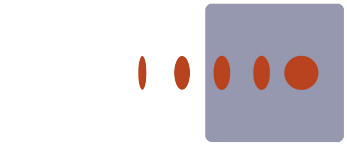


Manuel technique du langage Java

Jean-François Colin
TELECOM Lille 1

2008

Table des matières

Généralités

Ce document est un manuel d'introduction au langage Java. Deux prérequis sont nécessaires pour l'aborder;

- Une bonne connaissance du langage C et de sa syntaxe : les bases syntaxiques de Java ne sont en effet pas présentées car très proches de celles du C.
- Une bonne connaissance de l'approche orientée objet : les notions de classes, d'instances, d'héritage, de variable membre etc... n'y font pas non plus l'objet d'une présentation préalable.

Java est un langage foisonnant et évolutif. Ses bibliothèques, impressionnantes, sont en perpétuelle croissance. Ce document se concentre sur le langage et non sur les nombreuses bibliothèques qui l'accompagnent. Trois d'entre elles sont néanmoins présentées : la *Java Collection Framework (JCF),* la bibliothèque d'entrée-sortie et la bibliothèque de gestion des *threads*. Ces bibliothèques sont retenues en raison du caractère générique des outils qu'elles mettent à la disposition du programmeur.

Par ailleurs, pour mettre en évidence les caractéristiques spécifiques de Java, des comparaisons avec le langage C++ sont fréquemment présentées. C++ présentent beaucoup d'analogies syntaxiques avec Java, qu'il a historiquement précédé. Néanmoins le modèle d'exécution choisi par ces langages et la mise en oeuvre des concepts basés sur une syntaxe pourtant très proche sont souvent profondément différents

Ce document n'est pas utilisable seul : il doit être complété par les multiples ressources disponibles en ligne concernant Java et d'abord par celles fournies par Sun, en particulier la bibliothèque des API fournies par les *packages* standards : *http://java.sun.com/javase/6/docs/api/* .

Enfin un grand merci à Luc Betry pour sa relecture attentive.

Conventions d'écriture

Par sa très large bibliothèque Java a popularisé un certain nombre de conventions concernant le nommage des différentes entités. Le suivi de ces règles d'usage est fortement conseillé et constitue un style caractéristique de la programmation Java, même si la grammaire du langage Java n'impose aucune d'entre elles.

Ces règles sont très simples :

- les noms de package s'écrivent entièrement en minuscules. Exemple : *java.util.concurrent*
- les noms de classes ou interface sont en minuscules mais commencent par une majuscule. Si l'identifiant comporte plusieurs mots la première lettre de chacun est en majuscules. Exemple : *CyclicBarrier*, *SortedMap, ByteArrayOutputStream*
- les noms de méthodes sont en minuscules. Si l'identifiant comporte plusieurs mots la première lettre de chacun (sauf du premier) est en majuscule. Exemple : *write*, *newLine*, *divideAndRemainder*
- les noms de variables ou de paramètres suivent les mêmes règles que les noms de méthodes. Exemple : *calendar*, *outputBuffer*, *msgToSend, pathSeparator*
- les noms de constantes ou les énumérations sont entièrement en majuscules. Si l'identifiant est composé de plusieurs mots chacun d'entre eux est séparé par le symbole de soulignement "_". Exemple : *SIZE, MIN_VALUE, INFINITE*
- les identifiants d'annotation suivent les règles de nommage des interfaces, précédées du symbole "@". Exemple : *@Override*
- Les types formels des entités génériques apparaissent sous la forme d'une lettre unique et majuscule, souvent T, E, K, V.... Exemple : *class TreeMap<K,V>*

Les noms de classes sont très généralement des substantifs ou des constructions qui s'y ramènent: *String, StringBuffer, StackTraceElement* ...

Les noms de méthodes sont généralement des verbes ou des constructions verbales: *compareToIgnoreCase*, *concat*, *contains*, *contentEquals, copyValueOf ...*

Le couple de préfixe *get/set* est utilisé pour indiquer des méthodes d'accès aux propriétés d'un objet ou de modification d'un attribut de l'objet.

Exemple d'accès aux propriétés par des noms de méthodes préfixés *par* get et généralement sans paramètre: *getClassName getFileName*, *getLineNumber*, *getMethodName* etc...

Exemples de méthodes de modifications d'attributs, généralement avec paramètres : *setValue*, *setTimer* etc...

Pour le cas particulier des méthodes d'accès aux propriétés retournant un booléen on préférera souvent, pour la lisibilité, une forme verbale implicitement interrogative préfixée par *is : isNativeMethod*, *isEmpty etc...*

Structure générale d'un programme

Le programme *HelloWorld*

Tout programme java destiné à fournir une application autonome s'appuyant sur la machine virtuelle et utilisant le contexte standard appelé *JRE* (*Java Runtime Environnement*) doit comporter, comme son équivalent en *C*, un point d'entrée incarné par une méthode *main*.

D'autre part la syntaxe orientée objet ne comporte pas d'exception : tout commence par une classe, même le plus simple des programmes. Le nom de cette classe (*Launcher* ci-dessous) est libre et son nom sera également le nom du fichier source correspondant portant l'extension *java.*

```
public class Launcher { [1]
   public static void main(String[] args) {
      System.out.println("Hello World!");
   }
}
```

Ce texte, s'il est embarqué dans un fichier portant le nom de la classe et l'extension *java*, c'est à dire *Launcher.java*, pourra être présenté au compilateur fourni avec le JDK (*Java Development Kit*). Le nom de ce compilateur est *javac*. En ligne de commande la compilation du fichier précédent est obtenue par :

```
$ javac Launcher.java
```

Le résultat est la construction d'un fichier de byte-code *Launcher.class*. Ce fichier pourra être soumis à l'interpréteur java pour exécution. Le lancement de cet interpréteur, par l'invocation du programme *java*, correspond à la mise en oeuvre de la machine virtuelle java (la *JVM*).

```
$ java Launcher
Hello World!
```

Structure d'une programme simple

Généralement une classe prend sa classe dans le fichier *java* éponyme. La classe *A* sera donc écrite dans le fichier *A.java*.

Néanmoins il est parfois commode de disposer dans le fichier courant d'une classe locale au fichier et qui ne pourra être invoquée qu'à partir d'une instruction exprimée dans le fichier (ou le package) courant. L'écriture de courts exemples, tels que ceux qui figurent typiquement dans ce document, sera facilitée par cette possibilité.

Pour déclarer une telle classe il suffit de la déclarer localement dans ce fichier en omettant la mention *public* (ou *private*) qui normalement caractérise précisément sa visibilité pour les autres entités du programme[2].

Exemple :

```
// fichier Launcher.java
class Compteur {    // pas de mention public car cette classe n'est pas
                    // dans un fichier Compteur.java. Usage local au fichier ou
                    // package seulement.
   protected int value;
   public Compteur() { value = 0; }
   public void up() { value++; }
   public void raz() { value = 0; }
   public int getValue() { return value; }
   public String toString() { return "" + value;}
}

public class Launcher { // cette classe est dans son fichier Launcher.java. Mention public.
   public static void main(String[] args) {
      Compteur c1 = new Compteur();
      Compteur c2 = new Compteur();
      for (int i = 0; i < 1000; i++) c1.up();
      c2.up();
      System.out.println(c1); // invocation implicite de c1.toString()
      System.out.println(c2); // idem
   }
}
```

1 dans ce document nous nommerons systématiquement *Launcher* la classe constituant le point d'entrée d'une application, c'est à dire contenant la méthode *main*. Le nom de cette classe est libre.

2 Ce point est vu également dans le paragraphe *Protection des classes*

Résultat d'exécution :

```
1000
1
```

Instanciation

Pour les types de base l'instanciation fonctionne sur un modèle classique : leur simple déclaration crée l'objet.

```
int n1; // création d'une instance d'entier
int n2=300; // création et initialisation d'un entier
```

Pour les types objet, la création d'une instance s'effectue par l'opérateur *new* exclusivement, suivi de l'invocation du constructeur de la classe auquel on passe d'éventuels paramètres.

```
Compteur c1; // aucune instance créée ; c1 est une référence d'objet pas un objet
Compteur c2=new Compteur(); // une instance est créée. Elle sera référencée par c2
```

Pour les types objet la logique est celle de pointeur. Les variables référençant les objets sont des pointeurs de fait, même si Java, à l'inverse de C++, n'explicite pas au niveau du langage la notion de pointeur.

Un pointeur qui ne pointe pas possède la valeur *null* . La séquence suivante provoque une erreur :

```
Compteur c3=null;
c3.raz();
```

Résultat d'exécution :

```
Exception in thread "main" java.lang.NullPointerException
```

On invoque en effet une méthode *raz* sur un objet inexistant.

public, protected, private

Comme C++ Java dispose de plusieurs niveaux de protections des classes, des services ou des données. D'un façon générale *public* désigne une entité qui est accessible à toute autre entité : cela peut concerner la classe elle-même ou un membre de la classe (donnée, méthode ou autre déclaration (classe interne)).

protected désigne une entité dont l'accès sera réservé à la classe elle-même, à ses sous-classes ou aux autres membres de son paquetage.

private désigne une entité dont l'accès est strictement réservé à la classe elle-même.

Il convient d'ajouter un niveau supplémentaire de protection qui s'exprime en n'apposant aucune des mentions qui précèdent : dans ce cas, appelé <*default*> dans la suite, l'accès à l'entité n'est possible que pour les entités du *package* courant.

Protection des classes

Pour définir sa visibilité la déclaration d'une classe peut être qualifiée par le mot *public* ou par l'absence de mot. Les mots réservés *private* ou *protected* ne sont pas applicables pour une déclaration de classe.

Une classe déclarée *public* sera visible et donc utilisable de n'importe quel package (à condition bien entendu de préfixer le nom de la classe par le chemin de package qui y mène)

```
public class A { etc... } // visible de partout
```

Une classe ne spécifiant pas, comme ci-dessous, de clause de protection ne sera visible que dans son propre package.

Exemple :

```
// fichier A.java dans le paquetage test
package test;
class A {
   int n=5;
}
class B {
   A a=new A();
   B() { a.n=7; }
}
// fichier C.java dans le paquetage test
package test;
public class C {
   A a=new A();
   C() { a.n=11; }
}
// fichier Launcher.java dans un autre  paquetage
import test;
public class Launcher {
```

```
   A a=new A();  // ERREUR : A cannot be resolved to a type
}
```

Protection des membres

Les membres d'une classe (méthodes, données, classes internes) peuvent être qualifiés avec les quatre niveaux, le dernier consistant à ne rien apposer : *public, protected, private* et <*default*>.

Niveau de protection d'un membre	Accès depuis la classe elle-même	Accès depuis une entité du même package	Accès depuis une sous classe (dans un autre package)	Accès depuis une entité d'un autre package sans relation d'héritage
public	oui	oui	oui	oui
protected	oui	oui	oui	non
<default>	oui	oui	non	non
private	oui	non	non	non

L'exemple ci-dessous illustre, dans le cas d'un accès à un champ de donnée, les conséquences de ces règles.

```
// fichier A.java ; package test
package test;
public class A {
   public int n1;
   protected int n2;
   int n3;
   private int n4;
   public A() {
      n1=7; n2=7;n3=7;n4=7;
   }
}
// fichier C.java ; package test
package test;
public class C {
   A a=new A();
   C() {
      a.n1=7;
      a.n2=7;
      a.n3=7;
      a.n4=7; // ERREUR : The field A.n4 is not visible
   }
}
// fichier Launcher.java ; autre package
import test.A;
class D extends A {
   D() {
      n1=7;
      n2=7;
      n3=7; // ERREUR : The field A.n3 is not visible
      n4=7; // ERREUR : The field A.n4 is not visible
   }
}
public class Launcher {
   public static void main(String[] args) {
      A a=new A();
      a.n1=7;
      a.n2=7; // ERREUR : The field A.n2 is not visible
      a.n3=7; // ERREUR : The field A.n3 is not visible
      a.n4=7; // ERREUR : The field A.n4 is not visible
   }
}
```

Il n'y a pas en java de mécanisme permettant d'attribuer des privilèges d'accès d'une classe vers une autre classe (comme l'exportation sélective en Eiffel, ou les classes amis (*friend*) du C++).

Néanmoins une classe possède un accès privilégié aux instances d'elle-même (pas seulement *this*).

Dans l'exemple suivant la méthode *compareTo* de la classe *A* reçoit une instance extérieure à travers le paramètre formel *a* et accède à l'ensemble de ses données internes, y compris *n4* qui est *private*.

```
package test;
public class A implements Comparable <A>{
   public int n1;
   protected int n2;
   int n3;
   private int n4;
   public A() {
      n1=7; n2=7;n3=7;n4=7;
   }
   public int compareTo(A a) {
      return this.n1==a.n1 && this.n2==a.n2 && this.n3==a.n3 && this.n4==a.n4 ? 0 : 1;
   }
}
```

Notons enfin qu'il est possible que, dans une sous-classe, une méthode qui redéfinit une méthode héritée conserve explicitement (les mots *public* ou *protected* dans ce cas ne peuvent être omis) la visibilité de cette dernière (sauf quand cette dernière est *private* bien entendu). Il est néanmoins possible pour cette méthode d'élargir la visibilité d'une méthode héritée (de *protected (*ou *<default>)* à *public* par exemple). Il n'est en revanche pas possible de réduire la visibilité d'une méthode héritée.

Initialisation

Initialisation de variable locale

Les erreurs d'initialisation concernant les variables locales remontent en Java au niveau du compilateur. Ainsi :

```
public class Launcher {
   public static void main(String[] args) {
      Compteur c3;
      c3.raz(); // erreur du compilateur sur c3 : variable non initialisée
   }
}
```

Initialisation de variable membre

Les variables membres d'une classe font l'objet, en l'absence d'initialisation explicite, d'une initialisation par défaut. Il s'agit d'une valeur nulle pour les types numériques, de *false* pour le type *boolean* et de *null* pour les types objet.

Une variable membre peut être initialisée explicitement lors de sa déclaration en tant que variable membre (ce qui n'est pas possible en C++) ou dans le constructeur. Dans la première hypothèse l'initialisation de la variable membre intervient avant l'appel du constructeur.

```
// exemple 1 : initialisation de value
class Compteur {
   protected int value=0;   // initialisation directe
   public Compteur() {}
   public void up() {  value++; }
   public void raz() { value = 0; }
   public int getValue() { return value; }
   public String toString() { return "" + value;}
}
// exemple 2 : initialisation de value
class Compteur {
   protected int value;
   public Compteur() { value = 0; } // initialisation dans le constructeur
   public void up() {  value++; }
   public void raz() { value = 0; }
   public int getValue() { return value; }
   public String toString() { return "" + value;}
}
```

Package et classe

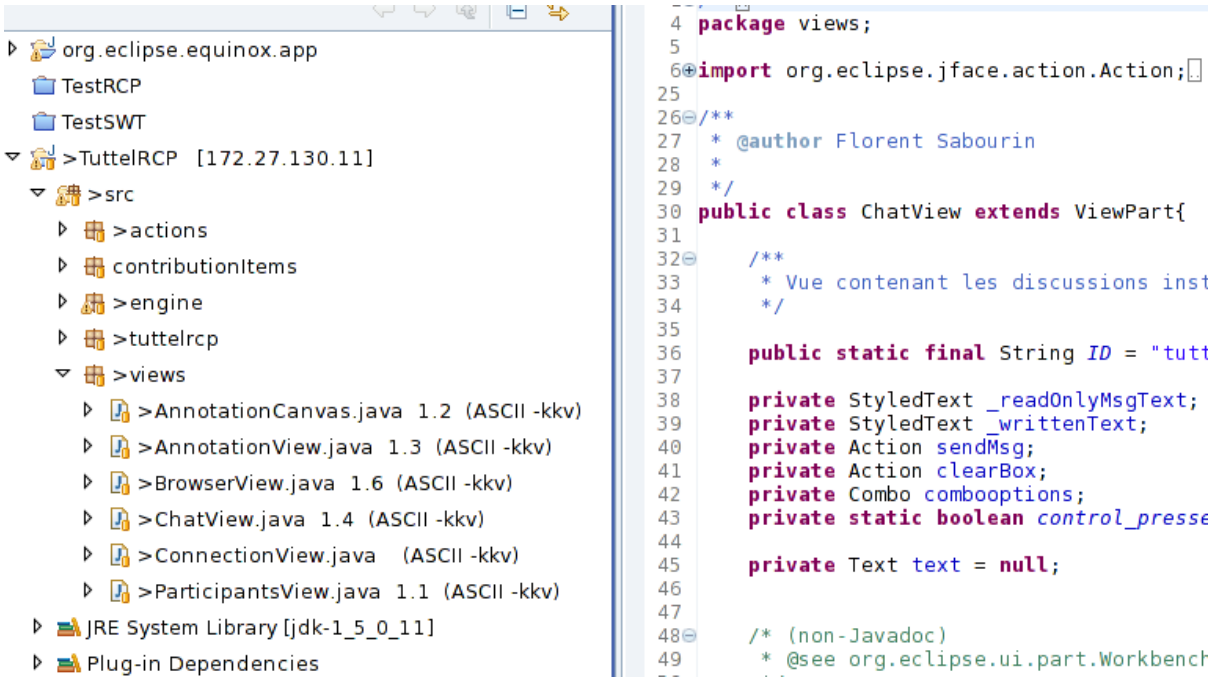
La déclaration d'une classe doit en général être précédée du mot *public*. Cela reste facultatif pour le cas où la classe ne doit être visible que dans le contexte du fichier ou du package où elle a pris place.

Une classe usuelle, déclarée *public* , doit prendre place dans le fichier qui porte son nom.

```
// Fichier : Compteur.java
public class Compteur {
   protected int value;
```

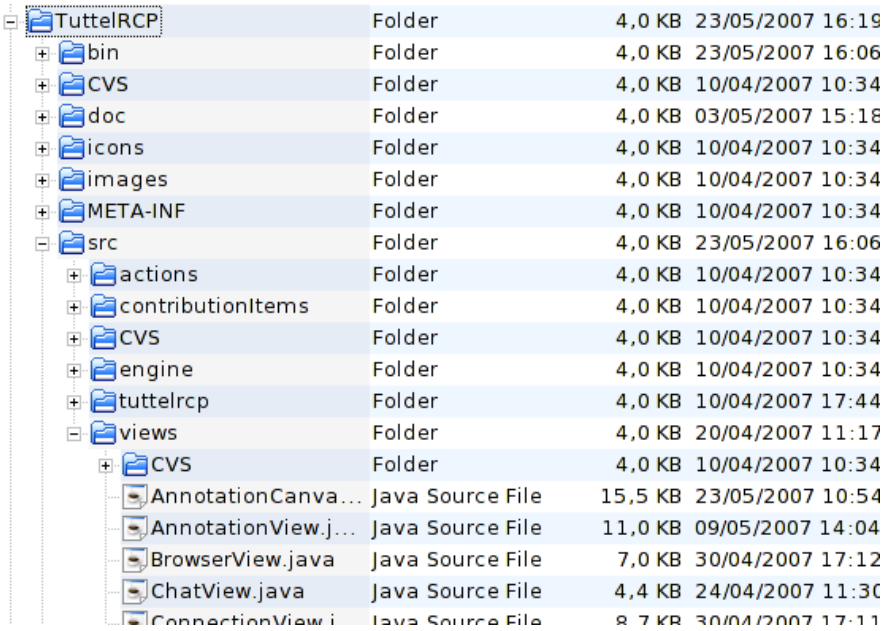
```
  public Compteur() { value = 0; }
  public void up() {  value++; }
  public void raz() { value = 0; }
  public int getValue() { return value; }
  public String toString() { return "" + value;}
}
```

En outre un système de nommage hiérarchique est imposé par java pour l'organisation des bibliothèques de classes. Dans la vue ci-dessous (une portion de l'environnement de développement *Eclipse*) d'un projet typique, les *packages* java (*actions, contributionitems, tuttelrcp, views,engine...)* apparaissent sous une forme hiérarchisée. A droite on voit que la classe *ChatView* déclare (par *package views*) en en-tête le nom du package auquel elle appartient.



Cette hiérarchisation des *packages* s'exprime également par une organisation hiérarchisée des fichiers et des répertoires correspondants. Les nom de *package* sont des noms de répertoires (généralement ils sont écrits entièrement en minuscules). Les fichiers *java* sont les feuilles de cette arborescence et comme il y a homonymie forcée entre un nom de fichier source (*.java*) et compilé (*.class*) et la classe correspondante sera entièrement référençable par le chemin qui suit cette hiérarchie d'entité.

Au plan du système de fichier le projet ci-dessus présente l'aspect suivant:



On y reconnaît le fichier *ChatView* et son positionnement.

A l'intérieur du projet courant, la classe *ChatView* peut invoquer une autre classe, par exemple la classe *ChatEngine* située dans le package *Engine*, à condition de préfixer le nom de la classe invoquée avec le nom du chemin hiérarchisé qui y mène.

```
public class ChatView extends ViewPart{
   public static final String ID = "tuttelrcp.chatview";
   private engine.ChatEngine chatEngine;  // engine est le nom du package
   private StyledText _readOnlyMsgText;
   private StyledText _writtenText;
   ...
}
```

Clause import

On préfère généralement, pour alléger le code et donc en améliorer la lisibilité, poser une fois pour toute les préfixes nécessaires pour les identifiants qu'on utilise au moyen de la clause *import*.

```
import engine.ChatEngine;
public class ChatView extends ViewPart{
   public static final String ID = "tuttelrcp.chatview";
   private ChatEngine chatEngine;  // engine est le nom du package
   private StyledText _readOnlyMsgText;
   private StyledText _writtenText;
   ...
}
```

Les environnement de développement permettent généralement d'automatiser la pose de ces clauses d'importations.

Tableaux

Comme la plupart des langages de programmation, Java propose une structure de données à accès indicé appelé tableau. C'est une structure de base du langage, indépendante de toute bibliothèque, et possédant sa syntaxe spécifique caractérisée par exemple par l'accès utilisant l'opérateur *[ ],* la consultation de la taille par *length* (sans parenthèse) ou encore la capacité à gérer les types natifs du langage (*int, double, boolean* etc..)

Néanmoins les tableaux de Java relèvent également d'une certaine logique objet puisque (à l'inverse de ce qui se passe en C++ par exemple) leur instanciation doit être explicite. Un tableau en Java doit être instancié préalablement au garnissage de ses éléments. Ces derniers, lorsqu'il ne sont pas issus d'un type de base, doivent être instanciés à leur tour.

Les tableaux, parce que leur logique d'accès aux éléments contenus exploite les instructions d'accès indicé du processeur, sont une structure de données efficace, mais pourvu de certaines limitations, comme l'impossibilité du redimensionnement après la création.

Le code ci-dessous crée deux tableaux :

- un tableau d'*int*. Il est instancié explicitement, mais chacune de ses cases est instanciée implicitement (avec un entier nul) parce qu'il s'agit d'un type de base , puis ensuite initialisée avec le carré des indices successifs
- un tableau de *date*. Il est instancié explicitement, et chacune de ses cases contient à ce stade une référence *null*. Il faut ensuite procéder explicitement à l'instanciation de chacune des dates.

```
final int MAX=100;
int [] tab1=new int[MAX]; // Le tableau est créé et 100 entiers nuls ont été créés
for(int i=0;i<MAX;i++) tab1[i]=i*i; // les valeurs nulles sont remplacées

Date [] tab2=new Date[MAX]; // Le tableau est créé mais aucun objet Date n'a été créé
for(int i=0;i<MAX;i++) tab2[i]=new Date(); // 100 objets Date ont été créés
```

Une fois créé un tableau ne peut pas être redimensionné. Dans le cas où un tel redimensionnement dynamique serait nécessaire des objets conteneurs spécifiques, tels que *ArrayList* proposé par la bibliothèque *JCF*, devraient être préférés.

