cloneable) 4. [Interface *Iterator*](http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/CoursJava/interface.html#iterator)    1. [Interface *ListIterator*](http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/CoursJava/interface.html#listIterator) 5. [Interface *Serializable*](http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/CoursJava/interface.html#serializable) 6. [Remarques](http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/CoursJava/interface.html#remarques)    1. [Paramètre méthode](http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/CoursJava/interface.html#paramMeth) |

1 Définition.

Une interface définit un comportement (d’une classe) qui doit être implémenté par une classe, sans implémenter ce comportement. C’est un ensemble de méthodes abstraites, et de constantes. Certaines interfaces ( *Cloneable*, *Serializable*, …) sont dites interfaces de «balisage» : elle n’imposent pas la définition d’une méthode, mais indiquent que les objets qui les implémentent ont certaines propriétés.

Les différences entre les interfaces et les classes abstraites :

* Une interface n’implémente aucune méthode.
* Une classe, ou une classe abstraite peut implémenter plusieurs interfaces, mais n’a qu’une super classe, alors qu’une interface peut dériver de plusieurs autres interfaces.
* Des classes non liées hiérarchiquement peuvent implémenter la même interface

2 L'interface *Comparable*

L’interface *Comparable*, nécessite l’implémentation de la méthode *compareTo* :

**interface** Comparable

**int** compareTo(Object o);

}

**Exemple :** ordonner les employés d’une entreprise par ordre croissant des salaires :

On peut créer une classe entreprise de la façon suivante :

**public** **class** Entreprise {

Employe [] lEntreprise;

**public** Entreprise(Employe [] e) {

lEntreprise = **new** Employe[e.length];

**for** (**int** i = 0; i<e.length; ++i)

lEntreprise[i] = e[i];

}

**public** **void** lister(){

**for** (**int** i = 0; i<lEntreprise.length; ++i)

System.out.println(lEntreprise[i]);

}

}

Pour lister  les employés de l’entreprise par ordre alphabétique,  on peut utiliser le tri de tableaux de la classe *Arrays*, mais il nécessite que les élément à trier implémentent l’interface *Comparable* :

**public** **void** listerOrdre(){

Arrays.sort(lEntreprise);

**for** (**int** i = 0; i<lEntreprise.length; ++i)

System.out.println(lEntreprise[i]);

}

Donc la classe *Employe* s’écrit :

**class** Employe **implements** Comparable{

. . .

**public** **int** compareTo( Object o ){

Employe e = (Employe)o; *// lève une exception*

*// ClassCastException si o*

*// n’est pas de la classe Employe*

**return** nom.compareTo(e.nom);*// par ordre alphabétique*

**return** (**int**)(salaire-e.salaire);*// par salaire croissant*

}

}

Si on veut pouvoir trier par salaire croissant ou par ordre alphabétique, on définira un attribut de classe booléen *triSalaire* :

**class** Employe **implements** Comparable{

**static boolean** triSalaire = **true**;

. . .

**public** **int** compareTo( Object o ){

Employe e = (Employe)o; *// lève une exception*

*//ClassCastException si o*

*// n’est pas de la classe Employe*

**if**(triSalaire)

**return** (**int**)(salaire-e.salaire);*//par salaire croissant*

**else** **return** nom.compareTo(e.nom); *//par ordre alphabétique*

}

}

La méthode *listerOrdre* de la classe *Entreprise* devient alors :

**public** **void** listerOrdre(**boolean** triSalaire){

Employe.triSalaire = trisalaire ;

Arrays.sort(lEntreprise);

**for** (**int** i = 0; i<lEntreprise.length; ++i)

System.out.println(lEntreprise[i]);

}

3 L'interface *Cloneable*.

C’est une interface de balisage : une classe qui implémente l’interface *Cloneable*, indique à *Object.clone()* qu’il est légal pour cet objet de faire une copie superficielle attribut par attribut pour les instance de cette classe. Une tentative de clonage pour des classes qui n’implémentent pas *Cloneable* se traduit par la levée d’une exception :  *CloneNotSupportedException*. La définition de *clone()* pour la classe *Object* est :

**protected** **native** Object clone() **throws** CloneNotSupportedException{

**if** ( !(**this** **instanceof** Cloneable)) **throw** **new** CloneNotSupportedException();

**else** {

*// copie superficielle des attributs*

}

}

* Une classe qui veut que ses instances puissent être clonés superficiellement devra implémenter l’interface *Cloneable* et redéfinir la méthode *clone()* :
* **class** XSurface **implements** Cloneable{
* . . .
* **public** Object clone() **throws** CloneNotSupportedException {
* **return** **super**.clone();
* }

}

* Une classe qui veut que ses instances puissent être clonés en profondeur devra implémenter l’interface *Cloneable* et redéfinir la méthode *clone()* :
* **class** XProfondeur **implements** Cloneable{
* X x;
* . . .
* **public** Object clone() **throws** CloneNotSupportedException {
* XProfondeur xp = (XProfondeur)**super**.clone() ;
* xp.x = (X)x.clone();
* **return** xp;
* }

}

4 L’interface *Iterator*.

L’interface *Iterator* nécessite l’implémentation des fonctions *hasNext()*, *next()*, *remove()*:

**interface** Iterator {

**boolean** hasNext();

Object next();

**void** remove();

}

L’opération *remove()* est dite optionnelle, c'est-à-dire que son implémentation  pourrait être dans ce cas :

**public** **void** remove() {

**throw** **new** UnsupportedOperationException();

}

*4.1 L'interface ListIterator.*

L'interface*ListIterator* est dérivée de *Iterator*, et ajoute des fonctionnalités de parcours dans le sens inverse, de calcul d’indice, et d’ajout et de modification.

|  |  |
| --- | --- |
| **boolean** hasPrevious() | Retourne vrai si l’élément courant à un élément le précédant |
| Object previous() | Retourne l’élément précédant. |
| **int** nextIndex() | Retourne l’indice de l’élément qui serait retourné par un appel de *next* |
| **int** previousIndex() | Retourne l’indice de l’élément qui serait retourné par un appel de *previous* |
| **void** add(Object o) | Ajoute un élément dans la liste (opération optionnelle) |
| **void** set(Object o) | Remplace le dernier élément retourné par *next* ou *previous* par *o* (opération optionnelle) |

5 L’interface *Serializable*.

La *sérialisation* d'un objet est le processus de sauvegarde d'un objet complet sur fichier, d'où il pourra être restauré à tout moment. Le processus inverse de la sauvegarde ( restauration ) est connu sous le nom de *désérialisation*.

Tous les attributs (qui ne sont pas spécifiés *transient* ) d’une classe implémentant l’interface *Serializable* sont sauvés et restaurés de façon simple en utilisant les classes *ObjectOutputStream* et *ObjectInputStream*.

Exemple : soit la classe suivante, permettant d’implanter une liste, dans laquelle chaque élément possède une référence vers le premier élément entré dans la liste.

**public** **class** MaClasse **implements** Serializable{

String nom;

**int** entier;

MaClasse premier ,suivant;

**public** MaClasse(String n, int e, MaClasse s, MaClasse p) {

**super**() ;

nom = n;

entier = e;

suivant = s;

premier = p ;

}

**public** MaClasse(String n, int e) {

**this**(n, e, null, null);

premier **= this;**

}

}

On peut alors définir un liste :

MaClasse liste = **null** ;

liste = **new** MaClass("un", 1) ; premier = liste ;

liste = **new** MaClass("deux", 2, liste, premier) ;

liste = **new** MaClass("trois", 3, liste, premier) ;

La sérialisation ( sauvegarde ) dans le fichier "serial.tmp" se fait alors simplement de la façon suivante :

**try**{

FileOutputStream sortie = **new** FileOutputStream("serial.tmp");

ObjectOutputStream p = **new** ObjectOutputStream(sortie);

p.writeObject(liste);

p.flush();

sortie.close();

}**catch**(IOException ioe){

System.out.println("erreur dans la sérialisation : "+ ioe);

}

L’intruction *p.writeObject(liste)* lève l’exception *NotSerializableException* si la classe *MaClasse* n’implémente pas l’interface *Serializable*.

La désérialisation est aussi simple :

**try**{

FileInputStream instream = **new** FileInputStream("serial.tmp");

ObjectInputStream p = **new** ObjectInputStream(instream);

liste = (MaClasse)p.readObject();

instream.close();

dlm.clear();

afficher(liste);

}**catch**(IOException ioe){

System.out.println("erreur dans la sérialisation : "+ ioe);

}**catch**(ClassNotFoundException cnfe){

System.out.println (" classe non trouvée : "+ cnfe);

}

Pour éviter les copies d’un objet référencé plusieurs fois, le procédé suivant est appliqué :

* Tous les objets sauvegardés reçoivent un numéro d’ordre 1, 2, 3 …
* Quand un objet est sauvegardé sur disque on commence par chercher si cet objet a déjà été sauvegardé :
  + Si c’est le cas son numéro l’identifie
  + Sinon il est sauvegardé.

|  |  |
| --- | --- |
| La liste précédente est donc stockée de la façon ci-contre : | ce qui est écrit dans le fichier |

6 Remarques

L’héritage multiple est autorisé pour les interfaces  :

**interface** MonInterfaceB {

**void** f();

}

**interface** MonInterface1 **extends** MonInterfaceB {

**void** f1();

}

**interface** MonInterface2 **extends** MonInterfaceB{

**void** f2();

}

**interface** MonInterface **extends** MonInterface1, MonInterface2 {

**void** fd();

}

Les classes implémentant *MonInterface* doivent implémenter *f(), f1(), f2(),* et *fd()*.

Une interface peut servir à définir des constantes :

**interface** Mois{

**final** **int** JANVIER = 1, FERVRIER = 2, … DECEMBRE = 12 ;

}

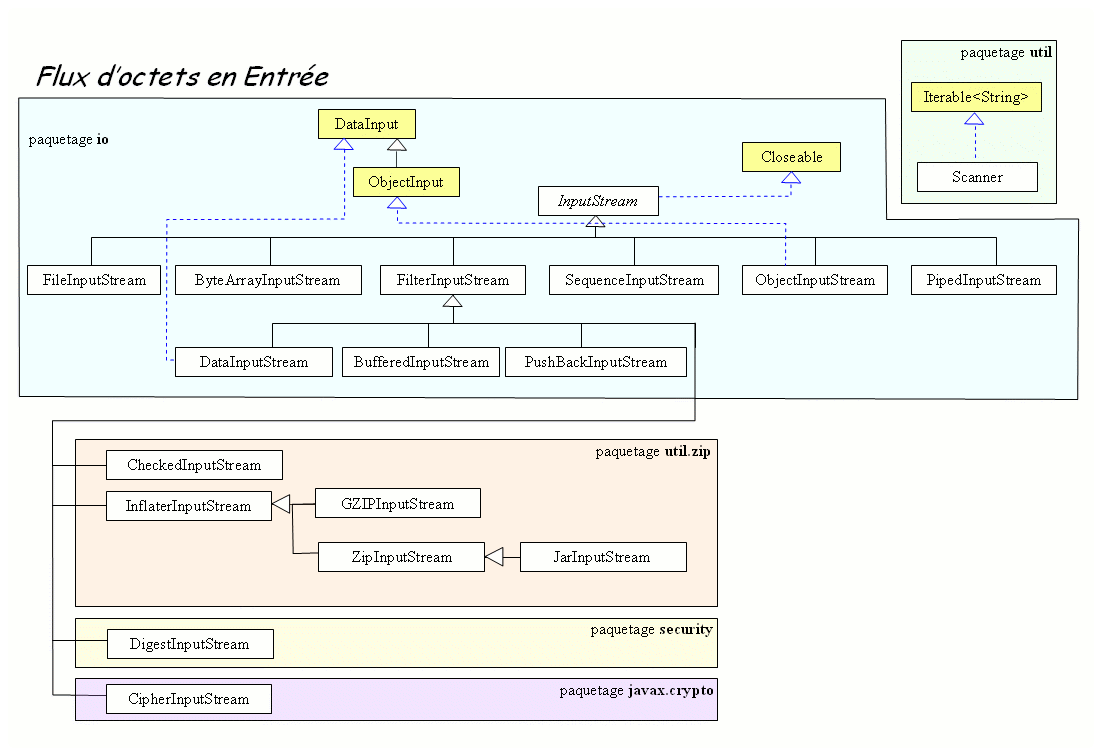
*6.1 Paramètres méthode*

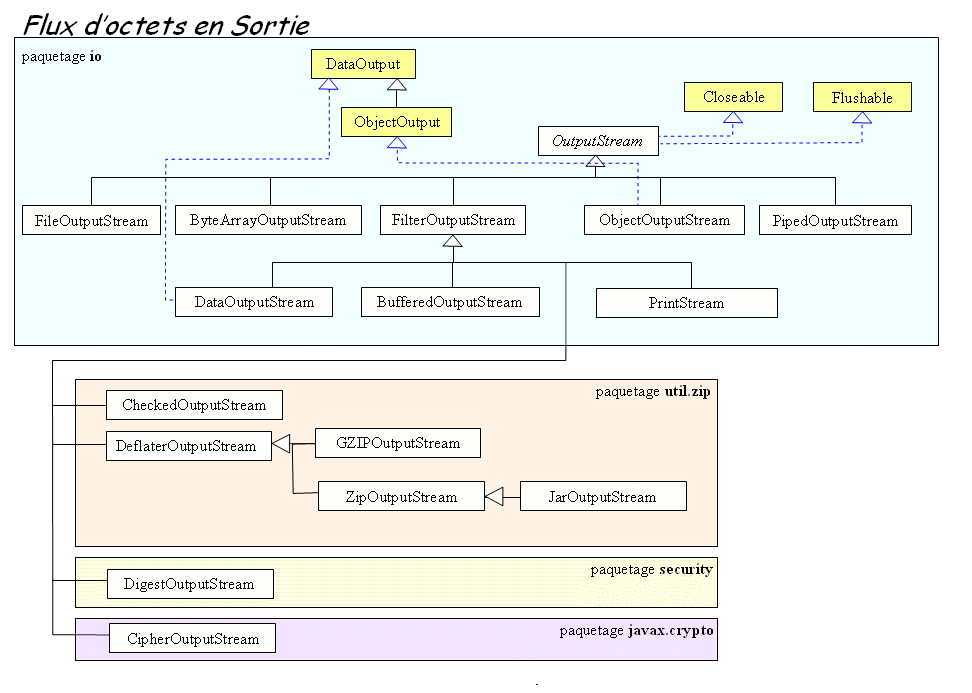
Contrairement à de nombreux langages de programmation, Java n’autorise pas de paramètres qui soient des méthodes. Les interfaces peuvent servir à simuler ce fonctionnement :

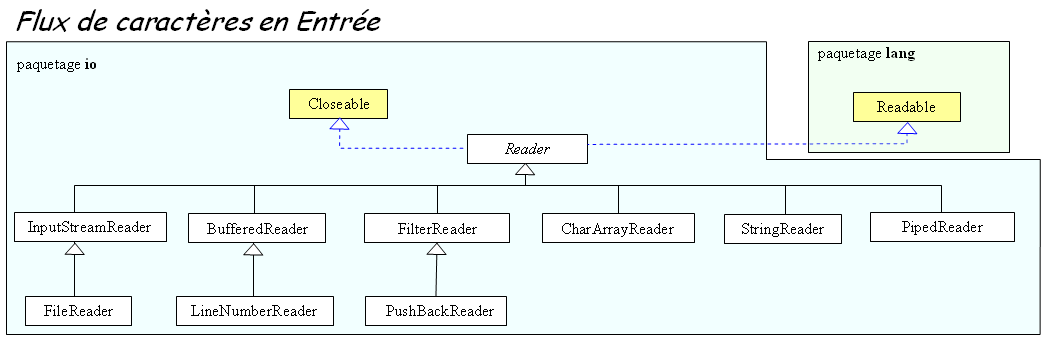
|  |  |
| --- | --- |
| **interface** X{  **void** f(){  }  }  **class** C{  . . .  **public** **void** m(X x){  . . .  x.f() ;  }  } | **class** maClasse  **implements** X{  . . .  **public** **void** f(){  . . .  }  }  C c = **new** C() ;  maClasse mc = **new** MaClasse() ;  c.m(mc)  *// la méthode m va appeler la méthode f*  *// définie dans maClasse.* |

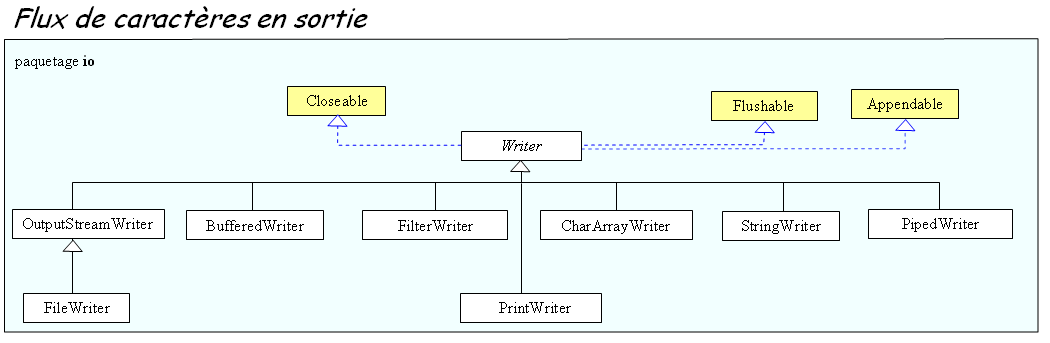
***Flux : généralités***

1. Les flux **séquentiels** : Ces flux peuvent être associés à des fichiers, des tableaux, ou des connexions réseau. Deux «styles» de flux séquentiels :
   * les flux d’octets  en [lecture](http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/Flux/FluxOctetsLectureHierarchie.html) et en [écriture](http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/Flux/FluxOctetsEcritureHierarchie.html) (*InputStream,* *OutputStream*).
   * les flux de caractères en [lecture](http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/Flux/FluxCaract%E8resLectureHierarchie.html) et en [écriture](http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/Flux/FluxCaract%25E8resEcritureHierarchie.html) (*Reader*, *Writer*). Les flux de caractères sont encodés. Si l'[encodage](http://fr.wikipedia.org/wiki/Codage_de_caract%C3%A8res) n'est pas spécifié, c'est l'encodage par défaut qui est utilisé. L'encodage par défaut dépend du système : cp1252 en France sous Windows, UTF-8 sous Linux, ...). Les classes *InputStreamreader*, et *OutputStreamReader* permetent de choisir l'encodage. Ceratins encodages ajoutent, en début de fichier, une marque d'ordre des octets ([BOM](http://fr.wikipedia.org/wiki/Marque_d%27ordre_des_octets)) de 2, 3, ou quatre octets, qui spécifie l'[encodage du fichier](http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/Applis/encodageFichier.jnlp).
2. Les flux à **accès direct**  [*RandomAccessFile*](http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/Flux/RandomAccessFile.html) : ces flux sont forcément associés à des fichiers.









***Scanner***

|  |
| --- |
| **Sommaire**   1. [Délimiteur](http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/Flux/Scanner.html#delimiteur) 2. [Localisation](http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/Flux/Scanner.html#localisation) 3. [Constructeurs](http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/Flux/Scanner.html#constructeurs) 4. [Méthodes](http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/Flux/Scanner.html#méthodes) 5. [Exceptions](http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/Flux/Scanner.html#exceptions) |

La classe *Scanner* permet d'annalyser dans un flux les types primitifs java et les chaînes de caractères en utilisant des expressions régulières.

Exemples :

* on peut lire une valeur entière sur *System.in* de la façon suivante :

Scanner sc = **new** Scanner(System.in);

**int** i = sc.nextInt();

* ou lire un fichier contenant des entiers longs :

Scanner = **new** Scanner(**new** FileInputStream("xxx"));

**while** (sc.hasNextLong(){

**long** l = sc.nextlong();

}

* ou décomposer une chaîne de caractères :

String s = "1 un 2 deux 3 trois";

Scanner sc = **new** Scanner(s);

System.out.println(sc.nextInt()+" "+sc.next());

System.out.println(sc.nextInt()+" "+sc.next());

sc.close();

1 Délimiteur

* Les différentes données du *Scanner* sont délimitées par des délimiteurs : le délimiteur par défaut est le [*Pattern*](http://java.sun.com/j2se/1.5.0/docs/api/java/util/regex/Pattern.html)

\p{javaWhiteSpace}+

c'est à dire un ou plusieurs javaWhiteSpace ( espace, tabulation, retour à la ligne, ...)

* Ce délimiteur peut être modifié par la méthode *useDelimiter(String s)* ou *useDelimiter(Pattern p)* :
* String s = "1/aun//a2////adeux/a3/atrois";
* Scanner sc = **new** Scanner(s);
* sc.useDelimiter( Pattern.compile("/+a")); *// un ou plusieurs / suivis de a*
* System.out.println(sc.nextInt()+" "+sc.next());
* System.out.println(sc.nextInt()+" "+sc.next());
* sc.close();

2 Localisation

* La classe *Scanner* est capable de scanner les nombres dans la forme locale :
* String s = "1,2 2,3";
* Scanner sc = **new** Scanner(s);
* System.out.println(sc.nextDouble());
* System.out.println(sc.nextDouble());

sc.close();

* On peut changer la localisation pour un Scanner par la méthode *useLocale(locale l)* :
* String s = "1.2 2.3";
* Scanner sc = **new** Scanner(s);
* sc.useLocale(Locale.ENGLISH);
* System.out.println(sc.nextDouble());
* System.out.println(sc.nextDouble());

sc.close();

3 Constructeurs

|  |  |
| --- | --- |
| Scanner(File source) | Construit un nouveau *Scanner* à partir du *File*  *f*. Les octets lus sont convertis en caractères en utilisant le *charSet* par défaut. |
| Scanner(File source, String charsetName) | Construit un nouveau *Scanner*File  *f*. Les octets lus sont convertis en caractères en utilisant le *charSet* spécifié. |
| Scanner(InputStream source) | Construit un nouveau *Scanner à partir du flux d'octets source. Les octets lus sont convertis en caractères en utilisant le charSet par défaut.* |
| Scanner(InputStream source, String charsetName) | Construit un nouveau *Scanner* à partir du flux d'octets *source*. Les octets lus sont convertis en caractères en utilisant le *charSet* spécifié. |
| Scanner(Readable source) | Construit un nouveau *Scanner* à partir du flux de caractères  *source*. |
| Scanner(String source) | Construit un nouveau *Scanner* à partir de la chaîne  *source*. |

Un *charSet* est un mapping entre le codage unicode des caractères et des codages de caractères sous forme de séquences d'octets :

|  |  |
| --- | --- |
| US-ASCII | 7-bit [ASCII](http://fr.wikipedia.org/wiki/ASCII), a.k.a. ISO646-US, a.k.a. l'ensemble des caractères latins de base. |
| [ISO-8859-1](http://fr.wikipedia.org/wiki/ISO-8859-1) | ISO Latin Alphabet No. 1, a.k.a. ISO-LATIN-1 |
| [UTF-8](http://fr.wikipedia.org/wiki/Unicode#UTF-8) | 8-bit UCS Transformation Format : chaque caractère est codé par une suite de 1 à 4 octets |
| UTF-16BE | 16-bit UCS Transformation Format, big-endian byte order |
| UTF-16LE | 16-bit UCS Transformation Format, little-endian byte order |
| [UTF-16](http://fr.wikipedia.org/wiki/Unicode#UTF-16) | 16-bit UCS Transformation Format, chaque caractère est codé par 2 ou 4 octets. |

4 Méthodes

Les principales méthodes ont la forme *boolean hasNextX()* et *X nextX()* pour *X*  qui est :

* un type primitif
* un des types *BigDecimal*, *BigInteger*

et les méthodes suivantes :

|  |  |
| --- | --- |
| **boolean** hasNext() | Retourne *true* s'il existe un autre token dans l'entrée. |
| **boolean** hasNext(Pattern p) | Retourne *true* s'il existe un autre token dans l'entrée, satifaisant le *Pattern* *p*. |
| **boolean** hasNext(String p) | Retourne *true* s'il existe un autre token dans l'entrée, satifaisant le pattern décrit par la*String* *p*. |
| **boolean** hasNextLine() | Retourne *true* s'il existe une autre ligne dans l'entrée. |
| String next() | Retourne le token suivant. |
| String next(Pattern p) | Retourne le token suivant, s'il correspond au *Pattern p*, et lève une exception *NoSuchElementException* sinon. |
| String next(String p) | Retourne le token suivant, s'il correspond au *Pattern p*, et lève une exception *NoSuchElementException*sinon. |
| String nextLine() | Retourne le reste de la ligne, et lève une exception *NoSuchElementException* sinon. |

|  |  |
| --- | --- |
| Pattern delimiter() | Retourne le *Pattern* delimiteur utilisé par le *Scanner*. |
| Scanner useDelimiter(Pattern p) | Retourne le *Scanner* après avoir changeé son *Pattern* délimiteur. |
| Scanner useDelimiter(String p p) | Retourne le *Scanner* après avoir changeé son *Pattern* délimiteur. |
| Locale locale() | Retourne la localisation du *Scanner.* |
| Scanner useLocale(Locale l) | Retourne le *Scanner après avoir changé sa localisation.* |
| **int** radix() | Retourne la base dans laquelle sont écrits les nombre pour le décodage du *Scanner*. |
| Scanner useRadix(**int** radix) | Retourne le *Scanner* après avoir cahngé la base des nombres. |

5 Exceptions

* Toutes les méthodes lèvent une exception *IllegalStateException* si elles sont appelées pour un *Scanner* qui est fermé.
* Toutes les méthodes next lèvent une exception *NoSuchElementException* si elles sont appelées alors que le *hasNext* précédent a répondu *false*.

***RandomAccessFile***

|  |
| --- |
| **Sommaire**   1. [Constructeurs](http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/Flux/RandomAccessFile.html#constructeurs) 2. [Méthodes](http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/Flux/RandomAccessFile.html#méthodes) 3. Exemples    1. [Exemple 1](http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/Flux/RandomAccessFile.html#exemple1)    2. [Exemple 2](http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/Flux/RandomAccessFile.html#exemple2) |

Permet de faire des lectures ou écritures  à une position aléatoire dans un fichier. Cette classe peut être utilisée pour des fichiers disque, mais pas pour des flux de données provenant du réseau. Le flux peut être lu ou écrit de la même façon que les flux séquentiels, avec levée d'exception *EOFException* en fin de flux. Le flux peut également être lu ou écrit en accès direct en positionnant la "tête de lecture/écriture"  par la méthode *seek*. La classe *RandomAccessFile* implémente les interfaces *DataInput*, *DataOutput* et *Closeable*.

1 Les constructeurs :

|  |  |
| --- | --- |
| RandomAccessFile(File f, String m) | Crée un nouveau flux à accès direct associé au *File f* et ouvert dans le mode *m*. |
| RandomAccessFile(String n, String m) | Crée un nouveau flux à accès direct associé au fichier de nom*n* et ouvert dans le mode *m*. |

Le mode d'ouverture d'un flux à accès direct peut être :

* "r" Ouverture en lecture seule. L'appel d'une méthode d'écriture lève une exception *IOException*.
* "rw" Ouverture en lecture/écriture. Si le fichier n'existe pas, on essaie de le créer.
* "rws" Ouverture en lecture/écriture. La mise à jour du contenu du fichier et de ses méta-données est faite de façon synchrone.
* "rwd"   Ouverture en lecture/écriture. La mise à jour du contenu du fichier est faite de façon synchrone.

2 Les méthodes :

|  |  |
| --- | --- |
| **long** getFilePointer() | Retourne la valeur du pointeur. |
| **void**seek(**long** pos) | Positionne le pointeur. |
| **long**length() | Retourne la longueur du fichier. |
| **void**setLength(**long** nl) | Change la longueur du fichier en *nl*. |

|  |  |
| --- | --- |
| **void**close() | Fermeture du fichier |
| FileDescriptor getFD() | Retourne un descripteur du fichier |
| **int**read() | Lit un octet dans le fichier |
| **int** read(byte[] b) | Lit jusqu’à taille de *b* octets dans le fichier et les range dans *b* |
| **int** read(byte[] b, **int** d, **int** l) | Lit jusqu’à *l* octets dans le fichier et les range dans *b*à partir de*d.* |
| **boolean** readBoolean() | Lit un booléen. |
| **byte** readByte() | Lit un octet. |
| **char** readChar() | Lit un caractère. |
| **double** readDouble() | Lit un flottant *double*. |
| **float** readFloat() | Lit un flottant *float*. |
| **int** readInt() | Lit un entier *int*. |
| String readLine() | Lit une suite d'octets jusq'à une marque de fin de ligne, ou la fin fu flux, et retourne ces octets sous forme de chaîne de caracères. |
| **long** readLong() | Lit un entier *long*. |
| **short** readShort() | Lit un entier *short*. |
| **int** readUnsignedByte() | Lit un octet non signé, et le retourne. |
| **int** readUnsignedShort() | Lit deux octets comme étant la représentation d'un *short* non signé, et le retourne. |
| String readUTF() | Lit une chaîne de caractères encodée UTF-8. |
| **int** skipBytes(**int** n) | Essaie d'avancer de *n* octets, et retourne le nombre réel d'octets sautés. |
| **void** write(**byte**[] b) | Ecrit le tableau d'octets*b.* |
| **void** write(**byte**[] b, **int** d, **int** l) | Ecrit *l* octets du  tableau d'octets *b*, à patir de *d*. |
| **void** write(**int** b) | Ecrit l'octet de droite contenu dans *b*. |
| **void** writeBoolean(**boolean** v) | Ecrit le booléen *v*. |
| **void** writeByte(**int** v) | Ecrit l'octet *v*. |
| **void** writeChar(**int** v) | Ecrit le caractère contenu dans *v*. |
| **void** writeDouble(**double** v) | Ecrit le flottant *v*. |
| **void** writeFloat(**float** v). | Ecrit le flottant *v*. |
| **void** writeInt(**int** v) | Ecrit l'entier *v*. |
| **void** writeLong(**long** v) | Ecrit l'entier *v*. |
| **void** writeShort(**int** v) | Ecrit l'entier *v*. |

|  |  |
| --- | --- |
| **void** writeBytes(String s) | Ecrit la chaîne de caractères sous la forme d'une suite d'octets. Seul l'octet de droite de chauqe caractères est écrit. |
| **void**writeChars(String s) | Ecrit la chaîne de caractères. |
| **void** writeUTF(String str) | Ecrit la chaîne de caractères en utilisant l'encodage UTF-8. |

3 Exemple :

*3.1 Exemple :*

Ecriture dans un fichier des lettres de ‘a’ à ‘z’, Puis remplacement d'une des lettres de façon aléatoire par "?".

**try**{

*// écriture*

RandomAccessFile out = **new** RandomAccessFile ("accesDirect","rw");

**for**(**char** i = 'z'; i>='a'; --i){

out.seek(i-'a');

out.write(i);

}

out.seek((**long**)(Math.random()\*out.length()));

out.write('?');

out.close();

}**catch**(IOException ex){ ... }

*3.2 Exemple :*

Génération d'un fichiers d'entiers aléatoires et tri de ce fichier.

**try** {

RandomAccessFile raf = **new** RandomAccessFile("x", "rw");

*// génération*

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

raf.writeInt((int)(Math.random()\*1000));

raf.close();

*// tri*

raf = **new** RandomAccessFile("x", "rw");

**for** (**int** i = 0; i < 99; ++i){

**int** iMin = i;

raf.seek(iMin\*4);

**int** min = raf.readInt();

**for**(**int** j = i+1; j<100; ++j){

raf.seek(j\*4); int courant = raf.readInt();

**if**(courant < min) {

iMin = j;

raf.seek(iMin\*4);  min = raf.readInt();

}

}

raf.seek(i\*4);

**int** a = raf.readInt();

raf.seek(iMin\*4); raf.writeInt(a);

raf.seek(i\*4);    raf.writeInt(min);

}

raf.close();

*// affichage*

raf = **new** RandomAccessFile("x", "r");

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

System.out.print("  "+raf.readInt());

System.out.println();

raf.close();

} **catch** (FileNotFoundException e) { ... }

**catch** (IOException e) { ... }

***File***

|  |
| --- |
| **Sommaire**   1. [Attributs et méthodes de classe](http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/Flux/File.html#attrConsC) 2. [Constructeurs](http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/Flux/File.html#constructeurs) 3. [Méthodes](http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/Flux/File.html#méthodes) 4. Exemples    1. [*dir*](http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/Flux/File.html#exemple1)    2. [gestionnaire de fichiers](http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/Flux/File.html#exemple2) |

La classe *File* encapsule les fonctionnalités de gestion de fichier : Suppression, renommage, date de la dernière modification, etc …

1 Attributs et méthodes de classe :

|  |  |
| --- | --- |
| **static** String pathSeparator | Le séparateur de chemins sous forme de chaîne, sous Windows le ";"  et ":" sous Unix; |
| **static** **char** pathSeparatorChar | Le séparateur de chemins sous forme de *char* |
| **static** String separator | Le séparateur sous forme de chaîne, sous Windows le "\"  et "/" sous Unix. |
| **static** **char** separatorChar | Le séparateur sous forme de *char* |
| **static** File createTempFile(String prefix,  String suffix) | Crée un fichier vide temporaire dans le répertoire temporaire. Equivalent à *createTempFile(prefix, suffix, null)* |
| **static** File createTempFile(String prefix,  String suffix, File directory) | Crée un fichier vide temporaire   * Le préfixe doit avoir au moins 3 caractères. * Si le suffixe est *null* le suffixe utilisé est "tmp". * Le fichier est créé dans le répertoire *directory*. Si celui-ci vat *null* il est créé dans le réperoire temporaire du système. |
| **static** File[] listRoots() | Liste les lecteurs. Un pour chaque lecteur sous Windows (X:) et "/" sous Unix. |

2 Constructeurs

|  |  |
| --- | --- |
| File(File parent, String child) |  |
| File(String chemin) | Crée un nouveau *File* associé au nom *chemin.* |
| File(String parent, String child) |  |

3 Méthodes

|  |  |
| --- | --- |
| **boolean** canRead() | Renvoie vrai si le fichier peut être lu, et faux sinon. |
| **boolean** canWrite() | Renvoie vrai si le fichier peut être écrit, et faux sinon. |
| **int** compareTo(File pathname) | Comparaison des 2 chemins |
| **int** compareTo(Object o) |  |
| **boolean** createNewFile() | Création d’un nouveau fichier vide, et renvoie vrai si le nouveau fichier n’existait pas avant sa création. |
| **boolean** delete() | Supprime le fichier |
| **void** deleteOnExit() | Supprime le fichier à la fin de l’exécution de la machine virtuelle. |
| **boolean** equals(Object obj) |  |
| **boolean** exists() | Renvoie vrai si et seulement si le fichier existe. |
| File getAbsoluteFile() |  |
| String getAbsolutePath() | Retourne le chemin absolu |
| File getCanonicalFile() |  |
| String getCanonicalPath() | Retourne le chemin absolu |
| String getName() | Retourne le nom du fichier. |
| String getParent() |  |
| File getParentFile() |  |
| String getPath() | Retourne le chemin du fichier. |
| **int** hashCode() |  |
| **boolean** isAbsolute() |  |
| **boolean** isDirectory() | Retourne vrai s’il s’agit d’un répertoire et faux sinon. |
| **boolean** isFile() | Retourne vrai s’il s’agit d’un fichier "normal" et faux sinon. |
| **boolean** isHidden() | Retourne *true* si le fichier est caché.   * Sous Windows le fichier est marqué "caché". * Sous Unix le nom du fichier commence par un point. |
| **long** lastModified() | Retourne la date de la dernière modification sous forma de *long*. *Date d = new Date(f .lastModified()) ;* |
| **long** length() | La longueur du fichier, en octets. |
| String[] list() | Retourne la liste des fichiers contenus dans le répertoire spécifié par le File. Si le File n'est pas un répertoire le tableau retourné vaut **null**. |
| String[] list(FilenameFilter filter) | Retourne la liste des fichiers contenus dans le répertoire spécifié, en utilisant un filtre. exemple :  **class** FiltreImage **implements** FilenameFilter {  **public** **boolean** accept(File dir, String name) {  **return** (name.endsWith(".gif")||name.endsWith(".GIF") ||  name.endsWith(".jpeg")||name.endsWith(".JPEG") ||  name.endsWith(".jpg")||name.endsWith(".JPG")  );  }  }  FiltreImage fImage = **new** FiltreImage();  File [] tf = **new** File(repertoire).listFiles(fImage);  Si le File n'est pas un répertoire le tableau retourné vaut **null**. |
| File[] listFiles() | Retourne la liste des fichiers contenus dans le répertoire spécifié par le File. Si le File n'est pas un répertoire le tableau retourné vaut **null**. |
| File[] listFiles(FileNameFilter filter) |  |
| **boolean** mkdir() | Crée un répertoire. |
| **boolean** mkdirs() | Crée un répertoire, et tous ses parents nécessaires. L'appel de *mkdir* avec un *File* constitué de "\\" retourne *false*. |
| **boolean** renameTo(File dest) | Renomme un fichier. retourne *false* si le file *dest* existe déjà, et le renommage n'est pas effectué. |
| **boolean** setLastModified(long time) |  |
| **boolean** setReadOnly() | Marque le fichier en lecture seule |
| String toString() |  |
| URL toURL() | *C:\toto* est transformé en *file:/C:/toto* |

4 Exemples.

*4.1 Exemple : liste des fichiers du répertoire courant.*

File rep = **new** File(System.getProperty("user.dir"));

File [] lesFichiers = rep.listFiles();

**for**(**int** i = 0; i < lesFichiers.length; ++i){

File f = lesFichiers[i];

**if**(!f.isDirectory())

System.out.println(lesFichiers[i].getAbsolutePath());

**else** System.out.println("REP : "+ lesFichiers[i].getAbsolutePath());

}

Liste récursive des fichiers du répertoire courant :

System.out.println("liste récursive des fichiers du répertoire courant");

File repR = **new** File(System.getProperty("user.dir"));

listeRecursive(repR);

Avec :

**void** listeRecursive(File rep){

File [] lesFichiers = rep.listFiles();

**for**(**int** i = 0; i < lesFichiers.length; ++i){

File f = lesFichiers[i];

**if**(!f.isDirectory())

System.out.println(lesFichiers[i].getAbsolutePath());

**else** listeRecursive( lesFichiers[i] );

}

}

*4.2 Exemple :*

un [gestionnaire de fichier](http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/Applis/listeFichiers.jnlp) « léger »

**public** **void** listeLecteurs() {

*// Liste des disques dans une JList dont le modèle est dlm*

dlm.clear();

File[] fs = File.listRoots();

**for** (**int** i = 0; i < fs.length; ++i)

dlm.addElement(fs[i].getAbsolutePath());

}

**public** **void** listeRepertoire(String nomFichier) {

*// Liste les fichier de nomFichier si c'est un répertoire*

*// ou un lecteur*

*// nomFichier peut être .. on liste alors le répertoire parent*

File rep;

**if** (nomFichier.equals("..")) { // répertoire parent

String repCourant = System.getProperty("user.dir");

**if** (repCourant.endsWith(":"+File.separatorChar)) {

*// le parent d'un lecteur c'est la liste des lecteurs*

listeLecteurs();

getJFrame().setTitle("");

**return**;

}**else** {

repCourant = repCourant.substring(0, repCourant.lastIndexOf(File.separatorChar));

**if** (repCourant.length() == 2) repCourant = repCourant + File.separatorChar;

rep = **new** File(repCourant);

System.setProperty("user.dir", repCourant);

getJFrame().setTitle(System.getProperty("user.dir"));

}

}**else**{

**if** (nomFichier.endsWith(":"+File.separatorChar)){*// nomFichier est un lecteur*

System.setProperty("user.dir", nomFichier);

rep = **new** File(System.getProperty("user.dir"));

}**else**

rep = **new** File(System.getProperty("user.dir") + File.separatorChar

+ nomFichier);

}

**if** (rep.isDirectory()) {

*// on change de répertoire courant*

System.setProperty("user.dir", rep.getAbsolutePath());

}

rep = **new** File(System.getProperty("user.dir"));

dlm.clear();

dlm.addElement("..");

File[] fss = rep.listFiles();

**if**(fss==**null**){ *// pas de lecteur ...*

listeLecteurs();

**return**;

}

**for** (**int** i = 0; i < fss.length; ++i)

dlm.addElement(fss[i].getName());

getJFrame().setTitle(System.getProperty("user.dir"));

}

***généricité***

|  |
| --- |
| **Sommaire**   1. [Pourquoi la généricité ?](http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/Conteneurs/genericite.html#pourquoi) 2. [Méthode générique](http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/Conteneurs/genericite.html#methode) 3. [Limite pour les type paramètre](http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/Conteneurs/genericite.html#limite) 4. [Effacement](http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/Conteneurs/genericite.html#effacement) 5. [Généricité et héritage](http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/Conteneurs/genericite.html#genHerit) 6. [Joker](http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/Conteneurs/genericite.html#joker) 7. [Capture du joker](http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/Conteneurs/genericite.html#capture) 8. [Généricité et reflexion](http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/Conteneurs/genericite.html#genReflex) 9. [Méthodes *pont*](http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/Conteneurs/genericite.html#pont) |

La généricité permet de définir des classes, et des méthodes paramétrées par une ou plusieurs classes.

1 Pourquoi la généricité ?

Exemple : Soit la classe suivante :

**public** **class** Paire {

Object premier ;

Object second;

**public** Paire (Object a, Object b){

premier= a; second = b;

}

**public** object getPremier(){

**return** premier;

}

**public** object getSecond(){

**return** second;

}

Les deux inconvénients sont :

Paire  p = **new** Paire ("abc", "xyz");

String x =  (String)p.getPremier(); *// le casting est obligatoire*

Double y =  (Double)p.getSecond(); *// Il faut attendre l'exécution pour avoir*

*// une levée d'exception(****ClassCastException****)*

On définit alors une classe paramétrée :

**public** **class** Paire<T> {

T premier ;

T second;

**public** Paire (T a, T b){

premier a; second = b;

}

**public** T getPremier(){

**return** premier;

}

**public** getSecond(){

**return** second;

}

}

Le programme est alors plus simple et plus sur :

Paire<String>  p = **new** Paire<String>  ("abc", "xyz");

String x =  p.getPremier(); *// pas de cast*

Double y =  p.getSecond(); *// erreur de compilation (****type mismatch****)*

2 Méthode générique

On peut définir une méthode générique dans une classe de la façon suivante :

**public** **class** X {

**public** <T> **void** affiche(Paire<T> p){

    System.out.println(p);

}

**public** <T> T choix(T a, T b){

**return** (**int**)(Math.random()\*2)==1?a:b;

}

**public** **static** **void** main(String []a){

     Paire<String> ps = **new** Paire<String>("un", "deux");

     Paire<Integer> pi = **new** Paire<Integer>(1, 2);

     X x = **new** X();

     x.affiche(ps);

x.affiche(pi);

Number n = x.choix(**new** Integer(2), **new** Double(3.14159));

}

}

Dans la dernière ligne, le compilateur fait une **inférence de type**, et calcule la première super classe commune aux deux classes *Integer* et *Double*, soit *Number*.

3 Limite pour les types paramètre

Supposons que nous voulions ajouter à la classe *Paire<T>* la méthode suivante :

**public** T min(){

**if**(premier.compareTo(second)<=0) **return** premier;

**else** **return** second;

}

Le compilateur signale alors que la méthode *compareTo* n'est pas définie pour le type *T*. Il faut restreindre *T* à une classe qui a cette méthode, et définir la classe *Paire* de la façon suivante :

**public** **class** Paire<T **extends** Comparable> {

T premier ;

T second;

**public** Paire (T a, T b){

premier a; second = b;

}

**public** T getPremier(){

**return** premier;

}

**public** getSecond(){

**return** second;

}

**public** T min(){

**if**(premier.compareTo(second)<=0) **return** premier;

**else** **return** second;

}

}

Le type limitant peut être une classe ou une interface. On définira :

**public** **class** Paire<T **extends** Number> { ... }

pour définir une paire de nombres... et

**public** **class** Paire<T **extends** A & Comparable> {  ... }

pour définir une paire d'élément de la classe *A* ou de classes dérivées de *A* et qui sont *Comparable*.

4 Effacement

Les classes paramétrées sont compilées vers une classe représentant le type *brut* : le type équivalent débarrassé des paramètres de la classe. On dit que les paramètres de type sont *effacés*. Ceci a des conséquences :

* On ne peut pas avoir une classe *Paire* et une classe *Paire<T>*
* Paire<String, Integer> p1 = **new** Paire<String>("abc");
* **if**(p1 **instanceof** Paire) ; *// OK*
* **if**(p1 **instanceof** Paire<String>) ;*// NON OK*

La dernière ligne provoque l'erreur de compilation suivante : Impossible d'effectuer une vérification instanceof sur le type paramétré Paire. A la place, utilisez sa forme brute Paire car les informations du type générique seront effacées lors de l'exécution.

* Lors de l'effacement la classe générique *Paire<T>* est générée en la classe brute *Paire* définie comme suit :
* **public** **class** Paire {
* Object premier ;
* Object second;
* **public** Paire (Object a, Object b){
* premier= a; second = b;
* }
* **public** object getPremier(){
* **return** premier;
* }
* **public** object getSecond(){
* **return** second;

}

* Lors de l'effacement la classe générique *Paire<T****extends****A>* est générée en la classe brute *Paire* définie comme suit :
* **public** **class** Paire {
* A premier ;
* A second;
* **public** Paire (A,A b){
* premier= a; second = b;
* }
* **public** A getPremier(){
* **return** premier;
* }
* **public** A getSecond(){
* **return** second;

}

* Paire<String> p1 = **new** Paire<String>("abc", "1");
* Paire p2 = **new** Paire("abc", "def");
* p2 = p1;  *// OK*
* p1 = p2;  *//* ***avertissement****, sécurité de type :*
* *// l'expression du type Paire requiert une conversion non controlée en Paire<String>*

5 Généricité et héritage

Soient les deux classes suivantes :

|  |  |
| --- | --- |
| **public** **class** Super {     ...  } | **public** **class** Sous **extends** Super {     ...  } |

Il n'existe pas de relation d'héritage entre *Paire<Super>* et *Paire<Sous>*.

Paire<Super> pSup = **new** Paire<Super>(**new** Super(), **new** Super());

Paire<Sous>  pSous = **new** Paire<Sous>(**new** Sous(), **new** Sous());

pSup = pSous ; *// INTERDIT*

Supposons que celà soit possible,nous pourrions alors écrire :

  pSup.setPremier(**new** Super());

Or *pSup* et *pSous* désigne le même objet et l'objet désigné par *pSous* aurait son attribut premier de type *Super* !!!

6 Joker

Supposons que nous définissions dans la classe *X* une autre méthode pour afficher des paires de *Super*, de la façon suivante :

|  |
| --- |
| **public** **void** afficheS(Paire<Super>p){   System.out.println(p);  } |
| Paire <Super> ps = **new** Paire<Super>();  Paire <Sous> pso = **new** Paire< Sous>();     |  |  | | --- | --- | | x.afficheS(ps); | **OK**, pas de problème. | | x.afficheS(pso); | **NON**, *pso* n’est pas une *Paire* de *Super* | |

Pour que la dernière instruction soit possible il faudrait définir la méthode *affiche*S de la façon suivante :

**public** **void** afficheS(Paire<?> p){

    System.out.println(p);

}

Mais dans ce cas tout est possible, et si on veut se limiter aux classes qui dérivent de *Super* :

**public** **void** afficheS(Paire<? **extends** Super>p){

    System.out.println(p);

}

Les jokers peuvent être limités vers le haut (*extends*) ou vers le bas (*super*).

On a les relations d'héritage suivantes :

|  |  |
| --- | --- |
| généricité1.gif | généricité 2 .gif |

La relation d'héritage entre *Paire*et *Paire<?>* est là pour assurer la compatibilité avec les codes existants avant la version 5 de Java.

7 Capture du Joker

Soit la classe :

**public** **class** Paire<T> {

T premier ;

T second;

**public** Paire (T a, T b){

premier a; second = b;

}

**public** T getPremier(){

**return** premier;

}

**public** getSecond(){

**return** second;

}

**public void** setPremier(T t){

premier = t;

}

}

On a alors :

Paire<?> p1 = **new** Paire<Integer>( 1, 2);

Paire<? **extends** Number> p2 = **new** Paire<Integer>( 1, 2);

Object o = p1.getPremier(); *// OK*

Number n = p1.getPremier(); *// ERREUR non concordance de types :*

*// impossible de convertir de capture-of ? en Number*

Integer i = p1.getPremier(); *// ERREUR non concordance de types :*

*// impossible de convertir de capture-of ? en Integer*

o = p2.getPremier(); *// OK*

n = p2.getPremier(); *// OK*

i = p2.getPremier(); *// ERREUR non concordance de types :*

*// impossible de convertir de capture-of ?* ***extends*** *Number en Integer*

Le compilateur désigne par **capture-of ?** le type inconnu qui paramètre la paire *p1*, le compilateur désigne par **capture-of ? extends Number** le type inconnu qui paramètre la paire *p2*.

8 Généricité et réflexion

On peut savoir si une classe est une classe générique (informations conservées à la compilation), mais on ne peut évidemment pas connaître l'instanciation précise d'un type.  
exemple :

**public** String toString(Class c){

StringBuffer res = **new** StringBuffer("");

TypeVariable[] tv = c.getTypeParameters();

res.append(c.getName());

**if**(tv.length!=0){

res.append("<");

**for** (**int** i = 0; i < tv.length-1; ++i)

res.append(tv[i]+", ");

res.append(tv[tv.length-1]);

res.append(">");

}

**return** res.toString();

}

L'usage de cette méthode pour une classe *c* permet d'obtenir le nom de la classe suivi de ses paramètres si la classe est générique.

9 Méthodes **Pont**

Soit les définitions suivantes :

**public** **interface** Comparable <T>{

**public** **int** compareTo(T t);

}

**public** **class** X **implements** Comparable<X> {

**public int** compareTo(X t) {

**return** 0;

}

}

L'effacement dans l'interface Comparable génère une méthode ***int****compareTo(Object t)*, et la classe *X* définit la méthode ***int****compareTo(X t)*. La classe *X* ne redéfinit donc pas ***int****compareTo(Object t)* : le compilateur ajoute la méthode pont :

**public** **class** X **implements** Comparable<X> {

**public int** compareTo(Object t) {

**return** compareTo((X)t);

}

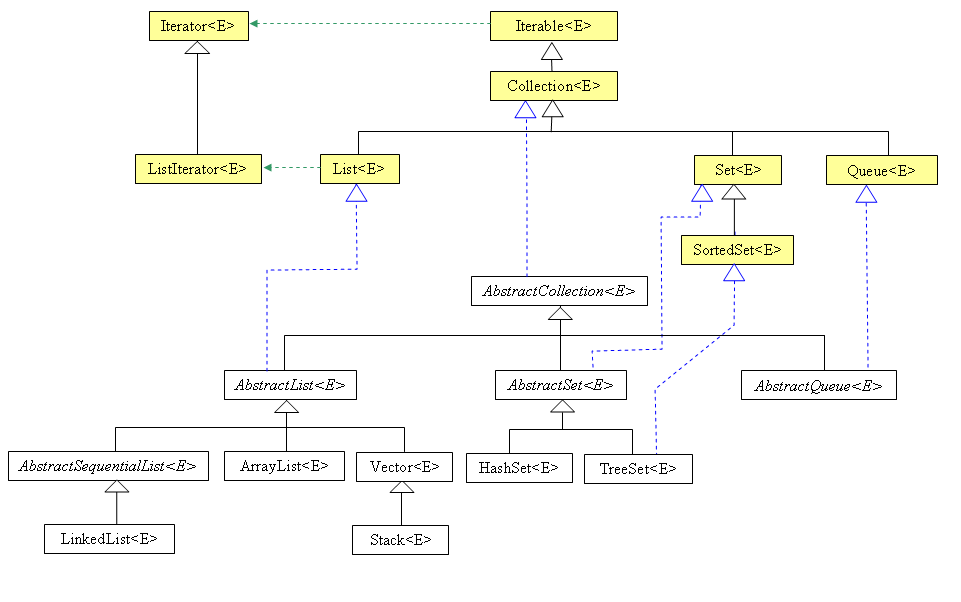
**public int** compareTo(X t) {

**return** 0;

}

}

# *Hiérarchie des Collection*



***Méthode optionnelle***

Il peut exister de nombreux cas particuliers de collections, par exemple :

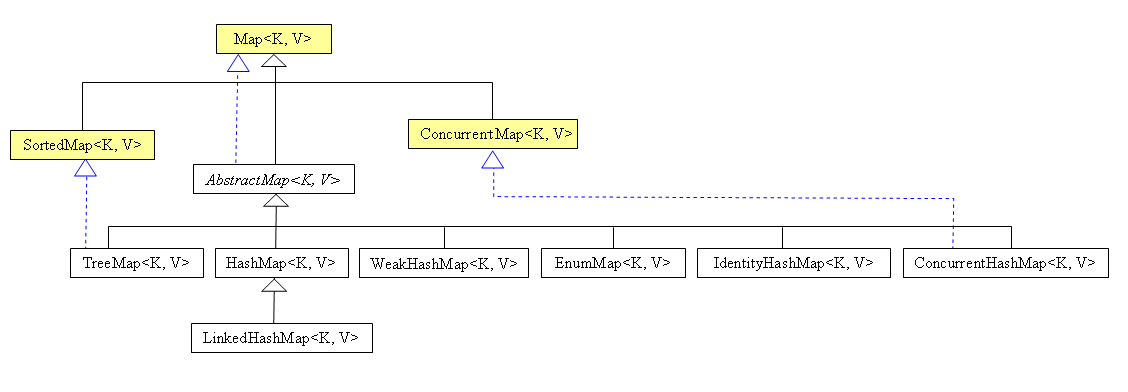
* collections composées d’objets non modifiables
* collections de taille fixe,
* collections dont on ne peut enlever des objets

Plutôt que de fournir une interface pour chaque cas particulier, l'API sur les collections comporte la notion de méthode optionnelle.

Une méthode optionnelle est une méthode qui lève une exception*UnsupportedOperationException* (sous-classe de *RuntimeException*) quand elle est utilisée pour une instance de classe pour laquelle la méthode n'est pas définie.

* Les méthodes optionnelles renvoient cette exception dans les classes abstraites du paquetage.
* Exemple d’utilisation : si on veut écrire une classe pour des listes non modifiables, il suffit de ne pas redéfinir dans cette classe,  les méthodes *set*, *add* et *remove* de la classe abstraite *AbstractList.*

# *Hiérarchie des Map*



# *Iterable*

Cette interface contient une seule méthode :

|  |  |
| --- | --- |
| Iterator<E> iterator() | Retourne un [itérateur](http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/Conteneurs/Iterator.html) sur le conteneur |

Toute collection qui implémente cette interface ( Y compris les tableaux Java) permet l'écriture de*for* simplifié :

**for**( E e : Iterable<E>) ... ;

LesTableaux Java sont également Iterables et par conséquent nous pouvons écrire un parcours de tableau de la façon suivante :

Object[] t;

...

**for**( Object e : t) ... ;

# *Collection*

L'interface Collection dérive de l'interface [*Iterable*](http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/Conteneurs/Iterable.html).

Une collection est un groupe d’objets appelés éléments. Certaines collections admettent des éléments ayant plusieurs occurrences ( [*ArrayList*,](http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/Conteneurs/ArrayList.html) [*LinkedList*,](http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/Conteneurs/LinkedList.html) *Vector*, *Stack*), d’autres non ([*HashSet*](http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/Conteneurs/HashSet.html)*,* [*TreeSet*](http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/Conteneurs/TreeSet.html)).

|  |  |
| --- | --- |
| **boolean** add(E o) | Ajoute *o* à la collection (opération optionnelle) retourne *true* si l’ajout a été fait, et *false* si l’ajout n’est pas fait ( parce que la collection n’autorise pas les doublons : cas des [*Set*](http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/Conteneurs/Set.html) ). |
| **boolean** addAll(Collection<? **extends** E> c) | Ajoute tous les éléments de la collection *c* à la collection (opération optionnelle) |
| **void** clear() | Supprime tous les éléments de la collection (opération optionnelle) |
| **boolean** contains(Object o) | Retourne *true* si la collection contient l’élément *o* |
| **boolean** containsAll( Collection<?> c) | Retourne *true* si la collection contient tous les éléments de la collection *c* |
| **boolean** equals(Object o) | Compare *o* avec la collection |
| **int** hashCode() | Retourne le *hash code* pour cette collection |
| **boolean** isEmpty() | Retourne *true* si la collection est vide |
| **boolean** remove(Object o) | Enlève une seule occurrence de *o* de la collection (opération optionnelle). Retourne *true* si l’élément a été enlevé. |
| **boolean** removeAll( Collection<?> c) | Enlève toutes les occurrences de la collection qui appartiennent à *c* (opération optionnelle). Retourne *true* si un élément a été enlevé. |
| **boolean** retainAll( Collection<?> c) | Enlève de la collection tous les éléments qui n’appartiennent pas à *c*(opération optionnelle). Retourne *true* si l’élément a été enlevé. Retourne *true* si la collection a changé. |
| **int** size() | Retourne le nombre d’éléments de la collection |
| Object[] toArray() | Retourne un tableau contenant tous les éléments de la collection |
| <T> T[] toArray(T[] t) | Retourne un tableau contenant tous les éléments de la collection qui sont d’un type dérivé du type des éléments du tableau *t* Si tous ces éléments tiennent dans le tableau *t*, alors ils sont rangés dedans et c’est le tableau *t*qui est retourné ( *null* est rangé dans le premier élément « vide »), sinon un nouveau tableau est créé Une exception *ArrayStoreException* est levée si un des éléments de la collection n’est pas d’un type dérivé du type des éléments du tableau *t* |

# *List*

L’interface *List* ajoute à l'interface *Collection* de nouvelles fonctionnalités d’accès à partir d’un indice : Les éléments ont un indice dans la collection.

|  |  |
| --- | --- |
| **void** add(**int** ind, E o) | Insère *o* à l’indice *ind* (opération optionnelle) |
| **boolean** addAll(**int** ind,  Collection<? **extends** E> c) | Insère tous les éléments de la collection *c* à partir de l’indice *ind*.(opération optionnelle) |
| get(**int** ind) | Retourne l’élément à la position *ind*. Peut lever une exception *IndexOutOfBoundsException* si *ind* n'est pas correct. |
| **int** indexOf(Object o) | Retourne l’indice de la première occurrence de *o*, ou -1 si la *list* ne contient pas *o*. |
| **int** lastIndexOf(Object o) | Retourne l’indice de la dernière occurrence de *o*, ou -1 si la *list* ne contient pas *o*. |
| ListIterator<E> listIterator() | Retourne un *ListIterator* sur la liste |
| ListIterator<E> listIterator(**int** ind) | Retourne un *ListIterator* sur la liste, démarrant à l’indice *ind*. Peut lever une exception *IndexOutOfBoundsException* si *ind* n'est pas correct. |
| E set(**int** ind>, E o) | Remplace l’élément à l’indice *ind* par *o* et retourne l'ancienne valeur. Peut lever une exception *IndexOutOfBoundsException* si *ind* n'est pas correct. (opération optionnelle) |
| E remove(**int ind)** | Enlève l'élément à l'indice ind, et retourne cet élément. Peut lever une exception *IndexOutOfBoundsException* si *ind* n'est pas correct. |
| List<E> subList(**int** d, **int** f) | Retourne la sous liste comprise entre les indices *d* (inclus) et *f* ( exclu) .Peut lever une exception *IndexOutOfBoundsException* si *ind* n'est pas correct. |

***Set***

Un *Set* est une collection qui ne contient pas d'éléments dupliqués. Plus formellement :

* il n'y a pas dans un *Set* deux éléments *e1* et *e2* tels que *e1.equals(e2)*.
* il peut n'y avoir qu'un seul élément qui vaut *null*.

Remarque : le comportement d'un *Set*, n'est pas spécifié pour des objets mutables. Si on stocke un objet *o* dans le *Set*, et que en changeant de valeur cet objet devient égal à un autre objet du *Set*, l'état du *Set* n'est plus correct !  
La liste des méthodes est identique à la liste des méthodes de l'interface [*Collection*](http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/Conteneurs/Collection.html).

***SortedSet***

Un *SortedSet* est un *Set* qui garantit que le parcours des éléments, du premier au dernier, se fera dans l'«ordre» des éléments. Les éléments ajoutés dans un *SortedSet* doivent implémenter l'interface *Comparable*.  
L'ordre sur les éléments peut être déterminé de deux façons :

1. Le SortedSet est construit sans [*Comparator*](http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/Conteneurs/SortedSet.html#Comparator), les élements ajoutés implémentent l'interface *Comparable*, alors la méthode *compareTo* des éléments est utilisée pour comparer deux éléments du *SortedSet.*
2. Un [*Comparator*](http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/Conteneurs/SortedSet.html#Comparator) est fournit au *SortedSet* au moment de sa création : alors les méthodes *compare* ou *equals* du *Comparator* sont utilisées pour comparer deux éléments.