Comparaison avec le C

Contrairement au langage C la taille d'un tableau ne doit pas en Java nécessairement être connue dès la compilation. Même s'il n'est pas redimensionnable la taille d'un tableau Java peut être déterminée à l'exécution seulement.

Dans cet exemple la taille du tableau est calculée à l'exécution sur une base aléatoire.

```
int [] tab=new int[(int)Math.round(Math.random()*100+1)];
System.out.println("Taille de tab ="+tab.length);
```

Résultat d'exécution (exemple):

```
Taille de tab =18
```

Cette souplesse de création ne peut être atteinte en C ou C++ qu'au moyen d'un tableau alloué dynamiquement . Un déclaration équivalent à celle ci-dessus serait donnée par :

```
int * tab1=new int[1+rand()*100];      // version C++
int * tab2=(int *)malloc((1+rand()*100)*sizeof(int));      // version C
```

Il faut noter que ces versions ne sont équivalentes que du point de vue de l'allocation mémoire. Un tableau en Java est une structure plus complexe qu'en C ou C++. En particulier, un tableau Java connaît son nombre d'éléments et est une structure capable de détecter ses débordements d'index.

D'une façon générale la création d'un tableau Java sur type de base ou sur classe possède les équivalents approximatifs suivant en C++ (on remarquera les variations du niveau d'indirection entre le type de base et la classe explicitées dans le cas du C++)

Java	C++
`int [] tab;`	`int * tab=0;`
`int [] tab=new int[100];`	`int * tab= new int[100];`
`A [] tab;`	`A ** tab=0;`

`A [] tab=new A[100];`	`A ** tab=new A*[100];`

Initialisation d'un tableau

Du fait de sa nature de structure de base du langage le tableau Java jouit de quelques privilèges syntaxiques, notamment pour l'initialisation.

```
int [] tab3={0,1,2,3,4,5,6,7,8,9 };
Date [] tab4={null,new Date(),null};
int [][] tab5={{1,2},{5,6}};
int [][][] tab6={tab5,tab5}; // attention aux effets de bords ; tab5 n'est pas dupliqué
System.out.println(Arrays.deepToString(tab6));
tab6[1][1][1]=14;
System.out.println(Arrays.deepToString(tab6));
```

Sortie :

```
[[[1, 2], [5, 6]], [[1, 2], [5, 6]]]
[[[1, 2], [5, 14]], [[1, 2], [5, 14]]]
```

Dans cet exemple nous avons utilisé une facilité offerte par la classe *Arrays* concernant la manipulation de tableau.

Débordements

Un tableau Java connaît le nombre d'éléments qui lui est attribué à sa création et celui-ci est accessible par *length* (*length* n'est pas une méthode mais une construction syntaxique propre aux tableaux).

En cas de débordement une exception *ArrayIndexOutOfBoundsException* est levée.

```
int [] tab=new int[(int)Math.round(Math.random()*10+1)];
System.out.println("Taille de tab ="+tab.length);
try {
   for(int i=0;i<100;i++) tab[i]=i;
}
catch (Exception e) {
   e.printStackTrace();
}
for(int elt:tab) System.out.print(" "+elt);
```

Résultat d'exécution (exemple) :

```
Taille de tab =5
java.lang.ArrayIndexOutOfBoundsException: 5
   at Launcher.main(Launcher.java:71)
 0 1 2 3 4
```

Manipulation d'un tableau avec *Arrays*

Le package *java.util* propose avec la classe *Arrays* (à la façon de la classe *Collections* de la *JCF*, ou de la bibliothèque *algorithm* de la *STD C++*) une boite à outils constituée d'un grand nombre de méthodes statiques permettant tout une gamme de traitements courants des tableaux : tri, parcours, conversion, affectation de valeurs, recherche d'élément etc...

```
public class Arrays {
    public static void sort(int[] a)
    public static void sort(int[] a, int fromIndex, int toIndex)
    public static void sort(Object[] a, int fromIndex, int toIndex)
    public static <T> void sort(T[] a, Comparator<? super T> c)
    public static int binarySearch(int[] a, int key)
    public static <T> int binarySearch(T[] a, T key, Comparator<? super T> c)
    public static boolean equals(int[] a, int[] a2)
    public static void fill(long[] a, int fromIndex, int toIndex, long val)
    public static void fill(Object[] a, Object val)
    public static void fill(Object[] a, int fromIndex, int toIndex, Object val)
    public static <T> List<T> asList(T... a)
    public int size()
    public Object[] toArray()
    public E get(int index)
    public E set(int index, E element)
    public int indexOf(Object o)
    public boolean contains(Object o)
```

```
    public static int hashCode(long a[])
    public static int deepHashCode(Object a[])
    public static boolean deepEquals(Object[] a1, Object[] a2)
    public static String deepToString(Object[] a)
   etc...
}
```

Le code suivant convertit un tableau de *Date* en liste et l'affiche:

```
Date [] tab4={null,new Date(),null};
List<Date> list=Arrays.asList(tab4);
for(Date date:list) System.out.println(date);
```

Sortie :

```
null
Fri Jul 06 12:39:34 CEST 2007
null
```

Le code suivant trie un tableau de chaînes de caractères:

```
String [] tab4={"hello","salut","bonjour"};
for(String s:tab4) System.out.println(s);
System.out.println();
Arrays.sort(tab4);
for(String s:tab4) System.out.println(s);
```

Sortie :

```
hello
salut
bonjour

bonjour
hello
salut
```

Tableau et généricité

Un tableau est nativement générique dans le sens où il est possible de spécifier le type des données gérées par ce tableau, y compris lorsqu'il s'agit d'un type de base.

```
int [] t1;
Date [] t2;
```

Mais la syntaxe utilisée n'est pas celle de la généricité qui a été introduite avec la version 1.5 du langage. En conséquence les tableaux Java présentent une certaine incompatibilité avec les types de données génériques au sens de cette généricité récente. En particulier les tableaux Java sont covariants (voir paragraphe Covariance des tableaux) alors que les classes génériques ne le sont pas (voir Héritage et Généricité).

Ainsi la déclaration suivante est correcte:

```
LinkedList<Integer> [] tabList;
```

Mais l'instanciation correspondante est incorrecte :

```
LinkedList<Integer> [] tabList=new LinkedList<Integer>[MAX];
        // Erreur : Cannot create a generic array of LinkedList<Integer>
```

Cette difficulté peut être contournée par l'utilisation de la version non générique de la classe générique problématique :

```
LinkedList<Integer>[] tabList=new LinkedList[MAX];     // OK
```

On dispose alors avec *tabList* d'une variable pourvue de toute la sémantique de sa classe déclarative. Exemple :

```
final int MAX = 5;
LinkedList<Integer>[] tabList=new LinkedList[MAX];
for(int i=0;i<MAX;i++) tabList[i]=new LinkedList<Integer>();
for(LinkedList<Integer> list:tabList) {
     list.add((int)(Math.random()*100)); // autoboxing
     list.add((int)(Math.random()*100)); // autoboxing
}
System.out.println(Arrays.deepToString(tabList));
```

La sortie montre bien qu'on a construit un tableau de 5 listes de 2 éléments :

```
[[2, 17], [41, 62], [85, 22], [63, 68], [24, 80]]
```

Ce type de limite concernant l'utilisation de générique est propre aux tableaux: les classes conteneurs de la *Java Collection Framework,* elles-mêmes génériques, ne présentent pas ce problème.

Covariance des tableaux

Pour une classe générique la covariance consisterait dans le fait qu'une classe générique suive le sous-typage qui résulte de l'instanciation de ses types génériques. Les classes génériques en Java ne sont pas covariantes : ainsi une *LinkedList<EtudiantSportif>* n'est pas un sous type de *LinkedList<Etudiant>* (en supposant bien sûr ici qu'*EtudiantSportif* hérite de *Etudiant*) . Ce point est abordé dans *Généricité et héritage*.

Au contraire les tableaux possèdent cette propriété de covariance. Un tableau d'*EtudiantSportif* peut être présenté là où on attend un tableau d'*Etudiant*.

Le programme suivant illustre cette différence de comportement entre tableau et classe générique.

```
public class Launcher {
   public static void main(String[] args) {
      String [] tab=new String[100];
      LinkedList<String> list;
      m1(tab);
      m2(list);  // erreur : The method m2(LinkedList<Object>) in the type Launcher is not
                 // applicable for the arguments (LinkedList<String>)
      m3(list);
   }
   static public void m1(Object[] tab) {
      for(Object o:tab) System.out.println("$"+o);
   }
   static public void m2(LinkedList<Object> list) {
      for(Object o:list) System.out.println("$"+o);
   }
   static public void m3(LinkedList<?>  list) {
      for(Object o:list) System.out.println("$"+o);
   }
}
```

L'appel *m1(tab)* passe une variable de type *String[]* là où la méthode *m1* attend une variable de type *Object[]*. Cet appel est accepté, ce qui signifie que du point de vue du compilateur *String[]* est un sous-type de *Object[]*.

La même expérience effectuée avec une classe générique (*LinkedList*) provoque une erreur qui révèle que du point de vue du compilateur une *LinkedList<String>* ne peut pas être fournie à la place de la *LinkedList<Object>* attendue. Autrement dit une *LinkedList<String>* n'est pas un sous-type de *LinkedList<Object>*, malgré le fait que *String* est bien sûr un sous type d'*Object*. La notation basée sur le *wildcard "?"* a été introduite pour pallier à la rigidité que cette absence de sous typage implique (voir le chapitre sur la généricité)

Ellipse

Java permet depuis sa version 5 la spécification de méthodes à nombre de paramètres variables et de types variables.

En *C* la fonction *printf* est une illustration typique de l'utilité de ce mécanisme, au prix de l'abandon du contrôle de type sur les paramètres concernés.

printf("salut");

printf("%s %d %s","J'ai",10,"ans");

Java propose une formulation qui permet à la fois un certain contrôle de type et une certaine souplesse comme l'illustre la portion de code ci-dessous.

```
public class Launcher {
   static <T> void print(T ...list) {
      System.out.println("list : "+list.length+ " elements");
      for(T t:list) System.out.println(t + " : "+ t.getClass());
   }
   static  void printInt(int ...list) {
      System.out.println("list : "+list.length+ " elements");
      for(int t:list) System.out.println(t );
```

```
   }
   public static void main(String [] args) {
      printInt(5,6,7);
      printInt(5,"hello",7); // Erreur "hello" n'est pas un int
      print("salut","hello","bonjour");
      print(4,7,"bonjour",new Date());
   }
}
```

La sortie est la suivante (une fois ôtée la ligne en erreur):

```
list : 3 elements
5
6
7
list : 3 elements
salut : class java.lang.String
hello : class java.lang.String
bonjour : class java.lang.String
list : 4 elements
4 : class java.lang.Integer // autoboxing
7 : class java.lang.Integer // autoboxing
bonjour : class java.lang.String
Mon Jul 09 17:38:10 CEST 2007 : class java.util.Date
```

Chaînes de caractères

Les chaînes de caractères permettent à un programme de manipuler l'information symbolique, c'est à dire constituée de caractères alphabétiques. En *C* les chaînes de caractères sont constituées des simples tableaux de *char* (dont l'équivalent est en Java le type *byte*) et le langage lui-même ne fournit que quelques facilités syntaxiques destinées à en faciliter un peu l'usage. Il ne s'agit donc pas réellement d'une construction spécifique.

En Java il en est un peu de même : une chaîne de caractères est banalement une instance de la classe *String*, et de même, Java propose quelques facilités pour en rendre l'usage plus commode.

Ainsi la création d'une chaîne et son initialisation peuvent se passer de l'opérateur *new*:

```
String msg="hello"; // équivaut à String msg=new String("hello");
```

D'autres facilités sont offertes, notamment l'usage de l'opérateur de concaténation **+**, ou sa variante **+=**, qui est atypique sur une instance (Java ne permet pas encore la surcharge d'opérateur).

```
msg=msg+"les amis";
```

La conversion automatique des types de bases fait partie de la panoplie:

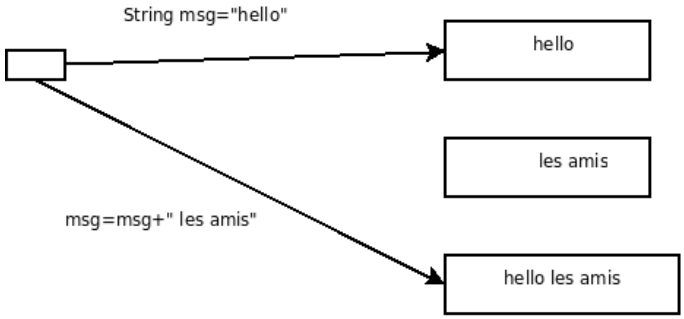
```
msg="La route aux " + 4 + " chansons";
```

Des instances inaltérables

Le point le plus remarquable concernant les chaînes de caractères gérées avec le type *String* est le caractère inaltérable des instances. Une fois créée, une chaîne de caractères ne change plus.

La séquence présentée précédemment semble contradictoire avec cette affirmation puisqu'on y voit la variable prendre deux valeurs successives différentes : "*hello"* puis *"hello les amis"*. En réalité chacune des ces chaînes est une instance différente, et inaltérable, de la casse *String* : ce qui varie en réalité ici est le fait que la variable *msg* référence successivement ces deux objets distincts dans la mémoire.

- Structure générale d'un programme



Un équivalent proche de la portion de code Java

```
String msg="hello";
msg=msg+" les amis";
```

est en langage C :

```
char *msg =(char *)malloc(6);
strcpy(msg,"hello);
char *aux1=(char *)malloc(8);
strcpy(aux1," les amis");
char *aux2=(char *)malloc(13);
strcpy(aux2,msg);
strcat(aux2,aux1);
free(msg);
free(aux1);
msg=aux2;
```

Cet exemple montre qu'en Java, c'est l'allocation dynamique qui est la forme exclusive d'allocation en mémoire pour les objets (mais non pour les types de base). En outre, et c'est évidemment un point fondamental, là où en *C* il faut se soucier d'éliminer les blocs mémoires désormais inutilisés (ceux contenant *"hello"* et *" les amis"*) on voit qu'en Java ce travail n'apparaît pas : c'est en effet le *garbage collector* (qu'on traduit par *ramasse miettes*) qui s'en chargera de façon transparente au programme et au programmeur.

Ce qui vient d'être exposé constitue le modèle de fonctionnement des chaînes de caractères en Java. Il est basé sur:

- des instances de chaînes inaltérables
- un grand nombre, même pour une opération simple, d'opérations d'allocations de blocs en mémoire (dans le tas) lors de l'exécution
- un *garbage collector* qui élimine les nombreuses et inévitables scories en mémoire engendrées par ce type de fonctionnement

Le coût néanmoins de ce mode de fonctionnement n'est pas aussi élevé qu'il y paraît : les chaînes étant inaltérables elles pourront sans problème et sans effets de bords indésirables être utilisées simultanément par plusieurs threads d'exécution. En outre leurs caractéristiques d'objets accessibles en lecture seule permet à la machine virtuelle de procéder à certaines optimisations. Le compilateur peut également optimiser l'exécution en remplaçant lorsque c'est possible une série d'allocations/libérations par la transformation directe des chaînes en interne au moyen des méthodes fournies par *StringBuilder*. Enfin, et compte tenu de l'importance de de mode d'allocation dans le modèle d'exécution de Java, le contexte d'exécution fourni par la machine virtuelle est optimisé pour rendre l'opération d'allocation dynamique aussi efficace que possible.

La classe *String* et ses méthodes

La classe *String* propose une grande variété de méthodes destinées à la manipulation des chaînes. Elles ont toutes un point commun : elles ne modifient pas (à l'exception des constructeurs) l'état de l'instance, c'est à dire de la chaîne de caractères, à laquelle elles s'appliquent. C'est ce fait qui détermine le caractère inaltérable des instances de *String*.

Puisqu'on ne peut pas modifier une chaîne de caractère les changements de valeur ne peuvent donc avoir lieu que par la construction d'une nouvelle chaîne retournée par les méthodes concernées.

Ainsi, formulée comme ci-dessous, l'extraction d'une sous-chaînes d'une chaîne donnée est un échec :

```
String msg="salut les amis";
msg.substring(10,14);
```

```
System.out.println(msg);
```

Sortie :

```
salut les amis                    // msg est resté inchangée
```

L'effet désiré (ici l'extraction du mot amis) doit s'obtenir ainsi :

```
String msg="salut les amis";
msg=msg.substring(10,14);
System.out.println(msg);
```

Sortie :

```
amis
```

Dans le première version l'appel à *substring* est donc inutile dans la mesure où la valeur de retour est ignorée.

Les méthodes citées ci-dessous à un seul exemplaire existent sous la forme de différentes variantes dans la classe *String*.

```
public class String {
String()
String(byte[] bytes)
String(char[] value)
String(String original)
String(StringBuffer buffer)
String(StringBuilder builder)
char  charAt(int index)
int   codePointAt(int index)
int   codePointBefore(int index)
int   codePointCount(int beginIndex, int endIndex)
int   compareTo(String anotherString)
int   compareToIgnoreCase(String str)
String  concat(String str)
boolean  contains(CharSequence s)
boolean  contentEquals(CharSequence cs)
boolean  contentEquals(StringBuffer sb)
static String    copyValueOf(char[] data)
boolean  endsWith(String suffix)
boolean  equals(Object anObject)
boolean  equalsIgnoreCase(String anotherString)
static String    format(Locale l, String format, Object... args)
static String    format(String format, Object... args)
byte[]  getBytes()
void  getChars(int srcBegin, int srcEnd, char[] dst, int dstBegin)
int   hashCode()
int   indexOf(String str, int fromIndex)
boolean  isEmpty()
int   lastIndexOf(String str)
int   lastIndexOf(String str, int fromIndex)
int   length()
boolean  matches(String regex)
boolean  regionMatches(boolean ignoreCase, int toffset, String other, int ooffset, int len)
boolean  regionMatches(int toffset, String other, int ooffset, int len)
String  replaceAll(String regex, String replacement)
String  replaceFirst(String regex, String replacement)
String[]   split(String regex, int limit)
boolean  startsWith(String prefix)
String  substring(int beginIndex, int endIndex)
char[]  toCharArray()
String  toLowerCase()
String  toString()
String  toUpperCase()
String  trim()
static String    valueOf(boolean b)
static String    valueOf(char c)
static String    valueOf(char[] data)
static String    valueOf(char[] data, int offset, int count)
static String    valueOf(double d)
```

```
static String    valueOf(float f)
static String    valueOf(int i)
static String    valueOf(long l)
static String    valueOf(Object obj)
}
```

La classe *StringBuilder*

Les limites du modèle *String* peuvent être contournées par l'usage de *StringBuilder*. Il s'agit d'une classe manipulant une chaîne de caractères qui peut cette fois, à la différence de *String*, être altérable. Elle pourra être utilisée dans tous les contextes où le modèle d'allocations multiples mis en oeuvre par *String* se révélera trop coûteux dans le contexte d'exécution du programme. Il faut néanmoins garder à l'esprit, avant de faire l'effort d'implémentation nécessaire pour passer à *StringBuilder*, que le compilateur procède avec les *String* classiques à des optimisations qui peuvent consister notamment à utiliser *StringBuilder* en interne.

La classe *StringBuilder* n'est pas protégée contre le multithreading à l'inverse de la classe *StringBuffer* qui avait été proposée auparavant, mais, pour cette raison précisément, la classe *StringBuilder* est plus efficace dans un contexte d'utilisation non partagé.

La classe *StringBuilder* permet donc l'insertion (*insert*), la concaténation (*append*), le remplacement (*replace*), la modification (*setAt*) en tout ou partie de la chaîne instanciée. L'usage de *StringBuffer* est très semblable à celui de *StringBuilder*.

La liste des méthodes ci-dessous est indicative et non exhaustive.

```
public class StringBuilder {
StringBuilder()
StringBuilder(int capacity)
StringBuilder(String str)
StringBuilder    append(boolean b)
StringBuilder    append(char c)
StringBuilder    append(char[] str)
StringBuilder    append(char[] str, int offset, int len)
StringBuilder    append(double d)
StringBuilder    append(float f)
StringBuilder    append(int i)
StringBuilder    append(long lng)
StringBuilder    append(Object obj)
StringBuilder    append(String str)
StringBuilder    append(StringBuffer sb)
StringBuilder    appendCodePoint(int codePoint)
int  capacity()
char charAt(int index)
int  codePointAt(int index)
int  codePointBefore(int index)
int  codePointCount(int beginIndex, int endIndex)
StringBuilder    delete(int start, int end)
StringBuilder    deleteCharAt(int index)
void ensureCapacity(int minimumCapacity)
void getChars(int srcBegin, int srcEnd, char[] dst, int dstBegin)
int  indexOf(String str)
int  indexOf(String str, int fromIndex)
StringBuilder    insert(int offset, boolean b)
StringBuilder    insert(int offset, char c)
StringBuilder    insert(int offset, char[] str)
StringBuilder    insert(int index, char[] str, int offset, int len)
StringBuilder    insert(int offset, double d)
StringBuilder    insert(int offset, float f)
StringBuilder    insert(int offset, int i)
StringBuilder    insert(int offset, long l)
StringBuilder    insert(int offset, Object obj)
StringBuilder    insert(int offset, String str)
int  lastIndexOf(String str)
int  lastIndexOf(String str, int fromIndex)
int  length()
int  offsetByCodePoints(int index, int codePointOffset)
StringBuilder    replace(int start, int end, String str)
StringBuilder    reverse()
```

```
void setCharAt(int index, char ch)
void setLength(int newLength)
String  substring(int start)
String  substring(int start, int end)
String  toString()
void trimToSize()
}
```

Exemple :

```
StringBuilder msg=new StringBuilder("salut les amis");
msg.replace(0,10,"");
System.out.println(msg);
```

Sortie :

```
amis
```

Héritage

L'héritage joue un rôle clé en orienté objet : c'est une technique permettant d'adapter ou de spécialiser des classes existantes, mais, au-delà de cet aspect, il apparaît que c'est également un moyen puissant d'organiser le domaine de connaissance associé au problème.

Combiné au polymorphisme, et aux notions de classes abstraites ou d'interfaces, cette technique est au coeur des architectures applicatives en Java.

Héritage par extension

L'héritage par extension, introduite par le mot réservé *extends* en Java, consiste à adapter une classe existante en la complétant ou en la spécialisant. La classe héritière hérite des membres (méthodes et données) de sa classe mère et accède à ceux d'entre-eux qui portent la mention *public* ou *protected*.

Une classe a la possibilité d'empêcher toute possibilité d'héritage en se déclarant *final*. De même une classe peut bloquer la redéfinition d'une méthode dans une sous-classe au moyen de clause *final* apposée cette fois sur le nom de la méthode.

Toutes les classe Java ont un ancêtre commun et implicite : la classe *Object*.

Héritage et super-invocation

Dans l'exemple qui suit la *classe CompteurDeb* spécialise *Compteur* en en redéfinissant en particulier les méthodes *up* et *raz.* Un *CompteurDeb* se distingue d'un *Compteur* dans la mesure où il est capable de gérer une valeur maximale (*999*, incarnée par la constante *MAX* en l'occurrence). Pour informer l'utilisateur de l'état d'éventuel dépassement de capacité atteint par le *CompteurDeb,* la méthode *getDeb* retourne à ce dernier la valeur du booléen interne d'état. Une instance de *CompteurDeb* occupe un espace plus important qu'une instance de *Compteur* puisqu'elle contient les variables membres de *Compteur*, c'est à dire *value*, et celle en propre de *CompteurDeb*, c'est à dire *deb*.