De nouvelles méthodes sont définies dans un *SortedSet* :

|  |  |
| --- | --- |
| Comparator<? **super** E> comparator(); | Retourne le *Comparator* associé au *SortedSet* , ou *null* si le *SortedSet* n'est pas muni d'un *Comparator*. |
| SortedSet<E> subSet(E depuis, E jusqua); | Retourne le sous ensembles des valeurs comprises entre *depuis* (inclus) et *jusqua* (exclu). Le *SortedSet* retourné est de même type que le *SortedSet* pour lequel est appelé la méthode. Une exception *IllegalArgumentException* est levée si :   * *depuis* est plus grand que *jusqua* * *depuis* ou *jusqua* ne sont pas dans le *SortedSet*. |
| SortedSet<E> headSet(E jusqua); | Retourne le sous ensembles des valeurs comprises entre le premier élément (inclus) et *jusqua* (exclu).Le *SortedSet* retourné est de même type que le *SortedSet* pour lequel est appelé la méthode. |
| SortedSet<E> tailSet(E depuis); | Retourne le sous ensembles des valeurs à partir de l'élément *depuis* (inclus) jusqu'à la fin.Le *SortedSet* retourné est de même type que le *SortedSet* pour lequel est appelé la méthode. |
| E first(); | Retourne le premier élément. Une exception *NoSuchElementException* est levée si le *SortedSet* est vide. |
| E last(); | Retourne le dernier élément. Une exception *NoSuchElementException* est levée si le *SortedSet* est vide. |

Un *Comparator* est un objet qui implémente les deux méthodes suivante :

|  |  |
| --- | --- |
| **int** compare(T o1, T o2); | Compare les deux arguments et retourne :   * une valeur négative si *o1* est plus petit que *o2*. * une valeur nulle si *o1* est égal à *o2*. * une valeur positive si *o1* est plus grand que *o2*. |
| **boolean** equals(Object obj) | Retourne true si l'objet *obj* est égal au *Comparator.* peut être utilisé pour savoir si deux *Comparator* imposent le même ordre. |

# *Iterator*

L'interface *Iterator* est utilisée par l'interface [*Iterable*.](http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/Conteneurs/Iterable.html)

De nombreuses classes conteneurs implémentent les méthodes de l’interface *Collection* : *Vector*, *Stack*, *ArrayList*, *LinkedList*, *HashSet*, *TreeSet*, Chacune de ces classes doit implémenter la méthode *Iterator iterator()* qui retourne un itérateur sur l’ensembles des éléments : un itérateur est un objet qui implémente l’interface *Iterator*, et qui a pour but de permettre le parcours des éléments d’un conteneur, sans avoir besoin de savoir de quelle nature est ce conteneur.

L’interface *Iterator* contient les méthodes suivantes :

|  |  |
| --- | --- |
| **boolean** hasNext() | Retourne *true* si l’itérateur n’est pas arrivé en fin de l’ensemble et *false* sinon. |
| E next() | Permet d’avancer l’itérateur, et qui retourne l’élément qui a été sauté. Lève une exception *NoSuchElementException* si l'itérateur n'a pas d'objet qui le suit. |
| **void** remove() | Supprime l’objet qui vient d’être obtenu par *next*. Cette méthode est facultative |

Quelque soit la collection *c*, on peut parcourir tous ses éléments de la façon suivante :

Iterable <E> c;

...

Iterator<E> it = c.iterator() ;

**while**(it.hasNext()){

E o = it.next();  *// o a pour valeur un objet de la collection*

. . .

}

|  |  |
| --- | --- |
| On peut représenter les itérateurs de la façon suivante : | http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/CoursJava/Conteneurs_fichiers/image004.gif |

# *ListIterator*

L'interface *ListIterator* dérive de l'interface *Iterator* ; elle est utilisée par l'interface *List*. Elle introduit notamment des méthodes qui permettent de parcourir une collection en sens inverse.

|  |  |
| --- | --- |
| **boolean** hasPrevious () | Retourne *true* s'il existe un élément avant. |
| E previous() | Retourn l'élément précédent s'il existe, et lève une exception *NoSuchElementException*, s'il n'existe pas. |
| **int**  nextIndex() | Retourne l'indice de l'élément qui serait retourné par un appel de *next()*. |
| **int** previousIndex() | Retourne l'indice de l'élément qui serait retourné par un appel de *previous()*. Retourne -1 si l'itérateur est au début de la liste. |
| **void** set(E o) | Remplace le dernier élément  retourné par *next()*, ou par *previous()*, par l'objet *o*. |
| **void** add(E o) | Remplace le dernier élément  retourné par *next()*, ou par *previous()*, par l'objet *o*. |

***AbstractCollection***

|  |
| --- |
| **Sommaire**   1. [Le constructeur par défaut](http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/Conteneurs/AbstractCollection.html#consDef) 2. [Méthodes abstraites et non supportées](http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/Conteneurs/AbstractCollection.html#methodeANS) 3. [Méthodes définies](http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/Conteneurs/AbstractCollection.html#methode) |

Cette classe abstraite fournit une implémentation minimale de l’interface *Collection* en utilisant essentiellement les itérateurs.

**public** **abstract** **class** AbstractCollection<E> **implements** Collection<E> {

...

}

1 Le constructeur par défaut :

**protected** AbstractCollection(){

}

2 Les méthodes qui sont abstraites, ou non supportées :

**public abstract** Iterator<E> iterator();

**public** **abstract** **int** size();

La méthode optionnelle *add* est définie dans la classe *AbstractCollection* et lèvent une exception *UnsupportedOperationException* :

**public** **boolean** add(E o) {

**throw** **new** UnsupportedOperationException();

}

3 Les méthodes définies :

La méthode *isEmpty()* peut être définie en utilisant la méthode *size()* :

**public** **boolean** isEmpty() {

**return** size() == 0;

}

La méthode *contains(Object o)* retourne *true* si l’objet *o* appartient à la collection et *false* sinon :

**public** **boolean** contains(Object o){

Iterator<E> e = iterator();

**if**(o==**null**) {

**while**(e.hasNext())

**if** (e.next()==**null**) **return** **true**;

}**else** {

**while** (e.hasNext())

**if** (o.equals(e.next())) **return** **true**;

}

**return** **false**;

}

Les deux méthodes suivantes convertissent un vecteur en un tableau d’*Object*:

|  |  |
| --- | --- |
| **public** Object[] toArray() {  Object[] result = **new**Object[size()];  Iterator<E> e = iterator();  **for** (**int** i=0; e.hasNext(); i++)  result[i] = e.next();  **return** result;  } | **public** Object[] toArray() {  Object[] result = **new** Object[size()];  **int** i = 0;  **for** (E x : **this**)  result[i++] = x;  **return** result;  } |

La méthode suivante, copie tous les éléments du vecteur dans le tableau *t* et retourne ce tableau si celui-ci est assez grand, sinon elle copie les éléments dans un tableau qui est créé et le retourne.

* Si le tableau *t* est plus grand que le nombre d’éléments du vecteur, alors l’élément qui suit le dernier copié est affecté de la valeur **null**.
* Le type du tableau retourné est le même que le type du tableau *t*. Une exception *ArrayStoreException* est levée si le vecteur contient un élément qui n’est pas de la classe (ou d’une classe dérivée ) des éléments du tableau *t*.

**public** <T> T[] toArray(T[] a ) {

**if** (a.length < size())

a = (T[])java.lang.reflect.Array.newInstance(

a.getClass().getComponentType(), size());

Iterator<E> it = iterator();

**for** (**int** i = 0; i < size(); i++)

a[i] = it.next();

**if** (a.length > size())

a[size()] = **null**;

**return** a;

}

Les méthodes  ayant en paramètre une *collection*  sont définies de la façon suivante:

|  |  |
| --- | --- |
| **public** **boolean**containsAll(Collection<?> c) {  Iterator<E> e = c.iterator();  **while** (e.hasNext())  **if**(!contains(e.next()))  **return** **false**;  **return** **true**;  } | **public** **boolean** containsAll(Collection<?> c) {  **for**(E x : c) **if**(!contains(x)) **return** **false**;  **return** **true**;  } |
| **public** **boolean** addAll(Collection<? **extends** E> c ){  **boolean** modifie = **false**;  Iterator<E> e = c.iterator();  **while** (e.hasNext()) {  **if**(add(e.next())) modifie = **true**;  }  **return** modifie;  } | **public** **boolean** addAll(Collection<? **extends** E> c){  **boolean** modifie = **false**  **for**(E e : c)  **if**(add(e)) modifie = **true**;  **return** modifie;  } |
| **public** **boolean** removeAll(Collection<?> c) {  **boolean** modifie = **false**;  Iterator<E> e = iterator();  **while** (e.hasNext()) {  **if**(c.contains(e.next())){  e.remove();  modifie = **true**;  }  }  **return** modifie;  } | Pas d'équivalent aussi performant avec *for*. |
| **public** **boolean** retainAll(Collection<?> c) {  **boolean** modifie = **false**;  Iterator<E> e = iterator();  **while** (e.hasNext()) {  **if**(!c.contains(e.next())) {  e.remove();  modifie = **true**;  }  }  **return** modifie;  } | Pas d'équivalent aussi performant avec *for*. |

Les méthodes suivantes sont optionnelles parce que la méthode *remove* de *Iterator* est optionnelle, elles peuvent être définies en utilisant *remove* :

**public** **void** clear() {

Iterator<E> e = iterator();

**while** (e.hasNext()) {

e.next();

e.remove();

}

}

**public** **boolean** remove(Object o) {

Iterator<E> e = iterator();

**if** (o==**null**) {

**while** (e.hasNext()) {

**if** (e.next()==**null**){

e.remove();

**return** **true**;

}

}

}**else**{

**while**(e.hasNext()) {

**if**(o.equals(e.next())) {

e.remove();

**return** **true**;

}

}

}

**return** **false**;

}

La méthode *toString* peut aussi être définie dans la classe *AbstractCollection* :

**public** String toString() {

StringBuffer buf = **new** StringBuffer();

Iterator<E> e = iterator();

buf.append("[");

**if**( nbElements!=0 ){

buf.append(String.valueOf(e.next()));

**for** (**int** i = 1; i < size(); i++)

buf.append(","+String.valueOf(e.next()));

}

buf.append("]");

**return** buf.toString();

}

***AbstractList***

|  |
| --- |
| **Sommaire**   1. [Les méthodes abstraites](http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/Conteneurs/AbstractList.html#methodeA) 2. [Les méthodes optionnelles](http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/Conteneurs/AbstractList.html#methodeO) 3. [Les autres méthodes](http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/Conteneurs/AbstractList.html#methode) 4. [Implémentation des itérateurs](http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/Conteneurs/AbstractList.html#iterateur) 5. [La méthode *subList*](http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/Conteneurs/AbstractList.html#subList) 6. [La méthode *equals*](http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/Conteneurs/AbstractList.html#equals) |

La classe abstraite *AbstractList* implémente certaines des nouvelles fonctionnalités de l’interface *List*. Ces nouvelles fonctionnalités sont des méthodes avec indice. Elles supposent que la structure sous-jacente servant à représenter la liste est une structure à accès direct par indice (tableau). Pour implanter un conteneur avec une liste chaînée, on utilisera plutôt la classe abstraite *AbstractSequentialList*

**public** **abstract** **class** AbstractList<E>

**extends** AbstractCollection<E>

**implements** List<E>

1 Les méthodes abstraites :

**abstract** **public** E get(**int** index);

2 Les méthodes optionnelles (à définir dans les sous-classes qui le souhaitent):

**public** **void** add(**int** index, E element) {

**throw** **new** UnsupportedOperationException();

}

**public** E set(**int** index, E element) {

**throw new** UnsupportedOperationException();

}

**public** E remove(**int** index) {

**throw** **new** UnsupportedOperationException();

}

3 Les autres méthodes :

**public** **boolean** add(E o) {

add(size(), o);

**return** **true**;

}

Les méthodes *indexOf* :

**public** **int** indexOf(Object o) {

ListIterator<E> e = listIterator();

**if** (o==null) {

**while** (e.hasNext())

**if** (e.next()==null)

**return** e.previousIndex();

} **else** {

**while** (e.hasNext())

**if** (o.equals(e.next()))

**return** e.previousIndex();

}

**return** -1;

}

**public** **int** lastIndexOf(Object o) {

ListIterator<E> e = listIterator(size());

**if** (o==null) {

**while** (e.hasPrevious())

**if** (e.previous()==null)

**return** e.nextIndex();

} **else** {

**while** (e.hasPrevious())

**if** (o.equals(e.previous()))

**return** e.nextIndex();

}

**return** -1;

}

Les opérations ensemblistes : 

**public** **void** clear() {

**for** (Iterator<E> it = Iterator(); it.hasNext(); ){

it.next();

it.remove();

}

}

**public** **boolean** addAll(**int** ind, Collection<? **extends** E> c) {

**boolean** modifie = **false**;

Iterator<? **extends** E> e = c.iterator();

**while** (e.hasNext()) {

add(index++, e.next());

modifie = **true**;

}

**return** modifie;

}

4 Implémentation des itérateurs :

**public** Iterator<E> iterator() {

**return** Itr();

}

La classe imbriquée *Itr* implémente l’interface *Iterator,*et est définie de la façon suivante :

**private** **class** Itr **implements** Iterator {

*// indice du prochain élément à retourner par next*

**int** curseur =0;

*// indice du dernier élément retourné par next*

**int** derRet = -1;

***modCompte*** est un attribut de la classe *AbstractList*qui représente le nombre de fois que la liste a été structurellement modifiée : une modification structurelle est un (ou plusieurs) ajout(s), un (ou plusieurs) retrait(s) qui risquent de rendre incohérents les parcours à l'aide d'itérateurs. Un itérateur stocke la valeur de *modCompte* quand il est créé, et lève une exception si, lors d'une utilisation de l'itérateur, le *modCompte* de l'objet collection ne correspond pas au *modCompte* de l'itérateur.

**int**modCompteAttendu = modCompte;

**public** **boolean** hasNext() {

**return** curseur != size();

}

|  |  |
| --- | --- |
| **public** E next() {  testModification();  **try** {  E next = get(curseur);  derRet = curseur++;  **return** next;  } **catch**(IndexOutOfBoundsException e) {  testModification();  **throw** **new** NoSuchElementException();  }  } | **final** **void** testModification() {  **if** (modCompte != modCompteAttendu)  **throw** **new** ConcurrentModificationException();  } |

**public** **void** remove() {

**if** (derRet == -1) *// l'opération précédente n'est pas un next*

**throw** **new** IllegalStateException();

testModification();

**try** {

AbstractList.this.remove(derRet);*// appel du remove de la classe englobante*

**if**(derRet < curseur) curseur--;

derRet = -1;

modCompteAttendu= modCompte; *// on remet à jour modCompteAttendu : modCompte a été modifié.*

} **catch**(IndexOutOfBoundsException e) {

**throw** **new** ConcurrentModificationException();

}

}

Les méthodes suivantes retournent un *ListIterator* :

ListIterator listIterator() {

**return** listIterator(0);

}

ListIterator listIterator(**final int** index) {

**if** (index<0 || index>nbElements)

**throw** **new** IndexOutOfBoundsException("Index: "+index);

**return** **new** ListItr(index);

}

La classe imbriquée *ListItr* implémente l’interface *ListIterator* :

**private** **class** ListItr **extends** Itr **implements** ListIterator<E>

ListItr(**int** index) {

curseur = index;

}

**public** **boolean** hasPrevious() {

**return** curseur != 0;

}

**public** E previous() {

testModification();

**try** {

**int** i = curseur - 1;

E previous = get(i);

derRet = curseur = i;

**return** previous;

} **catch**(IndexOutOfBoundsException e) {

testModification();

**throw** **new** NoSuchElementException();

}

}

**public** **int** nextIndex() {

**return** curseur;

}

**public** **int** previousIndex() {

**return** curseur-1;

}

**void** set(Object o) {

**if** (derRet == -1)

**throw** **new** IllegalStateException();

testModification();

**try**{

AbstractList.this.set(derRet, o);

modCompteAttendu= modCompte;

}**catch**(IndexOutOfBoundsException e) {

**throw** **new** ConcurrentModificationException();

}

}

**public** **void** add(Object o) {

testModification();

**try** {

AbstractList.this.add(curseur++, o);

derRet = -1;

modCompteAttendu= modCompte;

}**catch**(IndexOutOfBoundsException e) {

**throw** **new** ConcurrentModificationException();

}

}

5 La méthode *subList( int depuis, int jusqua)*

**public** List<E> subList(**int** fromIndex, **int** toIndex) {

**return** (this **instanceof** RandomAccess )

**new** RandomAccessSubList<E>(**this**, fromIndex, toIndex) :

**new** SubList<E>(**this**, fromIndex, toIndex));

}

Les classes *SubList* (sous classe de *AbstractList*) et sa dérivée *RandomAccessSubList* sont des classes privées du paquetage *java.util*.

6 La méthode *equals*

La méthode *equals* est définie de la façon suivante :

**public** **boolean** equals(Object o) {

**if** (o == **this**)

**return** **true**;

**if** (!(o **instanceof** List))

**return** **false**;

ListIterator<E> e1 = listIterator();

ListIterator e2 = ((List) o).listIterator();

**while**(e1.hasNext() && e2.hasNext()) {

E o1 = e1.next();

Object o2 = e2.next();

**if** (!(o1==**null** ? o2==**null** : o1.equals(o2)))

**return** **false**;

}

**return** !(e1.hasNext() || e2.hasNext());

}

# *AbstractSet*

Cette classe fournit  une implémentation minimale de quelques méthodes de l'interface [*Set*](http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/Conteneurs/Set.html). Elle redéfinit uniquement les méthodes *equals* et la méthode *removeAll*de la classe*AbstractCollection.*

**public** **boolean** equals(Object o) {

**if** (o == **this**) **return** **true**;

**if** (!(o **instanceof** Set)) **return** **false**;

Collection c = (Collection) o;<

**if** (c.size() != size()) **return** **false**;

**try** {

**return** containsAll(c);

} **catch**(ClassCastException unused) {

**return** **false**;

} **catch**(NullPointerException unused) {

**return** **false**;

}

}

L'algorithme de *removeAll* parcourt la *Collection* la plus petite en nombre d'éléments :

**public** **boolean** removeAll(Collection<?> c) {

**boolean** modifie = **false**;

**if** (size() > c.size()) {

**for** (Iterator<?> i = c.iterator(); i.hasNext(); )

modifie |= remove(i.next());

} **else** {

**for** (Iterator<?> i = iterator(); i.hasNext(); ) {

**if**(c.contains(i.next())) {

i.remove();

modifie = **true**;

}

}

}

**return** modifie;

}

# *AbstractSequentialList*

**public** **abstract** **class** AbstractSequentialList<E> **extends** AbstractList<E> {

. . .

}

Définition du *get*

**public** E get(**int** index) {

ListIterator<E> e = listIterator(index);

**try** {

**return**(e.next());

} **catch**(NoSuchElementException exc) {

**throw** **new** IndexOutOfBoundsException("Index: "+index);

}

}

Affectation de *element* à l’objet en position index dans le conteneur :

**public** E set(**int** index, E element) {

ListIterator<E> e = listIterator(index);

**try** {

Object oldVal = e.next();

e.set(element);

**return** oldVal;

} **catch**(NoSuchElementException exc) {

**throw** **new** IndexOutOfBoundsException("Index: "+index);

}

}

Insertion d’un élément à un indice donné :

**public** **void** add(**int** index, E element) {

ListIterator<E> e = listIterator(index);

e.add(element);

}

Oter un élément à un indice donné :

**public** E remove(**int** index) {

ListIterator<E> e = listIterator(index);

E ote;

**try** {

ote = e.next();

} **catch**(NoSuchElementException exc) {

**throw** **new** IndexOutOfBoundsException("Index : "+index);

}

e.remove();

**return** ote;

}

Ajout des éléments d’une collection :

**public** **boolean** addAll(int index, Collection<? **extends** E> c) {

**boolean** modifie = **false**;

ListIterator<E> e1 = listIterator<E>(index);

Iterator<? **extends** E>r e2 = c.iterator();

**while** (e2.hasNext()) {

e1.add(e2.next());

modifie = **true**;

}

**return** modifie;

}

La méthode *Iterator* renvoie un *ListIterator*, et *listIterator* est abstrait :

**public** Iterator<E> iterator() {

**return** listIterator();

}

**public** **abstract** ListIterator listIterator(**int** index);

***ArrayList***

|  |
| --- |
| **Sommaire**   1. [Constructeurs](http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/Conteneurs/ArrayList.html#constructeur) 2. [Les méthodes de l'interface *Collection*](http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/Conteneurs/ArrayList.html#methodeC) 3. [Les méthodes de l'interface *List*](http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/Conteneurs/ArrayList.html#methodeL) 4. [Les méthodes propres à *ArrayList*](http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/Conteneurs/ArrayList.html#methodeAL) |

La classe *ArrayList* implémente un tableau d’objets qui peut grandir ou rétrécir à la demande, ce qui débarrasse le programmeur de la gestion de la taille du tableau. Comme pour un tableau on peut accéder à un élément du *ArrayList*, par un indice.  
Les éléments de la liste tableau sont représentés dans un tableau d'objets de type *E* : *elementData*, qui sont tous tassés en tête du tableau dans les élément d’indice 0 à *size*nombre d’éléments présents dans le *ArrayList.*

|  |  |
| --- | --- |
| Une instance de *ArrayList* contenant quatre objets peut être représentée de la façon suivante : | représentation d'une instance de ArrayLIst |

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

|  |  |
| --- | --- |
| Insertion d'un élément, dans un *ArrayList* qui contient 4 éléments et qui est déjà plein : le tableau *elementData* est agrandi. | ArrayList avant insertion |
| Après ajout, la taille de *elementData* est 4\*3/2+1 = 7 | ArrayList après ajout (animation) |

**public** **class** ArrayList<E> **extends** [AbstractList](http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/Conteneurs/AbstractList.html)<E>

**implements**List<E>, RandomAccess, Cloneable, java.io.Serializable{

**private** **transient** E[] elementData;>

**private** **int** size;

**. . .**

1 Constructeurs

Le constructeur suivant construit un *ListeTableau* avec sa capacité initiale :

**public** ArrayList(**int** capacityInitial) {

**super**();

**if**(capacityInitial<0) **throw** **new** IllegalArgumentException("capacityInitial : "

+capacityInitial)

elementData = (E[])**new** Object[capacityInitial];

size = 0;

}

Le constructeur par défaut construit un *ListeTableau* de 10 emplacements.

**public** ArrayList() {

**this**(10);

}

Le constructeur suivant permet de construire un *ArrayList* à partir d'une collection d'objets qui sont de la classe *E* ou d'une sous-classe de *E* :

**public** ArrayList(Collection<? **extends** E> c) {

size = c.size();

*// On se donne 10% de plus pour grandir dans le futur*

elementData = (E[])**new** Object[

(**int**)Math.min((size\*110L)/100,Integer.MAX\_VALUE)];

c.toArray(elementData);

}

2 Les méthodes de l'interface *Collection* :

La méthode *add* ajoute un élément au *ArrayList*la méthode appelle  la méthode *ensureCapacity*, qui vérifie qu’il y a de la place pour réaliser l’ajout, et s’il n’y a pas de place agrandit le tableau. La méthode *ensureCapacity* pouvant servir à un utilisateur de la classe, elle est publique.

**public** **void** ensureCapacity(capaciteMin){

*// assure que la capacité du tableau représentant les éléments*

*// est au moins de capaciteMin*

**int** ancienneCapacite = elementData.length;

**if** (capaciteMin > ancienneCapacite) {

E [] ad = elementData;

**int** nouvelleCapacity = ancienneCapacite \* 3 / 2 + 1;

**if**(nouvelleCapacity < capaciteMin)

nouvelleCapacity = capaciteMin;

elementData = (E[])**new** Object[nouvelleCapacity];

System.arraycopy(ad, 0, elementData, 0, size);

}

}

La méthode *add*ajoute un élément en fin de tableau :

**public** **boolean** add(E o){

ensureCapacity(size+1);

elementData[size++] = o;

**return** **true**;

}

La méthode *clear* enlève tous les éléments du vecteur, *size* retourne le nombre d’éléments du ArrayList et *isEmpty* retourne vrai si et seulement si le ArrayListest vide :

**public** **void** clear(){

**for**( **int** i = 0; i< nbElements; ++i)

elementData = **null**;

size = 0;

}

**public** **int** size(){**return** size;}

**public** **boolean** isEmpty(){

**return** size ==0;

}

La méthode *contains* retourne *true* si *o* appartient au *ArrayList* et *false* sinon :

**public** **boolean** contains(Object o){

**return** indexOf(o)!=-1;

}

La méthode *remove*enlève la première occurrence de *o* du *ArrayList*, si elle existe et dans ce cas retourne *true*, sinon la méthode retourne *false*: dans le cas où l’élément est enlevé, il faut « retasser » le *ArrayList* :

**public** **boolean** remove(Object o){

**if**(o == **null**){

**for** (int index = 0; index < size; index++)

**if** (elementData[index] == **null**) {

fastRemove(index);

**return true**;

}

}**else**{

**for** (**int** index = 0; index < size; index++)

**if** (o.equals(elementData[index])){

fastRemove(index);

**return true;**

}

}

**return false**;

}

**private** **void** fastRemove(**int** index) {

**int** numMoved = size - index - 1;

**if** (numMoved > 0)

System.arraycopy(elementData, index+1, elementData, index, numMoved);

elementData[--size] = **null**; *// le Garbage Collector peut éventuellement récupérer l'espace*

}

Les deux méthodes suivantes convertissent un *ArrayList* en un tableau d’*Object*:

**public** Object [] toArray(){

Object [] resultat = **new** Object[size];

System.arraycopy(elementData, 0, resultat, 0, size);

**return** resultat;

}

La méthode suivante, copie tous les éléments du *ArrayList* dans le tableau *t* et retourne ce tableau si celui ci estassez grand,  sinon ellecopie les éléments dans un tableau qui est créé et le retourne.

* Si le tableau *t* est plus grand que le nombre d’éléments du vecteur, alors l’élément qui suit le dernier copié est affecté de la valeur **null**.
* Le type du tableau retourné est le même que le type du tableau *t*. Une exception *ArrayStoreException* est levée si le *ArrayList* contient un élément qui n’est pas de la classe (ou d’une sous classe ) des éléments du tableau *t*.>

**public** <T> T[] toArray(T[] t) {

**if** (t.length < size)

t = (T[])java.lang.reflect.Array.newInstance(t.getClass().getComponentType(),size);

System.arraycopy(elementData, 0, t, 0, size);

**if** (t.length > size) t[size] = **null**;

**return** t;

}

Parmi les opérations de masse, seule la méthode *addAll* est redéfinie, pour profiter de l'optimisation due au *arraycopy* :

**public** **boolean** addAll(Collection<? **extends** E> c) {

Object[] a = c.toArray();

**int** numNew = a.length;

ensureCapacity(size + numNew);

System.arraycopy(a, 0, elementData, size, numNew);

size += numNew;

**return** numNew != 0;

}

3 Les méthodes de l'interface *List* :

La méthode suivante permet d’ajouter un élément à un indice donné dans le *ArrayList* :

* Lève une exception si l’indice n’est pas « correct ».
* Déplace tous les éléments d’indice supérieur ou égal à index pour faire la place avant l’insertion.

**public** **void** add(**int** index, Object o){

**if**( index < 0 || index > size) **throw** **new** IndexOutOfBoundsException();

ensureCapacity( size + 1);

*// déplacement des elements d’indice >= index*

System.arraycopy(elementData, index, elementData, index+1,size - index );

elementData[index] = o;

++size;

}

La méthode *get* retourne l’élément se trouvant à un indice, ou lève une exception :

**public** Object get(**int** index){

**if**(index < 0 || index >= nbElements) **throw** **new** IndexOutOfBoundsException();

**return** elementData[index];

}

Les deux méthodes suivantes retournent l’indice d’un objet dans le *ArrayList*s’il est présent, -1 sinon. *indexOf* commence au début du *ArrayList*et *lastIndexOf* commence à la fin du vecteur :

|  |  |
| --- | --- |
| **public** **int** indexOf(Object elem) {  **if** (elem == **null**) {  **for** (**int** i = 0; i < size; i++)  **if**(elementData[i]==**null**)  **return** i;  }**else** {  **for** (**int** i = 0; i < size; i++)  **if** (elem.equals(elementData[i]))  **return** i;  }  **return** -1;  } | **public** **int** lastIndexOf(Object elem) {  **if** (elem == **null**){  **for** (**int** i = size-1; i >= 0; i--)  **if**(elementData[i]==**null**)  **return** i;  }**else**{  **for** (**int** i = size-1; i >= 0; i--)  **if** (elem.equals(elementData[i]))  **return** i;  }  **return** -1;  } |

La méthode *Object remove(int index)* permet d’enlever du *ArrayList*l’élément se trouvant à l’indice *index* si celuici est correct, et retourne l’objet enlevé du *ArrayList* :

**public** E remove(**int** index){

**if**( index < 0 || index >= size) **throw** **new** IndexOutOfBoundsException();

E o = elementData[index];

System.arraycopy(elementData, index+1, elementData, index, size - index - 1);

elementData[ size - 1]=**null**;

--size;

**return** o;

}

La méthode *Object set(int index, Object o)* modifie l’objet se trouvant en position *index* dans le vecteur, en lui affectant o, et retourne l’objet qui occupait cette place avant.

**public** e set(**int** index, e o){

**if**( index < 0 || index >= size)>**throw** **new** IndexOutOfBoundsException();

E x = elementData[index];

elementData[index] = o;

**return** x;

}

4 Méthode propres à *ArrayList*

*TrimToSize* ajuste la capacité du *ArrayList* à son nombre d'éléments.

**public** **void** trimToSize(){

**int** ancienneCapacity = elementData.length;

**if**(nbElements < ancienneCapacity) {

E[] oldData = elementData;

elementData = (E[])**new**Object[size];

System.arraycopy(oldData, 0, elementData,0, nbElements);

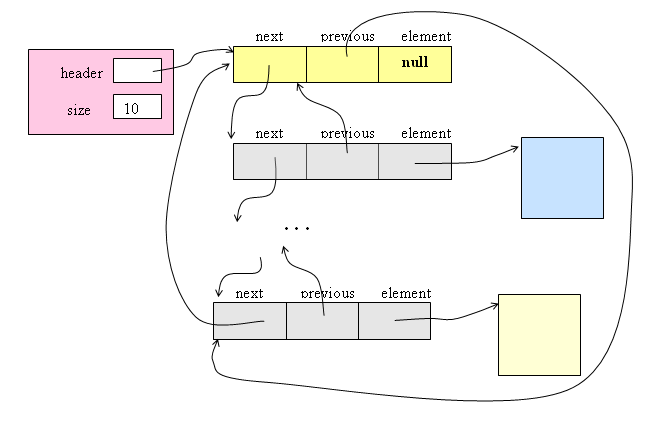
}

}

***LinkedList***

|  |
| --- |
| **Sommaire**   1. [Constructeurs](http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/Conteneurs/LinkedList.html#constructeur) 2. [Quelques outils](http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/Conteneurs/LinkedList.html#outils) 3. [Les méthodes de l'interface *Collection*](http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/Conteneurs/LinkedList.html#methodeC) 4. [Redéfinition des méthodes de *Object*](http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/Conteneurs/LinkedList.html#object) 5. [Les méthodes de l'interface *List*](http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/Conteneurs/LinkedList.html#methodeL) 6. [Les méthodes de propes à *LinkedList*](http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/Conteneurs/LinkedList.html#methodeLL) 7. [Implémentation des itérateurs](http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/Conteneurs/LinkedList.html#iterateur) |

La classe *LinkedList*implémente une liste d’objets représentée de la façon suivante :



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

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

La classe *Entry* peut être définie comme une classe interne privée : elle est simplement munie d’un constructeur.

**private** **static class** Entry<E> {

E element;

Entry<E> next;

Entry<E> previous;

Entry(E element, Entry<E> next, Entry<E> previous) {

**this**.element = element;

**this**.next = next;

**this**.previous = previous;

}

}

*LinkedList* est définie :

**public** **class** LinkedList<E> **extends** AbstractSequentialList<E>

**implements** List<E>, Cloneable, Serializable{

**private** **transient** Entry<E>head ;

**private transient int** size ;

**private transient int** modCount = 0;*// pour qu'un itérateur sur une liste puisse savoir*

*// si la liste a été modifiée par ailleurs (ajout ou retrait)*

...

}

1 Les constructeurs

Les constructeurs de la classe *LinkedList* sont  définis comme suit :

**public** LinkedList() {

header = **new** Entry<E>(**null**, **null**, **null**);

size = 0;

header.next = header.previous = header;

}

**public** LinkedList(Collection<? **extends** E> c) {

**this**();

addAll(c);

}

2 Quelques outils

Avant de définir les méthodes de l’interface *Collection*, puis de l’interface *List*, définissons quelques outils :

La première méthode permet d’enlever de la liste des chaînons, le chaînon *c* :

**private** E remove(Entry<E> e) {

*// enlève le chaînon e de la liste, et retourne l'élément retiré*

**if** (e == header) **throw new** NoSuchElementException();

E result = e.element;

e.previous.next = e.next;

e.next.previous = e.previous;

e.next = e.previous = **null**;

e.element = **null**;

size--;

**return** result;

}

Les deux méthodes suivantes permettent d’ajouter un chaînon contenant *o* dans son champ élément, avant, ou après le chaînon *e* :

**private** Entry<E> addBefore(E o, Entry<E> e) {

*// ajoute un nouveau chaînon contenant o avant e*

*// et retourne la référence au nouveau chaînon*

Entry<E> newEntry = **new** Entry<E>(o, e, e.previous);

newEntry.previous.next = newEntry;

newEntry.next.previous = newEntry;

size++;

modCount++;

**return** newEntry;

}

**private** Entry<E> addAfter(E o, Entry<E> e) {

*// ajoute un nouveau chaînon contenant o après e*

*// et retourne la référence au nouveau chaînon*

Entry<E> newEntry = **new** Entry<E>(o, e.next, e);

e.next = newEntry;

newEntry.next.previous = newEntry;

size++;

modCount++;

**return** newEntry;

}

La méthode suivante permet d’obtenir la référence au chaînon d’indice *index*, s’il existe :

**private** Entry<E> entry(**int** index) {

**if** (index < 0 || index >= size) **throw** **new** IndexOutOfBoundsException("Index: "+index+", Size: "+size);

Entry<E> e = header;

**if** (index < (size >> 1))

**for** (**int** i = 0; i <= index; i++) e = e.next;

**else**

**for** (**int** i = size; i > index; i--) e = e.previous;

**return** e;

}

3 Méthodes de l’interface *Collection*

La méthode *add* ajoute un élément en fin de liste :

**public** **boolean** add(E o){

*// ajout à la fin de la liste*

addBefore(o, header);

**return** **true**;

}

**public** **boolean** addAll(Collection<? **extends** E> c){

*// ajout des éléments de c à la fin de la liste*

**for**(E e : c) addBefore(e, header);

**return** c.size()!=0;

}

**public** **void** clear(){

*// le for est là pour aider gc*

**for** (Entry<E> e = header.next, next = e.next; e != header; e = next, next = e.next) {

e.next = e.previous = **null**;

e.element = **null**;

}

header.next = previous = header;

size = 0;

}

**public** **boolean** isEmpty(){

**return** size == 0;

}

**public** **int** size(){

**return** size;

}

**public**< **boolean** contains(Object o){

Entry<E> p;

**if**(o==b>null){

**for**( p = header.next; p != header; p = p.next)

**if**(p.element==**null**) **return** **true**;

}**else**{

**for**(p = header.next; p != header; p = p.next)

**if**(o.equals(p.element)) **return** **true**;

}

**return** **false**;

}

La méthode *containsAll* n'est pas redéfinie.

**public** **boolean** remove(Object o){

**if**(o == **null**){

**for**(Entry<E> p = header.next; p != header; p = p.next)

**if**(p.element==**null**){ remove(p); **return** **true**;}

}**else**{

**for**(Entry<E> p = header.next; p != header; p = p.next)

**if**(o.equals(p.element)){ remove(p); **return** **true**;}

}

**return** **false**;

}

Les méthodes *removeAll* et *retainAll* ne sont pas redéfinies.

**public** Object [] toArray(){

Object [] resultat = **new** Object[size];

Entry<E> p = header.next;

**for**( **int** i = 0; i < size; ++i, p = p.next)

resultat[i] = p.element;

**return** resultat;

**public** <T> T [] toArray(T [] t){

**if** (t.length < size)

t = (T[])java.lang.reflect.Array.newInstance(a.getClass().getComponentType(), size);

**int** i = 0;

Object[] result = t;

**for** (Entry<E> e = header.next; e != header; e = e.next)

result[i++] = e.element;

**if** (t.length > size) t[size] = **null**;

**return** t;

}

4 Redéfinition des méthodes de *Object*

*equals* et *toString* ne sont pas redéfinies.

**public** Object clone(){

**try** {

LinkedList<E> l = (LinkedList<E>)**super**.clone();

*// attention ne pas faire*

*// l.tete =* ***new*** *Chainon(null, l.tete, l.tete);*

l.header = **new** Entry<E>(**null**, **null**, **null**);

l.header.next = l.header.previous = l.header;

l.size = 0;

**for**( Entry<E> p = header.next; p!=header; p = p.next) l.add(p.element);

**return** l;

} **catch** (CloneNotSupportedException e) {

*// ne doit pas se produire :*

*// ListeLiee implémente Cloneable*

**throw** **new** InternalError();

}

}

5 Méthodes de l’interface *List*

**public** **boolean** add(**int** index, E o){

*// ajout de o au rang index*

**if** (index == size) add(o);

**else** addBefore(o, entry(index));

**return** **true**;

}

**public** **boolean** addAll(**int** index, Collection<? **extends** E> c){

*// ajout des éléments de c à partir de l'index*

e = entry(index);

**for**(E o : c) e.addBefore(o);

**return** c.size()!=0;

}

**public** E get(**int** index){

**return** entry(index).element;

}

**public** **int** indexOf(Object o){

Entry<E> p;

**int** index=0;

**if**(o==**null**){

**for**( p = header.next; p != header; ++index, p = p.next)

**if**(p.element==**null**) **return** index;

}**else**{

**for**( p = header.next; p != header; ++index, p = p.next)

**if**(o.equals(p.element)) **return** index;

}

**return** -1;

}

**public** **int** lastIndexOf(Object o){

Entry<E> p;

**int** index=size-1;

**if**(o==**null**){

**for**( p = header.previous; p != header; --index, p = p.previous)

**if**(p.element==**null**) **return** index;

}**else**{

**for**(p = header.previous; p != header; --index, p = p.previous)

**if**(o.equals(p.element)) **return** index;

}

**return** -1;

}

**public** E remove(**int** index){

**return** remove(entry(index));

}

**public** E set(**int** index, E o){

Entry<E> p = entry(index);

E ancienElement = p.element;

p.element = o;

**return** ancienElement;

}

6 Méthodes propres à *LinkedList*

**public** Object getFirst(){

**if**(size == 0) **throw** **new** NoSuchElementException();

**return** header.next.element;

}

**public** Object getLast(){

**if**(size == 0) **throw** **new** NoSuchElementException();

**return** header.previous.element;

}

**public** E removeFirst(){

remove(header.next);

}

**public** Object removeLast(){

**return** remove(header.previous);

}

**public** **void** addFirst(Object o){

addBefore(o, header.next);

}

**public** **void** addLast(Object o){

addBefore (o, header);

}

7 Itérateurs

 Les méthodes *iterator()* et *listIterator()* définies dans *AbstractList*, ne sont pas redéfinies. Seule la méthode *listIterator(int index )* est définie.

**public** ListIterator<E> listIterator(**int** index) {

**return** **new** ListItr(index);

}

**private** **class** ListItr **implements** ListIterator<E> {

**private** Entry<E> lastReturned = header;

**private** Entry<E> next;

**private** **int** nextIndex;

**private** **int** expectedModCount = modCount;

ListItr(**int** index) {

*// index est l'indice du prochain élément obtenu par next*

*// positionner next sur l'élément d'indice index*

*// et nextIndex à la valeur index*

**if** (index < 0 || index > size)

**throw new** IndexOutOfBoundsException("Index: "+index+ ", Size: "+size);

**if** (index < (size >> 1)) {

next = header.next;

**for** (nextIndex=0; nextIndex<index; nextIndex++)

next = next.next;

}**else**{

next = header;

**for** (nextIndex=size; nextIndex>index; nextIndex--)

next = next.previous;

}

}

**public** **boolean** hasNext() {

**return** nextIndex != size;

}

**public** E next() {

checkForComodification();

**if** (nextIndex == size)**throw** **new** NoSuchElementException();

lastReturned = next;

next = next.next;

nextIndex++;

**return** lastReturned.element;

}

**public** **boolean** hasPrevious() {

**return** nextIndex != 0;

}

**public** E previous() {

**if** (nextIndex == 0)**throw** **new** NoSuchElementException();

lastReturned = next = next.previous;

nextIndex--;

checkForComodification();

**return** lastReturned.element;

}

**public** **int** nextIndex() {

**return** nextIndex;

}

**public** **int** previousIndex() {

**return** nextIndex-1;

}

**public** **void** remove() {

checkForComodification();

Entry<E> lastNext = lastReturned.next;

**try** {

LinkedList.this.remove(lastReturned);

}**catch** (NoSuchElementException e) {

**thrownew** IllegalStateException();

}

**if** (next==lastReturned) next = lastNext;

**else**nextIndex--;

lastReturned = header;

expectedModCount++;

}

**public** **void** set(E o) {

**if** (lastReturned == header)**throw** **new** IllegalStateException();

checkForComodification();

lastReturned.element = o;

}

**public** **void** add(E o) {

checkForComodification();

lastReturned = header;

addBefore(o, next);

nextIndex++;

expectedModCount++;

}

**final** **void** checkForComodification() {

**if** (modCount != expectedModCount) **throw** **new** ConcurrentModificationException();

}

}

***Structure d'arbre***

|  |
| --- |
| **Sommaire**   1. [Définitions](http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/CoursJava/arbres.html#definition) 2. [Propriétés](http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/CoursJava/arbres.html#propriete) 3. [Représentation des arbres N-aires](http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/CoursJava/arbres.html#representationN) 4. [Représentation des arbres binaires](http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/CoursJava/arbres.html#representationB) |

1 Définitions.

      Graphe orienté.

Un graphe orienté est défini par le couple *(N, A)* :

* *N* : ensemble des noeuds ou sommets du graphe
* *A* : ensemble des arêtes du graphe. L'ensemble des arêtes est un sous-ensemble du produit cartésien *N*×*N*. Une arête est une ligne joignant un noeud de départ *d* à un noeud d'arrivée *a*. L'arête da si elle existe est différente de l'arête *ad*.

Un graphe orienté peut être considéré comme une représentation d'une relation binaire *R* sur l'ensemble*N* :

Soient *a* et *b* appartenant à *N* alors : *aRb ⇔  ∃x ∈ A et x = ab*

      Graphe non orienté

Un graphe est non orienté, si on ne distingue pas le sommet de départ et le sommet d'arrivée.

      Successeur, prédécesseur

La relation successeur est une application de *N* dans l'ensemble ℘*(N)*, ensembles des parties de *N*, qui à tout*n* appartenant à *N* associe l'ensembles des *a* appartenant à *N* tels que *na* appartient à *A*.

La relation prédécesseur est une application de *N* dans l'ensemble ℘*(N)*, ensemble des parties de *N*, qui à tout *n* appartenant à *N* associe l'ensembles des *a* appartenant à *N* tels que *da* appartient à *A*.

      Chaîne, cycle, chemin, circuit

Une chaîne est une séquence d'arêtes *{a1, ... ak}* telle que pour tout *i*de *1* à *k-1* les arêtes *ai* et *ai+1* ont une extrémité commune. Un cycle est une chaîne telle que *a*1 et *ak*ont une extrémité commune.

Un chemin est une séquence d'arêtes *{a1, ... ak}* telle que pour tout *i* de *1* à *k-1* les arêtes *ai* et *ai+1* ont une extrémité commune, cette extrémité étant de type arrivée pour *ai* et départ pour *ai+1*. Un circuit est un chemin tel que a1 et ak ont une extrémité commune, cette extrémité étant de type arrivée pour *ak* et départ pour *a1*.

La longueur du chemin, chaîne, circuit ou cycle est le nombre d'arêtes formant ce chemin, chaîne, circuit ou cycle.

      Racine

Un graphe admet une racine *r* si pour tout *n* appartenant à *N* il existe un chemin de *r* à *n*. Un graphe sans circuit qui admet une racine, en admet une seule.

      Connexe

Un graphe est connexe si et seulement si ∀*( n1, n2)∈N×N*, ∃ une chaîne de *n1* à *n2*.

      Arbre

Un arbre est un graphe connexe sans cycle et muni d'une racine.

Une structure d'arbre induit une partition sur l'ensemble des noeuds de l'arbre :

1. Il y a dans *N* un noeud r appelé la racine. La racine est le point d'arrivée d'aucune arête.
2. Les autres noeuds, s'il y en a, sont partitionnés en *m* sous ensembles *N1, N2, ... Nm* disjoints, qui sont eux mêmes des arbres de racines *r1, r2, ..., rm*. Ces *m*arbres sont appelés sous-arbres de la racine.

Cette définition peut s'écrire de la façon suivante :  
      *A→∅*  
      *A→ <r , A1, A2, ... An>*

exemple :

|  |  |
| --- | --- |
| N=(A,B,C,D,E,F,G,H,I,J,K).   * La racine de l'arbre est A. * m=4 * r1=B, r2=C, r3=D, r4=E. * N1=(B,H) N2=(C,F,G) N3=(D,I,J,K) N4=(E). | arbre |

Cet arbre peut être représenté sous forme indentée :

A  
    B  
         F  
   C  
         G  
   D  
         I  
                 K  
        J  
   E

Il peut être aussi représenté sous forme parenthésée : (A (B (F), C( G ), D(I(K), J),E ))

      Ascendance, Descendance

Soit *n1,n2, ... nk k* noeuds d'un arbre tels que : ∀*i, 1* ≤*i* ≤*k* on a *ni*  = prédécesseur(*ni+1*).

Une telle suite est appelée **chemin** entre le noeud *n1* et le noeud *nk*. La longueur du chemin est égale au nombre d'arêtes, c'est à dire au nombre de noeuds - 1.

S'il existe un chemin du noeud a vers le noeud b on dit que a est un ascendant de b, ou que b est un descendant de a.

      Hauteur, profondeur, niveau

La **hauteur** d'un noeud est la longueur du plus long chemin de ce noeud aux feuilles qui en dépendent plus 1 : c'est le nombre de nœuds du chemin. La **hauteur** d'un arbre est la hauteur de sa racine. L'arbre vide a une hauteur 0, et l'arbre réduit à une racine a une hauteur 1.  
La **profondeur** d'un noeud dans un arbre est le nombre de noeuds du chemin qui va de la racine à ce noeud. La racine d'un arbre est à une profondeur 1, et la profondeur d'un noeud est égale à la profondeur de son prédécesseur plus 1. Si un noeud est à une profondeur p, tous ses successeurs sont à une profondeur p+1.

Tous les noeuds d'un arbre de même profondeur sont au même niveau.

      Feuille, Noeud interne

Une feuille ou noeud terminal est un noeud qui n'a pas de successeur.

Tout noeud d'un arbre qui n'est pas une feuille est un noeud interne.

      Forêt

Nous appellerons forêt un ensemble ordonné d'arbres.

Remarque : les sous-arbres d'un noeud dans un arbre ordonné, forment une forêt.

      Arbre n-aire

Un arbre est un arbre n-aire si tous les noeuds de l'arbre ont au plus *n* successeurs.

      Arbre binaire

Un arbre est un arbre binaire, si tout noeud de l'arbre a 0, 1, ou 2 successeurs. Ses successeurs sont alors appelés respectivement successeur gauche et successeur droit. Lorsque tous les noeuds d'un arbre binaire ont deux ou zéro successeurs nous dirons que l'arbre binaire est **homogène**. Un arbre binaire est **arbre binaire dégénéré** si tous ses noeuds n'ont qu'un seul descendant.

Un **arbre binair**e **complet** est un arbre binaire tel que chaque niveau de l'arbre est complètement rempli. Un arbre binaire complet de hauteur *h* contient donc 2*h-1* nœuds, et son nombre de feuilles est : *Fh = 2h-1.*

On commence par calculer le nombre de feuilles d'un arbre binaire complet. La démonstration se fait par récurrence :

1. *F1 = 1 = 21-1.*
2. Supposons que *Fh = 2h-1* jusqu'au rang *h*. Alors *Fh+1 = 2*×*Fh =2*×*2h-1* = *2h* .

On peut alors calculer le nombre de noeuds d'un arbre binaire complet :

1. *N0 = 0*
2. *Nh = Nh-1 + Fh pour h≥1*

Soit *Nh = F1 + F2 + ...+ Fh  = 1+2+ ...+2h-1~~=~~Σi=0 h-12i= 2h-1*

Un **arbre binaire parfait** est tel que tous les niveaux sauf éventuellement le dernier sont remplis, et dans ce cas les feuilles du dernier niveau sont groupées à gauche.  
Un**arbre binaire parfaitement équilibré** est un arbre binaire complet dont les noeuds de l'avant dernier niveau peuvent n'avoir qu'un seul descendant.  
Un arbre binaire est l**ocalement complet** si tous les noeuds qui ne sont pas des feuilles ont deux fils.  
Un arbre **binaire étendu** est un arbre binaire dans lequel nous remplaçons tout descendant vide par un noeud spécial noté Noeud étendu.

Exemple :

|  |  |  |
| --- | --- | --- |
| Arbre sans noeud étendu | Est étendu en : | arbre avec naouds étendus |

On démontre facilement par récurrence sur le nombre *n* de noeuds de l'arbre initial que le nombre *C* de noeuds étendus est tel que :*C = n + 1*

1. La propriété est vraie pour *n=1*.
2. Supposons la vraie jusqu'au rang *n-1* et soit *A=<x, A1, A2>* alors *A1*et *A2*sont deux arbres ayant au plus *n-1* noeuds. Donc*C1=n1+1* et *C2=n2+1*. Nous avons alors : *C = C1+C2 = n1+1+n2+1 =n+1*

Le **cheminement interne** d'un arbre binaire est la somme, pour tous les noeuds de l'arbre, des profondeurs des noeuds. C'est la somme des profondeurs de tous les noeuds de l'arbre. Dans l'exemple précédent le cheminement interne est égal à 8.  
Le **cheminement externe** d'un arbre binaire est la somme, pour tous les noeuds de l'arbre binaire étendu, des longueurs des chemins de la racine à ce noeud. C'est la somme des profondeurs des noeuds . Dans l'exemple précédent le cheminement externe est égal à 17.

Ces deux quantités sont liées respectivement au coût de la recherche avec succès et sans succès d'un noeud dans un arbre binaire. Le cheminement interne divisé par *n* (le nombre de recherches) est le coût moyen de recherche d'un noeud présent dans un arbre binaire ordonné. Le cheminement externe divisé par *n* est le coût moyen de recherche pour un noeud non présent dans un arbre binaire ordonné.

On démontre par récurrence sur le nombre de noeuds que si *E* est le cheminement externe et *I* le cheminement interne :*E = I + 2×n + 1*

1. pour *N=1, I1=1* et*E1=4=I1+2+1*
2. Supposons la propriété vraie jusqu'à l'ordre *n* :*En = In + 2*×*n + 1*  
   Lorsque nous ajoutons un noeud à l'arbre initial, le nombre de noeuds est augmenté de 1, le nombre de noeuds Noeud étendu augmente de 1 également.  
   Soit *C* la longueur du chemin de la racine au noeud que nous venons d'ajouter à l'arbre ;   
   nous avons les relations :
   * *In+1 = In + C*
   * *En+1 = En - C + 2×(C + 1)*

D'où nous tirons : *En+1 - In+1 = En - In + 2*  
Soit*En+1 = In+1 + 2×(n + 1) + 1*

2 Propriétés.

**Propriété 0.**

Soit un arbre binaire de *n* noeuds et de *f* feuilles. On a :*f≤(n+1) /2*

1. la propriété est vraie pour *n=1*;
2. supposons la propriété vraie jusqu'au rang *n-1* : Un arbre de *n* noeuds est constitué de la façon suivante *<a, B1, B2>* et *B1* et *B2*ont au maximum *n-1* noeuds. Nous avons donc pour chacun de ces arbres : *f1≤(n1+1)/2* et *f2≤(n2+1)/2*  
   *n=n1+n2+1* et *f=f1+f2≤(n1+n2+2)/2=(n+1)/2.*

**Propriété 1.**

Un arbre binaire ayant *n* noeuds a une hauteur *h* telle que :*log2(n+1) ≤ h ≤ n*

* Un arbre ayant *n* noeuds de plus grande hauteur est l'arbre dégénéré, et sa hauteur est *n*.
* Un arbre ayant *n* noeuds de plus faible hauteur est un arbre parfait et sa hauteur est *log2(n+1)*

**Propriété 2.**

Tout arbre binaire ayant *f* feuilles a une hauteur supérieure ou égale à*log2(f)*. La fonction logarithme est croissante donc *log2(f) ≤ log2((n+1)/2)*

soit *1+ log2(f) ≤ log2(n+1)*

*1 + log2(f) ≤  log2(n+1) = 1 + log2(n)*

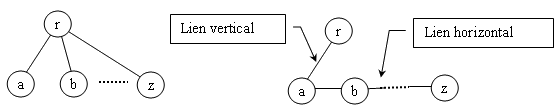
**Propriété 3.**

Le cheminement interne *I*d'un arbre binaire de *n* noeuds est :*(n+1) × (log2(n+1) - 1) +1 ≤  I ≤ n(n-1)/2*

* La deuxième inégalité est obtenue pour un arbre dégénéré.
* La première inégalité est obtenue pour un arbre binaire complet.
* Le cheminement interne dans un arbre parfait de hauteur *h* est *(h-1)×2k+1*

3 Représentation des arbres N-aires.

Les arbres N-aires sont représentés en utilisant deux liens : un lien horizontal et un lien vertical.

Exemple : 

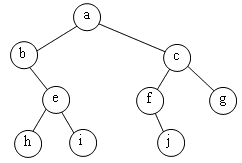
Un arbre N-aire est donc représenté comme un arbre binaire.

4 Représentation des arbres binaires.

*4.1 Représentation contigüe.*

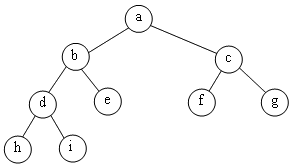
Les noeuds de l'arbre sont rangés dans un tableau à une dimension ; les successeurs gauche et droit du noeud de l'arbre représenté à l'indice I dans le tableau, sont représentés respectivement dans les élément d'indice **2\*I +1** et **2\*(I + 1)** , et la racine est à l’indice 0.

L'arbre :

  
Est représenté par :

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 |
| a | b | c |  | e | f | g |  | h | i |  | j |  |  |  |

Un arbre binaire parfait  a une représentation compacte sans trous dans le tableau:



|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 |
| a | b | c | d | e | f | g | h | i |  |  |  |  |  |  |

*4.2 Représentation chaînée.*

Pour représenter les arbres de façon chaînée nous utilisons une classe *Nœud* qui contient 3 champs : un champ pour représenter l’information liée au nœud, et deux champs pointeurs vers les sous-arbres droit et gauche.

***Arbre binaire, Arbre binaire ordonné.***

|  |
| --- |
| **Sommaire**   1. [Introduction](http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/CoursJava/arbresBinaires.html#introduction) 2. [Noeud](http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/CoursJava/arbresBinaires.html#noeud) 3. [Arbre Binaire](http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/CoursJava/arbresBinaires.html#arbreBinaire)    1. [Affichage](http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/CoursJava/arbresBinaires.html#affichage)    2. [Nombre d'éléments, hauteur, feuilles](http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/CoursJava/arbresBinaires.html#nbElement)    3. [Forme parenthésée](http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/CoursJava/arbresBinaires.html#formeP) 4. [Arbre binaire ordonné](http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/CoursJava/arbresBinaires.html#arbreBinaireO)    1. [Ajout](http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/CoursJava/arbresBinaires.html#ajout)    2. [Suppression](http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/CoursJava/arbresBinaires.html#suppression)    3. [Recherche](http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/CoursJava/arbresBinaires.html#recherche)    4. [Clonage](http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/CoursJava/arbresBinaires.html#clonage) 5. [Evaluation](http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/CoursJava/arbresBinaires.html#evaluation) |

1 Introduction.

Nous nous proposons d'implanter une classe *ArbreBinaire* permettant de stocker une collections d'objets.

Les opérations que nous voulons pouvoir faire sur un objet instance de la classe *ArbreBinaire*, sont les suivantes:

* Création.
* Nombre d'éléments de l'arbre binaire.
* Affichage de l'arbre binaire en infixé, préfixé, postfixé et indenté.

2 Noeud.

Chacun des élément de l'*ArbreBinaire* est représenté à l'aide d'un objet instance de la classe *Noeud*définie de la façon suivante:

|  |  |
| --- | --- |
| **class** Noeud <E> **implements** Cloneable{  **private** E element;  **private** Noeud<E> gauche, droit;  **public** Noeud (){  element = **null**;  gauche = **null**;  droit = **null**;  }  **public** Noeud ( Object valeur, Noeud<E> g,  Noeud<E> droit){  element = valeur;  gauche = g;  droit = d;  }    **public** Object clone() **throws** CloneNotSupportedException{  *// copie en profondeur d’un noeud*  Noeud<E> g = **null**;  **if**( gauche != **null** ) g = (Noeud<E>)gauche.clone();  Noeud<E> d = **null**;  **if**( droit != **null** ) d = (Noeud<E>)droit.clone();  **return** **new** Noeud<E>(element, g, d);  }  }; | Un *Noeud* peut être représenté par:  http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/CoursJava/arbresBinaires_fichiers/image027.gif |

3 ArbreBinaire.

Nous nous intéresserons d’abord à la classe *ArbreBinaire* qui peut être définie de la façon suivante:

**public** **class** ArbreBinaire<E> {

**private** Noeud<E> racine;

**private** **void** infixe(Noeud<E> r){...}

**private** **void** indente( Noeud<E> r, **int** n){...}

**public** ArbreBinaire(){...}

**public** **int** nombreElement(){...}

**public** **int** hauteur(){...}

**public** **int** nombreFeuilles(){...}

**public** **void** infixe (){...}

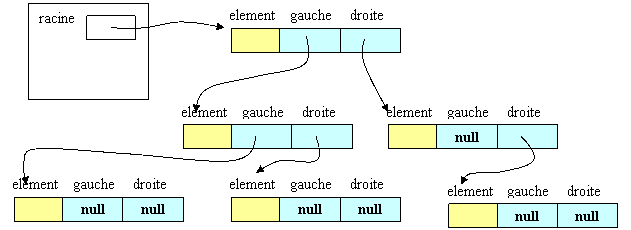
**public** **void** indente(){...}

**public** **String** parenthesee(){...}

**public** **Object** clone(){...}

}

Une instance de *ArbreBinaire* peut être représentée de la façon suivante:



*3.1 Affichage.*

L'affichage en infixé d'un *ArbreBinaire* non vide consiste en l'affichage en infixé du sous-arbre gauche, puis l'affichage de la racine, puis l'affichage en infixé du sous-arbre droit.

**void** infixe(){

infixe (racine);

}

**void** infixe(Noeud<E> r){

**if** (r!=**null**) {

infixe(r.gauche);

System.out.print(r.element.toString()+" ");

infixe(r.droit);

}

}

L'affichage en indenté d'un *ArbreBinaire* non vide consiste en l'affichage en indenté du sous-arbre gauche décalé vers la droite, puis l'affichage de la racine, puis l'affichage en indenté du sous-arbre droit, décalé vers la droite.

**void** indente(){

indente (racine, 0);

}

**void** (Noeud<E> r, **int** n){

**if** (r!=**null**){

indente(r.gauche, n+3);

**for**( **int** I = 0; i< n; ++i) System.out.print(" ");

System.out.print(r.element)

indente(r.droit, n+3);

}

}

*3.21 nombre d’éléments, hauteur, nombre de feuilles.*

**int** nombreElement(){

**return** nombreElement(racine);

}

**int** nombreElement(Noeud<E> r){

**if** (r==**null**)**return** 0;

**else** **return** nombreElement(r.gauche) + 1 + nombreElement(r.droite);

}

**int** hauteur(){

**return** hauteur(racine);

}

**int** hauteur(Noeud<E> r){

**if** (r==**null**) **return** 0;

**else** **return** max( hauteur(r.gauche), hauteur(r.droite)) + 1;

}

**int** nombreFeuilles(){

**return** nombreFeuilles(racine);

}

**int** nombreFeuilles(Noeud<E> r){

**if** (r==**null**)**return** 0;

**else**

**if**(r.gauche==**null** && r.droite==**null**) **return** 1;

**else** **return** nombreFeuilles(r.gauche) + nombreFeuilles(r.droite));

}

*3.3 Forme parenthésée.*

La forme parenthésée d’un arbre binaire est obtenue en construisant une chaîne de caractère contenant: l’élément, puis entre parenthèses et séparés par une virgule, les formes parenthésées de ses deux fils. *null* est représenté par la chaîne de caractères vide.