Un *CompteurDeb* est une sorte de *Compteur*. Pour se construire un *CompteurDeb* se construit comme un *Compteur* puis complète le travail en ajoutant l'initialisation de *deb* qui, elle, n'existe pas dans *Compteur.* C'est exactement ce qu'exprime le constructeur de *CompteurDeb* par *super();deb=false;*. Tout comme *this* exprime dans une définition de méthode l'objet courant, *super* exprime la part de l'objet courant qui relève de la classe mère (cette notion est d'ailleurs récursive dans le cas d'une arborescence d'héritage). Dans le contexte d'un constructeur *super()* exprime l'appel au constructeur de la classe mère pour la construction de la partie *super* de l'objet courant.

Dans le contexte de la méthode *up* l'expression *super.up()* signifie qu'on demande l'invocation de *up* à la façon de la classe mère sur l'instance courante.

Compte tenu du caractère *private* de *value* cette variable membre n'est pas directement accessible depuis la classe héritière. Néanmoins les méthodes de *Compteur* qui la manipule étant *public* on y accède indirectement à travers cette dernière. C'est le principe d'encapsulation des données au sein d'un objet : l'accès à ces données depuis l'extérieur n'est possible qu'à travers les méthodes que la classe met à la disposition de l'entité externe. La classe peut alors se poser en garante de la cohérence interne de ses données.

```
class Compteur {
   private int value;
```

```
   public Compteur() { value=0;}
   public void up() { value++;}
   public void raz() { value = 0; }
   public int getValue() {  return value; }
   public String toString() { return "" + value;}
}
class CompteurDeb extends Compteur {
   boolean deb;
   CompteurDeb() { super(); deb=false; } // super invocation dans le constructeur
   boolean getDeb() { return deb; }
   public void up() {
      if (getValue()==MAX) deb=true;      // super invocation de up
      else super.up();
   }
   public void raz() {
      super.raz();  // super invocation de raz
      deb=false;
   }
}
```

Héritage et polymorphisme

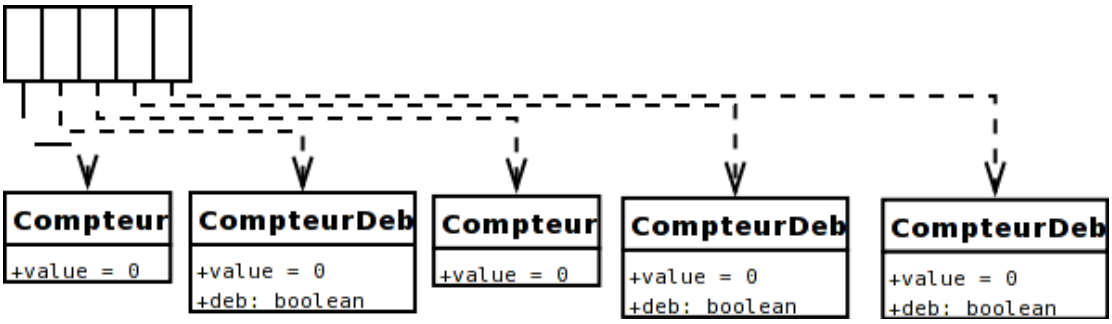
Le polymorphisme caractérise l'aptitude donnée à une variable en Java à référencer des objets d'un type différent que le type déclaratif de la variable. Il est supporté notamment par le mécanisme de liaison retardée exposé ci-après et l'introduction d'une relation de conformité entre types basée sur la notion d'héritage. Ainsi :

```
Compteur c1=new Compteur();
CompteurDeb c2=new CompteurDeb();
c1=c2;  // Ok ; car un CompteurDeb est une sorte de Compteur
        // tout ce que peut faire c1, c2 sait le faire
c2=c1;  // ERREUR : un Compteur n'est pas, en général, un CompteurDeb
```

Complétons les classes ci-dessus par un petit programme de test.

```
public class Launcher {
   public static void main(String[] args) {
      Compteur [] tab=new Compteur[5];
      // création aléatoire de Compteur ou CompteurDeb
      for(int i=0;i<tab.length;i++) {
         if (Math.random()>0.5) tab[i]=new Compteur();
         else tab[i]=new CompteurDeb();
      }
      // action et interrogation de chaque compteur
      for(Compteur c:tab) {
         for(int i=0;i<1500;i++) c.up();
         System.out.println(c.getClass()+" : "+c.getValue());
      }
   }
}
```

Le tableau *tab* est un tableau qui contient 5 compteurs et chacun d'entre-eux peut-être, sur une base aléatoire, soit un *Compteur* soit un *CompteurDeb*. Il pourrait avoir l'allure suivante.



La boucle suivante parcourt ce tableau et invoque 1500 fois la méthode *up* sur chaque compteur. Ceux qui gèrent le débordement (les *CompteurDeb*) resteront bloqués à la valeur *999* tandis que les autres, (les *Compteur*) se positionneront à *1500*.

La sortie donne le résultat suivant:

```
class Compteur : 1500
class CompteurDeb : 999
class Compteur : 1500
class CompteurDeb : 999
class CompteurDeb : 999
```

Ce résultat montre que chaque instance a bien été invoquée, concernant la méthode *up*, sur son type effectif (on dit également dynamique), car ce tableau de *Compteur* contient en fait des *Compteur* ou des *CompteurDeb*. En effet chacune de ces classes a une méthode *up* différente et lors de l'appel *c.up)* le compilateur est dans l'ignorance du type dynamique qui sera celui de *c* au moment de l'exécution. La détermination de l'adresse de *up* est donc retardée du moment de la compilation vers le moment de l'exécution: ce mécanisme est appelé liaison retardée et est à la base du polymorphisme, c'est à dire de la capacité d'une variable déclarée d'un type donné (*c* par exemple, de type déclaratif *Compteur*) à représenter autre chose que son type déclaratif (*CompteurDeb* par exemple) à l'exécution. Il faut noter que dans le contexte Java le polymorphisme est contrôlé par la relation d'héritage : ainsi une variable de type *Compteur* ne peut référencer que des instances de *Compteur* ou de classes qui héritent de *Compteur*.

Le polymorphisme est à la base de la conception d'architecture logicielle ouverte et évolutive.

Une méthode qui renonce à la capacité à être redéfinie dans une sous classe peut être déclarée *final* : dans ce cas le mécanisme de liaison retardée ne sera pas mobilisé pour l'appel de cette méthode.

Le contrôle dynamique de type

La possibilité pour une variable donnée de désigner à l'exécution un objet d'un autre type que son type déclaratif rend nécessaire l'adjonction d'outil permettant d'interroger dynamiquement un objet sur sa classe d'appartenance. Java fournit un ensemble d'outils pour cela : les plus sophistiqués sont les outils de *reflection* qui permettent d'interroger finement une instance sur sa classe et d'interroger également cette dernière. L'exposé de cette bibliothèque sort du cadre de ce paragraphe.

Pour les besoins courants (par exemple sélectionner une action en fonction du type dynamique de l'objet considéré) deux outils sont fournis : le transtypage et l'opérateur *instanceof*.

Transtypage ou cast

Transtypage numérique

Le transtypage numérique consiste à mettre en oeuvre un algorithme de conversion d'un format de donnée numérique vers un autre :

Les types *int* ou *double,* par exemple, diffèrent profondément par le format interne utilisé pour les représenter. Pourtant, d'un point de vue mathématique, un entier est un cas particulier de réel, et les situations où une valeur entière est fournie en tant que valeur réelle sont banales.

```
double x;
int n=5;
x=n;
```

Le compilateur acceptera l'expression *x=n* sans déclencher d'erreur malgré la différence de type. C'est qu'il met en place implicitement un algorithme de conversion qui se chargera d'effectuer la transformation entre le format *int* et le format *double.*

Dans les situations de conversions où la mise en place de ce mécanisme est un peu moins naturel il faudra que le programme procède à un *transtypage* explicite.

```
long n=3;
int n1=(int) 5L;
double x=3;
x=n;
n= x; // Erreur : Type mismatch: cannot convert from double to long
n= (long) x; // OK
```

Transtypage de classe

Cette problématique s'étend également aux instances de classes. La différence toutefois est qu'il s'agit ici d'une opération purement sémantique et donc qu'aucun algorithme de conversion n'est mis en oeuvre. Le

transtypage est utile chaque fois qu'il est nécessaire d'exploiter un objet au-delà des capacités de la variable polymorphe qui le référence.

```
Object object=new Date();
object.getHours(); // Erreur : The method getHours() is undefined for the type Object
```

Dans cet exemple *object* référence une instance de *Date,* ce que le polymorphisme rend possible (une *Date* est après tout une sorte d'*Object*). Mais à travers la variable *object,* connue du compilateur *comme étant de* type *Object,* seules les méthodes de la classe *Object* sont exposées. Le compilateur n'autorisera donc que les invocations des méthodes de cette classe très générale (comme *toString* ou *wait* par exemple).

Dans un contexte où le programmeur est sûr que cette variable *object* possède le type dynamique *Date,* il est possible, grâce au transtypage, d'accéder à une exploitation complète de cette date en disposant d'une variable exposant pleinement la sémantique du type *Date* car déclarée de type *Date*. L'ensemble des méthodes exposées par la classe *Date* sont mainteant accessibles, en particulier *getHours*.

```
Date date=(Date)object;
System.out.println(date.getHours());      // OK
```

Ce transtypage ne consiste pas à transformer l'objet lui-même : il s'agit au contraire d'un jeu d'écriture qui permet de contourner le contrôle de type mis en place par le compilateur et qui agit non pas sur l'objet mais sur la variable qui le référence.

Cette opération, précisément parce qu'elle permet dans une certaine mesure de contourner le contrôle de type du compilateur, est à manier avec parcimonie et prudence. Pour être autorisé le transtypage doit concerner deux classes liées entre elles par une relation directe ou indirecte d'héritage, mais même avec cette limite cela reste une opération à risque.

L'utilisation de la généricité permet d'éviter un recours trop important au transtypage et représente pour cette raison, et la sécurité que cela implique, une avancée importante du langage (version 1.5).

L'opérateur *instanceof*

L'opérateur *instanceof* complète le mécanisme de transtypage en permettant une interrogation dynamique de l'instance sur sa classe d'appartenance.

```
if (object instanceof Date) {
   date=(Date) object;
   h=date.getHours();
}
```

Il faut noter que cet opérateur tient compte de la relation de conformité de type résultant de l'héritage : l'expression *object instanceof Date* évoquée ci-dessus retournera *true*, dans le cas où le type dynamique de *object* est *Date*, bien sûr, ou n'importe quelle classe héritant de *Date*.

Classe abstraite

L'héritage étant aussi un moyen d'organisation des connaissances l'arbre d'héritage constitue souvent le reflet d'une organisatuion sémantique allant du général (le haut de l'arbre) au particulier (les feuilles de l'arbre).

Les classes situées dans les parties hautes de l'arbre ont donc vocation à représenter des catégories plus générales que celles du bas. Cela peut aller, les contraintes du polymorphisme contrôlé par l'héritage aidant, jusqu'à la conception de classes si générales qu'elles en deviennent abstraites : il s'agit alors de classes qui perdent leur prérogative de fabrique à instances pour devenir des pures entités classifiantes. Ces classes abstraites jouent un grand rôle en Java et font l'objet d'une utilisation systématique dans les bibliothèques Java.

Une classe est abstraite à partir du moment où elle possède au moins une méthode abstraite. Une méthode est abstraite lorsqu'elle ne reçoit pas de définition dans sa classe d'appartenance. Le mot réservé *abstract* permet de spécifier l'un et l'autre.

Dans l'exemple ci-dessous la classe *A* est abstraite car elle ne définit pas la méthode *computeInfo*. Celle-ci est en conséquence marquée *abstract*. La classe *A* elle même doit être marquée *abstract* pour que cette situation ne génère pas d'erreur de la part du compilateur. Cette caractéristique fait perdre à *A* la capacité à fabriquer une instance comme l'illustre l'erreur survenue dans le programme de test.

```
abstract class A {
   protected String info;
   public A() { info="general"; }
   public A(String info) { this.info=info; }
```

```
   abstract public void computeInfo();
   public String getInfo() { return info; }
   public String toString() { return info; }
}

public class Launcher {
   public static void main(String[] args) {
      A a; // OK car à ce stade pas de création d'objet
      a=new A(); // ERREUR : Cannot instantiate the type A
   }
}
```

La présence d'un constructeur dans une classe abstraite peut surprendre. Ce constructeur ne prendra son sens que lorsqu'il sera invoqué depuis un constructeur d'une classe héritière concrète.

```
abstract class A {
   protected String info;
   public A() { info="general"; }
   public A(String info) { this.info=info; }
   abstract public void computeInfo();
   public String getInfo() { return info; }
   public String toString() { return info; }
}
class B extends A {
   public B() { super("general B"); } // invocation du constructeur de la surclasse abstraite
   @Override
   public void computeInfo() {
      info+=" "+toString();
   }
}
public class Launcher {
   public static void main(String[] args) {
      A a; // OK car à ce stade pas de création d'objet
      a=new B(); // OK car B est concrète (elle a défini computeInfo)
   }
}
```

La sortie montre bien que le constructeur de *A* a été invoqué depuis le constructeur de *B* :

```
general B
```

La classe *Object*

La classe *Object* est l'ancêtre commune à toutes les classes. C'est une classe concrète, donc apte à produire des instances. Placée au sommet de l'arbre d'héritage, il s'agit d'une classe dont hérite toute classe sans nécessité de déclaration explicite d'héritage par *extends*.

Une variable de type *Object* peut donc, en vertu du polymorphisme, référencer n'importe quelle instance issue d'une classe quelconque. La classe *Object* définit un certain nombre de méthodes, qui vont donc être de fait disponible pour toutes les instances existantes. Compte tenu de sa place au sommet de l'arbre d'héritage, elle factorise ce qui est commun à toute instance et de ce fait comporte beaucoup de méthode *native* , c'est à dire interfacée directement avec le système d'exploitation sous-jacent et ses bibliothèques spécifiques.

```
package java.lang;
public class Object {
    private static native void registerNatives();
    static { registerNatives(); }
    public final native Class<? extends Object> getClass();
    public native int hashCode();
    public boolean equals(Object obj) { return (this == obj); }
    protected native Object clone() throws CloneNotSupportedException;
    public String toString() {
        return getClass().getName() + "@" + Integer.toHexString(hashCode());
    }
    public final native void notify();
    public final native void notifyAll();
    public final native void wait(long timeout) throws InterruptedException;
    public final void wait(long timeout, int nanos) throws InterruptedException {
```

```
        ... code ...
    }
     public final void wait() throws InterruptedException { wait(0); }
     protected void finalize() throws Throwable { }
}
```

La méthode *clone* a vocation à être redéfinie par surcharge dans les sous classes comme son caractère *protected* y invite. Elle ne sera invocable dans un programme qu'à la seule condition d'avoir été redéfinie dans la classe concernée et étendue du point de vue de la visibilité (passage au niveau *public*) ce qui est possible dans ce sens.

La méthode *finalize* est elle aussi destinée à être redéfinie dans les sous classes concernées. C'est typiquement la machine virtuelle, à travers son *garbage collector,* qui sera cliente de cette méthode.

La méthode *equals* pourra être également redéfinie dans les sous-classes pour affiner le test d'égalité de 2 objets. On voit qu'elle est définie par défaut par la comparaison des références et non des objets référencés.

Les méthodes *notify* ou *wait* sont dédiées aux problèmes de synchronisation.

La méthode *getClass* permet d'interroger une instance sur son type dynamique. Elle retourne une instance de *Class*<*T*> qui est l'objet représentant la classe *T* à l'exécution. La formulation générique de cette fonctionnalité dans le contexte d'une classe aussi universelle qu*'Object*, montre l'impact profond qu'a eu l'introduction de la généricité apportée par la version 1.5 sur l'existant.

Remarquons la méthode *toString* qui explique la raison pour laquelle toute instance semble douée de la capacité à s'afficher de façon intelligible (par son nom de classe en particulier). Dans de nombreuses circonstances où le compilateur attend une chaîne de caractères il est capable de lui substituer le résultat de l'invocation de *toString* sur l'instance qu'on lui fournit en lieu et place de la chaîne de caractères attendue. Cela confère une souplesse d'utilisation tout à fait agréable, d'autant qu'elle a été prolongée par cette même capacité concernant les types de base.

```
Object o1=new Object();
System.out.println( "Voici l'affichage de o1 : "+o1);
```

Sortie:

```
Voici l'affichage de o1 : java.lang.Object@1b90b39
```

Polymorphisme et covariance

Une méthode redéfinie dans une sous-classe doit, pour être prise en compte par le mécanisme de liaison retardée nécessaire au polymorphisme, posséder la même signature que la méthode de la surclasse. Si cette signature est différente le compilateur considère alors que la sous classe définit une nouvelle méthode qui lui est spécifique et qui, donc, n'existait pas dans la surclasse.

L'annotation @*Override* permise par Java est un appareillage du code qui permet au compilateur de vérifier qu'une méthode est bien un version redéfinie d'une autre méthode définie dans la surclasse.

Dans le code ci-dessous les classes très simples *Pen* et *ColoredPen* sont liées par une relation d'héritage.

```
class Pen {
   private int thickness;
   public Pen(int thickness) {this.thickness=thickness;}
}
class ColoredPen extends Pen{
   private int color;
   public ColoredPen(int thickness,int color) {super(thickness);this.color=color;}
   public int getColor() {  return color; }
}
```

La classe *Pair* ci-dessous utilise en interne deux *Pen*.

```
class Pair {
   Pen pen1,pen2;
   public Pair(Pen p1,Pen p2) { this.pen1=p1;this.pen2=p2; }
   public void setPen(Pen p1,Pen p2) { this.pen1=p1;this.pen2=p2; }
   public Pen getFirstPen() { return pen1; }
   public Pen getSecondPen() { return pen2; }
}
```

La classe *ColoredPair* est un raffinement de *Pair* obtenue par héritage. Elle se distingue par le fait que cette fois ce sont deux *ColoredPen* qui sont utilisés en interne (cette façon de procéder ne relève certainement pas d'un bon *design*, une construction fondée sur la généricité serait préférable)

```
class ColoredPair extends Pair {
   public ColoredPair(ColoredPen p1, ColoredPen p2) { super(p1, p2); }
   @Override
   public void setPen(ColoredPen p1,ColoredPen p2) {   // erreur déclenchée par @Override
      this.pen1=p1;this.pen2=p2;
   }
}
```

Avec la mention *@Override* la méthode *setPen* apparaît erronée : c'est qu'en effet le compilateur refuse, en raison de la présence de l'annotation *@Override*, qu'on la considère comme une redéfinition de la méthode *setPen* de la classe *Pair*.

Sans *@Override* cette méthode aurait été acceptée mais n'aurait pas été considérée comme une redéfinition (on dit aussi surcharge) de la méthode *setPen* de *Pair*. Il nous faut donc renoncer à redéfinir *setPen* (avec le risque d'introduire dans *ColoredPair* des simples *Pen*, non colorés)

Cette correspondance stricte entre la signature de la méthode redéfinie et celle de la méthode de la surclasse est néanmoins assouplie pour ce qui concerne le type de retour. Ainsi il est possible de préciser, par redéfinition de *getFirst* et *getSecond,* que les crayons retournés par ces appels sont des crayons colorés lorsqu'ils sont appliqués à *ColoredPair*

```
class ColoredPair extends Pair {
   public ColoredPair(ColoredPen p1, ColoredPen p2) { super(p1, p2); }
   @Override
   public ColoredPen getFirstPen() {return (ColoredPen)pen1;} // accepté en tant que redéfintion
   @Override
   public ColoredPen getSecondPen() {return (ColoredPen)pen1;} // accepté en tant que redéfintion
}
```

En s'adressant à une instance de *ColoredPair* le client récupère bien, avec *getFirst* par exemple, un crayon coloré.

```
public class Launcher {
   public static void main(String[] args) {
      ColoredPair colorPair=new ColoredPair(new ColoredPen(1,3),new ColoredPen(2,5));
      System.out.println(colorPair.getFirstPen().getColor());
   }
}
```

Cette tolérance n'est possible ici que parce qu'il s'agit d'un type de retour et qu'il existe une relation d'héritage entre *ColoredPen* et *Pen*. Ce mécanisme est parfois nommé principe de covariance des paramètres de retour et n'est pas toujours implémenté dans les langages orientés objet. Il l'est en Java.

Interface

Le mot *interface* est un mot réservé du langage Java. Une *interface* est en Java est une classe abstraite radicale. On a vu en effet qu'une classe abstraite l'était parce qu'elle présentait au moins une méthode non définie. L'exemple précédent montre qu'une classe abstraite peut néanmoins présenter des caractéristiques très concrètes telles que des données internes (le champ *info* ci-dessus) ou des méthodes et des constructeurs définies (comme *A()*, *A(String)*, *getInfo*, *toString* ci-dessus). Ces méthodes et ces données ne seront exploitées que par l'intermédiaire d'un mécanisme d'héritage (associé au polymorphisme) depuis une classe héritière.

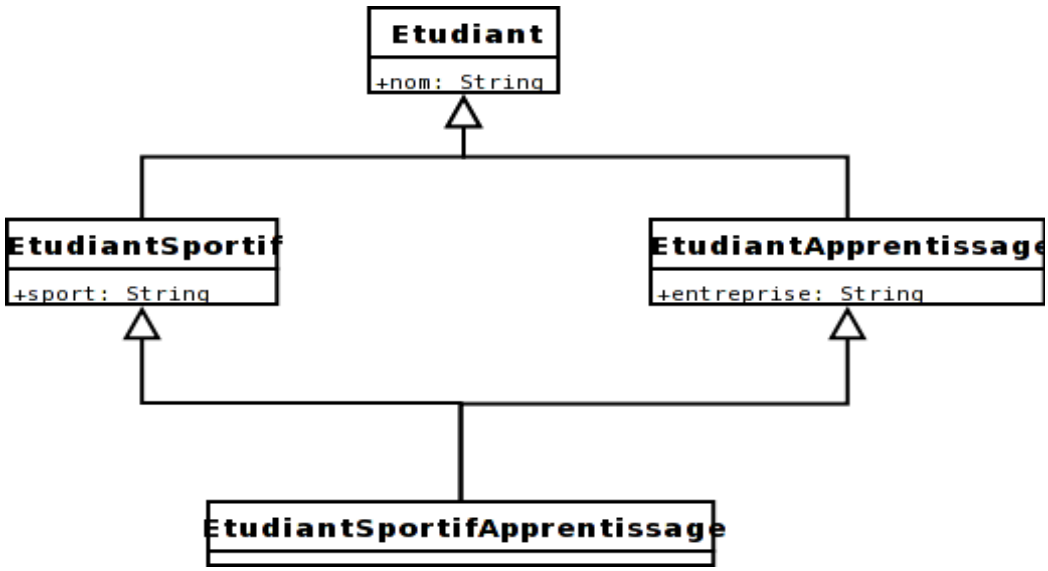
Une *interface* est en Java une classe totalement abstraite : elle ne comporte ni données internes, ni méthodes définies. Une *interface* ne présente donc que des méthodes non définies. Une *interface* est donc une entité qui ne fait qu'annoncer les services que *devront* implémenter les classes concrètes qui directement ou indirectement héritent du jeu de spécification exposé par cette interface.

L'interface *runnable* est, par exemple, entièrement définie par ce qui suit :

```
public interface Runnable {
    public abstract void run();
}
```

On peut s'interroger sur les raisons qui ont conduit les concepteur du langage à distinguer les interfaces des classes abstraites dans la mesure où une interface n'est, après tout, qu'un cas particulier de classe abstraite. La réponse réside dans les problèmes posés par l'héritage multiple. L'héritage multiple consiste à permettre à une classe d'avoir plusieurs classes mère. Des langages comme Eiffel ou C++ propose l'héritage multiple. L'héritage multiple complexifie notablement le modèle d'héritage. En effet si la hiérarchie d'héritage devient un graphe, et non plus seulement un arbre comme dans l'héritage simple, cela signifie qu'une classe hérite

potentiellement de toutes les données et les méthodes des classes situées au-dessus de sa position courante dans le graphe orienté.