String parenthesee(){

**return** parenthesee(racine);

}

String parenthesee(Noeud<E> r){

**if** (r==**null**) **return** "";

**else** **return** r.element.toString()+"("+ parenthesee(r.gauche)+", "+parenthesee(r.droit)+")";

}

<4 Arbre binaire ordonné.

Pour la suite, nous appellerons **Arbre Binaire Ordonné** un arbre binaire tel que l’attribut *element* ***e*** possède les propriétés suivantes :

1. ***e*** a une valeur supérieure aux valeurs des attributs *element* de tous les noeuds du sous-arbre gauche.
2. ***e*** a une valeur inférieure aux valeurs des attributs *element* de tous les noeuds du sous-arbre droit.

Les objets rangés dans un arbre binaire ordonné devrons implémenter l’interface *Comparable*.

**class** ArbreBinaireOrdonne <E  **extends** Comparable<E>> **extends** ArbreBinaire<E>{

**public** **void** ajout(E o){...}

**public** **void** suppression(E o){...}

**public** Noeud<E> chercher(E o){...}

**private** **void** ajout ( Noeud<E> r, E o){...}

**private** **void** suppression (Noeud<E> racine, E o){...}

**private** Noeud chercher(Noeud<E> r, E o){...}

**private** **void** supp (Noeud<E> r){...}

};

*4.1 Ajout.*

**void** ajout(E o){

*// o n'est pas ajouté s'il est déjà présent*

racine = ajout(racine, o);

}

Noeud ajout ( Noeud<E> r, E o){

**if**(r==**null**)

**return** **new** Noeud<E> (o, **null**, **null**);

**else**

**if** (r.element.compareTo(o)<0) r.droit = ajout(r.droit, o);

**else** **if** (r.element.compareTo(o)>0) r.gauche = ajout(r.gauche, o);

**return** r;

}

*4.2 Suppression.*

**void** suppression(E o){

racine = suppression(racine, o);

}

Noeud suppression(Noeud<E> r, E o){

**if** (r==**null**) **return** r;*// l’objet o n’est pas trouvé*

**else**{

**if** (r.element.compareTo(o)==0) **return** supp(r);

**else** **if**(r.element.compareTo(o)>0) r.gauche = suppression(r.gauche, a);

**else** r.droite = suppression(r.droite, a);

**return** r;

}

}

La fonction *supp* est définie comme suit : Il faut distinguer 3 cas:

1. Le champ *gauche* est *null* : *r* devient égal au champ *droit.*
2. Le champ *droit* est *null* : *r* devient égal au champ *gauche*
3. Les champs *gauche* et *droit* sont différents de *null* : on remplace la racine de l'arbre binaire par le plus grand élément du sous-arbre gauche.

Noeud<E> supp (Noeud<E> r){

**if**(r.gauche==**null**) **return** r = r.droite;

**else**

**if**(r.droite==**null**) **return** r = r.gauche;

**else**{

Noeud<E> r1 = r.gauche;

Noeud<E> pere = r;

**while**(r1.droite!=**null**) {

pere = r1;

r1 = r1.droite;

}

r.element = r1.element;

**if**(pere == r) pere.gauche = r1.gauche;

**else** pere.droite = r1.gauche;

**return** r;

}

}

*4.3 Recherche.*

La méthode de recherche associative dans un arbre binaire ordonné, si l'élément cherché n'est pas à la racine, effectue la recherche que dans un des sous-arbres:

Noeud<E> chercher(Noeud<E> r, Comparable o){

**if** (r==**null**) **return** **null**;

**else** **if** (r.element.compareTo(o)==0) **return** r;

**else**

**if**(r.element.compareTo(o)>0) **return** chercher(r.gauche, e);

**else** **return** chercher(r.droite, e);

}

*4.4 Clonage.*

Object clone() **throws** CloneNotSupportedException{

ArbreBinaireOrdonne abo = **new** ArbreBinaireOrdonne<E> ();

**if**(racine != **null**) abo.racine = (Noeud<E>) (racine.clone());

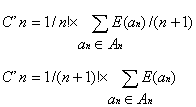
**return** abo;

}

5 Evaluation.

Nous nous proposons de calculer *C'n* coût moyen de la recherche d'un Noeud non présent dans un arbre binaire ordonné de *n* éléments. Cette moyenne nous la calculons sur l'ensemble des *n!* arbres binaires obtenus à partir des ***n!*** permutations de <b*n attributs element, ces permutations étant équiprobables.</b*

*Soit E(an) le cheminement externe d'un arbre binaire de n éléments, et An l'ensemble des arbres binaires ordonnés de n éléments. Le coût C'n peut alors s'écrire de la façon suivante :*

**

*Or pour obtenir un arbre binaire an de n éléments, une feuille (nœud étendu) est remplacée par un noeud interne, et il y a création de 2 nœuds étendus Nous pouvons donc écrire:*

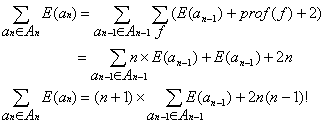
*http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/CoursJava/arbresBinaires_fichiers/image006.gif*

*De plus pour un arbre binaire an :*

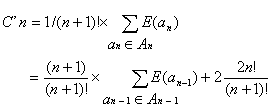
*http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/CoursJava/arbresBinaires_fichiers/image008.gif*

*et le nombre de feuilles (nœud étendu) est égal à n+1*

*Nous pouvons alors écrire :*

**

*Nous obtenons enfin la récurrence cherchée sur C'n:*

**

*http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/CoursJava/arbresBinaires_fichiers/image014.gif*

*La solution de cette récurrence est :*

*http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/CoursJava/arbresBinaires_fichiers/image016.gif*

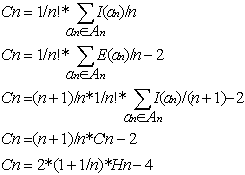
*Le calcul de Cn coût moyen de la recherche d'un Noeud présent dans l’arbre binaire se déduit facilement. En effet :*

*http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/CoursJava/arbresBinaires_fichiers/image018.gif*

*Or :*

*http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/CoursJava/arbresBinaires_fichiers/image020.gif*

*Donc :*

**

*Or pour n suffisamment grand nous avons :*

*http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/CoursJava/arbresBinaires_fichiers/image024.gif*

*Donc :*

*http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/CoursJava/arbresBinaires_fichiers/image026.gif*

***Rééquilibrage des arbres binaires ordonnés.***

|  |
| --- |
| **Sommaire**   1. [Définition](http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/CoursJava/arbresAVL.html#definition)    1. [Evaluation préliminaire](http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/CoursJava/arbresAVL.html#evaluationP) 2. [Algorithmes de rééquilibrage](http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/CoursJava/arbresAVL.html#algoR)    1. [Ajout](http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/CoursJava/arbresAVL.html#ajout)    2. [Suppression](http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/CoursJava/arbresAVL.html#suppression) 3. [Evaluation](http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/CoursJava/arbresAVL.html#evaluation) |

L'efficacité des algorithmes précédents dépend considérablement de la forme de l'arbre sur lequel on travaille : la situation la pire est celle d'un arbre binaire ordonné dont chaque noeud n'a qu'un seul successeur droit ou gauche : l'arbre se comporte alors comme une liste et les algorithmes précédents ont une complexité temporelle de l'ordre de *n*, nombre de noeuds de l'arbre. Les algorithmes précédents auront une complexité de l'ordre de *log2(n)* si l'arbre est dit équilibré, c'est le meilleur des cas.

En moyenne en revanche, nous venons de démontrer dans le paragraphe précédent que les algorithmes de recherche, d'ajout et de suppression ont un coût moyen de l'ordre de *oge(n)*. Ce qui est intéressant c'est de calculer le rapport coût moyen sur coût optimum. Nous avons :

*2\*loge(n)/log2(n) = 2\*loge(2) = 1,386*

Il faudra faire très attention à ce nombre avant de se lancer dans des opérations de rééquilibrage ; il faut que le coût supplémentaire entraîné par le rééquilibrage ne soit pas supérieur au bénéfice escompté par le rééquilibrage, sauf éventuellement si le nombre de recherches est de beaucoup supérieur au nombre des ajouts ou des suppressions qui engendrent des rééquilibrages.

Cependant, dans de nombreuses applications, la séquence des insertions n'est pas aléatoire : il faut alors se poser le problème de faire en sorte que l'arbre ne dégénère pas en une liste, de façon à conserver aux algorithmes toute leur rapidité.

1 Définition.

Un **arbre binaire équilibré**, ou **arbre AVL** (du nom des auteurs de la méthode : Adelson, Velskij, et Landis ) est un arbre binaire tel que les hauteurs des deux sous-arbres de tout noeud de l'arbre diffèrent de 1 au plus :

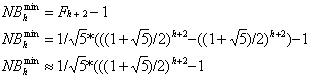
    |*h(Ag) - h(Ad)*|*≤1*

Ceci entraîne évidemment la propriété que tout sous-arbre d'un arbre binaire équilibré au sens AVL est un arbre équilibré au sens AVL.

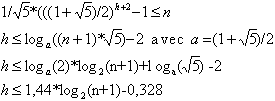
*1.1 Evaluation préliminaire.*

Avant d'étudier les algorithmes qui permettent de maintenir un arbre binaire équilibré au sens AVL, il est nécessaire de s'assurer que le gain sera intéressant.   
La question est de savoir quelle est la hauteur maximale que peut avoir un arbre binaire équilibré au sens AVL. Pour cela nous allons nous occuper de la question inverse, à savoir quel est le nombre minimal de noeuds que peut avoir un arbre binaire équilibré au sens AVL de hauteur *h*.   
Nous avons :  
  http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/CoursJava/arbresAVL_fichiers/image002.gif

Cette récurrence a pour solution le nombre de Fibonacci au rang *h+2* moins 1. Soit :



Donc réciproquement la hauteur maximale *h* d'un arbre AVL contenant *n* éléments est telle que :



La complexité en pire cas des arbres binaires éqilibrés au sens AVL est donc http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/CoursJava/arbresAVL_fichiers/image008.gif.

2 Algorithmes de rééquilibrage.

A chaque noeud de l'arbre, nous ajoutons une information indiquant quel est le sous-arbre le plus haut :

* **0**          indique que les deux sous-arbres du noeud ont même hauteur.
* **-1**indique que le sous-arbre gauche est plus haut que le sous-arbre droit.
* **1**indique que le sous-arbre droit est plus haut que le sous-arbre gauche.

**class** NoeudAVL{

**private** Comparable element;

**private** NoeudAVL gauche, droit;

**private int** equilibre;

**public** NoeudAVL(){

gauche = **null**;

droit = **null**

equilibre = 0;

}

**public** NoeudAVL(Comparable e, NoeudAVL g, NoeudAVL d){

element = e;

gauche = g;

droit = d;

equilibre = 0;

}

}

La classe *ArbreAVL* est définie comme suit :

**class** ArbreAVL{

**private** NoeudAVL racine;

**private** **boolean** equilibreD ( NoeudAVL r, NoeudAVL p, **boolean** g){

...

}

**private** **boolean** equilibreG ( NoeudAVL r, NoeudAVL p, **boolean** g){

...

}

**public** **void** ajout(Comparable o){ ... }

}

Supposons que le noeud *A* était précédemment équilibré :

Lors d'une insertion dans le sous-arbre gauche deux cas se présentent :

1. La hauteur du sous-arbre gauche n'augmente pas : l'arbre était équilibré par hypothèse, donc il le reste.
2. La hauteur du sous-arbre gauche augmente. Trois cas peuvent se présenter :
   * La hauteur du sous-arbre gauche était égale à la hauteur du sous-arbre droit moins un : l'arbre est alors encore mieux équilibré qu'avant l'insertion.
   * La hauteur du sous-arbre gauche était égale à la hauteur du sous-arbre droit, l'arbre reste équilibré.
   * La hauteur du sous-arbre gauche était égale à la hauteur du sous arbre droit plus 1 ; après l'insertion, l'arbre devient déséquilibré.

Les deux cas où l'arbre reste équilibré bien que la hauteur du sous-arbre gauche augmente sont les suivants :

|  |
| --- |
| http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/CoursJava/arbresAVL_fichiers/image009.gif |
| http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/CoursJava/arbresAVL_fichiers/image010.gif |

Dans le premier cas la hauteur de l'arbre n'augmente pas, alors que dans le second cas elle augmente.

 Etudions les différentes possibilité qui se présentent lorsque l'arbre est déséquilibré. Supposons qu'avant l'insertion la hauteur du sous-arbre droit soit *h-1*, la hauteur du sous-arbre gauche avant l'insertion était égale à *h*, elle a augmenté lors de l'insertion, et est donc maintenant de *h+1*. Soit *B* le sous-arbre gauche de *A*. Deux possibilités se présentent :

**B penche gauche.**

|  |
| --- |
| http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/CoursJava/arbresAVL_fichiers/image011.gif |

Devient après ré-équilibrage :

|  |
| --- |
| http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/CoursJava/arbresAVL_fichiers/image012.gif |

**2) B penche droite.**

Deux possibilités :

|  |
| --- |
| http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/CoursJava/arbresAVL_fichiers/image013.gif |

|  |
| --- |
| http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/CoursJava/arbresAVL_fichiers/image014.gif |

Ou un au moins des deux arbres*Cg* ou *Cd* a une hauteur**<="" b="">*h+1*, l'autre pouvant avoir une hauteur *h*, et c'est celui sur lequel se produit le déséquilibre.  
Cet arbre devient après ré-équilibrage :**

**<="" b="" style="color: rgb(0, 0, 0); font-family: Papyrus; font-size: medium; font-style: normal; font-variant-ligatures: normal; font-variant-caps: normal; letter-spacing: normal; orphans: 2; text-align: justify; text-indent: 0px; text-transform: none; white-space: normal; widows: 2; word-spacing: 0px; -webkit-text-stroke-width: 0px; text-decoration-style: initial; text-decoration-color: initial;">**

|  |
| --- |
| http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/CoursJava/arbresAVL_fichiers/image015.gif |

**Nous appellerons rotation chacune de ces opérations. Il est à remarquer que la deuxième rotation peut être obtenue en répétant deux fois la première rotation.**

**La méthode de rééquilibrage suivante est appelée après un ajout dans le sous-arbre droit qui a provoqué une augmentation de la hauteur du sous arbre droit :**

**boolean equilibreD ( NoeudAVL r, NoeudAVL p, boolean g){**

***// r est le fils gauche de p si g vaut true, r est le fils droit de p si g vaut false***

***// retourne true si après équilibrage l'arbre a grandi***

**NoeudAVL r1, r2;**

**switch(r.equilibre){**

**case -1 : r.equilibre = 0; return false;**

**case  0 : r.equilibre = 1; return true;**

**case  1 :**

**default : r1 = r.droit;**

**if(r1.equilibre == 1){**

**r.droit = r1.gauche; r1.gauche = r;**

**r.equilibre = 0;**

**r = r1 ;**

**}else{**

**r2 = r1.gauche; r1.gauche = r2.droit;**

**r2.droit = r1;**

**r.droit = r2.gauche; r2.gauche = r;**

**if(r2.equilibre == 1) r.equilibre = -1;**

**else r.equilibre = 0;**

**if(r2.equilibre == -1) r1.equilibre = 1;**

**else r1.equilibre = 0;**

**r = r2;**

**}**

***// refaire le chaînage avec le père***

**if(p==null) racine = r;**

**else if( g ) p.gauche = r ;**

**else p.droit = r ;**

**r.equilibre = 0;**

**return false;**

**}**

**}**

**La méthode de rééquilibrage suivante est appelée après un ajout dans le sous-arbre gauche qui a provoqué une augmentation de la hauteur du sous arbre gauche :**

**boolean equilibreG (NoeudAVL r, NoeudAVL p, boolean g){**

***// r est le fils gauche de p si g vaut true, r est le fils droit de p si g vaut false***

***// retourne true si après équilibrage l'arbre a grandi***

**NoeudAVL r1, r2;**

**switch (r.equilibre){**

**case 1 : r.equilibre=0; return false;**

**case 0 : r.equilibre = -1; return true;**

**case -1 :**

**default : r1 = r.gauche;**

**if(r1.equilibre==-1){**

**r.gauche = r1.droit; r1.droi t= r;**

**r.equilibre = 0; r = r1;**

**}else{**

**r2 = r1.droit; r1.droit = r2.gauche; r2.gauche=r1;**

**r.gauche=r2.droit; r2.droit = r;**

**if(r2.equilibre == -1) r.equilibre = 1;**

**else r.equilibre = 0;**

**if(r2.equilibre == 1) r1.equilibre =-1;**

**else r1.equilibre = 0;**

**r=r2;**

**}**

***// refaire le chaînage avec le père***

**if (p == null) racine = r;**

**else if( g ) p.gauche = r ;**

**else p.droit = r ;**

**r.equilibre = 0;**

**return false;**

**}**

**}**

***2.1 Ajout.***

**La fonction d'ajout est écrite alors de la façon suivante :**

**void ajouter ( Comparable x){**

**ajoutAVL( racine, null, true, x);**

**}**

**boolean ajoutAVL(NoeudAVL r, NoeudAVL p, boolean g, Comparable e){**

**if(r == null) {**

**r = new NoeudAVL(e, null, null);**

**if (p == null) racine = r**

**else if(g) p.gauche = r;**

**else p.droit = r;**

**return true;**

**}else{**

**int a = e.compareTo(r.element);**

**if (a==0) return false;*// a déjà présent dans l'arbre***

**if (a<0)**

**if(ajoutAVL(r.gauche, r, true, e)) return equilibreG(r, p, g);**

**else return false;**

**else**

**if(ajoutAVL(r.droit, r, false, e)) return equilibreD(r, p, g);**

**else return false;**

**}**

**}**

***2.2 Suppression.***

**Une suppression peut entraîner plusieurs rotations pour le re-équilibrage de l'arbre :**

|  |  |
| --- | --- |
| l'arbre avant suppression de 12. | l'arbre après suppression de 12. |
| L'arbre avant suppression de 12. | l'arbre après suppression de 12. |

**La suppression de 12 : entraîne les 2 rotations**

|  |  |
| --- | --- |
| Résultat après la première rotation | Et enfin on obtient |

**Une rotation peut entraîner une diminution de la hauteur, les fonction d’équilibrage sont donc modifiées :**

**boolean EquilibreDS ( NoeudAVL r, NoeudAVL p, boolean g){**

***// retourne true si la hauteur de l’arbre a diminué***

***// false sinon***

**NoeudAVL r1, r2 ;**

**switch (r.equilibre){**

**case -1 : r.equilibre = 0; return true; a**

**case 0 : r.equilibre = 1; return false;**

**case 1 :**

**default : boolean h; r1 = r.droit;**

**if(r1.equilibre==-1) {**

**r2 = r1.gauche; r1.gauche=r2.droit; r2.droit=r1;**

**r.droit=r2.gauche; r2.gauche = r;**

**if(r2.equilibre==1) r.equilibre = -1;**

**else r.equilibre = 0;p;**

**if(r2.equilibre==-1) r1.equilibre=1;**

**else r1.equilibre = 0;**

**r=r2; h = true; r.equilibre = 0;**

**}else{**

**r.droit=r1.gauche; r1.gauche=r;**

**if(r1.equilibre==0){**

**h = false;**

**r.equilibre=1; r1.equilibre=-1;**

**}else{**

***// entraîne une diminution de***

***// la hauteur de l’arbre***

**h = true;**

**r.equilibre=0; r1.equilibre=0;**

**}**

**r=r1;**

**}**

**if(p==null) racine = r;**

**else if( g ) p.gauche = r;**

**else p.droit = r;**

**return h;**

**}**

**}**

**boolean EquilibreGS ( NoeudAVL r, NoeudAVL p, boolean g){**

***// retourne true si la hauteur de l’arbre a diminué***

***// false sinon***

**NoeudAVL r1, r2 ;**

**switch (r.equilibre){**

**case 1 : r.equilibre = 0; return true;**

**case 0 : r.equilibre = -1; return false;**

**default : boolean h;**

**r1 = r.gauche;**

**if(r1.equilibre==1){**

**r2 = r1.droit; r1.droit=r2.gauche;**

**r2.gauche=r1; r.gauche=r2.droit;**

**r2.droit = r;**

**if(r2.equilibre ==-1) r.equilibre = 1;**

**else r.equilibre =0;**

**if(r2.equilibre == 1) r1.equilibre=-1;**

**else r1.equilibre=0;**

**r=r2;**

**r.equilibre = 0; h=true;**

**}else{**

**r.gauche=r1.droit;**

**r1.droit=r;**

**if(r1.equilibre==0){**

**h=false;**

**r.equilibre = -1; r1.equilibre = 1;**

**}else{**

***// entraîne une diminution de***

***// la hauteur de l’arbre***

**h=true;**

**r.equilibre = 0; r1.equilibre = 0;**

**}**

**r=r1;**

**if(p==null) racine = r;**

**else if( g ) p.gauche = r;**

**else p.droit = r;**

**return h;**

**}**

**}**

**La suppression est alors :**

**public void supprimer( Comparable x){**

**suppAVL(x, racine, null, true);**

**}**

**boolean suppAVL(Comparable e,  NoeudAVL r, NoeudAVL p, boolean g){**

***// retourne true si la hauteur de l’arbre a diminué***

**if (r == null)  return false;**

**else**

**if(r.element.compareTo(e)==0) return SuppAVL(r, p, g);**

**else if(e.compareTo(r.element)<0){**

**if(suppAVL(e, r.gauche, r, true)) return EquilibreDS(r, p, g);**

**else return false;**

**}else{**

**if(suppAVL(e, r.droit,r, false)) return EquilibreGS(r, p, g);**

**else return false;**

**}**

**}**

**boolean SuppAVL( NoeudAVL r, NoeudAVL p, boolean g){**

***// suppression effective du noeud r***

***// retourne true si la hauteur de l’arbre a diminué***

**NoeudAVL a, b;**

**a = r;**

**if(r.gauche==null) {**

**if( p==null) racine = r.droit;**

**else if( g ) p.gauche = r.droit;**

**else p.droit = r.droit;**

**return true;**

**}else**

**if (r.droit==null) {**

**if( p==null) racine = r.gauche;**

**else if( g ) p.gauche = r.gauche;**

**else p.droit = r.gauche;**

**return true;**

**}else{*// les 2 sous arbres sont non vides***

**b = r.gauche;**

**while(b.droit!=null) b = b.droit;**

**r.element = b.element;**

**if( suppAVL(b.element, r.gauche, r, true)) return EquilibreDS(r, p, g);**

**else return false;**

**}**

**}**

**3 Evaluation.**

**On a vu que la hauteur d'un arbre binaire équilibré au sens AVL est inférieure à *1,44\*log(n),* de plus lors d'un ajout, il y a au maximum une rotation effectuée, donc l'ajout d'un élément dans un arbre AVL est en pire cas en *(log(n))*. Une suppression en revanche peut entraîner jusqu'à *og(n)* rotations, bien que les résultats expérimentaux montrent qu'en moyenne il n'y a qu'une rotation toutes les 5 suppressions. La suppression est donc en pire cas en  *(log(n))*.**

**Les évaluations en moyenne des algorithmes sur les arbres binaires équilibrés au sens AVL sont des problèmes qui ne sont pas encore complètement résolus.**

***Arbres binaires équilibrés : arbre Rouge-Noir.***

|  |
| --- |
| **Sommaire**   1. [Propriété](http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/CoursJava/arbresRougeNoir.html#propriete) 2. [Programmation](http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/CoursJava/arbresRougeNoir.html#programmation)    1. [Ajout](http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/CoursJava/arbresRougeNoir.html#ajout)    2. [Suppression](http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/CoursJava/arbresRougeNoir.html#suppression) |

Les arbres rouge-noir sont des arbres binaires ordonnés (étendus) pour lesquels on a les propriétés suivantes :

* chaque noeud est soit noir, soit rouge.
* chaque feuille (nœud sentinelle) est noire.
* la racine est toujours noire.
* si un noeud est rouge, alors ses deux enfants seront noirs.
* pour tout noeud de l’arbre, les chemins de ce nœud vers les feuilles (nœud sentinelle) qui en dépendent ont le même nombre de noeuds noirs.

1 Propriété.

La hauteur noire d’un nœud *n* notée *hn(n)* est le nombre de nœuds noirs dans un chemin du nœud à une de ses feuilles (le nœud *n* n’est pas compris).  
La hauteur noire d’un arbre rouge-noir est la hauteur noire de sa racine.  
Un arbre rouge-noir contenant *n* nœuds internes a une hauteur inférieure ou égale à *2×log2(n+1)*.

Montrons d’abord par induction sur la hauteur noire d’un nœud, que le nombre de nœuds internes du sous-arbre enraciné en *x* est au moins égal à *2hn(x) - 1*.

* Si la hauteur noire de *x* est 0, alors c’est le nœud sentinelle, et le sous arbre enraciné en n contient *2hn(x) - 1 = 0* nœuds internes.
* Si la hauteur noire de *x* est >0, alors chacun de ses fils a une hauteur noire égale soit à *hn(x)* s’il est rouge, soit à *hn(x)-1* s’il est noir. Donc, en appliquant l’hypothèse d’induction aux deux sous arbres de *x*, le sous arbre enraciné en *x* contient au moins *2×(2hn(x)-1-1)+2 = 2hn(x)* nœuds internes.

Soit *h* la hauteur d’un arbre rouge-noir, la moitié au moins des nœuds vers une feuille doit être noirs. Donc la hauteur noire d’un arbre rouge-noir est au moins *h/2*. Nous pouvons donc écrire :

http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/CoursJava/arbresRougeNoir_fichiers/image019.gif  
    http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/CoursJava/arbresRougeNoir_fichiers/image020.gif  
    http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/CoursJava/arbresRougeNoir_fichiers/image021.gif

2 Programmation.

Pour représenter un arbre binaire rouge-noir nous utilisonsune classe *Noeud* qui permet de représenter la couleur de l’arbre, et un attribut permettant de référencer son *Noeud* père. Pour les besoins de l’algorithme de suppression, chaque feuille au lieu d’avoir une référence *null* vers son fils gauche et son fils droit, aura une référence vers un nœud statique sentinelle de couleur noire.

**static** Noeud sentinelle;

**static**{

ArbreRougeNoir.sentinelle = **new** Noeud(**null**, Color.black, **null**, **null**);

}

**class** Noeud{

Color couleur;

Object info;

Noeud gauche, droit, parent;

Noeud(Object o){

couleur = Color.black;

info = o;

gauche = droit = parent = **null**;

}

Noeud(Object o, Color c, Noeud g, Noeud d, Noeud p){

couleur = c;

info = o;

gauche = g;

droit = d;

parent = p;

}

**boolean** isSentinelle(){

**return** **this** == ArbreRougeNoir.sentinelle;

}

*2.1 Ajout.*

L’ajout d’un nœud se fait de façon traditionnelle :

Noeud noeudAjoute;

**public** **void** ajout( Comparable o){

racine = ajout(o, racine, **null**);

}

**private** Noeud ajout( Comparable o, Noeud r, Noeud p){

*// p est le parent de r*

**if** (r.isSentinelle()) r = noeudAjoute = **new** Noeud(o, Color.red, r, r, p);

**else**

**if**(o.compareTo(r.info)<0) r.gauche = ajout(o, r.gauche, r);

**else** r.droit = ajout(o, r.droit, r);

**return** r;

}

Il suffit après l’insertion de réorganiser l’arbre, en partant du nouveau nœud ajouté, s’il ne possède plus une des propriétés précédentes :

Noeud noeudAjoute;

**public** **void** ajout( Comparable o){

racine = ajout(o, racine, **null**);

reOrganiser(noeudAjoute) ;

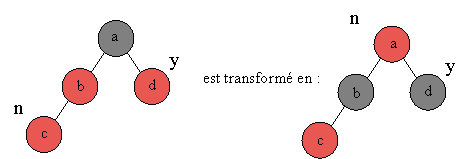
}

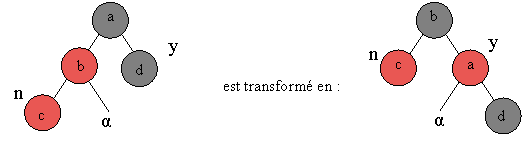
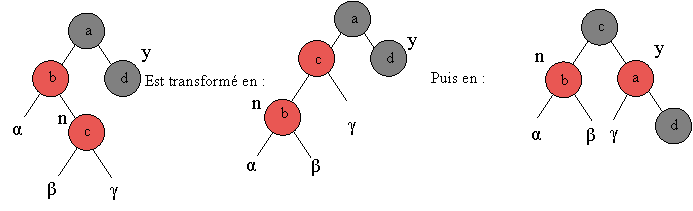
Un nœud ajouté est rouge. Si le nœud père est noir, l’arbre est toujours un arbre rouge-noir. En revanche si le nœud père du nœud ajouté est lui-même un nœud rouge, alors l’arbre a perdu sa propriété d’être un arbre rouge-noir.

Supposons que le parent du nœud *n* qui vient de briser la propriété de l’arbre d’être rouge-noir soit rouge et soit lui-même le descendant gauche de son père. Soit *y* le descendant droit du père du père de *n*. Dans ce cas il suffit de changer les couleurs du père de n de y et de leur père. Leur père devient rouge, et donc il faut réorganiser l’arbre à partir de ce nœud.

Le fait que *n*soit un fils droit ou un fils gauche de son père est indifférent.

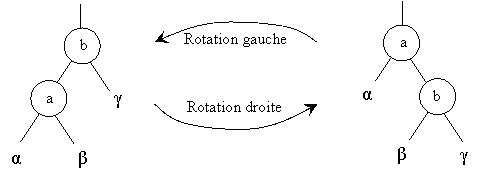
Nous pouvons représenter cette configuration de la façon suivante :

En revanche si la couleur de *y* est noir, deux possibilités se présentent :

Si le père du nœud *n*est un fils droit de son propre père, les transformations sont complètement symétriques.

Nous voyons apparaître deux transformations d’arbres qui sont des rotations à gauche ou à droite :



**private** **void** rotationGauche( Noeud n){

Noeud y = n.droit;

n.droit = y.gauche;

**if**(!y.gauche.isSentinelle())y.gauche.parent = n;*// on ne change pas le parent de la sentinelle*

y.parent = n.parent;

**if** (n.parent==**null**) racine = y;<

**else**

**if**( n.parent.gauche == n ) n.parent.gauche = y;

**else** n.parent.droit = y;

y.gauche = n;

n.parent = y;

}

**private** **void** rotationDroite( Noeud n){

Noeud y = n.gauche;

n.gauche = y.droit;

**if**(!y.droit.isSentinelle())y.droit.parent = n; *// on ne change pas le parent de la sentinelle*

y.parent = n.parent;

**if** (n.parent==**null**) racine = y;

**else**

**if**( n.parent.droit == n) n.parent.droit = y;

**else** n.parent.gauche = y;

y.droit = n;

n.parent = y;

}

La méthode de réorganisation de l’arbre est donc :

**private** **void** reOrganiser( Noeud n){

*// re organisation de l'arbre, en remontant vers la racine*

**while**(n != racine && n.parent.couleur == Color.red ){

**if**(n.parent == n.parent.parent.gauche){

Noeud y = n.parent.parent.droit;

**if**( y.couleur == Color.red ){

n.parent.couleur = Color.black;

y.couleur = Color.black;

n.parent.parent.couleur = Color.red;

n = n.parent.parent;

}**else**{

**if**(n == n.parent.droit){

n = n.parent;

rotationGauche(n);

}

n.parent.couleur = Color.black;

n.parent.parent.couleur = Color.red;

rotationDroite(n.parent.parent);

}

}**else**{

Noeud y = n.parent.parent.gauche;

**if**( y.couleur == Color.red ){

n.parent.couleur = Color.black;

y.couleur = Color.black;

n.parent.parent.couleur = Color.red;

n = n.parent.parent;

}**else**{

**if**(noeudAjoute == n.parent.gauche){

n = n.parent;

rotationDroite(n);

}

n.parent.couleur = Color.black;

n.parent.parent.couleur = Color.red;

rotationGauche(n.parent.parent);

}

}

}

racine.couleur = Color.black; *// la racine est toujours noire*

}

*2.2 Suppression.*

La suppression commence par une recherche classique du nœud à supprimer, puis enchaîne sur la réalisation de la suppression.

**public** **void** supprimer( Comparable o){

supprimer( racine, o );

}

**private** Noeud supprimer( Noeud r, Comparable o){

**if**(r.isSentinelle())**return** r; *// Pas trouvé*

**if**(o.compareTo(r.info)== 0) r = supprimer(r);

**else**

**if**(o.compareTo(r.info)<0) supprimer(r.gauche, o);

**else** supprimer(r.droit, o); **return** r;

}

La méthode suivante réalise la suppression du nœud, en le remplaçant par le plus petit élément de son sous-arbre droit, s’il a deux fils, et en le supprimant effectivement, s’il n’a qu’un seul fils.

**private** Noeud supprimer( Noeud z){

Noeud y, x;

**if**(z.gauche.isSentinelle() || z.droit.isSentinelle()) y = z;

**else** y = arbreSuccesseur(z);

**if**(!y.gauche.isSentinelle()) x = y.gauche;

**else** x = y.droit;

x.parent = y.parent;

**if**(y.parent== **null**) racine = x;

**else**

**if(** y == y.parent.gauche) y.parent.gauche = x;

**else** y.parent.droit = x;

**if**( y != z ) z.info = y.info;

*// si le noeud supprimé est un noeud rouge, il n’y a rien à*

*// faire, l’arbre conserve ses propriétés*

*// en revanche si le nœud supprimé est noir,*

*// il faut reorganiser l’arbre*

**if** ( y.couleur == Color.black) reOrganiserSuppression( x );

**return** y;

}

**private** Noeud arbreSuccesseur(Noeud x){

*// le noeud successseur de x dans l'arbre,*

*// sentinelle si c'est le plus grand*

**if**( !x.droit.isSentinelle()) **return** arbreMinimum(x.droit);

Noeud y = x.parent;

**while**(!y.isSentinelle() && x == y.droit){ x = y; y = x.parent;}

**return** y;

}

**private** Noeud arbreMinimum(Noeud x){

**while**( !x.gauche.isSentinelle()) x = x.gauche;

**return** x;

}

**private** **void** reOrganiserSuppression( Noeud n){

*// re organisation de l'arbre, en remontant vers la racine*

**while**(n != racine && n.couleur == Color.black ){

**if**(n == n.parent.gauche){

Noeud y = n.parent.droit;

**if**(y.couleur == Color.red ){

y.couleur = Color.black;

n.parent.couleur = Color.red;

rotationGauche(n.parent);

y = n.parent.droit;

}

**if**(y.gauche.couleur == Color.black && y.droit.couleur == Color.black){

y.couleur = Color.red;

n = n.parent;

}**else**{

**if**( y.droit.couleur == Color.black){

y.gauche.couleur = Color.black;

y.couleur = Color.red;

rotationDroite(y);

y = n.parent.droit;

}

y.couleur = n.parent.couleur;

n.parent.couleur = Color.black;

y.droit.couleur = Color.black;

rotationGauche(n.parent);

**break;**

}

}**else**{

Noeud y = n.parent.gauche;

**if**( y.couleur == Color.red ){

y.couleur = Color.black;

n.parent.couleur = Color.red;

rotationDroite(n.parent);

y = n.parent.gauche;

}

**if**(y.droit.couleur == Color.black && y.gauche.couleur == Color.black){

y.couleur = Color.red;

n = n.parent;

}**else**{

**if**( y.gauche.couleur == Color.black){

y.droit.couleur = Color.black;

y.couleur = Color.red;

rotationGauche(y);

y = n.parent.gauche;

}

y.couleur = n.parent.couleur;

n.parent.couleur = Color.black;

y.gauche.couleur = Color.black;

rotationDroite(n.parent);

**break**;

}

}

}

n.couleur = Color.black;

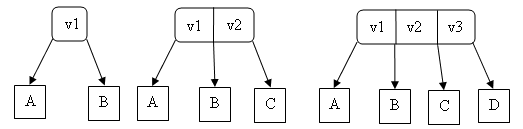
}

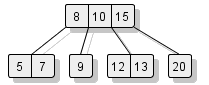
***Arbres binaires équilibrés : arbre 2-3-4.***

|  |
| --- |
| **Sommaire**   1. [Définition](http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/CoursJava/arbres234.html#definition) 2. [Propriété](http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/CoursJava/arbres234.html#propriete) 3. [Ajout](http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/CoursJava/arbres234.html#ajout) 4. [Suppression](http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/CoursJava/arbres234.html#suppression) |

1 Définition.

Les arbres 2-3-4 sont des arbres tels que

1. tous les nœuds ont 2, 3 ou 4 fils.  
     
   Les sous-arbres *A*, *B*, *C* et *D* ont les propriétés suivantes :
   * tous les nœuds du sous arbre *A* ont une valeur inférieure à *v1*.
   * tous les nœuds du sous arbre *B* ont une valeur supérieure ou égale à *v1* et inférieure à *v2*, pour les nœuds 3 ou 4.
   * tous les nœuds du sous arbre *C*ont une valeur supérieure ou égale à *v2* et inférieure à *v3*, pour les nœuds 4.
   * tous les nœuds du sous arbre *D* ont une valeur supérieure ou égale à *v3*.
2. tous les nœuds feuilles sont à la même profondeur.

Exemple d'arbre 2-3-4 obtenu par l'ajout des nombres de la séquence 10, 5, 8, 7, 15, 20, 12, 9, 13 : 

2 Propriété.

La hauteur *h* de l'arbre 2-3-4 qui contient *n* valeurs est telle que :  *log4(n+1) ≤ h ≤ log2(n+1)*.

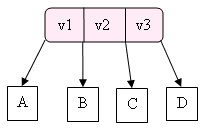
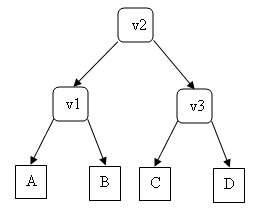
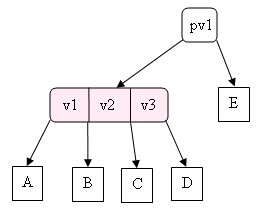
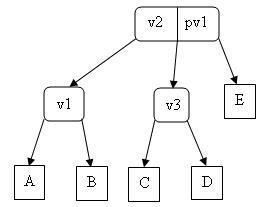
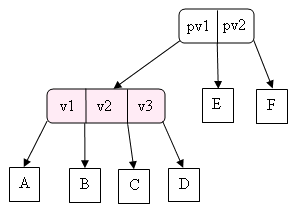
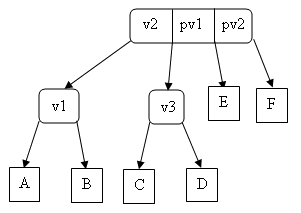
En effet :

* si tous les nœuds ont 2 fils, on a la hauteur maximum :  *log2(n+1)*
* si tous les nœuds ont 4 fils, on a la hauteur minimum :  *log4(n+1)*

3 Ajout.

L'insertion *top-down* se comporte de la façon suivante :

Quand on descend dans l'arbre vers un nœud 4, celui-ci est éclaté avant d'y parvenir. Ainsi, ont est certain de ne jamais ajouter une nouvelle valeur dans un nœud 4.

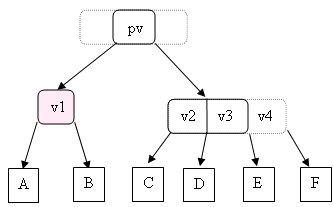
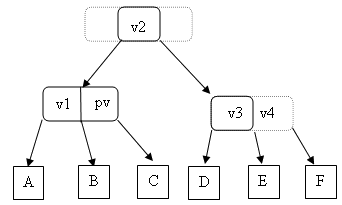
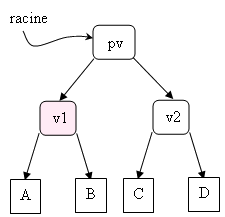
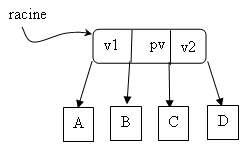
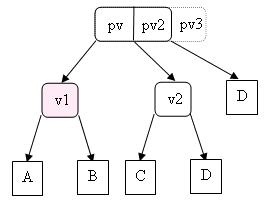
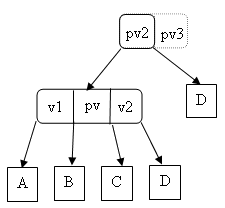
1. Si le nœud 4 est la racine il est éclaté de la façon suivante :  
    devient  C'est le cas où la hauteur de l'arbre augmente de 1.
2. Si le nœud 4 n'est pas la racine il est le fils d'un nœud 2 ou d'un nœud 3 (il ne peut pas être le fils d'un nœud 4 car celui-ci aura été éclaté lors de la descente) :
   * il est le fils d'un nœud 2 :   
      devient 
   * il est le fils d'un nœud 3 :   
      devient 

Dans ces deux cas la hauteur de l'arbre n'est pas augmentée. Les transformations sont identiques si le nœud 4 à éclater est en position *E* ou *F*.

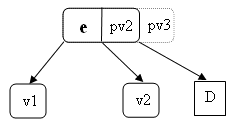
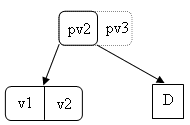
L'insertion conserve la propriété «toutes les feuilles sont à la même profondeur» : le seul moment où la hauteur de l'arbre augment, est lors de l"éclatement de la racine. Cet éclatement augmente les profondeurs de toutes les feuilles de 1.

4 Suppression.

Lors de la descente dans l'arbre, vers l'élément à supprimer, on fait les tranformations qu'il faut pour que le nœud où on arrive soit un nœud 3 ou un nœud 4 (sauf pour la racine).

1. J'ai un frère immédiat gauche ou droite qui est un nœud 3 ou un nœud 4 :   
   devient  
   Le principe est le même si c'est un frère à gauche.
2. Sinon, j'ai un frère immédiat gauche ou droite qui est un nœud 2 :
   * Mon père est un nœud 2 : c'est la racine :  
     devient
   * Mon père est un nœud 3 ou 4 :   
     devient

Quand on arrive sur un nœud qui contient l'élément à supprimer :

* le nœud est une feuille, il suffit de supprimer l'élément.
* le nœud n'est pas une feuille :
  + les deux descendants droite et gauche ont 1 élément, on fusionne ces deux descendants et on supprime l'élément.  
    devient
  + un des deux descendants a plus d'un élément, on remplace l'élément à supprimer par le succcesseur immédiat (ou le prédécesseur immédiat) de l'élément à supprimer.

***TreeSet***

Les données stockées dans un *TreeSet* sont représentées dans un [*TreeMap*,](http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/Conteneurs/TreeMap.html) ou suivant le constructeur plus généralement dans un *SortedMap*.  
Un *TreeMap* est un [arbre binaire](http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/CoursJava/arbresBinaires.html) [équilibré](http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/CoursJava/arbresAVL.html) par la technique [rouge-noir](http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/CoursJava/arbresRougeNoir.html).

**public** **class** TreeSet<E> **extends** AbstractSet<E>

**implements** SortedSet<E>, Cloneable, java.io.Serializable {

**private** **transient** SortedMap<E,Object> m; */// L'arbre binaire des données*

**private** **transient** Set<E> keySet; *// La vue clé de cet arbre binaire*

*// valeur "bidon" associée à un objet stocké dans le TreeMap*

**private** **static** **final** Object PRESENT = **new** Object();

...

1 Les constructeurs :

**private** TreeSet(SortedMap<E,Object> m) {

**this**.m = m;

keySet = m.keySet();

}

**public** TreeSet() {

**this**(**new** TreeMap<E,Object>());

}

**public** TreeSet(Comparator<? **super** E> c) {

**this**(**new** TreeMap<E,Object>(c));

}

**public** TreeSet(Collection<? **extends** E> c) {

**this**();

addAll(c);

}

**public** TreeSet(SortedSet<E> s) {

**this**(s.comparator());

addAll(s);

}

2 Les méthodes :

Les méthodes se résument toutes à un appel de la méthode correspondante de la classe *TreeMap*. Exemples :

* la méthode *add* ajoute un couple *(o,PRESENT)*, elle utilise la méthode *put* de *TreeMap*, qui retourne *null* si l'objet était absent du *TreeMap* avant ajout.
* **public** **boolean** add(E o) {
* **return** m.put(o, PRESENT)==**null**;

}

* la méthode *remove* enlève un couple *(o, PRESENT)*, elle utilise la méthode *remove* de *TreeMap*, qui retourne la valeur associée à la clé *o*, si celle-ci était présente dans le *TreeMap*, et *null* sinon.
* **public** **boolean** remove(Object o) {
* **return** m.remove(o)==PRESENT;<

}

* la méthode *clear* vide le *TreeSet.*
* **public** **void** clear() {
* m.clear();

}

* les méthodes *subSet, headSet* et *tailSet* :

|  |  |
| --- | --- |
| **public** SortedSet<E> subSet(E fromElement, E toElement) {  **return new** TreeSet<E>(m.subMap(fromElement, toElement));  } | Retourne l'ensemble des éléments supérieurs ou égaux à *fromElement* et inférieurs strictement à *toElement*. |
| **public** SortedSet<E> headSet( E toElement) {  **return** **new** TreeSet<E>(m.subMap(fromElement, toElement));  } | Retourne l'ensemble des éléments strictement inférieurs à *toElement.* |
| **public** SortedSet<E> tailSet(E fromElement) {  **return** **new** TreeSet<E>(m.tailMap(fromElement));  } | Retourne l'ensemble des éléments supérieurs ou égaux à *fromElement.* |

* les méthodes *first*et*last* :

**public** E first() {

**return** m.firstKey();

}

**public** E last() {

**return** m.lastKey();

}

* la méthode *contains* :
* **public** **boolean** contains(Object o) {
* **return** m.containsKey(o);

}

***TreeMap***

Le conteneur *TreeMap* permet de stocker des couples (clé, valeur), dans une structure d’[arbre binaire](http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/CoursJava/arbresBinaires.htm) [équilibré](http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/CoursJava/arbresAVL.htm) [rouge-noir](http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/CoursJava/arbresRougeNoir.htm). Cette classe garantit que la collection *Map* sera triée selon un ordre croissant, conformément à l'ordre naturel des clés ou à l'aide d'un comparateur fourni au moment de la création de l'objet *TreeMap*.

**public** **class** TreeMap<K,V>**extends** AbstractMap<K,V>

**implements** SortedMap<K,V>, Cloneable, java.io.Serializable {

**private** **transient** Comparator<? **super** K> comparator = **null**;

**private** **transient** Entry<K,V> root = **null**;

1 Les constructeurs :

**public** TreeMap() {

}

**public** TreeMap(Comparator<? super K> c) {

**this**.comparator = c;

}

**public** TreeMap(Map<? **extends** K, ? **extends** V> m) {

putAll(m);

}

Pour construire un *TreeMap* à partir d'un autre *TreeMap*, on peut s'y prendre comme précédemment, mais ceci va occasionner des restructurations de l'arbre au fur et à mesure des ajouts : la méthode *buildFromSorted*est faite pour éviter toutes ces restructurations :

**public** TreeMap(SortedMap<K, ? **extends** V> m) {

comparator = m.comparator();

**try**{

buildFromSorted(m.size(), m.entrySet().iterator(),**null**, **null**);

} **catch** (java.io.IOException cannotHappen) {

} **catch** (ClassNotFoundException cannotHappen) {

}

}

2 Les méthodes :

Les interrogations :

|  |  |
| --- | --- |
| **int** size() | Retourne le nombre de couples (clé, valeur) contenus dans ce *TreeMap*. |
| Comparator<? **super** K> comparator() | Retourne le Comparator utilisé par le *TreeMap*, ou null si le *TreeMap* n'utilise pas de *Comparator*. |
| **boolean** containsKey(Object cle) | Retourne *true* si le *TreeMap* contient la clé *cle*. |
| **boolean** containsValue(Object valeur) | Retourne *true* si le *TreeMap* contient la valeur *valeur*. |
| V get(Object cle) | Retourne la valeur associée à la clé *cle*. |
| K firstKey() | Retourne la première clé du *TreeSet*(la plus petite). Lève une exception *NoSuchElementException* si le TreeMap est vide. |
| K lastKey() | Retourne la dernière clé du *TreeSet*(la plus grande). Lève une exception *NoSuchElementException* si le *TreeMap* est vide. |

Les modifications :

|  |  |
| --- | --- |
| **void** clear() | Supprime tous les couples (clés, valeur) du *TreeMap*. |
| V put (K cle, V valeur) | Range un nouveau couple (clé, valeur) dans le *TreeMap*,  et retourne :   * l'ancienne valeur si la clé était déjà présente. * *null* si la clé n'était pas présente. |
| **void** putAll( Map<? **extends** K, ? **extends** V> map) | Ajoute tous les élément de map au TreeMap. |
| V  remove(Object cle) | Enlève le couple (clé, valeur) associée  à cle, et retourne :   * la valeur si le couple existe dans le *TreeMap*. * *null*si le couple n'existe pas dans le *TreeMap.* |

Les vues sur un *TreeMap* :

|  |  |
| --- | --- |
| Set <Map.Entry<K ,V >>entrySet() | Retourne l'ensemble des couples (clé, valeur) sous forme de *Set*. Permet  d'obtenir un iterateur sur le *TreeMap*. Un modification du *TreeMap* entraîne une modification du *Set* et vice versa.   Les opérations *add* et *addAll* ne sont pas autorisées pour  le *Set.* |
| Set<K> keySet() | Retourne une vue Set du *TreeMap*. Un modification du *TreeMap* entraîne une modification du *Set* et vice versa.   Les opérations *add* et *addAll* ne sont pas autorisées pour  le *Set.* |
| Collection<V> values() | Retourne une vue *Collection*  de ce *TreeMap*. Une modification du TreeMap entraîne une modification de la *Collection*, et vice versa.  Les opérations *add* et *addAll* ne sont pas autorisées pour  la *Collection*. |

Les sous *TreeMap* :

|  |  |
| --- | --- |
| SortedMap<K,V>headMap(K  jusqua) | Retourne un *TreeMap contenant tous les couples (clé, valeur) jusqu'à la clé jusqua exclue.* |
| SortedMap<K,V> subMap(K cleDepuis, K cleJusqua) | Retourne un *TreeMap* contenant tous les couples (clé, valeur) depuis la clé cleDepuis incluse, jusqu'à la clé *cleJusqua* exclue. |
| SortedMap<K,V> tailMap(K cleDepuis) | Retourne un *TreeMap* contenant tous les couples (clé, valeur) depuis la clé *cleDepuis* incluse. |

Pour créer une vue *Collection*sur le *TreeMap*, on crée une classe locale, dérivée de *AbstractCollection*, qui définit *iterator*, *size*, *contains*, *remove*, et *clear*. La *Collection* n'est pas fabriquée, elle est représentée dans le *TreeMap*,et ce qui est retourné est simplement une vue.

**public** Collection<V> values() {

**if** (values == **null**) {

values = **new** AbstractCollection<V>() {

**public** Iterator<V> iterator() {

**return** **new** ValueIterator();

}

**public** **int** size() {

**return** TreeMap.**this**.size();

}

**public** **boolean** contains(Object o) {

**for** (Entry<K,V> e = firstEntry(); e != **null**; e = successor(e))

**if** (valEquals(e.getValue(), o)) **return** **true**;

**return** **false**;

}

**public** **boolean** remove(Object o) {

**for** (Entry<K,V> e = firstEntry(); e != **null**; e = successor(e)) {

**if** (valEquals(e.getValue(), o)) {

deleteEntry(e);

**return** true;

}

}

**return** **false**;

}

**public** **void** clear() {

TreeMap.**this**.clear();

}

};

}

**return** values;

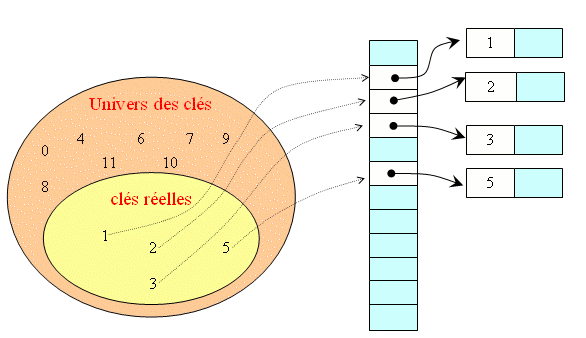
}

***Table de Hachage***

|  |
| --- |
| **Sommaire**   1. [Table à adressage direct](http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/Conteneurs/TableDeHachage.html#tableAD) 2. [Table de hachage](http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/Conteneurs/TableDeHachage.html#tableH)    1. [Résolution des collisions par chaînage](http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/Conteneurs/TableDeHachage.html#resolutionC)    2. [Analyse](http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/Conteneurs/TableDeHachage.html#analyse) 3. [Programmation](http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/Conteneurs/TableDeHachage.html#programmation)    1. [Recherche](http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/Conteneurs/TableDeHachage.html#recherche)    2. [La méthode *put*](http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/Conteneurs/TableDeHachage.html#put)    3. [La méthode *remove*](http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/Conteneurs/TableDeHachage.html#remove) |

1 Table à adressage direct

 Soit *U* l’univers des clés, si sa taille *n* est suffisamment petite, on peut  représenter les clés dans un tableau de *n* éléments.



Les méthodes d'ajout, de recherche et de suppression sont alors extrêmement simples :

Object chercher( Object cle){ **return** t[cle] ;}

**void** ajout( Object cle, Object valeur){

t[cle] = valeur;

}

Object suppression( Object cle){

Object o = t[cle] ;

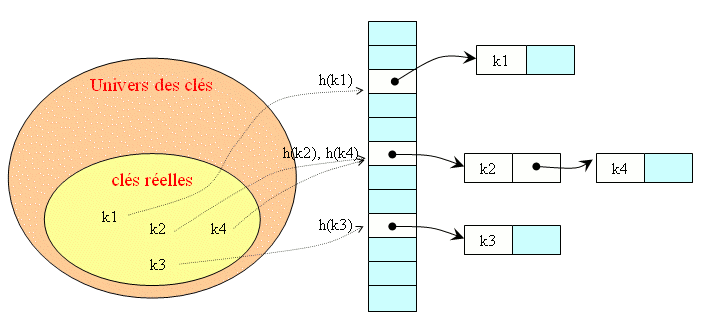
t[cle] = **null**

**return** o;

}

2 Table de Hachage

En général,   l’univers des clés est très grand alors que le nombre de clés présentes dans le conteneur est petit par rapport au nombre de clés possibles. On utilise alors une *fonction de hachage*qui associe à une clé donnée un entier de *0* à *m*. on range alors la clé au rang *h(cle)*dans la table.



Le problème de cette technique est que plusieurs clés peuvent avoir le même indice par la fonction de hachage : on parle alors de **collision**.

*2.1 Résolution des collisions par chaînage.*

Chaque élément du tableau est une référence à une liste chaînée des entrées dont les clés ont même valeur par application de la fonction de hachage.   
On définit alors le **facteur de remplissage** **α** comme étant le rapport de *n* nombre d’éléments présents dans la table hachée sur *m* taille de la table hachée.

*2.2 Analyse de la table hachée avec chaînage*

Dans le pire des cas : toutes les clés se retrouvent dans le même élément du tableau, alors le comportement est le même que pour une liste chaînée.  
Une recherche qui échoue prend un temps de l’ordre de 1+**α**. Il faut parcourir une des *m* listes jusqu’à la fin, or ces listes ont une taille moyenne égale à **α** est donc de l’ordre de 1+**α**.  
Une recherche qui réussit prend un temps de l’ordre de 1+**α**.  
Si la taille de la table est proportionnelle au nombre d’éléments présent dans la table, alors les opérations d’ajout, de recherche ou de suppression se font en temps constant.

3 Programmation

Pour représenter la liste chaînée, nous définissons la classe *Entree*

**class** Entree<K, V> {

**int** hash;

K cle;

V valeur;

Entree<K, V> suivant;

**public** Entree(**int** hash, K cle, V valeur, Entree<K, V> suivant){

**this**.hash = hash;

**this**.cle = cle;

**this**.valeur = valeur;

**this**.suivant = suivant;

}

**protected** Object clone() {

**return** **new** Entree<K,V>(hash, cle, valeur, (Entree<K,V>)(suivant==**null** ? **null** : suivant.clone()));

}

**public** K getKey() {

**return** cle;

}

**public** V getValue() {

**return** valeur;

}

**public** V setValue(V valeur) {

V aValeur = **this**.valeur;

**this**.valeur = valeur;

**return** aValeur;

}

**public** **boolean** equals(Object o) {

*// retourne true si les clés et les valeurs sont égales.*

**if** (!(o **instanceof** Entree)) **return** **false**;

Entree<K,V> e = (Entree)o;

**if**(cle == e.getKey() || (cle!=**null** && cle.equals(e.getKey())))

**if** (valeur == **null**) **return** e.getValue() == **null**

**else** **return** valeur.equals(e.getValue());

**else** **return** **false**;

}

**public** **int** hashCode() {

**return** hash ^ (valeur==**null** ? 0 : valeur.hashCode());

}

**public** String toString() {

**return** cle+"="+valeur;

}

}

La Classe *TableHachee* est alors définie de la façon suivante :

**public** **class** TableHachee<K,V> {

**private** Entree<K,V> table[];

**private** **int** nbEntrees; *// le nombre d’entrées présentes*

**private** **int** seuil; *// le seuil (en nombre d'entrées) à partir duquel*

*// on va augmenter la taille de la table*

**private** **float** facteurDeCharge; *// le facteur de charge qui sert*

*// à déterminer le seuil*

Les constructeurs :

**public** TableHachee(**int** capaciteInitiale, **float** facteurDeCharge) {

**if** (capaciteInitiale < 0)

**throw** **new** IllegalArgumentException( "Capacité initiale Illegale : "+ capaciteInitiale);

**if** (facteurDeCharge <= 0 || Float.isNaN(facteurDeCharge))

**throw** **new** IllegalArgumentException( "Facteur de charge Illegal : "+ facteurDeCharge);

**if** (capaciteInitiale==0)capaciteInitiale = 1;

**this**.facteurDeCharge = facteurDeCharge;

table = (Entree<K,V>[])**new** Entree [capaciteInitiale];

seuil = (**int**)(capaciteInitiale \* facteurDeCharge);

}

**public** TableHachee(**int** capaciteInitiale) {

**this**(capaciteInitiale, 0.75f);

}

**public** TableHachee() {

**this**(16, 0.75f);

}

Quelques méthodes simples

**public** **int** size() {**return** nbEntrees;}

**public** boolean isEmpty() { nbEntrees == 0;}

**public** **int** capacity() {**return** table.length;}

**public** **float** loadFactor() {**return** facteurDeCharge;}

*3.1 Recherche*

Recherche par valeur : dans ce cas il n’y a pas d’autre solution que faire un parcours de toute la table jusqu’à trouver ce qu’on cherche.

**public** **boolean** containsValue(Object valeur) {

Entree<K,V> tab[] = table;

**if** (valeur==**null**) {

**for** (**int** i = tab.length ; i-- > 0 ;)

**for** (Entree<K,V> e = tab[i] ; e != **null** ; e = e.suivant)

**if** (e.valeur==**null**) **return** **true**;

}**else**{

**for** (**int** i = tab.length ; i-- > 0 ;)

**for** (Entree<K,V> e = tab[i] ; e != **null** ; e = e.suivant)

**if** (valeur.equals(e.valeur)) **return** **true**;

}

**return** **false**;

}

Recherche par clé : la méthode de hachage des clés permet d’obtenir l’indice de la liste des entrées ayant même valeur de hachage :  la clé *null* est rangée dans l’élément de rang 0 de la table.

**boolean** containsKey(K cle) {

Entree<K,V> tab[] = table;

**if** (cle != **null**) {

**int** hash = cle.hashCode();

**int** index = (hash & 0x7FFFFFFF) % tab.length;

**for** ( Entree<K,V> e = tab[index]; e != **null**; e = e.suivant)

**if** (e.hash==hash && cle.equals(e.cle)) **return** **true**;

}**else**{

**for** (Entree<K,V> e = tab[0]; e != **null**; e = e.suivant)

**if** (e.cle==**null**)**return** **true**;

}

**return** **false**;

}

**public** V get(K cle) {

Entree<K,V> tab[] = table;

**if** (cle != **null**) {

**int** hash = cle.hashCode();

**int** index = (hash & 0x7FFFFFFF) % tab.length;

**for** ( Entree<K,V> e = tab[index]; e != **null**; e = e.suivant)

**if** ((e.hash == hash) && cle.equals(e.cle))**return** e.valeur;

}**else**{

**for** (Entree<K,V> e = tab[0]; e != **null**; e = e.suivant)

**if** (e.cle==**null**) **return** e.valeur;

}

**return** **null**;

}

*3.2 La méthode put*

La méthode *put* a l’effet suivant :

* Si la clé est présente : la méthode *put* change la valeur associée à la clé quand celle-ci est présente et retourne l’ancienne valeur qui était associée à la clé.
* Si la clé est absente : la méthode *put* ajoute une nouvelle entrée, et retourne *null*.