Dans ce schéma d'héritage multiple le champ *nom* hérité de *Etudiant* parvient à *EtudiantSportifApprentissage* par deux chemins différents. Compte tenu du principe d'agrégation des attributs hérités il est donc potentiellement dupliqué dans cette classe.

Un même jeu de données ou de méthodes peut donc parvenir par plusieurs chemins différents. Les difficultés sont multiples :

- Comment éviter, par exemple, la multi-instanciation de variables membres identiques mais provenant de chemins d'héritage différents?
- Comment gérer l'héritage multiple, par des chemins d'héritages différents, de méthodes homonymes, voire identiques.
- Que signifie *super*, quand l'instance en question relève de deux classes mères différentes.

Le C++ a introduit la notion de *classe virtuelle* pour tenter de résoudre ce problème, et a également opté pour une formulation de la super-invocation qui explicite la classe mère concernée et qui permet de discriminer les données ou les méthodes en fonction du chemin d'héritage, mais en raison de la complexité de ces techniques beaucoup d'auteurs préconisent tout simplement de ne pas utiliser l'héritage multiple.

Les interfaces ont été distinguées des classes abstraites parce qu'elles ne présentent pas ce type de problème. N'ayant aucune variable membre et ne présentant aucune méthode définie, l'héritage multiple d'*interface* ne présente pas l'inconvénient du schéma d'héritage classique pour lequel hériter signifie hériter des données et du code de la classe mère.

L'héritage d'interface est donc simplement une sorte de contrat d'*implémentation* : si une classe hérite de l'interface *Runnable*, exposée plus haut, cela signifie que cette classe s'engage à définir la méthode exposée par *Runnable* c'est à dire *run*.

Exemple :

```
public class A implements Runnable{
   public void run() { System.out.println("Hello");
}
```

Une instance de *A* sera, de fait, également une entité de type *Runnable* puisque par contrat cette instance est capable de faire tout ce qui est exposé dans *Runnable*.

Le mot réservé *implements* exprime cette forme d'héritage particulière. Java est donc un langage qui accepte l'héritage simple d'implémentation, c'est à dire classique avec *extends,* et l'héritage multiple de spécification, exprimé par *implements* et ne concernant que les *interfaces*.

Les bibliothèques Java font un usage intensif des interfaces. Ci-dessous une version incomplète de l'interface *List* qui expose les services que devront implémenter les classes héritières.

```
public interface List<E> extends Collection<E> {
   int size();
```

```
   boolean isEmpty();
   boolean contains(Object o);
   Iterator<E> iterator();
   boolean add(E o);
   boolean remove(Object o);
   boolean removeAll(Collection<?> c);
   int indexOf(Object o);
   ListIterator<E> listIterator();
   public void clear();
   etc...
}
```

La classe *AbstractList* tente au niveau suivant de l'héritage d'implémenter celles des méthodes de *List* dont elle hérite, par exemple *add* ou *indexOf* . Dans la mesure où elle ne peut pas le faire pour toute les méthodes, par exemple *remove*, elle reste abstraite. Noter les clauses d'héritage.

```
public abstract class AbstractList<E> extends AbstractCollection<E> implements List<E> {
     public boolean add(E o) { add(size(), o); return true; }
     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 void clear() { removeRange(0, size()); }
    public Iterator<E> iterator() { return new Itr(); }
     ...
    private class Itr implements Iterator<E> {  ...   }
    protected transient int modCount = 0;
}
```

Enfin la *LinkedList* est une classe concrète, une feuille de l'arbre d'héritage. Noter l'héritage multiple d'interface, et l'héritage simple concernant la classe abstraite *AbstractSequentialList* .

```
public class LinkedList<E>
    extends AbstractSequentialList<E>
    implements List<E>, Queue<E>, Cloneable, java.io.Serializable {
    private transient Entry<E> header = new Entry<E>(null, null, null);
    private transient int size = 0;
    public LinkedList() {header.next = header.previous = header;  }
    public LinkedList(Collection<? extends E> c) { this(); addAll(c);
     }
    public void addFirst(E o) { addBefore(o, header.next); }
     ...
}
```

Avec cette dernière classe il va être possible de créer concrètement des objets incarnant autant de listes chaînées.

Dans le programme de test ci-dessous on crée plusieurs listes chaînées en utilisant la conformité de type offerte de facto par les clauses d'héritage qui concerne *LinkedList*.

```
public class Launcher {
   public static void main(String[] args) {
      LinkedList<Integer> l1=new LinkedList<Integer>();
      AbstractSequentialList l2=new LinkedList<Integer>();
      List<Integer> l3=new LinkedList<Integer>();
      Queue<Integer> l4=new LinkedList<Integer>();
      Cloneable l5=new LinkedList<Integer>();
      java.io.Serializable l6=new LinkedList<Integer>();
      // héritage indirect
      List<Integer> l7=new LinkedList<Integer>();
      Object l8=new LinkedList<Integer>();
      Iterable<Integer> l9=new LinkedList<Integer>();
      Collection<Integer> l10=new LinkedList<Integer>();
```

```
    }
}
```

La série d'affectations de 10 *LinkedList* montre qu'un objet *LinkedList<Integer>* est aussi une sorte de *List<Integer>* ou de *Iterable<Integer>* etc... Grâce à la relation sémantique *est une sorte de* introduite par la relation d'héritage (quelque soit sa nature , implémentation, spécification) une *LinkedList<Integer>* est aussi une sorte de *List<Integer>* ou de *Iterable<Integer>*. C'est à la base des architectures logicielles typées ouvertes: partout où on attend une entité *List<Integer>*, par exemple, un objet *LinkedList<Integer>* convient puisque c'est une sorte de *List<Integer>* . On est sûr en particulier que cet objet implémente tous les services exposés par *List<Integer>,* il en implémente d'ailleurs aussi d'autres, parce qu'il est issu d'une classe concrète, *LinkedList<Integer>* , qui, par définition, a été en mesure de produire une implémentation des services exposés par *List<Integer>*.

Le graphe d'héritage induit par la notion d'*interface* interfère avec celui résultant de l'héritage classique. Une interface héritière, à la différence des classes abstraites ou concrètes, ne peut avoir qu'une autre interface comme ancêtre. En particulier elle n'hérite donc pas d'*Object.*

Le mot réservé *static*

Le mot réservé *static* peut qualifier une variable membre, une méthode, une classe interne ou un bloc d'initialisation. La connotation générale du terme est la suivante : une entité statique est une entité pour laquelle la notion d'instance courante, incarnée par le mot réservé *this*, n'a pas de sens.

Ainsi une variable membre *static* existe indépendamment de l'existence d'une quelconque instance de la classe dans laquelle elle est déclarée, une méthode *static* est invocable sans instanciation préalable d'un objet, une classe interne *static* peut être à l'origine d'instance sans instanciation préalable de la classe englobante, un bloc de code *static* détermine une initialisation indépendante de toute instanciation.

Variable membre *static*

Une variable membre non statique (c'est à dire une variable membre normale) existe à autant d'exemplaires qu'il existe d'instances de la classe dans laquelle elle est embarquée. Si aucune instance n'est créée à partir d'une classe donnée, les variables membres (non statiques) de cette classe n'ont pas d'existence physique. S'il existe *n* instances de cette classe chaque variable membre est donc elle -même instanciée à *n* exemplaires.

En revanche une variable membre *static* existe du début à la fin de l'exécution du programme, et à un seul exemplaire, que la classe où elle est déclarée soit instanciée ou non.

La classe *A* ci-dessous déclare une cinquième variable *n5* qualifiée par *static*.

```
package test;
public class A implements Comparable <A>{
   public int n1;
   protected int n2;
   int n3;
   private int n4;
   public static int n5=7;
   public A() {
      n1=7;
      n2=7;
      n3=7;
      n4=7;
      n5=7; // OK
      A.n5=7; // OK
      this.n5=7; // OK, mais warning : The static field A.n5 should be accessed in a static way
   }
   public int compareTo(A a) {
      return n1==a.n1 && n2==a.n2 && n3==a.n3 && n4==a.n4?0:1;
   }
}
```

La variable membre *n5* est ici déclarée *static* et initialisée dans le constructeur. Trois formulations sont utilisées ici pour l'accès depuis l'objet lui-même à cette variable. La troisième suscite un *warning* qui rappelle que cette variable n'a en réalité pas besoin d'un instance courante pour être accédée. La formulation *this.n5* n'est pas fausse (car on est dans un constructeur, donc une entité pour laquelle *this* a un sens) mais semble attacher *n5* à l'existence de *this* .

En réalité *n5* est une variable membre qui n'a pas besoin d'instance de *A* pour exister. Son référencement depuis l'extérieur de *A* implique donc un autre schéma que *objet.variable*, puisque l'*objet* n'est ici pas nécessaire. C'est le nom de la classe qui très logiquement viendra se substituer à l'objet dans ce schéma.

Dans le code ci-dessous aucune instance de *A* n'existe, mais la variable membre *static n5* existe, et n'est invocable de l'extérieur de la classe que par le préfixage par le nom de la classe.

```
public class Launcher {
   public static void main(String[] args) {
      A.n5=11;// accès par le nom de la classe; aucune instance de A n'existe
   }
}
```

Compte tenu de ce qui précède, une variable membre statique est donc par la force des choses partagée entre toutes les instances de la classe correspondantes si elles existent, et existe même en l'absence de ces dernières comme l'illustre la portion de programme ci-dessus.

```
public class Launcher {
   public static void main(String[] args) {
```

```
      A a1=new A(),a2=new A();
      a1.n5=178; // warning : accès par un nom d'instance
      System.out.println(a2.n5);
   }
}
```

La sortie montre que les expressions *a1.n5* et *a2.n5* désigne en fait la même entité : *n5* est un entier partagé entre *a1* et *a2*.

```
178
```

Compte tenu de son caractère partagé une formulation moins ambiguë de l'accès à cette variable utilisera de préférence le préfixage par nom de classe.

```
public class Launcher {
  public static void main(String[] args) {
     A a1=new A(),a2=new A();
     A.n5=178;
     System.out.println(A.n5); // il semble ici évident que le résultat d'affichage soit 178
  }
}
```

Les variables *static* sont donc plutôt attachées sémantiquement à la notion de classe qu'à la notion d'instance. Pour cette raison dans certains langages (comme *SmallTalk*) on les désigne sous l'appellation *variable de classe*.

Constante static

Le mot *final* permet d'initaliser définitivement une variable, et il est utilisé notamment pour permettre la définition de constantes.

```
class A {
   public final int MAX=100;
   public final B BOX=new Box();
   ...
}
```

Cette déclaration de la constante *MAX,* ou de l'attribut *BOX*, est d'abord la déclaration d'une variable d'instance ordinaire, ce qui implique que cette variable existera (et consommera les ressources mémoires correspondantes) en autant d'exemplaire qu'il y aura d'instances à l'exécution de la classe *A*.

Il est évident que, pour MAX particulièrement à cause de la sémantique de valeur induite par le type de base *int* utilisé, il s'agit d'une duplication inutile. Dans ce contexte il faudrait donc faire de *MAX* une variable *static* pour éviter ce problème, voir à ce sujet le paragraphe *Déclarations de variables membres statiques final*.

Méthode static

Comme tout membre statique les méthodes statiques sont attachées à la notion de classe et non aux instances de cette dernière. La construction syntaxique (déclaration et appel) est analogue à celle des variables membres statiques.

N'étant pas attachées à une instance ces méthodes ne peuvent invoquer, dans le code qui les définit, l'objet courant *this* que cela soit explicite, par l'usage du mot réservé *this*, ou que cela soit implicite, lorsqu'une méthode invoque un membre (non statique) ou une méthode (non statique) sans préciser, comme le langage le permet, la mention explicite de l'objet courant concerné.

```
class A {
   int n;
   public void m() { n=5; }
   public static void m1() { this.n=7; } // Erreur :Cannot use this in a static context
   public static void m2() { n=7; }      // Erreur:Cannot make a static
                                         // reference to the non-static field n
   public static void m3() { m(); }      // Erreur : Cannot make a static reference
                                         // to the non-static method m() from the  type A
}
```

La classe *Math* est une illustration de l'usage intensif de méthodes statiques. En effet pour calculer *sin(x)* il suffit à la méthode *sin* de connaître *x,* et aucun support d'un quelconque objet courant ne semble utile pour cette méthode.

```
public final class Math {
```

```
 public static final double E = 2.7182818284590452354;
 public static final double PI = 3.14159265358979323846;
   public static double sin(double a)
   public static double sinh(double x)
   public static double sqrt(double a)
   public static double tan(double a)
   ... etc
}
```

L'utilisation de ces méthodes prend la forme suivante :

```
System.out.println(Math.sin(3)*Math.sin(3)+Math.cos(3)*Math.cos(3));
```

Il est loisible d'utiliser la clause d'importation *import static* qui permet, comme son nom l'indique, d'importer des données ou des méthodes déclarées statiques. Cela permet ici de retrouver le formalisme mathématique habituel.

```
import static java.lang.Math.sin;
import static java.lang.Math.cos;
...
System.out.println(sin(3)*sin(3)+cos(3)*cos(3));
```

ou encore :

```
import static java.lang.Math.*;
...
System.out.println(sin(3)*sin(3)+cos(3)*cos(3));
```

Une méthode statique ne peut invoquer dans son code que des données statiques ou des méthodes statiques de la classe courante (qu'elles soient héritées ou non). L'invocation des méthodes non statiques ou de données internes non statiques (héritées ou non) seraient parfaitement contradictoire avec un mécanisme d'exécution qui ne suppose pas l'existence de l'objet courant.

La déclaration *static* reste compatible avec la notion d'héritage, mais le polymorphisme, en particulier le mécanisme de liaison retardée sur lequel il s'appuie, n'ont, en l'absence d'objet, pas de sens dans le contexte d'une méthode statique. Une méthode *static* est donc liée à son adresse au moment de la compilation, comme le serait une méthode *final*.

A quoi servent les méthodes statiques ?

L'existence de la classe *Math* illustre bien l'utilité des méthodes statiques. En effet pour préserver le formalisme mathématique, pour lequel par exemple le calcul du sinus d'une valeur *x* s'exprime par *sin(x)*, Java a conservé aux type de base numérique la syntaxe classique. Ces derniers ne sont pas des classes, et le calcul du sinus d'un *double* ne s'effectue pas par *x.sin()* . En conséquence le formalisme objet se marie mal, ici, avec le formalisme mathématique, alors que la programmation structurée classique fournissait, avec la notion de fonction un concept qui s'ajuste harmonieusement avec le formalisme mathématique:

- *sin(x)* désigne en mathématique la valeur qui résulte du calcul du sinus de x
- *sin(x)* désigne en *C*, par exemple, un appel à la fonction *sinus*. En tant qu'expression elle vaut la valeur retournée par ce calcul.

Les méthodes statiques sont donc aussi un moyen de retrouver le formalisme parfois commode des procédures ou des fonctions de la programmation structurée. L'appel *Math.sin(x)* est donc équivalent à un appel fonctionnel classique caractéristique de ces langages.

Néanmoins l'usage, d'ailleurs intensif, des *static* en Java a largement débordé la contrainte d'adaptation de formalismes présentée ici. Les *static* ont trouvé une place dans d'autres contextes (adaptation aux librairies systèmes, construction *de Factory, singleton*, etc...) qui démontre la richesse du concept lui même. En *SmallTalk* les méthodes dites *de classe* (par opposition aux méthodes d'instances) sont assez proches de ce concept de méthode statique. En outre, tout comme les méthodes *final*, incompatibles avec le concept de liaison retardée, elles peuvent être intéressantes également dans les contextes où une recherche de performance particulière doit être mise en oeuvre.

Import static

La version 1.5 de JAVA a introduit la possibilité d'automatiser le préfixage de l'invocation d'une méthode statique par le nom de la classe à laquelle elle se rapporte. La clause *import static* permet de formuler, à la façon d'*import*, ce préfixage.

Le bloc de code

```
double x=Math.sin(MATH.PI/4);
System.out.println(x);
```

devient, après l'apposition des clauses d'importation statiques :

```
import static java.lang.Math.*;
import static java.lang.System.*;
...
double x=sin(PI/4);
out.println(x);
```

Bloc d'initialisation *static*

L'introduction du mot *static* met en évidence qu'il y a finalement deux aspects d'utilisation d'une classe : par les instances qu'elle produit, ou directement en tant que classe, sans nécessiter d'instance, au moyen des attributs ou méthodes statiques.

Le constructeur d'une classe est dédié à l'initialisation de ses instances. On voit qu'il est assez naturel de compléter ce dispositif pour la partie *static* de la classe : c'est le rôle dévolu au *bloc d'initialisation statique* d'une classe. Ce bloc d'initialisation pourra intervenir, par exemple, lorsque les capacités d'expression de l'initialisation directe des membres statiques se révèlent insuffisantes. Ce bloc est exécuté lors du premier chargement de la classe dans la machine virtuelle, et il n'est exécuté qu'une fois par classe.

```
class H {
   private static int n;
   static {
      System.err.println("Hello 1");
      n=0;
   }
   public H() {
      System.err.println("Hello 2");
      n++;
      System.err.println("n="+n);
   }
}
```

Programme de test :

```
public class Launcher {
   public static void main(String[] args) {
      H h1=new H(),h2=new H(),h3=new H();
   }
}
```

Sortie :

```
Hello 1
Hello 2
n=1
Hello 2
n=2
Hello 2
n=3
```

En cas d'héritage, l'exécution du bloc d'initialisation statique de la sous-classe est précédée de celle du bloc d'initialisation de la surclasse. Toutefois un bloc d'initialisation statique donné dans une classe n'est exécuté qu'une seule fois. Dans l'exemple ci-dessous 2 sous-classes de *A* sont créées et le bloc statique de *A* ne sera appelé qu'une seule fois.

```
class A {
   static int n=14;
   static { System.out.println("Hello1"); }
}
class B extends A {
   static int n=15;
   static { System.out.println("Hello2"); }
```

```
}
class C extends A {
   static int n=16;
   static { System.out.println("Hello3"); }
}

public class Launcher {
   public static void main(String[] args) {
      System.out.println(B.n);
      System.out.println(C.n);
   }
}
```

La sortie est :

```
Hello1
Hello2
15
Hello3
16
```

Classe *static*

La notion de classe *static* n'est valide que pour une classe interne. Voir le chapitre concernant les classes internes statiques.

Le mot réservé final

Le mot *const* , bien que réservé par le langage, n'a pas été repris. Java lui a substitué le mot réservé *final* dont le sens ne recouvre pas celui de *const* en C++.

final est un mot à sémantique plurielle en fonction de son contexte d'utilisation.

Variables locales *final*

Dans le cas d'une déclaration qui concerne un type de base on est en présence de la déclaration classique d'une constante.

```
final int MAX=30;
final double PI=3.14159;
```

Une tentative ultérieure de modification de la variable est refusée par le compilateur :

```
MAX=80 ;  // erreur
```

Cette écriture permet de définir des constantes dont la visibilité et le cycle de vie sont liés à la portée du bloc déclaratif. Dans ce cas ces constantes (qu'on écrit conventionnellement entièrement en majuscules) doivent impérativement être initialisées lors de la déclaration.

Dans le cas d'une déclaration qui concerne un type objet la signification du mot *final* s'éloigne de ce qu'on entend généralement par constante. Il faut alors distinguer la variable et l'objet référencé par cette variable.

```
final Date TODAY=new Date();
```

La variable *TODAY* est bien constante dans le sens où toute assignation ultérieure serait refusée :

```
TODAY=new Date(); // erreur
```

Par contre l'objet référencé par *TODAY* ne bénéficie lui pas de cette protection, et peut être modifié y compris au moyen de sa référence *TODAY*.

```
TODAY.setHours(5); // modification de l'heure de l'objet référencé par TODAY
```

Java ne nous propose donc pas, à l'inverse de C++ avec le mot réservé *const*, la possibilité de garantir l'état constant d'une instance après sa création. En effet cette possibilité est assortie, en C++, de celle de désigner par leur signature celle des méthodes garantissant l'invariance de l'objet auquel elles s'appliquent (les méthodes *getX* dont la signature se terminent par la mention *const* en sont un exemple typique). Faute d'avoir capturé cette sémantique particulière relative aux méthodes, Java se trouve donc dans l'impossibilité de garantir l'invariance d'un objet après sa création.

Une variable locale d'objet *final* doit, comme pour les variables locales concernant un type de base, impérativement référencer l'objet lors de sa déclaration.

Variables membres non statiques *final*.

Pour une variable membre non statique la problématique précédente peut être reprise à l'identique pour ce qui concerne la variabilité de la référence ou de l'objet.

Néanmoins une variable membre possède un cycle de vie lié à l'instance dans laquelle elle est embarquée, Pour cette raison ce sont les conditions de son initialisation qui s'expriment différemment que pour une variable locale.

La règle est la suivante : il faut qu'à la fin du processus de construction de l'instance la variable membre *final* ait reçu sa valeur.

Ainsi la déclaration suivante est correcte, dans la mesure où les variables membres initialisées sont traitées avant l'invocation du constructeur:

```
public class A {
   public final Date date=new Date();
   public A() {}
}
```

Cette déclaration est également correcte, bien que le champ *date* ne reçoive pas de valeur lors de sa déclaration. En effet le constructeur affectant une valeur à *date* la condition précédente est respectée.

```
public class A {
   public final Date date;
   public A() {
      date=new Date();
   }
}
```

L'exemple suivant conduit à une erreur car il existe un chemin de construction qui n'initialise pas la variable membre *final date*.

```
public class A {
   public final Date date;
   private String title;
   public A() {
      date=new Date();
   }
   public A(String title) { // The blank final field date may not have been initialized
      this.title=title;
   }
}
```

Variables membres statiques *final*.

La combinaison de *static* et *final* est un moyen commode de déclarer des constantes de classes. L'avantage du mot *static* est qu'il évite d'avoir à embarquer (donc de créer) autant de représentants de la valeur qu'il y a d'instances. D'autre part l'existence d'une telle constante n'étant pas liée à celle d'une instance de sa classe, son initialisation ne peut pas être, comme dans le cas précédent, rattrapée par le constructeur. Elle doit donc être soit immédiate, soit pris en charge par un bloc d'initialisation statique.

```
public class A {
   public static final int MAX=30;
      public A() {
         for(int i=0;i<MAX;i++) System.out.println(i);
      }
}
```

ou :

```
class A {
   public static final int MAX;
      public A() {
         for(int i=0;i<MAX;i++) System.out.println(i);
      }
      static { MAX=30; }
}
```

Ce mode de déclaration de constante numérique doit être mis en balance avec la possibilité offerte le mot réservé (introduit avec java 1.5) *enum*.

Méthode *final*

Le mot *final* peut également qualifier une méthode. L'apposition du mot *final* à une méthode la marque comme étant non redéfinissable dans une sous-classe.

```
class A {
   final public String getTitle() {
      return "A";
   }
}
class B extends A {
   public String getTitle() { // erreur ; getTitle déclarée final dans la surclasse
      return "B";
   }
}
```

L'intérêt de cette utilisation de la clause *final* peut être ici de deux ordres :

- elle permet de verrouiller certaines méthodes en empêchant leur modification y compris par l'héritage comme illustré ci-dessus.
- elle permet également de remplacer le mécanisme d'appel retardée qui est celui par défaut mis en oeuvre en Java (à l'inverse de C++). Ce mécanisme, indispensable à la mise en oeuvre du polymorphisme caractéristique des architectures orientées objets, permet de reporter le calcul de l'adresse de la méthode de la phase de compilation vers la phase d'exécution. C'est un mécanisme qui peut apparaître comme coûteux dans certains contexte, et il peut parfois gêner certaines optimisations basées sur des mécanismes de branchement prédictif au plus bas niveau. L'invocation de la méthode *getTitle* dans l'expression *a.getTitle()* peut ainsi être directement remplacée par un *call* vers la méthode *getTitle* de la classe *A*. Plus efficace encore un mécanisme d'*inlining* pourrait être mis en place par

certain compilateur consistant à remplacer purement et simplement l'expression *a.getTitle()* par la valeur *"A"*.