**public** V put(K cle, V valeur) {

Entree<K,V> tab[] = table;

**int** hash = 0;

**int** index = 0;

**if** (cle != null) {

hash = cle.hashCode();

index = (hash & 0x7FFFFFFF) % tab.length;

**for** (Entree<K,V> e = tab[index]; e != **null** ; e=e.suivant){

**if** ((e.hash == hash) && cle.equals(e.cle)) {

V aValeur = e.valeur;

e.valeur = valeur;

**return** aValeur;

}

}

}**else**{

**for** (Entree<K,V> e = tab[0] ; e != **null**; e = e.suivant) {

**if** (e.cle == **null**) {

V aValeur = e.valeur;

e.valeur = valeur;

**return** aValeur;

}

}

}

*// la clé n’a pas été trouvée dans la table*

**if** (nbEntrees >= seuil) {

*// Rehash la table si le seuil est dépassé*

rehash();

tab = table;

index = (hash & 0x7FFFFFFF) % tab.length;

}

*// Création de la nouvelle entrée*

tab[index] = **new** Entree(hash, cle, valeur, tab[index]);

nbEntrees++;

**return** **null**;

}

La méthode *rehash* agrandit  la table de façon que le nombre d’éléments ne dépasse pas le seuil :

**private** **void** rehash() {

**int** aCapacite = table.length;

Entree<K,V> aTab[] = table;

**int** nCapacite = aCapacite \* 2 + 1;

Entree<K,V> nTab[] = (Entree<K,V>[])**new** Entree[nCapacite];

seuil = (**int**)( nCapacite \* facteurDeCharge);

table = nTab;

**for** (**int** i = aCapacite; i-- > 0 ;) {

**for** (Entree<K,V> a = aTab [i] ; a != **null** ; ) {

Entree<K,V> e = a;

a = a.suivant;

**int** index = (e.hash & 0x7FFFFFFF) % nCapacite;

e.suivant = nTab [index];

nTab [index] = e;

}

}

}

*3.3 méthode remove*

La suppression d’une clé dans la table :

**public** V remove(K cle) {

Entree<K,V> tab[] = table;

**if** (cle != **null**) {

**int** hash = cle.hashCode();

**int** index = (hash & 0x7FFFFFFF) % tab.length;

**for** (Entree<K,V> e = tab[index], prec = **null**;

e != null; prec = e, e = e.suivant) {

**if** ((e.hash == hash) && cle.equals(e.cle)) {

**if** (prec != **null**)prec.suivant = e.suivant;

**else** tab[index] = e.suivant;

nbEntrees--;

V aValeur = e.valeur;

e.valeur = **null**;

**return** aValeur;

}

}

}**else**{

**for** (Entree<K,V> e = tab[0], prec = **null**;

e != null; prev = e, e = e.suivant) {

**if** (e.cle == **null**) {

**if** (prec != **null**) recv.suivant = e.suivant;

**else** tab[0] = e.suivant;

nbEntrees--;

V aValeur = e.valeur;

e.valeur = **null;**

**return** aValeur;

}

}

}

*// la clé n’a pas été trouvée*

**return** **null**;

}

Suppression de toutes les clés dans la table :

**public** **void** clear() {

Entree<K,V> tab[] = table;

**for** (**int** index = tab.length; --index >= 0; )

tab[index] = **null**;

nbEntrees = 0;

}

Clonage d’une table hachée : ni les clés, ni lesvaleurs stockées ne sont clonées :

**public** Object clone() {

**try** {

TableHachee t = (TableHachee)super.clone();

t.table = **new** Entree[table.length];

**for** (**int** i = table.length ; i-- > 0 ; ) {

t.table[i] = (table[i] != **null**)? (Entree)table[i].clone() : **null**;

}

**return** t;

} **catch** (CloneNotSupportedException e) {

*// ça ne devrait pas arriver : la table est cloneable*

**throw** **new** InternalError();

}

}

# *HashMap*

Cette classe dérive de la classe [*AbstractMap*](http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/Conteneurs/AbstractMap.html), et implémente l'interface *Map*, en utilisant une [table hachée](http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/Conteneurs/TableDeHachage.html). Les valeurs ou les clés peuvent être *null*. Il n'y a aucun ordre garanti sur les éléments stockés dans le *HasMap*, et de plus l'ordre des éléments n'est pas forcément le même à deux instants différents.  
Les opérations *get* *et put* sont en temps constant. La capacité est le nombre de listes. Le *load factor*(0.75 par défaut) est  la limite de remplissage de la table à partir de laquelle la table est automatiquement agrandie.

1 Constructeurs :

|  |  |
| --- | --- |
| HashMap() | Crée une *HashMap* de capacité initiale 16 et de loadFactor 0.75. |
| HashMap( **int** capaciteInitiale) | Crée une *HashMap* de capacité initiale *capaciteInitiale*et de loadFactor 0.75. |
| HashMap( **int** capaciteInitiale, **float** loadFactor) | Crée une *HashMap* de capacité initiale *capaciteInitiale*et et de loadFactor *loadFactor.* |
| HashMap(Map<? **extends** K, ? **extends** V> m) | Crée une *HashMap* contenant tous les élément de la *Map m*. |

1 Méthodes :

|  |  |
| --- | --- |
| **void** clear() | Supprime toutes les entrées de la table. |
| **boolean** containsKey(Object cle) | Retourne *true* si la clé *cle* est dans la table et *false* sinon. |
| **boolean** containsValue(Object valeur) | Retourne *true* si la valeur *valeur*est dans la table et *false* sinon. |
| Set<Map.entry<K, V>> entrySet() | Retourne une vue *Set* de la table Hachée. |
| V get(Object cle) | Retourne la valeur associée à la clé *cle* si cette clé existe dans la table, et *null* sinon. |
| **boolean** isEmpty() | Retourne *true* si la table hachée est vide et *false* sinon. |
| Set<K>keySet() | Retourne une vue *Set* sur le clés de la table hachée. |
| V put(K cle, V valeur) | Associe la *cle* et la *valeur* dans la table hachée. Si la clé était déjà présente, la valeur est remplacée. Retourne *null* si la clé n'était pas déjà présente dans la table hachée, et l'ancienne valeur associée à la clé sinon. |
| **void** putAll(Map<? extends K, ? extends V> m) | Ajoute (*put) à la table hachée tous les couples (cle, valeur) contenus dans le Map m.* |
| V remove(Object cle) | Enlève de la table hachée l'entrée associée à *cle*, et retourne la *valeur* associée à *cle* si la clé est présente dans la table hachée, et *null* sinon. |
| **int** size() | Retourne le nombre d'entrées de la table hachée. |
| Collection<V>values() | Retourne une vue Collection de l'ensemble des valeurs de la table hachée. |

# *HashSet*

Cette classe implémente l'interface *Set*, en utilisant une [table hachée](http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/Conteneurs/TableDeHachage.html). Un *HashSet* est implanté comme une [*HashMap*](http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/Conteneurs/HashMap.html), dont les clés sont les éléments du *hashSet* et les valeurs sont toutes une même valeur *PRESENT*.  
Les opérations *get* et *put* sont en temps constant.   
La capacité est le nombre de listes. Le *load factor* (0.75 par défaut) est  la limite de remplissage de la table à partir de laquelle la table est automatiquement agrandie.

1 Constructeurs :

|  |  |
| --- | --- |
| HashSet() | Crée une *HashSet* de capacité initiale 16 et de loadFactor 0.75. |
| HashSet( **int** capaciteInitiale) | Crée une *HashSet* de capacité initiale *capaciteInitiale*et de loadFactor 0.75. |
| HashSet( **int** capaciteInitiale, **float** loadFactor) | Crée une *HashSet* de capacité initiale *capaciteInitiale*et et de loadFactor *loadFactor.* |
| HashSet(Collection<? **extends E> c)** | Crée une *HashSet* contenant tous les élément de la *Collection c*. |

2 Méthodes :

|  |  |
| --- | --- |
| **void** clear() | Supprime tous les élément du *Set*. |
| **boolean** add(E o) | Ajoute *o* dans le *Set* et  retourne *true* si *o* a été effectivement ajouté et *false* sinon (*o* était déjà présent dans le *Set*) |
| **boolean** contains(Object o) | Retourne *true* si l'objet *o* est dans le *Set* et *false* sinon. |
| **boolean** isEmpty() | Retourne *true* si le *Set* est vide et *false* sinon. |
| **boolean** remove(Object o) | Enlève du *Set* l'objet *o* et retourne *true*, retourne *false*si *o* n'est pas dans le *Set*. |
| **int** size() | Retourne le nombre d'éléments du *Set*. |
| Iterator<E> iterator() | Retourne un itérateur sur les éléments du *Set*. |

3 Définition de quelques méthodes :

**public** **class** HashSet<E> **extends** AbstractSet<E> **implements** Set<E>{

**private** **transient** HashMap<E,Object> map;

// valeur bidon

**private** **static** **final** Object PRESENT = **new** Object();

**public** HashSet(**int** initialCapacity, **float** loadFactor) {

map = **new** HashMap<E,Object>(initialCapacity, loadFactor);

}

**public** **boolean** add(E o) {

**return** map.put(o, PRESENT)==**null**;

}

**public** **boolean** remove(Object o) {

**return** map.remove(o)==PRESENT;

}

. . .

# *Comparaison des performances HashSet-TreeSet*

La classe *HashMap* a des temps d'ajout et de recherche légèrement meilleurs que la classe *TreeSet*, pour des objets qui n'ont pas besoin d'être *Comparable*. En revanche la classe *TreeSet* maintient des objets ordonnés.  
[Application de Test](http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/Conteneurs/testHashTree.jnlp).

***Collections***

Cette classe contient des méthodes statiques permettant de manipuler des collections :

* Recherche, recherche dichotomique
* Tris
* Remplissage
* Egalité

Ces méthodes lèvent toutes une *NullPointerException*, si la ou les collections en paramètre ont valeur *null*.

1Recherche et tris :

|  |  |
| --- | --- |
| **int** binarySearch(List<? **extends** Comparable<? **super** T>> liste, T cle)  **int** binarySearch(List<? **extends** T> liste, T cle, Comparator<? **super** T> c) | Recherche dichotomique de *cle* dans une liste, en utilisant *compareTo*, ou un *Comparator*. |
| T max(Collection<? **extends** T> coll)  T max(Collection <? **extends** T> coll, Comparator<? **super** T> comp) | Maximum des objets d'une collection, en utilisant le *compareTo* des objets ou le *Comparator*. |
| T min(Collection <? **extends** T> coll)  T min(Collection <? **extends** T> coll, Comparator<? **super**T> comp) | Minimum des objets d'une collection, en utilisant le *compareTo* des objets ou le *Comparator*. |
| **void** sort (List<T> liste)  **void** sort (List<T> liste, Comparator<? **super** T>) | Tri des éléments de la liste en utilisant la méthode *compareTo* des éléments. Ce tri est stable. La méthodes crée un tableau des éléments de la liste et trie ce tableau. |
| **int** indexOfSubList(List source, List target)  **int** lastIndexOfSubList(List source, List target) | Retourne l’indice de la première occurrence  de *source* dans *target* ( -1 si pas trouvé) |

http://imss-www.upmf-grenoble.fr/prevert/Prog/Java/Conteneurs/3dnum_002.GIFChanger l'ordre des éléments :

|  |  |
| --- | --- |
| **void** reverse(List<?> l) | Inverse l'ordre des éléments de la liste *l*. |
| **void** rotate( List<?> l, **int** distance) | Effectue une rotation sur les éléments de la liste *l*. l'élément d'indice *i* devient l'élément d'indice *(i+distance) % l.size()*. |
| **void** shuffle (List <?> l) | Permute de façon aléatoire les éléments de la liste. Le germe de la suite aléatoire est le germe par défaut.. |
| **void** shuffle (List <?> l, Random r) | Permute de façon aléatoire les éléments de la liste. Le  germe de la suite aléatoire *R*. |
| **void** swap(List<?> l, **int**i, **int** j) | Echange les éléments de la liste *l* d'indices *i*et *j*. |

3 Divers :

|  |  |
| --- | --- |
| Comparator<T> reverseOrder()  Comparator<T> reverseOrder(Comparator<T> comp) | Retourne un *Comparator* qui imposerait l'ordre inverse du *compareTo*, ou de *comp*. |
| **void** copy (List<? **super** T> dest, List<? **extends** T> source) | Copie tous les éléments de la liste *source*, vers la liste *dest*. La liste *dest* doit être au moins aussi longue que *source*. Si *dest* est plus long que *source*, les éléments supplémentaires sont inchangés. |
| **void** fill(List<? **super** T>l, T o) | Remplace tous les éléments de lal iste *l*par *o*. |
| **boolean** replaceAll(List<T> l, T aV, T nV) | Remplace, dans la liste *l*, toutes les occurrences de *aV*, par *nV*. Retourne *true* s'il y a eut au moins un remplacement, et *false* sinon. |
| **int** frequency(Collection<? >c, Object o) | Retourne le nombre d'occurrences de l'objet *o* dans la collection *c*. |
| List<T> nCopies(**int** n, T o) | Retourne une liste constituée de *n* fois l'objet *o*. La liste retournée est immuable,  et ne contient que des références au même objet *o*. |

4Collections synchronisées :

Les  différentes méthodes des classes collection ne sont pas synchronisées,  la classe *Collections* offre des méthodes permettant d'obtenir pour chaque classe une enveloppe synchronisée de la classe.

|  |  |
| --- | --- |
| Collection<T> synchronizedCollection(Collection<T> c) | Retourne une vue synchronisée de la collection *c*. Toutes les méthodes de la collection *c* sont synchronisées. Les classes synchronisées sont construites sur le schéma suivant :  SynchronizedX<E> **implements** X<E>, Serializable {  X<E> c; *// le X sous jacent*  Object mutex;  *// l'objet sur lequel synchroniser*  SynchronizedX(X<E> c) {<  **if** (c==**null**)  **throw** **new** NullPointerException();  **this**.c = c;  mutex = **this**;  }  **public** Type f() {  **synchronized**(mutex) { **return** c.f(); }  }  . . .  } |
| List<T> synchronizedList(List<T> l) |
| Map<T> synchronizedMap(Map<T> m) |
| Set<T> synchronizedSet(Set<T> s) |
| SortedMap<T> synchronizedSortedMap(SortedMap<T> s) |
| SortedSet<T> synchronizedSortedSet(SortedSet<T> s) |

5unmodifiable :

Les méthodes suivantes retourne une collection qui est non modifiable : c'est à dire que les opérations d'ajout et de suppression sont devenues interdites ; elles lèvent l'exception UnsupportedOperationException.

|  |  |
| --- | --- |
| unmodifiableCollection(Collection<? **extends** T>) | Retourne une vue non modifiable de la collection. |
| unmodifiableList(List<? **extends** T>) | Retourne une vue non modifiable de la liste. |
| unmodifiableMap(Map<? **extends** K, ? **extends** V>) | Retourne une vue non modifiable de la mapppe. |
| undunmodifiableSet(Set<? **extends** T>) | Retourne une vue non modifiable du set. |
| unmodifiableSortedMap(SortedMap<K, ? **extends** V>) | Retourne une vue non modifiable du sortedMap. |
| unmodifiableSortedSet(SortedSet<T>) | Retourne une vue non modifiable du sortedSet. |

La collection non modifiable et la collection d'origine, partagent la même représentation des données. Celà entraine que si on modifie ( ajout ou suppression) la collection d'origine, la collection non modifiable est modifiée !

List<String> lOrigine = **new** ArrayList<String>();

list.add("un");

Collection<String> nonModifiable = Collections.unmodifiableCollection(lOrigine);

lOrigine.add("deux");

System.out.print(nonModifiable);

Affiche **[un, deux]** à la console.

# *Exercices de programmation : nombres*

1. Pourquoi les expressions suivantes sont-elles illégales ?
2. ++i--
3. 12 = 11+1
4. 12 == 11+1 && 12
5. Parenthéser et évaluer les expressions suivantes.
6. **int** a, b, c ;
7. a = 1-+-1
8. a = a+++a ;
9. a = b = c ;
10. a = a>5 ? c : 10 ;
11. a = 7+3\*6/2-1 ;
12. a = 2%2+2\*2-2/2 ;
13. a = 3\*9\*(3+9\*3/3) ;
14. Quelle(s) expression(s) correspond(ent) à la formule mathématique y = ax3+7 ?
15. y = a\*x\*x\*x+7
16. y = a\*x\*x\*(x+7)
17. y = (a\*x)\*x\*(x+7)
18. y == a\*x\*x\*x+7
19. y == a\*x\*x\*x+7
20. y = a\*(x\*x\*x)+7
21. Quelle est la valeur de *a* après chacune des instructions suivantes ?
22. **short** a = 215 ;
23. a = a\*a\*a\*a ;
24. a += 10733022 ;
25. ++a
26. Quelles sont les valeurs de *a* et *b* après chacune des instructions suivantes ?
27. **int** a = 215, b = 150, c ;
28. c = a ;
29. a = b ;
30. b = c ;
31. Quelles sont les valeurs de *a* et *b* après chacune des instructions suivantes ?
32. **int** a = 215, b = 150 ;
33. a = a - b ;
34. b = a + b ;
35. a = b - a ;
36. La séquence suivante énumère 10 entiers aléatoires, compris entre 0 et 1000 (exclu) :
37. Random rd = **new** Random();
38. **for** (**int** i = 0; i<10; ++i){
39. **int** a = rd.nextInt(1000);
40. }

Dans la séquence suivante, remplacer la partie jaune par du code qui permet de calculer :

* + La somme des nombres.
  + Le produit des nombres.
  + Le maximum des nombres.

Random rd = **new** Random();

**int** res =                 ;

**for** (**int** i = 0; i<10; ++i){

**int** a = rd.nextInt(1000);

res =                 ;

}

1. Donner une expression qui est vraie si le nième bit d'un entier long est à 1 et fausse sinon.
2. Pour calculer la surface d’un triangle dont on connaît les longueurs des trois cotés ( *a*, *b* et *c*), on peut utiliser la formule de [Heron](http://fr.wikipedia.org/wiki/H%C3%A9ron_d'Alexandrie) :
3. p = (a+b+c)/2

s = sqrt(p\*(p-a)\*p-b)\*(p-c))

Ecrire deux définitions de méthode qui calculent la surface d’un triangle connaissant la longueur de ses trois cotés. Une méthode utilise des **double** et l’autre des **float** .  
Faire le test suivant :

System.out.println(heronD(12345678., 12345678., 1.));

System.out.println(heronF(12345678.F, 12345678.F, 1.F));

System.out.println(heronD(1234567., 1234567., 1.));

System.out.println(heronF(1234567.F, 1234567.F, 1.F));

Expliquer les résultats.

1. Ecrire une séquence d’instructions qui permet à l’ordinateur de deviner un nombre compris ente 1 et 16 choisi par l’utilisateur. L’ordinateur pose des questions du style « le nombre est-il <= à 7 », et l’utilisateur répond par « oui » ou « non ».
2. Définir les méthodes qui calculent :
   * La somme des *n* premiers entiers.
   * La somme des carrés de *n* premiers entiers.
   * La somme des cubes de *n* premiers entiers.
3. Un [nombre premier](http://fr.wikipedia.org/wiki/Nombre_premier) est un nombre qui n'est divisible que par 1 et par lui même. Ecrire la définition d'une méthode qui retourne ***true*** si le nombre *n* en paramètre est premier et ***false*** sinon.

Ecrire la définition d'une méthode qui retourne la somme des chiffres du nombre *n* écrit dans la base *b*. Exemple :

sommeChifres(63, 10) = 9

sommeChifres(63, 2) = 6

Un [nombre de Smith](http://fr.wikipedia.org/wiki/Nombre_de_Smith) est un nombre qui est tel que la somme de ses chiffres est égale à la somme des chiffres de tous ses facteurs premiers. Ecrire la définition d'une méthode qui retourne ***true*** si le nombre *n* en paramètre est un nombre de Smith et ***false*** sinon. Exemple :

estSmith(666, 10) = true

estSmith(666, 2) = false

estSmith(4937775, 10) = true

estSmith(4937775, 2) = true

|  |
| --- |
|  |

1. Pour calculer la racine carrée de *a*, l'algorithme de [Newton](http://fr.wikipedia.org/wiki/Isaac_Newton) ( aussi connu depuis l'antiquité sous le nom d'[algorithme de babylone](http://mathematiques.ac-bordeaux.fr/pedalyc/dosped/analyse/pb_1ts/alg_bab.htm) ou d'algorithme de [Héron d'alexandrie](http://fr.wikipedia.org/wiki/H%C3%A9ron_d%27Alexandrie)) énumère les éléments de la suite :
2. *x0 = a/2*
3. *xn+1 = 1/2×(xn+a/xn)*
   * Ecrire la définition d'une méthode qui calcule la racine carrée de *a* en calculant *n* termes de la suite.
   * Ecrire la définition d'une méthode qui calcule la racine carrée de *a* en s'arrêtant quand l'intervalle entre deux termes successifs de la suite est plus petit que un *ε* donné.
   * Ecrire la définition d'une méthode qui calcule la racine carrée d'un nombre en s'arrêtant quand l'intervalle entre deux termes successifs de la suite est plus petit que *10-n*. On utilisera la classe *BigDecimal*.

La méthode de [Halley](http://fr.wikipedia.org/wiki/It%C3%A9ration_de_Halley) pour calculer la racine carrée d'un nombre est la suivante :

*x0 = a/2*

*xn+1 = (xn×xn×xn+3×a×xn)/(3×xn×xn+a)*

La méthode de [HouseHolder](http://fr.wikipedia.org/wiki/It%C3%A9ration_de_Householder) pour calculer la racine carrée d'un nombre est la suivante :

*x0 = a/2*

*xn+1 = (3×xn×xn×xn×xn+6×a×xn×xn-a×a)/(8×xn×xn×xn)*

Ecrire la définition d'une méthode qui calcule la racine carrée d'un nombre en s'arrêtant quand l'intervalle entre deux termes successifs de la suite est plus petit que *10-n*. On utilisera la classe *BigDecimal*. Comparer les *vitesses* (nombre d'itérations nécessaires) des différentes méthodes.

1. Le [coefficient binomial](http://fr.wikipedia.org/wiki/Coefficient_binomial) est défini par (binom{n}{k})= {fact n} over {fact k fact(n-k)} ou par :   
   (binom{n}{n})= (binom{n}{1})=1  
   (binom{n}{k})= (binom{n-1}{k})+(binom{n-1}{k-1}) pour *1<k<n*. Ecrire la définition d'une méthode qui calcule le coefficient binomial pour deux entiers *n* et *k* pour chacune des définitions précédentes.
2. Un nombre de [Fibonacci](http://fr.wikipedia.org/wiki/Suite_de_Fibonacci) à l’ordre n est défini comme suit :
3. F0=F1=1

Fn= Fn-1 + Fn-2

Ecrire la définition d’une méthode récursive qui calcule le nombre de Fibonacci au rang *n*. Ecrire la définition d’une méthode itérative qui calcule le nombre de Fibonacci au rang *n*. Tester ces méthodes pour le calcul du nombre de Fibonacci aux rangs 40, 50, 60. Expliquer !

1. Un nombre de [Padovan](http://fr.wikipedia.org/wiki/Suite_de_Padovan) est défini par :
2. P0=P1= P2=1

Pn+1= Pn-1 + Pn-2

Ecrire la définition d’une méthode qui calcule le nombre de Padovan au rang *n*.

1. Un nombre de [Catalan](http://fr.wikipedia.org/wiki/Nombres_de_Catalan) est défini de la façon suivante :  
   http://imss-www.upmf-grenoble.fr/prevert/Prog/exercices/image006.gif  
   http://imss-www.upmf-grenoble.fr/prevert/Prog/exercices/image008.gif  
   Ecrire la définition d’une méthode qui calcule le nombre de Catalan au rang *n*.
2. Ecrire la définition d’une méthode qui énumère la suite de [Syracuse](http://fr.wikipedia.org/wiki/Conjecture_de_Syracuse) d’un nombre *n*.
   * Si *n* est pair on le divise par 2 et on recommence
   * Si *n* est impair on le multuplie par 3, on ajoute 1 et on recommence.
   * On s’arrête lorsque *n* vaut 1.

|  |
| --- |
|  |

1. Le [code de Gray](http://fr.wikipedia.org/wiki/Code_de_Gray) est un codage binaire des nombres entiers, tel que lorsqu'on passe d'un entier au suivant, un seul bit change dans la représentation de l'entier. Exemple :

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| 0000 | 0001 | 0011 | 0010 | 0110 | 0111 | 0101 | 0100 | 1100 |

1. Un algorithme permettant de passer d'un nombre de Gray au suivant :
   * si le nombre de 1 est pair, on inverse le dernier chiffre de la représentation.
   * si le nombre de 1 est impair, on inverse le chiffre situé à gauche du 1 le plus à droite.

Ecrire la définition d'une méthode *int graySuivant(int s)* qui, à partir d'une représentation d'un nombre en code de Gray, calcule la représentation du nombre suivant.

Ecrire la définition d'une méthode *int grayPrecedent(int s)* qui, à partir d'une représentation d'un nombre en code de Gray, calcule la représentation du nombre précédent.

Ecrire la définition d'une méthode *int int2gray(int n)* qui, à partir d'un nombre *n* retourne sa représentation en code de Gray, sans calculer les représentations des nombres qui précèdent *n*. On pourra utiliser la formule *codeGray(n) = n ^(n>>1)*.

ouh la laEcrire la définition d'une méthode *int gray2int(int s)* qui, à partir d'une représentation d'un nombre en code de Gray, calcule l'entier dont il est représentation.

1. Un nombre de [Lychrel](http://fr.wikipedia.org/wiki/Nombre_de_Lychrel) est un nombre naturel qui ne peut pas former de nombre palindrome lorsqu'il est soumis au processus itératif qui consiste à l'additionner au nombre formé de l'inversion de ses chiffres en base 10. Le nom *Lychrel* a été inventé par Wade VanLandingham : il s'agit d'une anagramme du nom de sa fiancée, Cheryl. Les nombres de Lychrel sont des nombres théoriques. On n'en connait aucun, bien que de nombreux nombres soient suspectés. Le plus petit nombre suspecté d'être de Lychrel est 196.  
   Définir une méthode qui programme le processus décrit, et qui s’arrête si le nombre donné en paramètre n’est pas un nombre de Lychrel.
2. Définir la méthode qui dit si deux nombres sont [amicaux](http://fr.wikipedia.org/wiki/Nombre_amical) ou non. Deux nombres *n* et *m* sont amicaux si la somme des diviseurs de l’un est égale à la somme des diviseurs de l’autre et égale à *n+m*.

Définir la méthode qui dit si un nombre est [parfait](http://fr.wikipedia.org/wiki/Nombre_parfait) ou non. Un nombre est parfait s’il est égal à la somme de ses diviseurs stricts. 6 (1+2+3 est parfait).

Définir la méthode qui dit si un nombre est [abondant](http://fr.wikipedia.org/wiki/Nombre_abondant) ou non. Un nombre est abondant s’il est inférieur à la somme de ses diviseurs stricts.

Définir la méthode qui dit si un nombre est [déficient](http://fr.wikipedia.org/wiki/Nombre_d%C3%A9ficient) ou non. Un nombre est déficient s’il est supérieur à la somme de ses diviseurs stricts.

|  |
| --- |
|  |

1. L'[algorithme d'Euclide](http://fr.wikipedia.org/wiki/Algorithme_d%27Euclide) permet de calculer le plus grand commun diviseur de deux entiers *a* et *b*.
   * si *b* = 0  le pgcd est *a*
   * si *b* != 0 le pgcd de *a* et *b* est le même que le pgcd de *b* et *a%b*

attention !L'[algorithme d'Euclide-Bezout](http://fr.wikipedia.org/wiki/Algorithme_d%27Euclide_%C3%A9tendu) permet de calculer le plus grand commun diviseur de deux entiers *a* et *b*, ainsi que deux entiers *u* et *v* tels que *a×u + b×v = pgcd*.

1. Ecrire la définition d'une méthode qui retourne vrai si l'entier en paramètre est un nombre [premier](http://fr.wikipedia.org/wiki/Nombre_premier) et faux sinon. Le polynome x2+x+41 est réputé [générer](http://mathworld.wolfram.com/Prime-GeneratingPolynomial.html) des nombre premiers. Définir une méthode qui trouve le premier nombre non premier généré par ce polynome.

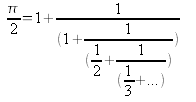
|  |
| --- |
|  |

1. L'algorithme d'[Archimède](http://fr.wikipedia.org/wiki/Archim%C3%A8de) pour calculer la valeur du nombre π consiste à calculer les périmètres de polygones réguliers inscrits et exinscrits dans un cercle de rayon 1.  
   Les demi-périmètres des polygones inscrits sont donnés par la suite :
2. I4=2×sqrt(2)
3. I2n=sqrt(In×E2n)
4. Les demi-périmètres des polygones exinscrits sont donnés par la suite :
5. E4=4
6. E2n=2×In×En/(In+En)
7. On commence par les carrés inscrit et exinscrit, puis à chaque étape on double le nombre de cotés des polygones.

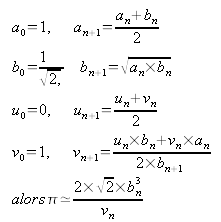
|  |
| --- |
|  |

1. Une mauvaise méthode de calcul de π est une méthode monte-carlo. On tire au hasard un point dans le carré de coté 2. La probabilité pour que ce point se trouve à l'intérieur du cercle de rayon 1 inscrit dans ce carré est π/4. Ecrire la définition d'une méthode qui calcule π en tirant *n* points au hasard.

|  |
| --- |
|  |

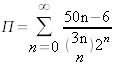
1. Le calcul de π peut se faire en utilisant la fraction continue suivante :   
     
   Ecrire la définition d'une méthode qui calcule π en utilisant la formule précédente au rang *n*.

|  |  |
| --- | --- |
|  |  |

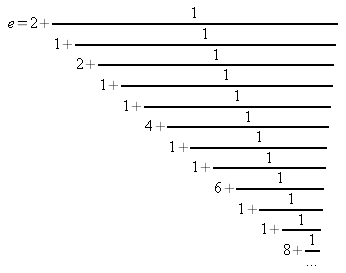
1. Le calcul de [π](http://www.pi314.net/) peut se faire en utilisant l'une des formules suivantes
   * [Leibniz](http://fr.wikipedia.org/wiki/Leibniz)    http://imss-www.upmf-grenoble.fr/prevert/Prog/exercices/leibniz.gif
   * [Wallis](http://fr.wikipedia.org/wiki/John_Wallis)     http://imss-www.upmf-grenoble.fr/prevert/Prog/exercices/wallis.gif
   * [Viète](http://fr.wikipedia.org/wiki/Vi%C3%A8te)      http://imss-www.upmf-grenoble.fr/prevert/Prog/exercices/viete.gif
   * [Brent et Salamin](http://www.pi314.net/sitepdf/salamin.PDF)    
   * [Plouffe](http://fr.wikipedia.org/wiki/Simon_Plouffe)    http://imss-www.upmf-grenoble.fr/prevert/Prog/exercices/plouffe.gif

Pour chacune de ces formules, écrire la définition d'une méthode qui calcule π en utilisant la formule au rang *n*.

|  |  |
| --- | --- |
|  |  |

1. En 1974 Gosper a découvert la formule . Ecrire la définition d'une méthode qui retourne la valeur de π en calculant la somme précédente au rang *n*.

|  |
| --- |
|  |

1. Le calcul de [E](http://www-history.mcs.st-andrews.ac.uk/HistTopics/e.html) peut se faire en utilisant l'une des formules suivantes :
   * [Jacob Bernoulli](http://fr.wikipedia.org/wiki/Jacob_Bernoulli)    http://imss-www.upmf-grenoble.fr/prevert/Prog/exercices/bernouilli.gif
   * [Euler](http://fr.wikipedia.org/wiki/Euler)     http://imss-www.upmf-grenoble.fr/prevert/Prog/exercices/euler.gif
   * [Euler, fractions continues](http://fr.wikipedia.org/wiki/Fraction_continue)      

Pour chacune de ces formules, écrire la définition d'une méthode qui calcule E en utilisant la formule au rang *n*.