La clause *final* ne modifie les caractéristiques que de la méthode à laquelle elle s'applique. Ainsi :

```
class A {
   public String getTitle() {
      return "A";
   }
}
class B extends A {
   final public String getTitle() {
      return "B";
   }
}
public class Launcher {
   public static void main(String[] args) {
      A a;
      B b=new B();
      a=b;
      System.out.println(a.getTitle()); // appel retardée : affiche B
      System.out.println(b.getTitle()); // appel immédiat : affiche B
}
```

Cette fois c'est au niveau de la classe *B* que *getTitle* est déclarée *final*.

Classe *final*

Une classe peut également être déclarée *final*. De la même façon que *final* appliquée à une méthode ferme les capacités de redéfinition de cette méthode dans les sous-classes, *final* appliqué à une classe ferme définitivement la classe à tout héritage. Cela a pour conséquence en particulier de rendre toutes les méthodes de cette classe activables par liaison immédiate (voir paragraphe précédent) donc, de fait, *final*.

Cette mention est souvent utilisée pour des classes de bas niveaux et fréquemment utilisées et pour lesquelles l'abandon de la liaison retardée peut donc être intéressant. Dans le package de base *java.lang* on trouve ainsi de nombreuses classes *final*, par exemple: *Boolean*, *Byte*, *Character*, *Class*, *Double*, *Float*, *Integer*, *Long*, *Math*, *Short*, *String*, *StringBuffer*, *StringBuilder*, *System*, *Void*.

Les instances de classe *final* ne sont pas des objets constants.

```
final class A {
   String title="A";
   public String getTitle() {
      return title;
   }
   public void setTitle(String title) {
      this.title = title;
   }
}

A a=new A();
System.out.println(a.getTitle()); // affiche A
a.setTitle("B");
System.out.println(a.getTitle()); // affiche B
```

Certaines classes produisent des instances constantes (dites *immutables* ou inaltérables). C'est le cas de la classe *String* qui est, par ailleurs, une classe également *final*. Les instances *de* String *sont* inaltérables simplement parce que la classe *String* ne propose aucune méthode permettant de modifier une instance après sa création.

Paramètre *final*

Dans certain cas particulier le mot *final* peut être apposé sur un paramètre de méthode, ou une variable locale, pour en permettre la transmission de valeur à une classe interne locale. Voir *Classe interne locale*.

Classes internes

Java permet, tout comme C++, la définition de classes internes à une autre classe.

En C++ on parle plutôt de classes imbriquées (*nested classes*), et il s'agit d'une relation assez simple qui n'implique que les classes et dont les conséquences ne concernent finalement que le nommage et la visibilité de ces classes. Une instance de classe interne n'a en C++ a priori pas de relation avec une instance de sa classe englobante, sauf dans le cas bien sûr où le programmeur a organisé cette visibilité explicitement.

Java a choisi sur ce point d'étendre les possibilités offertes par cette notion en la complexifiant : la différence essentielle est qu'en Java la notion de classe interne implique généralement une relation entre les instances des classes imbriquées.

Cette sophistication, déroutante au premier abord, du schéma d'imbrication de classes se révèle néanmoins utile dans la pratique. Elle permet en particulier d'éviter une certaine lourdeur dans l'écriture des méthodes *callback* si employées dans un contexte d'application événementielle. Nous distinguerons dans la suite quelques catégories de classe interne : classe interne simple, classe interne locale, classe interne anonyme, classe interne statique.

Classe interne simple

Les classes *B1* et *B2* sont deux classes internes de la classe *A1* respectivement *public* et *private*.

```
public class A1 {
   private String title="A1";
   public A1() {
      System.out.println(new B1());
      System.out.println(new B2());
   }
   public class B1 {
      public String toString() { return "B1:"+title; }
   }
   private class B2 {
      public String toString() { return "B2:"+title; }
   }
}
```

Le programme de test est le suivant :

```
public class Launcher {
   public static void main(String[] args) {
      A1 a1=new A1();
   }
}
```

Il produit l'affichage suivant :

```
B1:A1
B2:A1
```

Le point qui marque la différence avec les classes imbriquées du C++ est la possibilité pour *B1* et *B2* d'accéder au champ *title* de la classe *A1*. Une instance de classe interne peut donc accéder aux attributs et méthodes, même privés, de sa classe englobante. Le nom d'instance de la classe englobante est ici implicite car il n'y a pas d'ambiguïté (tout comme l'implicitation de *this* est possible à chaque fois que le contexte permet au compilateur de déterminer l'entité dont il s'agit). Son explicitation est néanmoins possible par *<nom de la classe englobante>.this*.

Ci-dessous une version équivalente avec explicitation de l'instance de la classe englobante et de son instance (remarquer également l'explicitation de l'instance courante dans le *constructeur A1*()) :

```
public class A1 {
   private String title="A1";
   public A1() {
      System.out.println(this.new B1());
      System.out.println(this.new B2());
   }
   public class B1 {
      public String toString() { return "B1:"+A1.this.title; }
   }
   private class B2 {
      public String toString() { return "B2:"+A1.this.title; }
   }
}
```

Il ne s'agit pourtant pas, entre l'instance de la classe englobante et de la classe interne, d'une relation d'agrégation. L'instance de la classe interne, que nous appellerons par commodité *instance interne* dans la suite, n'est pas automatiquement créée par l'instanciation qui a lieu sur la classe englobante. Dans notre exemple la création des instances internes est explicite, et prise en charge par le constructeur *A1()* avec *new B1()* et *new B2()*.

Autrement dit une instance interne ne peut être créée sans qu'au préalable une instance de la classe englobante n'ait été créée. Elle référence cette dernière par <*nom de la classe englobante*>*.this*.

Réciproquement néanmoins, la création d'une instance de la classe englobante n'implique pas automatiquement la création d'une instance interne : cette création, si elle a lieu, peut se faire dans la classe englobante (comme ci-dessous dans le constructeur de la classe *A1*) ou même à l'extérieur de cette classe. L'instance englobante ne possède a priori pas de référence sur les instances éventuelles de ses classes internes, sauf bien entendu si le programmeur organise ce référencement. Au contraire chaque instance interne, si elle existe, possède nativement une référence sur l'instance englobante.

La syntaxe de création d'instance interne reflète les nécessités précédemment évoquées : l'existence d'une instance englobante de rattachement est un préalable à la création d'une instance interne comme l'illustre l'exemple ci-dessous (qui montre également une des conséquences du caractère private de *B2*)

```
public class Launcher {
   public static void main(String[] args) {
      A1 a1=new A1(); // cette instanciation de a1 créé deux instances internes
                      // de type B1 et B2
      A1.B1 b1=a1.new B1(); // une instance supplémentaire de type B1 est créée.
                            //Elle référence a1
      A1.B1 b1=new A1.B1(); // erreur, car cette syntaxe permettrait de créer une instance
                            // interne sans instance englobante de rattachement
      A1.B2 b2=a1.new B2(); // erreur, car B2 est private
   }
}
```

Dans cet exemple *a1* se trouve donc muni de 2 instances de *B1* et d'une instance de *B2*. Ces trois instances référencent *a1*. Tout se passe comme si elles étaient munies d'un champ supplémentaire référençant *a1*. Au plus bas niveau, dans la machine virtuelle, c'est bien comme cela que les choses sont implémentées et ces instances de *A1*, *B1* et *B2* sont des instances de classes a priori indépendantes et dont les instances présentent entre elles des interactions normales.

La notation *A1.B1* n'est pas sans rappeler la hiérarchisation de nommage offerte par les *nested classes* du C++. Mais ce mécanisme de classes internes a également des conséquences, en, Java, sur la constructions des instances correspondantes.

Par ailleurs, une classe interne (locale ou non) non statique ne peut pas comporter de variables ou de méthodes statiques (en réalité la seule tolérance en matière de variables statiques est celle constituée des variables *s*tatiques qui soient également *final* et qui possède un type de base). En revanche les classes internes elles-mêmes *static* ne présentent aucune de ces restrictions.

```
public class B {
   public class A {
      public static final int MAX=30;
      public final static  Date date1=new Date(); // interdit
      public static  Date date2=new Date();      // interdit
      public static int m() { ... }                    // interdit
   }
}
public class B {
   public static class A {
      public static final int MAX=30;
      public final static  Date date1=new Date(); // ok
      public static  Date date2=new Date();      // ok
      public static int m() { ... }                    // ok
   }
}
```

Classe interne locale

Une classe peut également être définie dans un bloc, comme dans l'exemple ci-dessous :

```
public class A2 {
   private String title="A2";
   public A2() {
      class B1 {
         public B1() { System.out.println(this); }
         public String toString() { return "B1:"+A2.this.title;      }
      }
   }
}
```

La classe *B1* est ici définie dans un bloc d'instruction, celui qui correspond au constructeur de *A2* en l'occurrence. Cet exemple montre que les privilèges conférés aux classes internes sont attribués également à ce type de classe définie localement : elles peuvent accéder aux attributs et méthodes de la classe englobante également comme le montre l'expression *A2.this.title*.

La visibilité de ces classes est locale : pour cette raison elles ne sont pas pourvues du qualificatif *private* ou *public* qui n'aurait pas de sens dans leur contexte.

Le programme minimum d'exploitation ci-dessous montre que là aussi l'instanciation de ce type de classe n'est pas implicite et ce programme ne produit aucun affichage sur sa sortie standard.

```
public class Launcher {
   public static void main(String[] args) {
      A2 a2=new A2();
   }
}
```

Pour l'instruction d'affichage *System.out.println(this)* soit affichée il faut qu'une instance de *B1* soit créée. Elle ne peut être créée que dans le bloc déclaratif de *B1* compte tenu de son caractère local. Ci-dessous cette instanciation est ajoutée a minima.

```
public class A2 {
   private String title="A2";
   public A2() {
      class B1 {
         public B1() { System.out.println(this); }
         public String toString() {return "B1:"+A2.this.title;}
      }
      new B1();
   }
}
```

Cette fois le programme de test lancé par *Launcher* produit l'affichage *B1:A2*.

Accès aux variables de la classe englobante

Les privilèges d'accès à l'instance englobante sont donc conservés pour ces classes locales et le modèle de création d'instances associé relève de la même logique. Mais ces privilèges sont même étendus. Ces classes internes locales ont en effet la possibilité d'accéder, sous certaines conditions, aux variables locales et aux paramètres éventuels passés au bloc courant.

Considérons le code suivant :

```
public class A2 {
   private String title="A2";
   private Object b1;
   public A2(int value1, final int value2) {
      int value3=3;
      final int value4=4;
      class B1 {
         public B1() {
            System.out.println(this+":"+value1);         // erreur
            System.out.println(this+":"+value2);
            System.out.println(this+":"+value3);         // erreur
            System.out.println(this+":"+value4);
         }
         public String toString() {return "B1:"+A2.this.title;}
      }
      b1=new B1();
```

```
   }
}
public class Launcher {
   public static void main(String[] args) {
     A2 a2=new A2(1,2);
   }
}
```

Cette fois le programme de test lancé par *Launcher* (une fois ôtées les parties grisées) produit l'affichage:

```
B1:A2:2
B1:A2:4
```

Le jeu de variables locales constituant le contexte de la déclaration de la classe *B1* comporte ici 4 variables: *value1*, *value2*, *value3* et *value4*. Deux d'entre elles, celles qui sont déclarées *final*, peuvent être accédées par l'instance de *B1*. La règle est donc la suivante : une classe définie localement n'accède qu'à celles des variables locales qui ont été déclarées *final*.

On peut s'interroger sur ce que signifie cet accès aux variables locales *final,* d'autant qu'elles sont ici de type basique, caractérisé par la sémantique de valeur qui y est associée. En effet les instances locales de type objet peuvent avoir un cycle de vie très différent des variables locales qui les référencent *(b1 par* exemple). Ces dernières n'existent en effet, comme toute variable locale, que pendant la traversée du bloc déclaratif par le flux d'exécution. Elles sont créées et détruites à chaque passage du flux. Mais les variables d'un type de base (*value3* ou *value4* par exemple) sont à la fois *variable* et *objet* (de type *int*) : la disparition de la variable implique donc la disparition de l'objet. et cela rend donc la nature de *value4* un peu problématique lorsqu'il s'agit de son invocation depuis le constructeur *B1*.

Pour cette raison, le véritable mécanisme, malgré la proximité syntaxique que présente cette possibilité avec l'accès aux attributs ou méthodes de l'instance englobante, est tout autre. Les variables locales *final* sont recopiées dans l'instance de la classe locale à sa création (bien sûr s'il s'agit de type objet ce sont les références qui sont simplement recopiées) . L'invocation, comme ci-dessus, de *value4* par exemple, est en fait un accès à un champ implicite de l'instance de *B1* obtenu par recopie de la variable locale *value4*.

La restriction aux seules variables *final* résulte du fait que, sans un marqueur particulier apposé sur les variables concernées, toutes les variables locales, et les éventuels paramètres passés au bloc local, seraient embarqués dans les instances de la classe interne, ce qui n'est pas souhaitable en toute généralité.

Le mot *final* utilisé pour ce marquage peut se justifier par le fait que, si la variable locale fait bien l'objet d'une recopie dans l'instance éventuelle de la classe interne, ce mécanisme est syntaxiquement complètement transparent, ce qui pourrait facilement induire des erreurs d'interprétation. Il s'agit ici en effet d'une situation inédite dans un langage fortement déclaratif et typé comme Java : une variable est créée sans aucun formalisme déclaratif.

Exemple : en supposant que le marqueur *final* ne soit pas nécessaire pour la transmission du paramètre *i* la tentation serait grande d'oublier que la variable *i* de la classe est une duplication de la variable *i* d'origine qui nous conduirait à conclure que la portion de code ci-dessous incrémente cette variable.

```
int i=7;
new Serializable() {{ i++; }}; // attention erreur car i non final
```

L'apposition *de* final a comme conséquence de rendre *l'instruction i*++ illégale et finalement nous oblige à prendre en compte la nature duale de *i*.

```
final int i=7;
new Serializable() {{ i++; }}; // attention erreur car i est considérée comme final
```

Si on veut modifier l'information transmise *par* i il faut prévoir une autre variable.

```
final int i=7;
new Serializable() {{ int j=i; j++; }}; // OK j vaut 8
```

Enfin rappellons qu'une classe interne (locale ou non) ne peut pas comporter de variables ou de méthodes statiques.

Classe interne locale et anonyme

La classe interne anonyme est un aboutissement naturel de la notion de classe interne locale. Le nom d'une classe interne locale n'est utilisable que dans le bloc où la déclaration de la classe prend place, et donc l'usage de ce nom est par construction très limité. Si le but de la classe locale est la construction d'une instance unique ce nom devient inutile à condition que la construction de cette instance s'effectue dans la continuité immédiate de la déclaration de cette classe.

La syntaxe qui permet de créer une instance de classe anonyme est du type :

```
new <super class>() { ... };
```

ou

```
new <interface>() { ... };
```

La classe anonyme est positionnée dans un éventuelle hiérarchie d'héritage par invocation de sa surclasse (abstraite ou non) ou par le nom de l'interface qu'elle se charge d'implémenter. Une classe anonyme ne peut pas être générique.

L'exemple précédent en version anonyme donne :

```
public class A3 {
   private String title="A3";
   public A3() {
      new Object() {
         public String toString() {return "Anonyme:"+A3.this.title; }
      };
   }
}
```

Le code test minimum est le suivant :

```
public class Launcher {
   public static void main(String[] args) {
      A3 a3=new A3();
   }
}
```

L'exécution ne produit aucun affichage puisqu'aucune instruction d'entrée-sortie n'est invoquée. Pour obtenir l'équivalent exact de la classe locale nommée précédente il faudrait munir cette classe anonyme d'un constructeur. Cela suppose là aussi une construction syntaxique spécifique puisque Java, tout comme C++, a choisi de distinguer le constructeur des autres méthodes en lui attribuant le nom de sa classe, et le nom de la classe est précisément ce qui manque à une classe... anonyme.

Aussi dans ce cas particulier d'une classe anonyme le constructeur prend la forme d'un bloc de code anonyme délimité par les classiques accolades.

L'exemple ci-dessous reprend le précédent avec l'adjonction d'un constructeur pour la classe anonyme.

```
public class A3 {
   private String title="A3";
   public A3() {
      new Object() {
         { System.out.println(this); }
         public String toString() {return "Anonyme:"+A3.this.title; }
      };
   }
}

public class Launcher {
   public static void main(String[] args) {
      A2 a2=new A2(1,2);
   }
}
```

Cette fois le programme de test produit un effet visible qui signe la création de l'instance anonyme et l'invocation qui l'accompagne de son constructeur anonyme.

```
Anonyme:A3
```

Ce constructeur anonyme n'est pas en mesure d'accueillir directement les paramètres passées à l'instanciation : ceux-ci seront passés au constructeur de la surclasse de la classe anonyme invoquée à la création. La capacité du constructeur à récupérer ces informations dépend donc de cette surclasse.

Dans l'exemple ci-dessous une classe *B* a été introduite (sans qualifiant car présente dans le fichier *A3.java*) pourvu d'un constructeur muni de deux paramètres *n* et *info*. La valeur de *n*, parce qu'elle a été recopiée dans une variable membre, est bien récupérable dans la sous classe anonyme et son constructeur, tandis que le paramètre *info* est perdu. D'autre part, il faut garder à l'esprit qu'un mécanisme de passage d'informations locales au moyen du marqueur *final* est également fourni pour ces classes internes.

```
class B {
   protected int n;
```

```
   public B(int n, String info) {this.n = n; }
}

public class A3 {
   private String title="A3";
   public A3() {
      new B(7,"hello") {
         { System.out.println(this+":"+n); }
         public String toString() {return "Anonyme:"+A3.this.title; }
      };
   }
}
```

Classe interne statique

La classe interne statique est la déclinaison la plus simple et la plus proche des *nested classes* de C++. Elle permet d'enfouir au sein d'une classe la définition d'une autre classe, généralement parce que cette dernière est l'usage exclusif de la première.

Cette définition interne de la classe n'implique cette fois, en raison de son caractère *static,* aucune conséquence sur les instances correspondantes de ces classes.

Dans l'exemple suivant (l'implémentation classique et simplifiée d'une liste chaînée) la classe *LinkList* construit en interne une série de noeuds (classe interne *Node*) chaînés. Ces derniers sont donc des instances de node et il ne serait d'aucune utilité pour chacune de ces instances de posséder une référence implicite à l'objet de la classe englobante *LinkList*.

La mention *static* permet d'éviter ces références inutile.

```
public class LinkList {
   Node root=null;
   static private class Node {
      private String info;
      private Node next;
      public Node(String info) { this.info=info; }
   }
   public void add(String info) {
      Node node=root;
      if (root==null) root=new Node(info);
      else {
         while (node.next!=null) node=node.next;
         node.next=new Node(info);
      }
   }
   @Override
   public String toString() {
      String s="";
      Node node=root;
      while (node!=null) {
         s+=node.info;
         node=node.next;
      }
      return s;
   }
}
public class Launcher {
   public static void main(String[] args) {
      LinkList list=new LinkList();
      for(int i=0;i<5;i++) list.add("string"+i+" ");
      System.out.println(list);
   }

}
```

L'affichage est le suivant :

```
string0 string1 string2 string3 string4
```

A quoi servent les classes internes ?

Le premier objectif des classes internes est illustré par la notion de classe interne statique : elle permet de limiter la pollution de l'espace de nommage de l'application par des noms de classes dont la raison d'être est purement interne à d'autres classes. En ce sens c'est un moyen de hiérarchiser l'espace des noms de classes et cela constitue une technique complémentaire à celle des paquetages qui relève eux mêmes déjà d'une conception hiérarchisée.

Pour cet objectif la notion de classe interne statique suffisait. Mais en Java les classes internes visent également à procurer la possibilité de construire à la volée un objet muni, également à la volée, de son code, ce qui apporte dans certaines circonstances une souplesse appréciable. Ainsi la combinaison de l'usage d'interface ou de classe abstraite avec celle de classe interne locale et anonyme est très fréquemment employée par exemple dans un contexte de programmation événementielle.

Prenons l'exemple de la mise en oeuvre d'un *Timer*.

Une instance de *Timer* (du package *java.util*) permet, par exemple, de lancer une tache à intervalle régulier. On crée un objet *Timer* ainsi:

```
Timer timer=new Timer(task,0,1000); // lancement de task dès maintenant et toutes les 1000 ms
```

L'objet *task* doit être une instance dérivée de *TimerTask* qui est une classe abstraite. Cette classe hérite de l'interface *Runnable* la méthode non définie *run*. C'est cette méthode qui sera invoquée par l'objet *timer* sur l'objet *task.*

Sans la possibilité de définir de classe interne la mise en oeuvre de cette fonctionnalité comporte donc les éléments incontournables suivant :

- création de l'objet référencé par *task*
- création de la classe concrète, héritée de *TimerTask* (cette dernière étant abstraite elle ne peut pas fournir d'instance)
- choix d'un nom pour cette classe
- création d'un fichier portant ce nom
- pollution éventuelle de l'espace global de nommage par l'ajout d'un nom d'usage pourtant très local.
- organisation du passage d'informations (variables locales, paramètres, variables membres etc...) à cette nouvelle classe au moyen de constructeurs ou de méthodes dédiées.
- formulation du passage d'informations côté appelant.

L'ensemble de ces taches n'est pas forcément rédhibitoire, néanmoins il faut dans tous les cas ajouter à l'ensemble des classes de l'application une classe qui ne correspond pourtant qu'à un besoin très local. Dans certaines applications, dans les contextes événementiels où les fonctions *callback* sont nombreuses par exemple, l'absence de ces facilités d'écriture conduirait à une prolifération de petites classes dédiées chacune à la gestion d'un type d'évènement.

La facilité offerte par l'utilisation ici d'une classe anonyme constitue une facilité appréciable :

```
public class Launcher {
   public static void main(String[] args) {
      new Timer().schedule(
            new TimerTask(){
               public void run(){
                     System.out.println("Hello !");
            }
         },0,1000);
   }
}
```

Le résultat est d'exécution est constitué par l'affichage en boucle du mot *Hello !* toutes les secondes.

Dans la variante suivante le mot *Hello* est passé en ligne de commande lors du lancement de l'application. Elle illustre la capacité de récupérer dans une instance de classe anonyme locale la valeur d'une variable locale (*arg*), à condition qu'elle soit *final.*

```
public class Launcher {
   public static void main(String[] args) {
      final String arg=args[0];
      new Timer().schedule(new TimerTask(){
            public void run() {
               System.out.println(arg);
            }
```

```
        },0,1000);
      }
}
```

Le lancement par *java Launcher Hello* produit l'affichage de *Hello* toutes les secondes jusqu'à interruption forcée de l'application. On voit que la durée de vie de l'objet anonyme, issu de la classe locale anonyme, déborde très largement celui de la variable locale *arg* qu'il référence : rappelons que cette référence faite à *arg* dans la méthode *run* est en fait une référence à une recopie de *arg* sous la forme d'un champ implicite de cet objet anonyme.