# *Exercices de programmation : tableaux*

|  |
| --- |
| 1. [Tabeaux à 1 dimension](http://imss-www.upmf-grenoble.fr/prevert/Prog/exercices/tableaux.html#uneDim) 2. [Tabeaux à 2 dimensions](http://imss-www.upmf-grenoble.fr/prevert/Prog/exercices/tableaux.html#deuxDim) |

1Tableaux à une dimension.

1. Ecrire la définition d'une méthode qui initialise les éléments d’un tableau d’entiers à 0.
2. Ecrire la définition d'une méthode qui affiche le contenu d’un tableau à la console. On écrira deux versions , une qui affiche tous les éléments du tableau, l'autre qui affiche que les *n* premiers éléments du tableau.
3. Ecrire la définition d'une méthode qui retourne la chaîne de caractères "[", les éléments du tableau séparés par des "," et "]". On écrira deux versions , une qui prend en compte tous les éléments du tableau, l'autre qui ne prend en compte que les *n* premiers éléments du tableau.
4. Ecrire la définition d'une méthode qui augmente de 1 tous les éléments d’un tableau.
5. Ecrire la définition d'une méthode qui, pour tous les éléments d’un tableau d’entiers, affiche si cet élément est pair ou non.
6. Ecrire la définition d'une méthode calcule la somme des éléments d’un tableau. On écrira deux versions , une qui prend en compte tous les éléments du tableau, l'autre qui ne prend en compte que les *n* premiers éléments du tableau.
7. Ecrire une séquence d’instructions qui calcule le nombre d’éléments pairs dans un tableau d’entiers. On écrira deux versions , une qui prend en compte tous les éléments du tableau, l'autre qui ne prend en compte que les *n* premiers éléments du tableau.
8. Ecrire une séquence d’instructions qui calcule le maximum des éléments d’un tableau d’entiers. On écrira deux versions , une qui prend en compte tous les éléments du tableau, l'autre qui ne prend en compte que les *n*premiers éléments du tableau.
9. Ecrire une séquence d’instructions qui calcule si un tableau d’entiers est ordonné dans l’ordre croissant. On écrira deux versions , une qui prend en compte tous les éléments du tableau, l'autre qui ne prend en compte que les *n* premiers éléments du tableau.
10. Ecrire la définition de deux méthodes qui ordonnent les *n* premiers éléments d'un tableau d'entiers par ordre croissant, en utilisant les algorithmes suivants :
    * Insertion : on suppose que la partie de tableau qui va de l'indice 0 à l'indice *i* est ordonnée. On insère alors l'élément d'indice *i+1* dans cette partie ordonnée. on obtient alors une partie qui va de 0 à *i+1* qui est ordonnée.
    * Sélection/échange : On cherche le plus petit élément dans la partie qui va de l'indice *i* à l'indice *n-1*, puis on échange cet élément avec l'élément d'indice *i*. On recommence en faisant varier *i*.
11. Soit *t* un tableau d'entiers. Une série dans *t* est une suite d'éléments consécutifs et égaux de *t*. Ecrire une séquence d'instruction qui calcule l'indice du premier élément et la longueur de la plus longue série de *t*. exemple :
12. **int** [] t = {1, 1, 2, 2, 2, 9, 9, 9, 9, 2, 2, 3};

Alors la plus longue série commence à l'indice 5 et a pour longueur 4. On écrira deux versions , une qui prend en compte tous les éléments du tableau, l'autre qui ne prend en compte que les *n* premiers éléments du tableau.

1. Ecrire la définition d'une méthode qui comprime un tableau d'entiers. Pour réaliser la compression, on remplacera toute suite de *n* fois la valeur *a* par *n* suivi de *a*;
2. **int** [] t = {1, 1, 2, 2, 2, 9, 9, 9, 9, 2, 2, 3};

est comprimé en :

**int** [] t1 = {2, 1, 3, 2, 4, 9, 2, 2, 1, 3};

On écrira deux versions , une qui prend en compte tous les éléments du tableau, l'autre qui ne prend en compte que les *n* premiers éléments du tableau.

1. Ecrire la définition d'une méthode qui retourne l'indice de la première occurrence de l'entier *a* dans un tableau qui contient *n* éléments entiers. Cette méthode retourne -1 si l'entier *a* ne se trouve pas dans la tableau.

Même question que la précédente, en supposant que la tableau est ordonné par ordre croissant.

1. attention !Ecrire la définition d'une méthode qui retourne l'élément médian d'un tableau d'entiers. L'élément médian est l'élément qui se trouverait au milieu du tableau si celui-ci était ordonné. On écrira deux versions , une qui prend en compte tous les éléments du tableau, l'autre qui ne prend en compte que les *n* premiers éléments du tableau.
2. Soit *t* un tableau de *n* entiers (*n* <= *t.length*).
   * Ecrire la définition d'une méthode qui ajoute un élément à l'indice *i* dans le tableau, si c'est possible, et lève une exception sinon.
   * Ecrire la définition d'une méthode qui enlève l'élément d'indice *i* du tableau, si c'est possible, et lève une exception sinon.
3. Nous supposons que le tableau *t* de *n+1* nombres réels est le tableau des coefficients d'un polynome de degré *n*. Définir une méthode qui a en paramètre le tableau *t* et un nombre réel *x*, et qui retourne la valeur du polynome :  xn×tn+xn-1×tn-1+...+x0×t0. On utilisera la méthode de Méthode de [Ruffini-Horner](http://fr.wikipedia.org/wiki/M%C3%A9thode_de_Ruffini-Horner).
4. Programmer le [crible](http://fr.wikipedia.org/wiki/Crible_d%27%C3%89ratosth%C3%A8ne) [d’Erathosthène](http://fr.wikipedia.org/wiki/%C3%89ratosth%C3%A8ne) pour calculer l’ensemble des nombres premiers inférieurs à 1000.
5. Définir une méthode qui retourne un tableau constitué des éléments du tableau d'entiers *t*. Dans le tableau retourné, chaque élément n'a qu'une seule occurrence.
6. Définir une méthode qui inverse l'ordre des éléments du tableau d'entiers en paramètre.
7. Définir une méthode qui retourne le tableau obtenu par concaténation des deux tableaux d'entiers en paramètres.
8. Définir une méthode qui modifie de façon aléatoire l'ordre des éléments du tableau d'entiers en paramètre. On écrira deux versions , une qui prend en compte tous les éléments du tableau, l'autre qui ne prend en compte que les *n* premiers éléments du tableau.
9. Supposons que deux tableaux de *n* éléments réels représentent deux vecteurs. Ecrire la définition d'une méthode qui retourne le [produit scalaire](http://fr.wikipedia.org/wiki/Produit_scalaire) de ces deux vecteurs. La valeur retournée est t10×t20+ ... +t1n-1×t2n-1.
10. Le langage [Python](http://www.python.org/) définit l'opérateur *slice* sur les listes.Si *l* est une liste, *l[d:f]* est une liste constituée de tous les éléments de *l* qui vont de l'indice *d* (inclus) à l'indice *f* (exclu). Ecrire la définition d'une méthode *slice*qui a le même sens pour les tableaux à une dimension en Java.
11. Un diviseur de *n* qui est différent de *n* est appelé [diviseur strict](http://fr.wikipedia.org/wiki/Diviseur_strict) de *n*. Ecrire la définition d'une méthode qui retourne un tableau contenant tous les diviseurs stricts de *n*.

Un nombre [abondant](http://fr.wikipedia.org/wiki/Nombre_abondant) est un nombre dont la somme de ses diviseurs stricts est supérieure au nombre. Ecrire la définition d'une méthode qui énumères tous les nombres abondants inférieurs à 100.

Un nombre [parfait](http://fr.wikipedia.org/wiki/Nombre_parfait) est un nombre dont la somme de ses diviseurs stricts est égales au nombre. Ecrire la définition d'une méthode qui énumères tous les nombres parfaits inférieurs à 1000.

Deux nombres *n* et *m* sont [amicaux](http://fr.wikipedia.org/wiki/Nombres_amicaux) si la somme de ses diviseurs stricts de *n* est égale à *m*, et la somme de ses diviseurs stricts de *m* est égale à *n*. Ecrire la définition d'une méthode qui énumères tous les couples de nombres amicaux inférieurs à 5000.

|  |
| --- |
| a_{0}+{1}over{a_{1}+{1}over{a_{2}+{1}over{a_{3}+{1}over{...}}}} |

1. Un tableau *a* contient les coefficients intervenant dans une [fraction continue](http://fr.wikipedia.org/wiki/Fraction_continue).
2. Ecrire la définition d'une méthode qui retourne la valeur de la fraction continue calculée avec les coefficients contenus dans *a*.

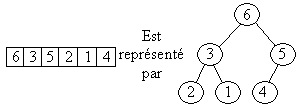
|  |
| --- |
| a_{0}+{b_{0}}over{a_{1}+{b_{1}}over{a_{2}+{b_{2}}over{a_{3}+{b_{3}}over{...}}}} |

1. Un tableau *a* et un tableau *b* contiennent les coefficients intervenant dans une [fraction continue généralisée](http://fr.wikipedia.org/wiki/Fraction_continue).
2. Ecrire la définition d'une méthode qui retourne la valeur de la fraction continue calculée avec les coefficients contenus dans *a* et *b*.

|  |  |
| --- | --- |
| attention !Ecrire la définition d'une méthode qui calcule le tableau des coefficients pour un paramètre *BigDecimal d*, avec au maximum *n* termes. (voir [ici](http://fr.wikipedia.org/wiki/Fraction_continue#Algorithme_de_d.C3.A9veloppement_en_fraction_continue_pour_un_r.C3.A9el)) |  |

|  |
| --- |
|  |

1. Un arbre binaire ordonné horizontalement est défini par : tout noeud de l’arbre binaire a une valeur plus grande que la valeur de chacun de ses fils. Un arbre binaire ordonné horizontalement peut être représenté dans un tableau :
   * La racine est à l’indice 0.
   * L’élément à l’indice *i* est le père de l’élément à l’indice *2×i+1* s’il existe, et le père de l’élément de rang *2×i+2*s’il existe.

Cette structure de tableau se nomme un [tas](http://fr.wikipedia.org/wiki/Tas_(informatique))  
Exemple :   


* + **Ajouter** un élément dans le tas consiste à ajouter un élément en fin du tableau représentant le tas, puis à le remonter à sa place dans l’arbre et donc dans le tableau.
  + **Retirer** un élément du tas, c’est retirer le premier élément, puis le remplacer par le dernier élément du tableau, et on descend cet élément dans l’arbre et donc dans le tableau.

Définir ces deux méthodes.

|  |
| --- |
|  |

1. Une suite de [Skolem](http://fr.wikipedia.org/wiki/Suite_de_Skolem) est constituée d'entiers de 1 à *n* répétés deux fois. De plus, chaque entier *i* de la suite est à une distance *i* du même entier dans la suite. On vérifiera que la suite suivante est une suite de Skolem :
2. 4, 1, 1, 5, 4, 2, 3, 2, 5, 3
3. On démontre qu'il n'existe des suites de Skolem que pour *n* tel que *n%4==0* ou *(n-1)%4==0*.
   * Ecrire la définition d'une méthode qui retourne ***true*** si un tableau d'entiers en paramètre contient une suite de Skolem et ***false*** sinon.
   * hou la la !Ecrire la définition d'une méthode qui retourne un tableau contenant une suite de Skolem pour un *n* donné, si c'est possible, et ***null*** sinon. A chaque appel la méthode ne retourne pas en général la même suite.

Une suite de [Langford](http://en.wikipedia.org/wiki/Langford_pairing) est constituée d'entiers de 1 à *n* répétés deux fois. De plus, chaque entier *i* de la suite est à une distance *i+1* du même entier dans la suite.

2Tableaux à deux dimensions.

1. Ecrire une séquence d’instructions qui initialise les éléments d’un tableau d’entiers à 2 dimensions (*n* et *m*) à 0.
2. Ecrire une séquence d’instructions qui initialise les éléments de la diagonale d’un tableau d’entiers à 2 dimensions (*n* et *m*) à 1.
3. Ecrire une séquence d’instructions qui affiche les éléments d’un tableau d’entiers à 2 dimensions (*n* et *m*). Chaque ligne du tableau est affichée sur une ligne de la console.

|  |
| --- |
| http://imss-www.upmf-grenoble.fr/prevert/Prog/exercices/tableau1.gif |

1. Ecrire une séquence d’instructions qui construit le tableau d'entiers (figure 1) :

|  |
| --- |
| http://imss-www.upmf-grenoble.fr/prevert/Prog/exercices/pascal.gif |

1. Ecrire la définition d'une méthode qui affiche le triangle de [Pascal](http://fr.wikipedia.org/wiki/Triangle_de_Pascal) au rang *n*. On commencera par construire un tableau d'entiers contenant les valeurs du triangle de Pascal.  
   Exemple pour *n*=5 :
2. Ecrire la definition d'une méthode qui retourne le transposé d'un tableau d'entiers à 2 dimensions (*n* et *m*). Le transposé *tt* d'un tableau *t* est tel que pour tout *i* et *j* on a *tt[i][j] == t[j][i]*.
3. Ecrire la definition d'une méthode qui retourne le tableau à deux dimensions (*n*, *o*) représentant le produit de deux matrices données en paramètre(*n*, *m*) et (*n*, *o*). Si le produit ne peut pas être calculé, on lèvera une exception. l'élément de ran *i* et *j* du résultat est : t1i,0×t20,j+ t1i,1×t21,j + ... +t1i,m-1×t2m-1, j.
4. Ecrire la definition d'une méthode qui retourne le tableau à deux dimensions (*n×m*, *2*) représentant le [produit *cartésien*](http://fr.wikipedia.org/wiki/Produit_cart%C3%A9sien) de deux tableaux d'entiers à une dimension (*n*) et (*m*). Exemple :
5. **int** [] a = {1, 2, 3};
6. **int** [] b = {10, 11};

Alors *produitCartesien(a,b)* retourne le tableau :

{{1,10},{1,11},

{2,10},{2,11},

{3,10},{3,11}

}

# *Exercices de programmation : chaînes de caractères*

1. Définir les méthodes de la classe *String*
2. Définir les méthodes de la classe *StringBuffer*

1. Ecrire la définition d'une méthode qui a en paramètre deux chaînes de caractères *s* et *r*, et retourne une chaîne de caractères constituée de tous les caracères de *s* sauf ceux qui appartiennent à *r*.
2. Ecrire une séquence d’instructions qui calcule le nombre d’occurrences de chaque voyelles dans une chaîne de caractères.
3. Ecrire la séquence d’instruction qui calcule si une chaîne de caractères est un [palindrome](http://fr.wikipedia.org/wiki/Palindrome) ou non. Un palindrome est une chaîne de caractères qui se lit indifféremment de gauche à droite ou de droite à gauche. exemple *«ici»* , *«kayak»* ou *«esoperesteicietserepose»*. Reprendre le même problème en considérant que les mots peuvent être séparés par des caractères ‘espace’ *«esope reste ici et se repose»* est alors un palindrome.
4. Ecrire la définition d'une méthode qui retourne ***true*** si *s1* est une [anagramme](http://fr.wikipedia.org/wiki/Anagramme) de *s2*, et ***false*** sinon.

Exemple : *«albert einstein»* est une anagramme de *«rien n est etabli»*.

1. Ecrire la définition d'une méthode qui a un paramètre chaîne de caractères *s*, et retourne une chaîne de caractères constituée de *s* concaténé à son inverse.
2. Ecrire la définition d'une méthode qui a un paramètre chaîne de caractères *s*, et retourne une chaîne de caractères constituée de tous les caractères de *s* sauf les voyelles.
3. Ecrire la définition d'une méthode qui a un paramètre chaîne de caractères *s*, et retourne une chaîne de caractères constituée de tous les caractères de *s* , le début de chaque mot étant en majuscule.
4. Ecrire la définition d'une méthode *toString* qui a un paramètre tableau de réels *t*, et retourne une chaîne de caractères constituée de tous les éléments du tableau *t* , séparés par des "," et entre "[" et "]".  
   Exemple :
5. **double**[]t={1.5, 5.2, 45.567};
6. s = toString(t);

Alors *s* = [1.50, 5.20, 45.57]

1. Ecrire la définition d'une méthode qui retourne ***true*** si une chaîne en paramètre est constituée de deux chaîne identiques concaténées, ***false*** sinon.
2. Une chaîne de caractères contient un texte formé de mots séparés par des espaces, des retour à la ligne, des tabulations, et des signes de ponctuation (. , ; : ? ! " '). Ecrire la définiton d'une méthode qui retourne un tableau des mots du texte.
3. Un [tautogramme](http://fr.wikipedia.org/wiki/Tautogramme) est un texte dont tous les mots commencent pat la même lettre
4. Mazarin, ministre malade,
5. méditait même moribond malicieusement mille maltôtes.

ou presque tous les mots commencent par la même lettre

Dans la zone zoologique, bon zigue,

zizagait l'ouvrier zingueur,

zieutant les zèbres

mais zigouillant plutôt les zibelines.

* + Ecrire la définition d'une méthode qui retourne ***true*** si un texte est un tautogramme et ***false*** sinon.
  + Ecrire la définition d'une méthode qui retourne la probabilité d'un texte d'être un tautogramme : le plus fort pourcentage de l'initiale de mot qui apparaît le plus souvent dans le texte (divisé par 100).

1. Ecrire la définition d'une méthode qui retourne ***true*** si une chaîne en paramètre est constituée de deux chaîne identiques concaténées, ***false*** sinon.
2. Les mots de [Fibonacci](http://fr.wikipedia.org/wiki/Mot_de_Fibonacci) sont définis de la façon suivante :
3. MF0="1"
4. MF1="0"
5. MFn=MFn-1+MFn-2

Définir une méthode qui retourne le mot de Fibonacci au rang *n*.

1. Soit la suite de chaînes (de [Thue-Morse](http://fr.wikipedia.org/wiki/Suite_de_Prouhet-Thue-Morse)) de caractères définie par :
2. S0="0"
3. Sn= la chaine obtenue en remplaçant dans Sn-1
4. toute occurrence de *0* par *01*
5. et toute occurrence de *1* par *10*

Ecrire la définition d'une méthode qui calcule Sn.

Soit la suite de chaînes (de [Mephisto Waltz Sequence](http://mathworld.wolfram.com/MephistoWaltzSequence.html)) de caractères définie par :

M0="0"

Mn= la chaine obtenue en remplaçant dans Mn-1

toute occurrence de *0* par *001*

et toute occurrence de *1* par *110*

Ecrire la définition d'une méthode qui calcule Mn.

1. Le cryptage de [César](http://fr.wikipedia.org/wiki/Chiffrement_par_d%C3%A9calage) consiste à un décalage de *n* positions des lettres du message. Définir la méthode *String cesar(String s, int n)* qui retourne la chaîne *s* cryptée suivant la méthode de César (on appliquera le cryptage à tout caractère, qu'il soit une lettre ou non). Comment décrypter ?

|  |
| --- |
|  |

1. La suite de [Conway](http://fr.wikipedia.org/wiki/Suite_de_Conway) est définie de la façon suivante :
   * C0 1
   * C1 : 11 (parce que le terme précédent contient un 1)
   * C2 : 21 (parce que le terme précédent contient deux 1)
   * C3 : 1211(parce que le terme précédent contient un 2, puis un 1 …)
   * …
   * C9 : 13211311123113112211

Définir la méthode qui calcule le terme de rang *n* de la suite de Conway.

|  |
| --- |
|  |

1. La suite de [Robinson](http://fr.wikipedia.org/wiki/Suite_de_Robinson) est définie de la façon suivante :
   * R0 : 0
   * R1 : 10 (parce que le terme précédent contient un 0)
   * R2 : 1110 (parce que le terme précédent contient un 1 et un 0)
   * R3 : 3110 (parce que le terme précédent contient trois 1, puis un 0 )
   * R4 : 132110 (parce que le terme précédent contient un 3, deux 1, puis un 0 )
   * R5 : 13123110 (parce que le terme précédent contient un 3, un 2, trois 1 puis un 0 )
   * …
   * R10 : 1433223110

Définir la méthode qui calcule le terme de rang *n* de la suite de Robinson.

1. Les deux méthodes suivantes retournent l'inverse d'une chaîne de caractères (les caractères sont dans l'ordre inverse). Une des deux méthodes est moins *«bonne»*; que l'autre : expliquer pourquoi.
2. String inverse1(String s){
3. **if**(s.equals("")) **return** "";
4. **else** **return** inverse1(s.substring(1))+s.charAt(0);
5. }
6. String inverse2(String s){
7. **int** lg = s.length();
8. StringBuffer sb = **new** StringBuffer(lg);
9. **for**(**int** i = 0; i < lg; ++i)
10. sb.insert(i, s.charAt(lg-i-1));
11. **return** sb.toString();
12. }

|  |
| --- |
|  |

1. Ecrire la définition des méthodes suivantes :
2. String versChaineBinaire(**byte** b)
3. String versChaineBinaire(**short** s)
4. String versChaineBinaire(**int** s)
5. String versChaineBinaire(**long** l)
6. Qui convertissent une valeur de type entier en sa représentation en binaire. On commencera par définir la méthode:
7. String versChaineBinaire(**long** l, **int** n)
8. Qui convertit les *n* bits de droite du long *l*, en leur représentation en binaire.

|  |
| --- |
|  |

1. Ecrire la définition des méthodes suivantes :
2. String versChaineBinaire(**float** s)
3. String versChaineBinaire(**double** s)
4. Qui convertissent une valeur de type flottant en sa représentation en binaire. On pourra utiliser les méthodes :
5. **int** floatToRawIntBits(**float** f)
6. **long** doubleToRawLongBits(**double** d)
7. Des classes *Float* et *Double* qui retournent un *int* ou un *long* contenant exactement les même bits que le *float* ou le *double* en paramètre.

# *Exercices de programmation : classes*

|  |
| --- |
|  |

1. Définir la classe [*Temperature*](http://fr.wikipedia.org/wiki/Temp%C3%A9rature), qui permet de représenter une température. La température pourra être obtenue, ou modifiée en utilisant les échelles de température *Kelvin*, *Celsius*, *Fahrenheit*et *Newton*.

|  |
| --- |
|  |

|  |
| --- |
|  |

1. Définir la classe [*Devise*](http://fr.wikipedia.org/wiki/Devise_%28monnaie%29), qui permet de représenter une somme exprimée en *Euros*. La somme pourra être obtenue, ou modifiée en utilisant d'autres devises *Dollar US*, *Yuan*, *Rouble*et *Livre Sterling*. (On pourra obtenir des cours plus récents en utilisant le webservice de [www.webservicex.net](http://www.webservicex.net/WS/WSDetails.aspx?WSID=10&CATID=2) qui retourne les taux de change entre devises)

|  |  |
| --- | --- |
| Dollar US |  |
| Yuan |  |
| Rouble |  |
| Frac Suisse |  |
| Livre anglaise |  |
| Zloty |  |

|  |
| --- |
|  |

|  |
| --- |
|  |

1. Définir la classe [*Angle*](http://fr.wikipedia.org/wiki/Angle), qui permet de représenter un angle exprimé en [*radians*](http://fr.wikipedia.org/wiki/Radian). L'angle pourra être obtenu, ou modifié en utilisant d'autres systèmes de mesure [*degré*](http://fr.wikipedia.org/wiki/Sous-unit%C3%A9s_du_degr%C3%A9) et [*grade*](http://fr.wikipedia.org/wiki/Grade_%28angle%29). La classe devra être munie des méthodes *plus* et *moins*.
2. Définir la classe [*Pression*](http://fr.wikipedia.org/wiki/Pression), qui permet de représenter une pression exprimée en [*Pascal*](http://fr.wikipedia.org/wiki/Blaise_Pascal). La pression pourra être obtenue, ou modifiée en utilisant d'autres systèmes de mesure *athmosphère*, [*bar*](http://fr.wikipedia.org/wiki/Bar_%28unit%C3%A9%29), *mm de mercure* ou [*PSI*](http://fr.wikipedia.org/wiki/PSI_%28unit%C3%A9%29). La classe devra être munie des méthodes *plus* et *moins*.

|  |  |  |  |
| --- | --- | --- | --- |
| **×** | **i** | **j** | **k** |
| **i** | -1 | k | -j |
| **j** | -k | -1 | i |
| **k** | j | -i | -1 |

1. Un [quaternion](http://fr.wikipedia.org/wiki/Quaternion) est un nombre *h = a + i×b + j×c + k×d* avec *a*, *b*, *c* et *d* réels. Les nombres *i*, *j*, et *k* sont tels que :  
   Définir un contructeur qui a quatre paramètres réels.  
   Définir les méthodes *toString*, *addition*, *multiplication* et *norme*.
2. Un [polynome](http://fr.wikipedia.org/wiki/Polyn%C3%B4me) en mathématique est une expression de la forme :  p(x)= c0×x0+c1×x1+c2×x2+ ... +cn×xn. Les coefficients *c* et la variable *x* appartiennent à un [anneau](http://fr.wikipedia.org/wiki/Anneau_%28math%C3%A9matiques%29) unitaire : un ensemble muni de deux lois de composition interne :
   * Une loi notée +, qui est commutative, associative et a un élément neutre noté 0. Tout élément a un inverse pour cette loi.
   * Une loi notée × , qui est associative, distributive par rapport à + et a un élément neutre noté 1.

L'exposant le plus grand de la variable qui a un coefficient non nul est le degré du polynome. Par convention le degré du polynome dont tous les coefficients sont nuls est -∞.

* + Définir l'interface *AnneauU* possédant les méthodes suivantes :
    - *AnneauU getZero();*
    - *AnneauU getUn();*
    - *AnneauU plus(AnneauU a);*
    - *AnneauU mult(AnneauU a);*
    - *String toString();*
  + Définir une classe *Entier* qui implémente *AnneauU*.
  + Définir la classe *Polynome* qui a des coefficients qui sont des *AnneauU*. La classe *Polynome* a les méthodes suivantes :
    - *Polynome plus (Polynome p)*
    - *Polynome mult(Polynome a);*
    - *AnneauU evaluer(AnneauU x);* qui évalue le polynome pour une valeur de la variable *x*.
    - *String toString();*
    - *polynome getZero();*
    - *polynome getUn();*

|  |
| --- |
|  |

1. Un peu de géométrie ...
   1. Définir la classe *Point*. Un *Point* est représenté par deux *double* qui sont les coordonnées du point dans un plan. On définira les méthodes suivantes :
      * *equals(Object o)* retourne *true* si les deux points sont égaux et *false* sinon. Deux points sont égaux si leurs coordonnées sont égales à *Point.PRECISION* près.
      * *toString()* retourne une chaîne de caractères contenant les coordonnées du point entre parenthèses.
      * *distance(Point p)* retourne la distance du point au point *p*.
      * *estProche(Point p)* retourne *true* si la distance du point à *p* est inférieure ou égale à *Point.PROXIMITE*.
      * *deplaceDe(int dx, int dy)* qui déplace le point de *dx* et *dy*.

* 1. Définir la classe [*Ligne*](http://fr.wikipedia.org/wiki/Droite_%28math%C3%A9matiques%29). Une *Ligne* est représentée par deux points non confondus. On définira les constructeurs suivants :
     + *ligne(point d, Point a)* qui définit la ligne passant par les deux points *d* et *a*, non confondus.
     + *ligne(point d, double angle)* qui définit la ligne passant par le point *d* et faisant un angle *angle* avec l'horizontale.

On définira les méthodes suivantes :

* + - *getAngle()* qui retourne l'angle de la ligne avec l'horizontale.
    - *contient (Point p)* qui retourne *true* si le point *p* est sur la ligne et *false* sinon.
    - *estConfondue(Ligne l)*

retourne *true* si les deux lignes sont confondues et *false* sinon.

* + - *estParallele(Ligne l)* retourne *true* si les deux lignes sont parallèles (non confondues) et *false* sinon.
    - *intersection(Ligne l)* retourne le point d'intersection des deux lignes s'il existe. La méthode retourne *null* si les lignes sont parallèles et lève une exception si les lignes sont confondues.
    - *deplaceDe(int dx, int dy)* qui déplace la ligne de *dx* et *dy*.

* 1. Définir la classe [*Segment*](http://fr.wikipedia.org/wiki/Segment_%28math%C3%A9matiques%29) sous-classe de *Ligne*, qui représente un segment de droite : on ne prend en compte que les points de la ligne qui sont compris entre *d* et *a*. Définir le constructeur et rédéfinir les méthodes qui doivent l'être dans la classe *Segment*.

* 1. Une [ligne brisée](http://fr.wikipedia.org/wiki/Ligne_polygonale) est un ensemble de points reliés par des segments. Définir la classe *LigneBrisee* munie du constructeur et des méthodes suivantes :
     + *LigneBrisee(Point[] ps)* qui construit une ligne brisée à partir d'un tableau de points.
     + *deplaceDe(int dx, int dy)* qui déplace la ligne brisée de *dx* et *dy*.
     + *longueur()* qui retourne la longueur de la ligne brisée.
     + *estCroise()* qui retourne *true* si un des segments de la ligne brisée coupe un autre segment de la ligne brisée et *false* sinon.

* 1. Définir la classe [*Polygone*](http://fr.wikipedia.org/wiki/Polygone), sous-classe de la classe *LigneBrisee*, qui définit trois méthodes supplémentaires :
     + *estConvexe()* qui retourne *true* si le polygone est convexe et *false* sinon.
     + *centreGravite()* qui retourne un point, [centre de gravité](http://fr.wikipedia.org/wiki/Aire_et_centre_de_gravit%C3%A9_d%27un_polygone) du polygone.
     + *surface()* qui retourne la [surface](http://fr.wikipedia.org/wiki/Aire_et_centre_de_gravit%C3%A9_d%27un_polygone) du polygone.