Ces classes internes sont traduites en classes *normales* par le compilateur. Le programme suivant créée 3 instances anonymes de trois classes anonymes (contenant d'ailleurs le même code)

```
public class Launcher {
   public static void main(String[] args) {
      final String arg=args[0];
      new Timer().schedule(new TimerTask(){public void run()
{System.out.println(arg+"0");}},0,1000);
      new Timer().schedule(new TimerTask(){public void run()
{System.out.println(arg+"1");}},0,1000);
      new Timer().schedule(new TimerTask(){public void run()
{System.out.println(arg+"2");}},0,1000);
   }
}
```

Après le lancement par *java Launcher Hello, l*a sortie boucle sur l'affichage de :

```
Hello0
Hello1
Hello2
```

L'exploration du répertoire de création des fichiers *.class* correspondant à la version compilée des classes de l'application produit dans ce dernier cas :

```
Launcher$1.class  Launcher$2.class  Launcher$3.class  Launcher.class
```

On reconnaîtra l'expression de la classe *Launcher* sous le fichier *Launcher.class*. Pour les trois classes anonymes locales à *Launcher* le compilateur aura construit les fichiers *Launcher$1.class Launcher$2.class Launcher$3.class* construisant de fait les classes éponymes, ce que confirme le programme suivant :

```
public class Launcher {
   public static void main(String[] args) {
      Object task=new Runnable(){public void run() {System.out.println("Hello");}};
      System.out.println(task.getClass().toString());
   }
}
```

Sortie :

```
class Launcher$1
```

L'imbrication de classes est une opération récursive : une classe interne peut elle-même comporter une classe interne. Le système de nommage des fichiers de byte-code correspondant est lui même récursif (par exemple *Launcher$1$1* etc...)

Enumération

La version 1.5 de Java a incorporé la notion d'énumération dans le langage. Inspirée du C++, supportée par l'introduction du mot réservé *enum*, la notion d'énumération permet de définir un type associé explicitement à un jeu de valeur fini et définitif.

L'énumération est en Java plus riche, plus complexe mais aussi plus consommatrice de ressources que son équivalent C++.

Ci-dessous deux classes d'énumération sont créées *PizzaSize* et *ColorPixel.*

PizzaSize crée 4 instances *final,* même si les *mots final et* new n'apparaissent pas, *PizzaSize.SOLO, PizzaSize.DUO, PizzaSize.SUPER, PizzaSize.EXTRA*.

```
enum PizzaSize {
   SOLO, DUO, SUPER, EXTRA
};
```

Chacune de ces instances conserve le nom symbolique qui lui a été choisi en tant que valeur (*"SOLO"* pour *PizzaSize.SOLO* par exemple) ainsi qu'une valeur ordinale reproduisant l'ordre d'énumération (pour *PizzaSize.SOLO c'est 0,* pour *PizzaSize.DUO c'est 1* etc.).

Une instance d'énumération est donc au minimum constituée par une information de type *String* (son nom symbolique donné à la déclaration) et de type *int* (son numéro ordinal).

Il est possible d'enrichir les données internes associées à une énumération. Ci-dessous l'énumération *ColorPixel* associe explicitement une valeur numérique, incarnée *par colorValue,* à chacune de ses 3 instances. Remarquer alors l'intervention du constructeur.

ColorPixel crée 3 instances *final ColorPixel.RED, ColorPixel.BLUE, ColorPixel.GREEN*.

```
enum ColorPixel {
   RED(0x00FF0000), BLUE(0x0000FF00), GREEN(0x000000FF);
   private ColorPixel(int color) {
      this.colorValue = color;
   }
   public int colorValue;
};
```

Plus généralement, du point de vue de la bibliothèque Java, une énumération, introduite par *enum*, est une classe *final* héritière de la classe générique *Enum* . Une énumération ne peut toutefois pas être elle-même générique.

Cette dernière fournit à la classe d'énumération ou à ses instances un ensemble de méthodes pour exploiter les données internes des instances, par exemples: *compareTo(E o), equals(Object other), getDeclaringClass(), name(),ordinal(), toString(), valueOf(Class<T> enumType, String name)* . On constate notamment que, par héritage, les instances d'énumération sont comparables par leurs valeurs ordinales.

Le programme suivant exploite ces deux classes d'énumération :

```
enum PizzaSize {
   SOLO, DUO, SUPER, EXTRA
};

enum ColorPixel {
   RED(0x00FF0000), BLUE(0x0000FF00), GREEN(0x000000FF);
   private ColorPixel(int color) { this.colorValue = color; }
   public int colorValue;
};
public class Launcher {
   public static void main(String[] args) {
      System.out.println(PizzaSize.SOLO);
      System.out.println(PizzaSize.SOLO.ordinal());
      System.out.println(PizzaSize.SOLO.getClass());
      for (PizzaSize ps : PizzaSize.values()) System.out.println(ps);
      PizzaSize ps1 = null; // OK
      PizzaSize ps2 = PizzaSize.DUO;      // OK
      PizzaSize ps3="SMALL";// ERREUR
      System.out.println(ColorPixel.BLUE);
      System.out.println(ColorPixel.BLUE.ordinal());
      System.out.println(ColorPixel.BLUE.getClass());
      System.out.println(ColorPixel.BLUE.colorValue);
      ColorPixel.BLUE.colorValue++;       // OK ! final ne veut pas dire constant
      for (ColorPixel ps : ColorPixel.values()) System.out.println(ps.colorValue);
   }
}
```

Remarquer l'usage de la méthode statique values sur les classes d'énumération. Une fois éliminée la ligne générant une erreur, la sortie est la suivante :

```
SOLO
0
class PizzaSize
SOLO
DUO
SUPER
EXTRA
BLUE
1
```

```
class ColorPixel
65280
ff0000
ff01
ff
```

Comme on le voit, au niveau de la machine virtuelle, les instances d'énumération sont des instances classiques du modèle. Les classes d'énumération sont un cas particulier de classe, dont les instances sont définies lors de la déclaration de la classe et à qui on attribue une référence (*final*) . Les classes d'énumération sont elles-mêmes *final* et donc fermées, en particulier, à l'héritage. Elles ne peuvent pas être génériques. En outre elles ont un comportement, lorsqu'elles sont internes à une autre classe ce qui est souvent le cas, de classes statiques (par exemple leurs instances peuvent exister indépendamment de toute instanciation de la classe englobante).

```
class A {
   public int n;
   enum B { COUNT };
}

public class Launcher {
   public static void main(String[] args) {
      System.out.println(A.B.COUNT); // OK : affiche COUNT ; pas d'instanciation de A
   }
}
```

Autoboxing

Les types de base (*int, short, long, float, char, byte, double...*) en Java n'obéissent pas à la même sémantique que les types objet. Il s'agit d'une sémantique de *valeur*. Lorsqu'une variable construite sur un type de base est passée en paramètre sa valeur est recopiée dans la pile, alors que lorsqu'une variable construite sur un type objet est passée en paramètre cette recopie ne concerne que l'adresse de l'objet et donc pas l'objet lui-même.

Dans une classe générique le type générique doit être un type objet. Ainsi une déclaration telle que *LinkedList<int>* serait refusée par le compilateur en raison du fait que le type *int* est un type de base (en *C++* par contre les types de base restent compatibles avec l'utilisation des génériques).

Pour faciliter la manipulation des types de bases avec des classe génériques Java a introduit des facilités d'écriture regroupées sous le nom d'*autoboxing* .

Elles consistent à chaque fois que cela est possible à rendre implicite la conversion d'un type de base vers la classe de *wrapping* (*java.lang*) qui lui est associée:

Type de base	Classe associée
int	**Integer**
short	**Short**
long	**Long**
byte	**Byte**
char	**Character**
float	**Float**
double	**Double**
boolean	**Boolean**
void	**Void**

Ces classes associées sont *inaltérables* : une instance d'*Integer*, par exemple, ne peut pas changer de valeur après sa création. On ne peut en particulier pas s'en servir pour implémenter un mécanisme de passage par référence en Java. Ces classes sont également un endroit naturel pour placer des fonctionnalités (sous la forme de méthodes statiques) logiquement associées au type en question. Ainsi dans la classe *Integer* on trouve des méthodes statiques permettant la conversion de valeur entière vers les formats *double*, *String* etc...

L'*autoboxing* permet d'automatiser la construction d'une instance de *Integer* (par exemple) à chaque fois qu'un tel type est attendu et qu'on fournit un *int*. A l'inverse la conversion dans le sens *Integer* vers *int* est également prise en compte chaque fois que cela a un sens.

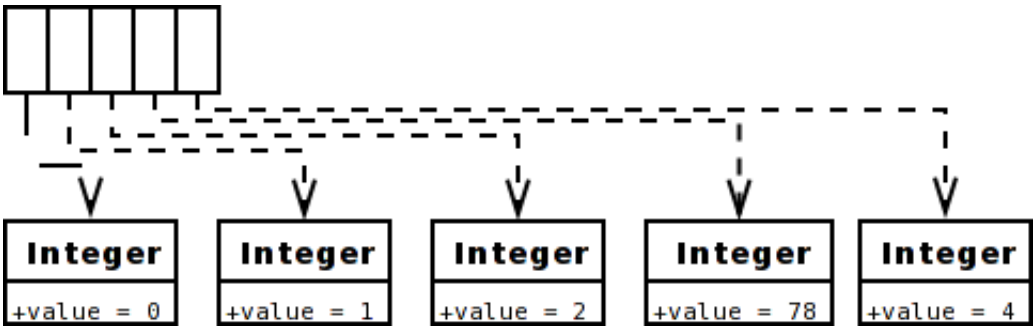
Exemple :

```
Integer m=14; // Autoboxing : équivaut à Integer m=new Integer(14);
int n=12;
m++; // Autoboxing : équivaut à m=new Integer(m.intValue()+1);
m=11+n+m; // Autoboxing : équivaut à m=new Integer(m.intValue()+11+n);
System.out.println(m);
System.out.println(m.getClass());
ArrayList<Integer> tab=new ArrayList<Integer>();
for(int i=0;i<5;i++) tab.add(i); // Autoboxing : équivaut à
                                 // for(int i=0;i<5;i++) tab.add(new Integer(i));
System.out.println(tab);
tab.set(3, 78);  // Autoboxing : équivaut à tab.set(3,new Integer(78));
System.out.println(tab);
```

Affichage :

```
38
class java.lang.Integer
[0, 1, 2, 3, 4]
[0, 1, 2, 78, 4]
```

Le tableau tab présente l'aspect suivant :



Généricité

Introduction

La version 1.5 de java a introduit la notion de généricité. Ce concept avait déjà été introduit par des langages comme *Ada*, *Eiffel* ou *C++*.

La généricité permet de passer un type, concrètement une classe en java, en paramètre. Pour l'introduire,dans sa version 1.5, java a procédé à une reformulation assez profonde de nombre de ses classes de base.

Le besoin de structure générique se fait spontanément sentir lorsqu'il s'agit de construire des classes, appelées parfois *conteneur*, qui ont vocation à organiser l'accès et le rangements d'objets : il s'agit typiquement de classes implémentant des strutures de données telles que les tableaux, les listes, les arbres binaires etc.

Pour les tableaux, Java fournit nativement une structure de données génériques permettant de les gérer.

Ainsi les déclaration suivantes déclarent et créent deux tableaux.

```
int [] tab1=new int[50];
Runnable [] tab2=new Runnable [100];
```

Chacun de ces tableaux est, comme tout tableau, une struture de données spécifique permettant un mode d'accès sur les éléments contenus basé sur la notion d'indice. Ces deux tableaux disposent néanmoins d'une information supplémentaire concernant le type des éléments qu'ils ont vocation à stocker : tout se passe comme si on avait fourni à chacune de ces structure de données le type des éléments à manipuler. Il s'agit, autrement dit, d'une passage de type en paramètre.

Jusqu'à la version 1.5 le langage ne nous permettait pas de concevoir nos propres classes conteneurs sur le modèle des tableaux, c'est à dire en pouvant spécifier le type des éléments contenus. Avec la généricité, une construction syntaxique spécifique a été introduite pour le permettre.

Considérons l'exemple d'un autre conteneur, qui à l'inverse des tableaux, ne fait pas partie nativement du langage, mais est implémenté sous la forme d'une classe : la *LinkedList*. du package *java.util*

Jusqu'à la version 1.4 son nom est *LinkedList*. A partir de la version 1.5 son nom est devenu *LinkedList<E>*.L'adjonction de *<E>* correspond à la syntaxe permettant de spécifier un type générique.

Une *LinkedList* permet de construire une liste chaînée d'objets. Dans la version antérieure à 1.5 nous ne pouvions pas préciser la nature des objets en question. Une portion d'interface de *LinkedList était la suivante :*

```
public class LinkedList {
   public:
   LinkedList();
   void add(Object object);
   Object getFirst();
   ListIterator listIterator();
   etc...
}
```

Un code d'exploitation typique de cette liste serait le suivant :

```
   LinkedList l1=new LinkedList();
   ListIterator li1=l1.listIterator();
   li1.add(new A());
   li1.add(new A());
   li1.add(new A());
   A a;
   while (li1.hasNext()) {
      a=(A)li1.next();
      System.out.println(a.n);
   }
```

La liste *l1* ne possède pas d'indication du type des éléments qu'elle contient : ils sont donc a priori du type *Object*.

On procède ensuite à l'insertion concrète de trois instances de type *A*, qui seront acceptées car un *A* est bien une sorte d'*Object*. Une instance de *ListIterator*, *li1,* est ensuite créée pour exploiter en lecture la liste *l1*. Cette instance ne possède aucune information sur le type des éléments de *l1*. L'invocation de *li1.next()* retourne donc, pour le compilateur, un objet de type *Object*. Un *cast* est donc nécessaire pour permettre

l'exploitation complète de l'expression *l1.next()* en tant qu'instance de type *A,* puisqu'un *Object* n'est pas une sorte de *A*.

Tout se passe comme si le programmeur devait renoncer au typage, et à la cohérence et la sécurité qu'il apporte, lorsqu'il manipule un conteneur faute de pouvoir spécifier à la déclaration le type des objets manipulés par ce dernier.

La version générique de l'exemple précédent serait la suivante :

```
LinkedList<A> l2=new LinkedList<A>();
ListIterator<A> li2=l2.listIterator();
li2.add(new A());
li2.add(new A());
li2.add(new A());
A a;
while (li2.hasNext()) {
   a=li2.next();
   System.out.println(a.n);
}
```

Cette fois le typage de la liste chaînée est complet. On peut bénéficier du contrôle de cohérence qui l'accompagne. Un itérateur d'un type incohérent avec celui de la liste chaînée serait, par exemple, statiquement détecté.

```
ListIterator<B> li2=l2.listIterator(); // erreur; incohérence de type
```

Le *cast* affectant l'expression *li2.next()* n'est plus nécessaire : du point de vue du compilateur cette expression est bien du type *A* et ne nécessite donc pas de transtypage.

Le but de la généricité est donc de permettre au compilateur de pousser plus loin les limites du contrôle sémantique sur les types auquel il procède pour assurer le maximum de cohérence et de sécurité au code. Cela revient à déplacer à déplacer la manifestation de ces erreurs de la période d'exécution, c'est à dire trop tard, vers la période de compilation.

Classe générique en Java

L'expression syntaxique de la généricité de Java présente certaines similitudes avec celles des *templates* de C++. Néanmoins l'implémentation en est très différente.

En C++ une classe générique (par exemple *A*<*T*>) est un pur modèle. Ce n'est que lorsque l'utilisateur spécifie le type de donnée sur lequel ce modèle doit se construire (par exemple *A*<*int*>) que cette classe acquiert une existence analogue à celle d'une classe ordinaire. Ce n'est qu'à cette condition que le code ou les données contenus dans la classe acquièrent une consistance suffisante pour pouvoir faire l'objet d'une compilation ou d'une édition de lien. Le code, notamment celui utilisant le type générique *T* n'est qu'un modèle de code et ne devient du code au sens classique du terme, c'est à dire, par exemple, compilable, que lorsque le type générique *T* a été instancié, c'est à dire remplacé par un type réel (c'est le mot qu'avait introduit Ada pour cette opération et qui dans ce contexte ne doit pas être confondu avec l'instanciation classique d'un objet). Les *templates* du C++ apparaissent alors davantage comme un jeu d'écriture permettant d'élaborer un modèle de classe, accompagné de son modèle de données et du modèle de code accompagnant des méthodes elles-mêmes génériques. Une même classe générique, *A*<*T*> par exemple, produit autant de fichier objet qu'elle connaîtra d'instanciations de son type générique. *A*<*T*> seule n'est pas compilable, mais *A*<*int*>, *A*<*double*> et *A*<*string*> produiront trois versions compilées distinctes.

En java l'approche est très différente. Une classe générique est déjà une classe, même en l'absence d'instanciation de son type générique.

```
public class C<T> {
   public C() { this.t=null; }
   public C(T t) { this.t=t; }
   public T t;
}
```

Cette déclaration purement générique produit à la compilation, comme on peut le vérifier facilement, un fichier *C.class.*

Une variable issue d'une version instanciée de *C* peut être déclarée ainsi (on suppose que la classe A est une classe existante):

```
C<A> c1;
```

L'instanciation (au sens de la création d'un objet) s'exprime par exemple ainsi :

```
c1=new C<A>();
```

On peut aussi vérifier qu'ici, à l'inverse de ce qui se passe en C++, aucune nouvelle classe n'a été créée par la déclaration *C<A>*.

D'autre part l'instanciation d'un type générique ne concerne pas en Java les types de base, à sémantique de valeur (*int, double*, *char* etc...)

```
C<int> c1; // erreur ; un type générique ne peut pas être instancié par une type de base en Java
```

Cela montre que l'implémentation de la généricité relève en Java d'un travail sur le typage des références. La variable membre générique *public T t;* évoquée ci dessus dans la classe *C<T>* n'est finalement pas très éloignée de la déclaration *public Object t;* dont l'implémentation peut être préparée par le compilateur (au plus bas niveau *t* est un pointeur, dont la taille est donc parfaitement définie). Plus tard lorsque le type effectif est connu (par exemple *A)* il suffit au compilateur de substituer le type *A* au type *T*, en effectuant, sous son contrôle sémantique, le travail de transtypage que le programmeur aurait dû faire lui-même dans un contexte où la généricité serait absente. Concrètement le fichier résultant de la compilation d'une classe générique est un fichier *.class* ordinaire où les références concernant *T* sont remplacées par des références concernant le type *Object* (ou le type constraint associé au type générique, voir plus loin). Si elle a impacté et complexifié le compilateur, la généricité n'a eu aucun impact sur la machine virtuelle. Ce mécanisme d'implémentation d'une classe générique, où le type générique est remplacé par le type *Object*, est appelée effacement *du type générique.*

Syntaxe de la généricité côté utilisateur

La position d'utilisateur de classe générique est incontournable en Java puisque la bibliothèque standard a fait l'objet d'un remaniement assez profond pour incorporer cette technique là où elle est utile (dans les classes conteneurs notamment).

Par exemple la classe *LinkedList* déjà évoquée est présentée ainsi dans l'arborescence Java :

```
+ java.lang.Object
   + java.util.AbstractCollection<E>
      + java.util.AbstractList<E>
           +  java.util.AbstractSequentialList<E>
              - java.util.LinkedList<E>
```

On y voit que les entités génériques peuvent être en Java des interface, des classes abstraites ou des classes. Les types génériques eux-mêmes peuvent relever de ces trois catégories. On constate également sur cet exemple que la généricité et l'héritage peuvent cohabiter harmonieusement : il est possible de spécialiser, au moyen de l'héritage, une classe elle-même générique. Ce point est discuté spécifiquement plus loin.

L'exemple ci-dessous crée une *LinkedList* d'objets de type *Runnable*. *Runnable* est une interface qui expose la méthode *run*. Un conteneur de type *LinkedList<Runnable>* permet, grâce au contrôle du type des objets contenus, rendu possible du fait de la généricité, de garantir la capacité des objets listés à supporter la méthode *run*.

```
public class Launcher {
   public static void main(String[] args) {
      LinkedList<Runnable> listRun=new LinkedList<Runnable>();
      ListIterator<Runnable> listRunIterator=listRun.listIterator();
      listRunIterator.add(new Runnable(){
         public void run() {
            System.out.println(("Run 1"));
         }});
      listRunIterator.add(new Runnable(){
         public void run() {
            System.out.println("Run 2");
         }});
      listRunIterator.add(new Runnable(){
         public void run() {
            System.out.println("Run 3");
         }});
      for(Runnable r:listRun) r.run();
   }
}
```

Dans ce code une liste chainée pour l'accueil d'objets *Runnable* est créée par :

```
LinkedList<Runnable> listRun=new LinkedList<Runnable>();
```

Un itérateur, dont la compatibilité de type avec celle du conteneur est contrôlé par le compilateur, est créé par :

```
ListIterator<Runnable> listRunIterator=listRun.listIterator();
```

Les objets *Runnable* sont créés à la volée, ainsi que leur classe d'appartenance, au moyen de 3 classes anonymes locales. Une tentative d'insérer un objet qui ne supporterait pas *run* serait vouée à l'échec.

```
listRunIterator.add(new Runnable(){
   public void run() {
      System.out.println(("Run 1"));
   }});
```

Enfin on utilise une variante de la boucle *for* (dite variante *foreach*) pour parcourir et exploiter les objets de la liste. Aucun transtypage n'est nécessaire.

```
for(Runnable r:listRun) r.run();
```

En sortie nous obtenons :

```
Run 1
Run 2
Run 3
```

Une classe peut présenter plusieurs types génériques. La classe *TreeMap*, issue de la *Java Collection Classes*, la bibliothèque Java dédiée aux conteneurs, en est une illustration. Elle est étudiée dans le paragraphe suivant.

Généricité contrainte

La généricité contrainte a été historiquement introduite pour la première fois dans le langage *Eiffel.* Elle n'existe pas en C++.

Ce concept permet d'exprimer une contrainte concernant le paramètre générique (par exemple *T*) utilisé dans une classe générique.

Pour illustrer ce point considérons l'exemple de la classe *Collections*. Cette classe est un catalogue de fonctionnalités génériques permettant le traitement de conteneurs : tri, permutation, insertion, parcours etc...

La méthode de tri est intéressante car elle suppose que les éléments à trier soient munis d'une relation d'ordre. Que se passe-t-il si nous donnons à traiter une liste d'élément qui ne possède pas cette capacité à la *comparaison*?

Dans l'exemple qui suit nous considérons une liste de *A*. Cette classe *A* n'offre rien d'intéressant : héritant d'*Object* elle est néanmoins capable de produire des instances. Nous construisons une liste de 3 instances de *A* et tentons de la trier au moyen de la méthode statique *sort* de la classe *Collections.*

```
class A {
}
// programme d'exploitation
LinkedList<A> list=new LinkedList<A>();
list.add(new A());
list.add(new A());
list.add(new A());
for(A a:list) System.out.println(a);
Collections.sort(list); // Erreur : voir message ci-après
for(A a:list) System.out.println(a);
```

Le message d'erreur est assez complexe :

```
Bound mismatch: The generic method sort(List<T>) of type Collections is not applicable for
the arguments (LinkedList<A>). The inferred type A is not a valid substitute for the bounded
parameter <T extends Comparable<? super T>>
```

Ce message signifie que A ne correspond pas au type générique attendu par *sort*. Cette méthode est définie ainsi :

```
public static <T extends Comparable<? super T>> void sort(List<T> list)
```

La mention qui précède *void* pose un certain nombre de contraintes pour le type générique *T* .

<T extends Comparable<? super T>> signifie que *T* doit être un type qui étend (hérite de) l'interface *Comparable* (le sens de l'expression *<? super T>* sera étudié plus loin et peut être ici négligé en première approximation). En clair *sort* veut s'assurer que le type *T* soit muni d'une méthode ce comparaison car celle-ci lui est nécessaire pour procéder au tri.

L'interface *Comparable* est la suivante :

```
public interface Comparable<T> {
    public int compareTo(T o);
}
```

Il nous faut donc ajouter une clause d'implémentation à la classe *A* et lui ajouter la méthode *compareTo.*

```
class A implements Comparable<A> {
   public int compareTo(A o) {
      return (int)(Math.round(Math.random()*2)-1); // comparaison aléatoire
   }
}
```

Le programme d'exploitation cette fois est accepté :

```
      LinkedList<A> list=new LinkedList<A>();
      list.add(new A());
      list.add(new A());
      list.add(new A());
      for(A a:list) System.out.println(a);
      System.out.println("---");
      Collections.sort(list);
      for(A a:list) System.out.println(a);
```

La sortie est la suivante :

```
A@1e0bc08
A@158b649
A@127734f
---
A@158b649
A@1e0bc08
A@127734f
```

Généricité et héritage

L'exemple de la classe *LinkedList<E>* et de son insertion dans un arbre d'héritage faisant largement appel à la généricité montre que les relations d'héritage et de généricité peuvent cohabiter et s'enrichir mutuellement.

```
+ java.lang.Object
   + java.util.AbstractCollection<E>
      + java.util.AbstractList<E>
            +  java.util.AbstractSequentialList<E>
               - java.util.LinkedList<E>
```

Ainsi une classe générique peut étendre ou implémenter une autre classe générique , une classe abstraite générique ou une interface générique. *LinkedList* étend les classes abstraites *AbstractList* ou *AbstractSequentialList*, mais elle implémente également (ce qui ne se voit pas dans cette arborescence dédiée à l'héritage et non à l'implémentation d'interface, qui serait d'ailleurs un graphe) l'interface *Iterable<E>* , par exemple.

Une classe générique peut hériter d'une classe non générique comme le montre ici la présence de la classe *Object* au sommet de cet arbre d'héritage.

Classe générique et classe *Raw*

Pour des raisons de compatibilité avec la version non générique de l'existant Java, et de la machine virtuelle en particulier, il existe une certaine forme de conformité entre une classe générique et sa version brute (*raw*) c'est à dire sans classe générique.

Ainsi, considérons une classe *A* générique réduite à sa plus simple expression :

```
class A<T> {
   private T t;
   public void setItem(T t) { this.t=t; }
}
```

Il est possible d'exploiter cette classe ainsi, sans générer d'erreurs :

```
A<Integer> a=new A<Integer>();
A aRaw;
a=aRaw;
aRaw=a;
```

Bien entendu cette tolérance est pleine de danger:

```
A<Integer> a=new A<Integer>();
A aRaw;
a=aRaw;
aRaw=a;
a.setItem(5);
a.setItem("salut"); // erreur - le contrôle sémantique fonctionne
aRaw.setItem("salut"); // pas d'erreur ! - Il y a pourtant violation de type
```

Cette tolérance trop grande ne joue que pour la version *raw* de la classe générique et nous rappelle qu'en Java le type générique est effacé au profit du type réel *Object* dans la machine virtuelle. Ainsi la classe *A,* dans sa version *raw*, est analogue, au niveau implémentation, à la classe *A<Object>.*

Pourtant avec *A<Object>* le contrôle de type reprend ses droits :

```
A<Integer> a=new A<Integer>();
A<Object> aRaw;
a=aRaw;    // erreur : type mismatch cannot convert from A<Object> to A<Integer>
aRaw=a; // erreur : type mismatch cannot convert from A<Integer> to A<Object>
```

Il y a incompatibilité de type totale entre *A<Object>* et *A<Integer>,* et en particulier aucun lien d'héritage entre ces deux instanciations d'une même classe générique.

Il y a des situations où un tel lien d'héritage, ou au moins de conformité, serait pertinent. Java a introduit la possibilité d'exprimer cela grâce au type *joker* (*wilcard*)

Invariance des classes génériques

Les classes génériques ont du point de vue de l'héritage, et plus généralement du sous typage, un comportement classique. La classe *LinkedList<T>* hérite au sens classique du terme de la classe *List<T>* à condition bien sûr que le type générique soit le même. Ainsi, par exemple, *LinkedList<Integer>* hérite de *List<Integer>.*

Mais cela est une question distincte de celle qui consiste à étudier les relations de sous-typage éventuellement induites par l'existence de sous-type au niveau du type générique. De quelle façon le type incarné par une classe générique *varie-t-il* vis à vis des variations du type générique lui-même.

Pour illustrer cette problématique nous partons du fait, dans la suite, que la classe *Number* est une surclasse de *Integer.*

La classe *LinkedList<Integer>* est-elle un sous-type de *LinkedList<Number>*? Ou l'inverse? Lorsque *T* varie, cela implique-t-il dans les différents versions induites de *LinkedList<T>* des relations de sous-typage qui suivent (*covariance*) ou contrarient (*contravariance*) celles du paramètre générique?

La réponse à cette question est claire et peut être aisément testée : il n' y aucune relation a priori entre les variations du paramètre générique et celles de la classe générique correspondante.

Par exemple :

```
// LinkedList<T> n'est pas covariant par rapport à T
LinkedList<Number> list1=new LinkedList<Integer>(); // erreur !
```

Autrement dit : bien que *Integer* soit un sous-type de *Number, LinkedList<Integer>* n'est pas un sous-type de *LinkedList<Number>*

```
// LinkedList<T> n'est pas contravariant par rapport à T
LinkedList<Integer> list2=new LinkedList<Number>(); // erreur !
```

Autrement dit : bien que Integer soit un sous-type de *Number, LinkedList<Integer>* n'est pas non plus un sur-type de *LinkedList<Number>*

Covariance ou contravariance?

L'intérêt des questions qui précèdent réside dans l'objectif assigné au typage d'un façon générale. Plus celui-ci est sémantiquement riche, meilleur sera le contrôle de code effectué en amont par le compilateur.

Or l'option choisie ici par les concepteurs du langage semble précisément aller vers un appauvrissement sémantique. En effet l'intuition nous dicte ici qu'il y a certainement quelque chose à déduire concernant les relations entre les différentes versions d'une même classe générique obtenues par différentes instanciations de son paramètre générique. Examinons les deux hypothèses : covariance ou contravariance.

Covariance

La covariance impliquerait que *LinkedList<Integer>* soit un sous-type de *LinkedList<Number> :* autrement dit, tout ce que sait faire une *LinkedList<Number> , une LinkedList<Integer>* sait-elle le faire aussi?

Si la réponse à cette question était oui on pourrait écrire...

```
LinkedList<Number> list1=new LinkedList<Integer>(); // erreur !
```

… ce qui n'est pas le cas.

Une *LinkedList<Number>* nous permet d'itérer à travers un ensemble de *Number.* A cet égard une *LinkedList<Integer>* remplit parfaitement cette part du contrat puisqu'elle nous permet, elle, d'itérer à travers une liste d'*Integer* et que ces *Integer* sont, bien sûr, des *Number*.

Du point de vue de l'itération à travers la structure, et donc du point de vue de la lecture des éléments contenus, on voit que *LinkedList<Integer>* mériterait d'être, par covariance, un sous-type de *LinkedList<Number>.*

Examinons maintenant une autre caractéristique de la classe *LinkedList<Number>* : elle est capable d'accueillir de nouveau *Number.* Il s'agit typiquement d'une caractéristique associée à la méthode *add* de cette classe. Cette propriété là est en revanche plus problématique pour une instance de *LinkedList<Integer>.* En effet une telle instance ne peut pas accueillir, sans mettre en danger sa propre cohérence, un *Number,* puisqu'elle est censée ne contenir que des *Integer,* et qu'un *Number* n'est pas une sorte d'*Integer* !

La conclusion est la *covariance* d'une classe générique ne pourrait pas être assumée complètement sans introduire d'incohérence dans le système de typage.

Contravariance

Il vaut la peine de poser le même problème vis à vis d'une contravariance éventuelle de la classe générique vis à vis de son type générique.

Si la réponse à la question de la contravariance était oui on pourrait écrire...

```
LinkedList<Integer> list1=new LinkedList<Number>(); // erreur !
```

… ce qui n'est pas le cas.

La contravariance impliquerait que *LinkedList<Number>* soit un sous-type de *LinkedList<Integer> :* autrement dit tout ce que sait faire une *LinkedList<Integer> , une LinkedList<Number>* sait-elle le faire aussi?

LinkedList<Integer> nous permet d'itérer à travers un ensemble de *Integer.* Il est clair que *LinkedList<Number>* ne remplit pas cette part du contrat puisqu'elle ne nous permet, elle, que d'itérer à travers une liste de *Number* et que, bien sûr, ces *Number* ne sont en général pas des *Integer*.

A l'inverse, la classe *LinkedList<Integer>* est capable d'accueillir de nouveau *Integer* : il s'agit typiquement d'une caractéristique associée à la méthode *add* de cette classe. Cette propriété là est en revanche possible pour une instance de *LinkedList<Number>.* En effet une telle instance pouvant accueillir, sans mettre en danger sa propre cohérence, un *Number,* elle est tout à fait capable d'accueillir en particulier un *Integer,* puisqu'un *Integer* est une sorte de *Number.*

La conclusion est là aussi que la *contravariance* d'une classe générique ne pourrait pas être assumée complètement sans introduire d'incohérence dans le système de typage.

Conclusion provisoire

Il y a dans les paragraphes précédents des arguments qui montrent que sémantiquement le typage d'une classe *A<T>* évolue :

- dans le sens du paramètre générique *T* vis à vis des opérations à sémantique de "lecture". C'est la covariance.
- dans le sens inverse du paramètre générique *T* vis à vis des opérations à sémantique d' "écriture". C'est la contravariance.

On s'efforcera dans la suite de donner un sens aux mots *lecture* ou *écriture* qui ne sont pas des catégories clairement identifiées au sein du langage Java (elles le seraient davantage en C++).

Faute de pouvoir choisir entre ces deux aspects contradictoires on comprend mieux le choix effectué par le langage : celui d'une absence de relation a priori entre les différentes instanciations d'une même classe générique.

Faut-il pour autant se résoudre à la perte de sens qui en résulte ? Une *LinkedList<Number>* est-elle condamnée à refuser d'accueillir de nouveaux *Integer* ? Une *LinkedList<Integer>* est-elle condamnée à refuser qu'on la parcourt simplement pour les *Number qu'elle contient ?* La réponse provisoire est *oui* dans cette forme là exactement. La réponse définitive est *non*, mais au prix d'une complexification du système de typage générique incarnée notamment par l'introduction du *jocker.*

Type générique joker

Covariance des tableaux

On a déjà introduit les tableaux en indiquant qu'ils constituaient, avant la lettre, des sortes de structures génériques. Les tableaux sont des structures natives du langages et leur comportement vis à vis de la variance a été choisi dès le début : c'est celui de la covariance. Ainsi *Integer []* est un sous-type de Object*[]*.

Cela est commode comme l'illustre cette portion de code:

```
Number [] tab=new Integer[100];
for(int i=0;i<100;i++) tab[i]=i*i;
for(Number n:tab)  System.out.println(n);
```

Mais cela ne va pas bien sûr sans incohérence, comme le révèle la portion de code suivante dans laquelle une erreur de type n'est révélée qu'au *run-time* ce qui contradictoire avec l'objectif même du typage.

```
Object [] t=new Integer[100];
t[3]=5;
t[4]="salut";  // compilation OK
===> Exécution : java.lang.ArrayStoreException: java.lang.String
```

Rappelons que les tableaux ne supportent que la seule généralisation covariante. Ainsi la déclaration suivante serait refusée par le compilateur:

```
Integer [] tab=new Object[100];
```

Classe générique covariante

La notation *jocker,* incarné par le symbole *?,* combiné avec l'usage du mot clef *super, va* permettre de construire une classe générique possédant des propriétés de covariance analogues à celles exposées pour les tableaux, mais sans présenter les incohérences de ces derniers.

La notation est la suivante : *A<? super T>*.

Par exemple :

```
LinkedList<? super Integer> list=new LinkedList<Integer>();
```

Comme son formalisme le suggère la notation *<? super T>* désigne une compatibilité avec toute classe qui est *T* ou une surclasse de *T*. Dans l'exemple ci-dessus l'expression *<? super Integer>* désigne un type plus général, au sens large, que la classe *Integer*. L'acceptation de la déclaration *LinkedList<? super Integer> list=new LinkedList<Integer>()* signifie que du point de vue du compilateur une *LinkedList<? super Integer>* est également plus générale qu'une *LinkedList<Integer>*: la sémantique de la classe conteneur suit donc celle de son type générique, d'où la covariance annoncée.

L'expérimentation de list donne les résultats suivants:

```
LinkedList<? super Integer> list=new LinkedList<Integer>();
list.add(5);  // idem list.add(new Integer(5));
Integer n=list.get(0);
```

Il est en effet possible d'ajouter à *list* un *Integer* puisque cette structure est censée contenir des objets plus généraux qu'*Integer*. Il n'est en revanche pas possible de lire ces éléments en tant qu'*Integer* pour la même raison (il serait possible bien entendu de les lire en tant qu'*Object*)

Classe générique contravariante

La notation *jocker,* incarnée par le symbole *?,* combinée avec l'usage du mot clef *extends , va* permettre de construire une classe générique possédant des propriétés de contravariance.

La notation est la suivante : *A<? extends T>*.

Par exemple :

```
LinkedList<? extends Integer> list=new LinkedList<Integer>();
```

Comme son formalisme le suggère la notation *<? extends T>* désigne une compatibilité avec toute classe qui est *T* ou une sous-classe de *T*. Dans l'exemple ci-dessus l'expression *<? extends Integer>* désigne quelque chose de plus spécialisée, au sens large, que la classe *Integer*. L'acceptation de la déclaration *LinkedList<? extends Integer> list=new LinkedList<Integer>()* signifie que du point de vue du compilateur une *LinkedList<? extends Integer>* est plus générale qu'une *LinkedList<Integer>*: la sémantique de la classe conteneur varie donc ici en sens inverse de celle de son type générique, d'où la contravariance annoncée.

L'expérimentation de list donne les résultats suivants:

```
LinkedList<? extends Integer> list=new LinkedList<Integer>();
list.add(5); // idem list.add(new Integer(5));
Integer n=list.get(0);
```

Il n'est en effet pas possible d'ajouter à *list* un *Integer* puisque cette structure est censée contenir des objets plus spécialisés qu'*Integer*. Il est par contre possible de lire ces éléments en tant qu'*Integer* pour la même raison.

Variance et paramètres

Les résultats précédents montrent que la notation avec *joker* nous permet de généraliser de deux façons différentes une même classe générique. Avec *super* nous obtenons une généralisation covariante qui ne permet une exploitation satisfaisante de la structure qu'en écriture, alors qu'avec *extends* nous obtenons une généralisation covariante qui ne permet une exploitation satisfaisante de la structure qu'en écriture. Ces termes *lecture* et *écriture* nous sont suggérés par la sémantique même d'une *LinkedList* qui est d'être une structure conteneur destinée à accueillir des éléments ou à en permettre la lecture.

Néanmoins, pour le comprendre, le comportement très cohérent du compilateur est à mettre en relation avec des concepts plus fondamentaux du langage. Pour cela nous définissons une classe générique beaucoup plus simple que la classe *LinkedList*, la classe *Ref*.

```
class Ref<T> {
   private T t;
   public void set(T t) { this.t = t; }
   public T get() { return t; }
}
```

L'invocation de la méthode *set* sur une instance de *Ref* doit être accompagnée d'un apport d'information exprimé à travers le paramètre *t* de cette méthode. Vu du point de vue de l'objet nous parlerons de paramètre *in*.

A l'inverse le type *T* tel qu'il apparaît dans la méthode *get* correspond à une situation où l'instance de *ref* transmet de l'information à l'extérieur : nous parlerons de paramètre *out*.

Le comportement du compilateur est le suivant :

```
// Covariance
Ref <? super Integer> ref1=new Ref<Integer>();
ref1.set(new Integer(5)); // Paramètre IN OK
Integer i=ref1.get(); // Paramètre OUT erreur

// Contravariance
Ref <? extends Integer> ref2=new Ref<Integer>();
ref2.set(new Integer(5));// Paramètre IN erreur
Integer i=ref2.get(); // Paramètre OUT OK
```

Nous pouvons ainsi préciser les règles limitant l'usage de la classe généralisée avec joker:

- la généralisation covariante obtenue avec *super* autorise l'usage spécialisé du type réellement instancié pour les paramètres de méthodes (*in)* mais non pour les paramètres de retour (*out*)
- la généralisation contravariante obtenue avec *extends* autorise l'usage spécialisé du type réellement instancié pour les paramètres de retour de méthodes (*out)* mais non pour les paramètres d'entrée (*in*)

Exprimé plus simplement nous pouvons résumer en écrivant :

- ? super T préserve l'utilisation de la classe en "entrée" (in)
- ? extends T préserve l'utilisation de la classe en "sortie" (out)

Exemple d'utilisation des types joker

Cette complexification formelle du typage générique a un enjeu : celui d'être capable d'exprimer en Java des relations de généralisations ou de spécialisations entre types, y compris lorsque ceux-ci résultent de constructions génériques. L'argument est le même que celui qui a poussé l'approche orientée objet à introduire l'héritage et les relations de sous-typage qui en résultent : être capable d'écrire du code présentant un haut niveau de réutilisation et de découplage parce qu'il s'applique non seulement aux types référencés dans le code mais également à toutes leurs spécialisations.

Pour illustrer cela considérons la classe *Collections* fournie par le package *java.util* et constituant une des nombreuses classe de la JCF. Cette classe est une boite à outils : elle propose ensemble de méthodes permettant d'agir sur les conteneurs en *général*. Cette classe est décrite plus complètement dans le chapitre sur la JCF. Cette classe contient un catalogue de méthodes statiques qui sont des utilitaires pour la manipulation des collections. La plupart de ces méthodes sont génériques.

Sur ce modèle considérons la méthode statique et générique *add* qui permet d'ajouter le contenu d'une liste à une autre liste. La signature et le code de cette méthode sont les suivants :

```
public static <T> void add(List<? super T> dst,List<? extends T> src) {
    for(T t:src) dst.add(t);
}

// test
LinkedList<Object> list1=new LinkedList<Object>();
LinkedList<Integer> list2=new LinkedList<Integer>();
for(int i=0;i<1000;i++) list2.add(i*i);
add(list1,list2);
System.out.println(list1.size());  // affiche 1000
add(list2,list1);
```

La méthode *add* est compatible avec toutes classes dérivées de *List* (*ArrayList*, *LinkedList* etc...), cela résulte des règles habituelles de l'héritage .... Et cette méthode *add* est également compatible avec tous les types des éléments contenus dans ces listes, et préservant la cohérence du typage, cela résulte de l'utilisation de *?, extends* et *super*.

Sans ce formalisme la méthode *add* pouvait se présenter ainsi, et rendre, faute de pouvoir formuler un principe de généralisation sur le type générique, illégale l'adjonction d'un ensemble d'*Integer* dans une liste d'*Object*. *:*

```
public static <T> void add(List<T> dst,List<T> src) {
   for(T t:src) dst.add(t);
}

public static void main(String[] args) {

// test
LinkedList<Object> list1=new LinkedList<Object>();
LinkedList<Integer> list2=new LinkedList<Integer>();
for(int i=0;i<1000;i++) list2.add(i*i);
add(list1,list2);
System.out.println(list1.size());  // affiche 1000
add(list2,list1);
```

Remarques et recommandations

Principe de l'effacement

Les principe d'effacement des types génériques repose sur l'idée qu'en Java le code généré à partir d'une classe générique est identique à celui généré à partir de la classe équivalente non générique. Ainsi la présence de deux classes génériques instanciées *A<Integer> et A<Date>* ne génère qu'une seule version compilée (celle issue de *A<T>*) sous la forme du fichier *A.class*.

L'information sur le type associé au paramètre générique (c'est à dire *Integer* et *Date* dans cet exemple) est, après avoir été utilisée par le compilateur pour effectuer son contrôle sémantique, perdue pour le *runtime*.

Rappelons qu'à l'inverse en C++ chaque classe générique instanciée produit autant de version d'elle-même, au *runtime*, que de types d'instanciation.

Le principe d'effacement des génériques propre à Java suit certaines règles, illustrées ci-dessous :

- l'effacement de *A<Integer>* et *A<Date>* est *A*
- *l'effacement de Integer et Date est respectivement Integer et Date*
- l'effacement de *int* est *int*
- l'effacement d'un paramètre générique *T* est généralement *Object*
- l'effacement d'un paramètre générique contraint *T* est le type contraignant. Par exemple *<T extends Comparable>* est effacé par *Comparable*

Rappelons que c'est ce principe d'effacement des paramètres génériques qui permet au code généré par un compilateur 1.5 (version qui a introduit les génériques en java) d'être largement compatible avec les machines virtuelles antérieures et au code source antérieur de l'être avec cette version du compilateur.

Il est utile d'avoir en tête ce principe d'effacement pour comprendre certaines des limites de l'usage des génériques, illustrées ci-après.

Paramètres génériques et membres statiques

Un membre statique ne peut faire référence à un type générique de sa classe.

```
class A<T> {
   static private T t; // interdit
}
```

L'attribut statique *t* ci-dessus est partagé, comme toute entité statique, par toutes les instances de sa classe d'appartenance. Mais en raison de l'effacement des types génériques cela signifie aussi que toutes les instances de *A<Date>* et de *A<Integer>* partagent également cet attribut unique. On voit donc le problème sémantique que cela pose : le même objet *t* ne peut être considéré comme une *Date* dans *A<Date>* et comme un *Integer* dans *A<Integer>.*

De même une méthode statique ne peut pas invoquer un paramètre générique de sa classe d'appartenance.

```
class A<T> {
   static private void m(T t) {} // interdit
}
```

Néanmoins une méthode statique peut faire usage d'un type générique si ce n'est pas celui de sa classe. C'est une situation fréquente, voir, par exemple, la classe *Collections* de la *JCF*.

```
class A<T> {
   static private <K> void m(K k) {}
}
```

Restrictions d'usage sur les jokers

La création d'une instance en orienté objet prend typiquement la forme suivante : *A a=new A()*

Les deux occurrences de la lettre *A* n'ont pas le même positionnement sémantique:

- le *A* de gauche est déclaratif. Il indique au compilateur que la variable *a* devra être dans la suite considérée en tant que *A*.
- le *A* de droite est d'une nature plus opératoire. Il doit permettre au compilateur de disposer des informations lui permettant très concrètement de créer une instance en mémoire.

Dans un contexte de polymorphisme il est possible de découpler ces deux aspects. Ainsi dans l'expression suivant *Comparable n=new Integer()* l'instance référencée par *n* est créée sur la base des informations fournies par *Integer,* mais par la suite c'est uniquement en tant que *Comparable* que la variable *n* sera considérée par le compilateur et elle en perdra même sa capacité à être additionnée à un autre entier...

```
Comparable n=6;  // équivaut à Comparable n=new Integer(6)
n=n+2; // erreur
```

Le type de gauche (ici *Comparable*) est ici une *interface*, c'est à dire une pure spécification. Le type de droite (ici *Integer*) est une classe concrète, c'est à dire à la fois une spécification et une machine à produire des instances.

Autrement dit :

- Une *classe* est un *type* ; un *type* n'est pas forcément une *classe*
- Un *type* est une *spécification* ; une *classe* est une *spécification* et une réalité *opératoire*

L'usage des joker permet, d'une façon générale, d'enrichir le langage de spécification dont dispose le langage. Les jokers permettent d'exprimer des nouveaux types, mais non de nouvelles classes.

Pour cette raison on retrouvera le plus souvent les jokers dans la partie type déclaratif et non dans la partie droite.

Ainsi :

```
List<?> list1=new LinkedList<Integer>();
LinkedList<?> list2=new LinkedList<Integer>();
List<? extends Number> list3=new LinkedList<Integer>();
List<? super Number> list4=new LinkedList<Integer>(); // erreur : cannot convert from
LinkedList<Integer> to LinkedList<? super Number>
List<? super Integer> list5=new LinkedList<Integer>();
List<?> list6=new LinkedList<?>(); // erreur : cannot instanciate the type LinkedList<?>
List<Integer> list7=new LinkedList<? extends Number>(); // erreur : cannot instanciate the
                                                        // type LinkedList<? extends Number>
```

Java Collections Framework.

Une arborescence

Les classe de conteneurs sont destinées à fournir les structures de données évoluées (listes, files, tableau, arbre etc...), les outils pour les manipuler efficacement, et un typage strict révélant autant que possible les incohérences éventuelles de leur mise en oeuvre.

Le projet est analogue à celui mené dans le cadre *C++* avec la *STL* (*Standard Template Library*). Les techniques orientées objets mises en oeuvre s'en rapprochent parfois : usage intensif de la généricité, approche fonctionnelle.

Par contre là où la *STL* utilise systématiquement la surcharge d'opérateurs (qui n'existe pas en Java) la Java Collection Framework utilise massivement le polymorphisme, l'héritage et les interfaces.

Concevoir une telle librairie est particulièrement difficile. Les problèmes de complexité et d'efficacité sont cruciaux pour la manipulation des données et aucune structure de données fondamentale ne peut posséder toutes les qualités pour tous les types de manipulations (recherche d'élément, insertion, suppression, caractère ordonné ou non, tri, réarrangement, accès indicé, accès par clé de hachage etc...). La *STL* en est une illustration qui a réussi la gageure de fournir une bibliothèque complète d'objets que l'utilisateur peut adapter finement à ses besoins au moyen, en particulier, de la généricité. Mais le prix de cette réussite est une certaine complexité d'utilisation, qui résulte des choix de conception déroutant (surcharge d'opérateur, subtilité du passage par référence, paramètre générique par défaut systématique, écart avec le modèle objet etc.) et confère à cette bibliothèque une incontestable dimension ésotérique.

Java, qui revendique un positionnement de *C++* simplifié, ne pouvait proposer une bibliothèque caractérisée par une sophistication formelle aussi importante.

Au plus bas niveau cette bibliothèque prend la forme d'un ensemble de classes concrètes caractérisées souvent par le type d'implémentation de structure de données sous-jacente. C'est cette structure de données qui va conditionner l'efficacité de chaque algorithme de manipulation. Elle est parfois appelée *structure de données fondamentales* (*SDF*). Quelques grandes catégories de *SDF* sont listées ci-dessous : les tableaux, les tables de hachage, les listes chaînées, les arbres. Des opérations telles la recherche d'un élément, l'ajout ou la suppression d'élément, l'insertion, le tri etc... auront une complexité qui dépendra de façon décisive de la structure choisie.

Le tableau ci-dessous présente les classes concrètes principales en relation avec leur structure de données fondamentale :

- Héritage

Java Collection Framework		*Implementation (SDF)*				
		Table de hachage	Tableau redimen-sionnable	Arbre équilibré	Liste chaînée	Hachage + liste chaînée
Interface	Set	`HashSet`		`TreeSet`		`LinkedHashSet`
	List		`ArrayList`		`LinkedList`	
	Deque		`ArrayDeque`		`LinkedList`	
	Map	`HashMap`		`TreeMap`		`LinkedHashMap`

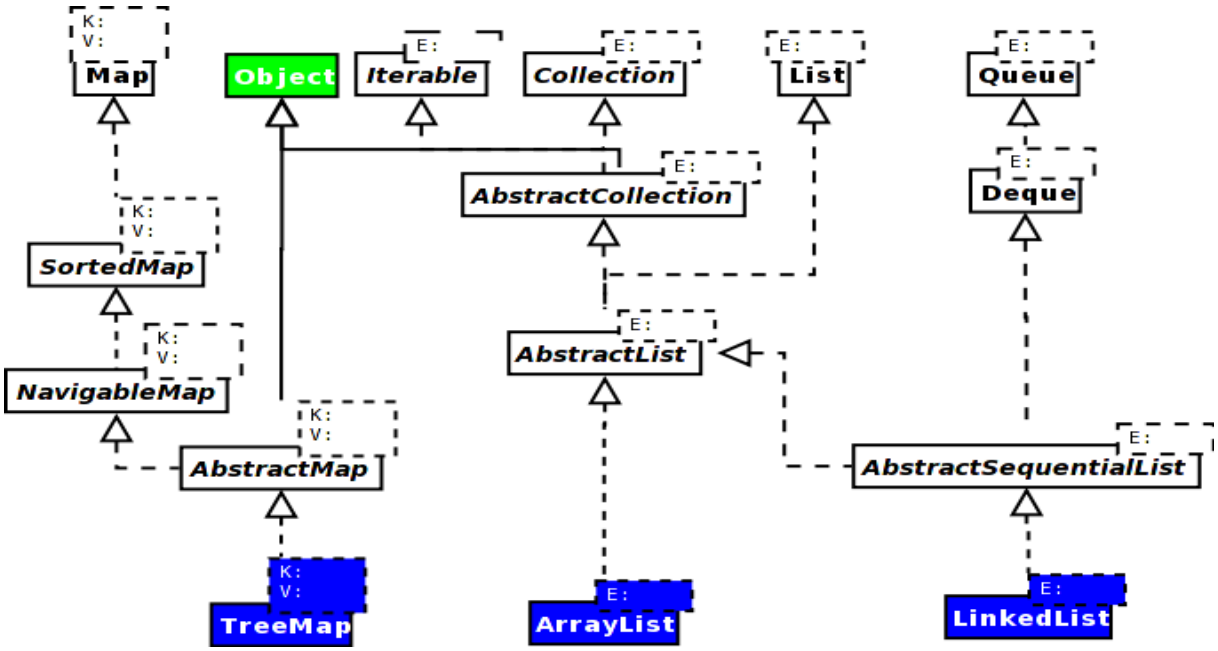
Ces classes, dont la liste ici n'est pas exhaustive, sont des exemples de classes concrètes finalement utiles à l'utilisateur. Elles sont l'aboutissement d'une hiérarchie de classes complétée par un graphe d'interface globalement structurée du haut en bas en fonction du niveau d'abstraction de l'entité.

Dans les couches supérieures on trouve les interfaces, qui ne font donc que spécifier sans les définir les services offerts à leur niveau.

Les classes commençant par le mot *Abstract* sont abstraites : elles se distinguent des interfaces dont elles héritent par le fait qu'elles ont tenté, chaque fois que le niveau d'abstraction auquel elles se situent le permet, de définir certaines des méthodes qu'elles ont reçues en héritage. Elle n'ont bien sûr pas pu le mener complètement ce travail sinon elles ne seraient pas restées abstraites : néanmoins celles des méthodes qui ont reçu une définition sont d'emblée disponibles par héritage pour les sous-classes. Globalement, on voit que les classes abstraites permettent de mettre en oeuvre une factorisation de code efficace et structurée.

Enfin les classes concrètes, parce qu'elles sont situées à un niveau qui permet de les définir, offrent pour tous les services exposés un comportement défini.

Le schéma suivant montre une portion de cet arbre d'héritage. Les classes concrètes (susceptibles donc de produire des instances opératoires) sont dessinées avec un fond grisé.



On remarquera également le rôle important joué par la généricité dans cette architecture.

Les itérateurs

Tout comme *SmallTalk* ou *C++* et sa *STL* Java a repris le concept d'itérateurs pour ses objets conteneurs. Les classes concrètes héritant de l'interface générique *Iterator* mettent, par la méthode iterator, à la disposition de l'utilisateur une entité appelée *itérateur* . Cet objet est spécialisé dans la navigation à travers l'objet conteneur auquel il a été associé à sa création. Lorsqu'une classe conteneur ne semble pas vous fournir directement toutes fonctionnalités que vous souhaitez en matière de navigation ou de suppression d'élément c'est qu'il faut compléter ce jeu opératoire en instanciant l'itérateur associé. L'interface *Iterator* annonce les services très simples suivants :

```
boolean hasNext() : Teste la présence d'un élément suivant
E next() : passe à l'élément suivant et le retourne.
void remove(): Supprime de la collection le dernier élément retourné par l'itérateur au moyen de next.
```

L'itérateur permet de spécifier une position d'insertion d'un élément (pour les conteneurs qui sont compatibles avec cette notion).

La suppression d'une élément d'une collection est toujours une opération délicate. Elle s'effectue généralement au moyen de l'itérateur. La règle spécifie que la suppression concerne le dernier élément retourné par *next*.

Exemple :

```
LinkedList<Integer> ll=new LinkedList<Integer>();
for(int i=0;i<5;i++) ll.add(i);
System.out.println(ll);
Iterator<Integer> iter=ll.iterator();
System.out.println(iter.next());
System.out.println(iter.next());
iter.remove();
System.out.println(ll);
```

Sortie :

```
[0, 1, 2, 3, 4]
0
1
[0, 2, 3, 4]
```

La classe *Collections*

La classe *Collections* présente un catalogue de méthodes statiques, toutes génériques, permettant la manipulation des conteneurs génériques. La *STL* propose une approche très similaire avec sa bibliothèque *algorithm*.

```
static<T> boolean  addAll(Collection<? super T> c, T... elements)
static<T> Queue<T> asLifoQueue(Deque<T> deque)
static<T> int binarySearch(List<? extends Comparable<? super T>> list, T key)
static<T> int binarySearch(List<? extends T> list, T key, Comparator<? super T> c)
static<E> Collection<E>     checkedCollection(Collection<E> c, Class<E> type)
static<E> List<E>  checkedList(List<E> list, Class<E> type)
static<K,V> Map<K,V> checkedMap(Map<K,V> m, Class<K> keyType, Class<V> valueType)
static<E> Set<E> checkedSet(Set<E> s, Class<E> type)
static<K,V> SortedMap<K,V>  checkedSortedMap(SortedMap<K,V> m, Class<K> keyType, Class<V>
valueType)
static<E> SortedSet<E>      checkedSortedSet(SortedSet<E> s, Class<E> type)
static<T> void copy(List<? super T> dest, List<? extends T> src)
static boolean disjoint(Collection<?> c1, Collection<?> c2)
static<T> List<T> emptyList()
static<K,V> Map<K,V> emptyMap()
static<T> Set<T> emptySet()
static<T> Enumeration<T>    enumeration(Collection<T> c)
static<T> void fill(List<? super T> list, T obj)
static int frequency(Collection<?> c, Object o)
static int indexOfSubList(List<?> source, List<?> target)
static int lastIndexOfSubList(List<?> source, List<?> target)
static<T> ArrayList<T> list(Enumeration<T> e)
static<T extends Object & Comparable<? super T>> T max(Collection<? extends T> coll)
```

```
static<T> T max(Collection<? extends T> coll, Comparator<? super T> comp)
static<T extends Object & Comparable<? super T>> T min(Collection<? extends T> coll)
static<T> T min(Collection<? extends T> coll, Comparator<? super T> comp)
static<T> List<T>  nCopies(int n, T o)
static<E> Set<E> newSetFromMap(Map<E,Boolean> map)
static<T> boolean  replaceAll(List<T> list, T oldVal, T newVal)
static void   reverse(List<?> list)
static<T> Comparator<T> reverseOrder()
static<T> Comparator<T> reverseOrder(Comparator<T> cmp)
static void rotate(List<?> list, int distance)
static void shuffle(List<?> list)
static void shuffle(List<?> list, Random rnd)
static<T> Set<T> singleton(T o)
static<T> List<T>  singletonList(T o)
static<K,V> Map<K,V> singletonMap(K key, V value)
static<T extends Comparable<? super T>> void    sort(List<T> list)
static<T> void sort(List<T> list, Comparator<? super T> c)
static void swap(List<?> list, int i, int j)
static<T> Collection<T> synchronizedCollection(Collection<T> c)
static<T> List<T>  synchronizedList(List<T> list)
static<K,V> Map<K,V> synchronizedMap(Map<K,V> m)
static<T> Set<T> synchronizedSet(Set<T> s)
static<K,V> SortedMap<K,V>  synchronizedSortedMap(SortedMap<K,V> m)
static<T> SortedSet<T> synchronizedSortedSet(SortedSet<T> s)
static<T> Collection<T>unmodifiableCollection(Collection<? extends T> c)
static<T> List<T>  unmodifiableList(List<? extends T> list)
static<K,V> Map<K,V> unmodifiableMap(Map<? extends K,? extends V> m)
static<T> Set<T> unmodifiableSet(Set<? extends T> s)
static<K,V> SortedMap<K,V>  unmodifiableSortedMap(SortedMap<K,? extends V> m)
static<T> SortedSet<T> unmodifiableSortedSet(SortedSet<T> s)
```

Exemples comparés (*JCF* / *STL*)

Nous comparons ci-dessous l'usage de la *Java Collectin Framework* (*JCF*) et de la *Standard Template Library* (*STL*). Ces deux librairies constituent les propositions standards des environnements Java et C++ concernant la manipulation des conteneurs.

Afficher les éléments d'un conteneur

Java Collection Classes	C++-STL
`// Version 1 simple` `public static void main(String[] args) {` `  // init` `  ArrayList<Integer> v1=new ArrayList<Integer>();` `  for(int i=0;i<10;i++) v1.add(i);` `  // affichage` `  for(Integer n:v1)` `  System.out.println(n);` `}`	`// Version 1 simple` `void main() {` `  // init` `  vector <int> v1;` `  for(int i=0;i<10;i++) v1.push_back(i);` `  // affichage` `  for(vector<int>::iterator p=v1.begin();` `  p!=v1.end();p++)` `  cout<<*p<<endl;` `}`
`// pas d'équivalence stricte`	`// Version 2 avec fonction-objet générique` `template <class T> struct Printor {` `  void operator()(T t) { cout<<t<<endl; }` `};` `void main() {` `  // init` `  vector <int> v1;` `  for(int i=0;i<10;i++) v1.push_back(i);` `  // affichage` `  for_each(v1.begin(),v1.end(),Printor<int>());` `}`
`// pas d'équivalence stricte`	`// Version  stream` `// init` `vector <int> v1;` `for(int i=0;i<10;i++) v1.push_back(i);` `// affichage` `copy(v1.begin(),v1.end(),` `  ostream_iterator<int>(cout,"\n"));`

Ordre des éléments d'un conteneur

Java Collection Classes

```
// mettre en désordre
// init
ArrayList<Integer> v1=new ArrayList<Integer>();
for(int i=0;i<10;i++) v1.add(i);
//action
Collections.shuffle(v1);
// affichage
for(Integer n:v1)
   System.out.println(n);
}
```

```
// permuter les deux premiers éléments
// init
ArrayList<Integer> v1=new ArrayList<Integer>();
for(int i=0;i<10;i++) v1.add(i);
//action
Collections.swap(v1, 0, 1);
// affichage
for(Integer n:v1)
    System.out.println(n);
}
```

```
// Incrémenter chaque élément de la liste
// init
ArrayList<Integer> v1=new ArrayList<Integer>();
for(int i=0;i<10;i++) v1.add(i);
//action
for(int i=0;i<v1.size();i++) {
    int v=v1.get(i)+1;
    v1.set(i, v);
}
// affichage
for(Integer n:v1) System.out.println(n);
```

```
// supprimer les éléments pairs d'une liste
// init
LinkedList<Integer> v1=new LinkedList<Integer>();
for(int i=0;i<10;i++) v1.add(i);
// action
Iterator<Integer> iter=v1.iterator();
while (iter.hasNext()) {
    if (iter.next()%2 == 0) iter.remove();
}
// affichage
for(Integer n:v1) System.out.println(n);
```

```
// Construire une liste des éléments pairs par
// extraction d'une liste d'entiers
// init
ArrayList<Integer> v1=new ArrayList<Integer>(),
ArrayList<Integer> v2=new ArrayList<Integer>();
for(int i=0;i<10;i++) v1.add(i);
    // action: construction de la seconde liste
    // et suppresion des éléments pairs de la iste 1
    Iterator<Integer> iter=v1.iterator();
    while (iter.hasNext()) {
       int numPair=iter.next();;
       if (numPair%2 == 0) {
           v2.add(numPair);
           iter.remove();
       }
    }
```

C++-STL

```
// mettre en désordre
// init
vector <int> v1;
for(int i=0;i<10;i++) v1.push_back(i);
//action
random_shuffle( v1.begin( ), v1.end( ) );
// affichage
for(vector<int>::iterator
          p=v1.begin();p!=v1.end();p++)
    cout<<*p<<endl;
```

```
// permuter les deux premiers éléments
// init
vector <int> v1;
for(int i=0;i<10;i++) v1.push_back(i);
//action
vector<int>::iterator p1=v1.begin(),p2=v1.begin()+1;
swap(*p1,*p2);
// affichage
for(vector<int>::iterator
        p=v1.begin();p!=v1.end();p++)
    cout<<*p<<endl;
```

```
// Incrémenter chaque élément de la liste
// init
vector <int> v1;
for(int i=0;i<10;i++) v1.push_back(i);
//action
for(vector<int>::iterator p=v1.begin();p!=v1.end();p++)
    (*p)++;
// affichage
for(vector<int>::iterator p=v1.begin();p!=v1.end();p++)
       cout<<*p<<endl;
```

```
// supprimer les éléments pairs d'une liste
template <class T> struct ParityPredicator {
   bool operator()(T &t) {
      return t%2==0 ;
   }
};
void main() {
// init
list <int> v1;
for(int i=0;i<10;i++) v1.push_back(i);
// action
list<int>::iterator begin_modif;
begin_modif=remove_if(v1.begin(),v1.end(),
                      ParityPredicator<int>());
v1.erase(begin_modif,v1.end());
// affichage
for(list<int>::iterator p=v1.begin();p!=v1.end();p++)

    cout<<*p<<endl;
}
```

```
// Construire une liste des éléments pairs par
// extraction d'une liste d'entiers
template <class T> struct ParityPredicator {
    bool operator()(T &t) {
       return t%2==0 ;
    }
};
void main() {
    // init
    vector <int> v1,v2;
    vector<int>::iterator p;
    for(int i=0;i<10;i++) v1.push_back(i);
    // action
    p=v1.begin();
    for(p=v1.begin();p!=v1.end();p++)
       if (*p%2==0) v2.push_back(*p);
```

```
// affichage
System.out.println("Liste 1 :");
for(Integer n:v1) System.out.println(n);
System.out.println("Liste 2 :");
for(Integer n:v2) System.out.println(n);
```

```
    // effacement
    vector<int>::iterator modified;

modified=remove_if(v1.begin(),v1.end(),ParityPredicator<int
>());
    v1.erase(modified,v1.end());
    // affichage
    cout<<"Liste 1:"<<endl;
    for(p=v1.begin();p!=v1.end();p++) cout<<*p<<endl;
    cout<<"Liste 2:"<<endl;
    for(p=v2.begin();p!=v2.end();p++) cout<<*p<<endl;
```

```
// Construire une suite de Fibonacci
// init
ArrayList<Integer> v1=new ArrayList<Integer>();
v1.add(0);
v1.add(1);
// action
for(int i=2;i<10;i++)
    v1.add(v1.get(i-1)+v1.get(i-2));
// affichage
for(Integer n:v1) System.out.println(n);
```

```
// Construire une suite de Fibonacci
// init
vector <int> v1;
vector<int>::iterator p,p1,p2;
v1.push_back(0);
v1.push_back(1);
// action
for(int i=0;i<10;i++) {
    p1=v1.end()-1;
    p2=v1.end()-2;
    v1.push_back(*p1+*p2);
}
// affichage
cout<<"Liste 1:"<<endl;
for(p=v1.begin();p!=v1.end();p++)
    cout<<"fib("<<p-v1.begin()<<")="<<*p<<endl;
```

Autres Exemples

Annuaire téléphonique

Java Collection Classes

```
// Construire et trier un annuaire téléphonique
class Client implements Comparable<Client> {
    Client(String nom, String prenom, String tel) {
        _nom = nom;_prenom = prenom;_tel = tel;
    }
    String _nom;
    String _prenom;
    String _tel;
    public int compareTo(Client c) {
        return _nom.compareTo(c._nom);
    }
    @Override
    public String toString() {
        return _nom + " " + _prenom + " " + _tel;
    }
}
LinkedList<Client> annu = new LinkedList<Client>();
// init
annu.add(new Client("durand", "jean", "0425146898"));
annu.add(new Client("dupont", "anne", "0325146128"));
annu.add(new Client("martin", "luc", "0458146898"));
annu.add(new Client("duchateau", "leon","0625416898"));
annu.add(new Client("valles", "jules", "0963256898"));
annu.add(new Client("antoine", "charles",
                                        "0625446898"));
// action
Collections.sort(annu);
// affichage
for (Client client : annu)
    System.out.println(client);
```

```
// Supprimer de l'annuaire les entrées dont les
// numéros téléphoniques commencent par 06
class Client ... {
    ... comme ci-dessus
}
LinkedList<Client> annu = new LinkedList<Client>();
// init
```

C++-STL

```
// Construire et trier un annuaire téléphonique
struct Client {
    Client(char *nom,char *prenom,char *tel):
        _nom(nom),_prenom(prenom),_tel(tel){}
    string _nom;
    string _prenom;
    string _tel;
    bool operator<(const Client & c) {
        return _nom < c._nom; }
    };
ostream &operator<<(ostream &out,const Client &c) {
    return out<<c._nom<<" "<<c._prenom<<
                          " "<<c._tel<<endl;
}
typedef list <Client> Annuaire;
Annuaire annu;

void main() {
// init
    annu.push_back(Client("durand",
                         "jean","0425146898"));
    annu.push_back(Client("dupont",
                         "anne","0325146128"));
    annu.push_back(Client("martin",
                          "luc","0458146898"));
    annu.push_back(Client("duchateau",
                          "leon","0625416898"));
    annu.push_back(Client("valles",
                           "jules","0963256898"));
    annu.push_back(Client("antoine",
                           "charles","0625446898"));
    // action
    annu.sort();
    // affichage
    for(list <Client>::iterator p=annu.begin();
                                  p!=annu.end();p++)
        cout<<*p<<endl;
}
```

```
// Supprimer de l'annuaire les entrées dont les
// numéros téléphoniques commencent par 06
struct Client {
    ... idem ci-dessus
};

struct TelTerminator{
```

```
... comme ci-dessus
// action
Iterator<Client>  iter=annu.iterator();
while (iter.hasNext()) {
    Client client=iter.next();
    if (client._tel.startsWith("06")) iter.remove();
}
// affichage
for (Client client : annu)
System.out.println(client);
```

```
    TelTerminator(string prefix):_prefix(prefix) {}
    string _prefix;
    bool operator()(const Client &c) {
       return c._tel.find ( _prefix , 0 )==0;
    }
};
typedef list <Client> Annuaire;
Annuaire annu;

void main() {
    // init
    ...idem ci-dessus
    // suppression entrée telephone debutant par 06
    list <Client>::iterator begin_modif,p;
    begin_modif=remove_if(annu.begin(),annu.end(),
                                TelTerminator("06"));
    annu.erase(begin_modif,annu.end());
    // affichage
    for(p=annu.begin();p!=annu.end();p++)
          cout<<*p<<endl;
}
```

```
// Supprimer de l'annuaire les entrées dont les
// numéros téléphoniques commencent par 06
// (version avec Map)
public class Launcher {
    public static void main(String[] args) {
    class Client implements Comparable<Client> {
        Client(String nom, String prenom, String tel) {
            _nom = nom;_prenom = prenom;_tel = tel;
        }
        String _nom;
        String _prenom;
        String _tel;
        @Override
        public int compareTo(Client c) {
           return _nom.compareTo(c._nom);
        }
        public String toString() {
           return _nom + " " + _prenom + " " + _tel;
       }
}
    TreeMap<String,Client> annu;
    annu=new TreeMap<String, Client>();
    annu.put("durand",new
              Client("durand","jean","0425146898"));
    annu.put("durand",new
              Client("durand","jean","0425146898"));
    annu.put("dupont",new
              Client("dupont","anne","0325146128"));
    annu.put("martin",new
               Client("martin","luc","0458146898"));
    annu.put("ducha",new
               Client("ducha","leon","0625416898"));
    annu.put("val",new
                Client("val","jules","0963256898"));
    annu.put("hu",new
              Client("hu","charles","0625446898"));
    LinkedList<String> listKeyToRemove;
    listKeyToRemove=new LinkedList<String>();
    for(Map.Entry<String ,Client>
                        set:annu.entrySet())
        if (set.getValue()._tel.startsWith("06"))
           listKeyToRemove.add(set.getValue()._nom);
    for(String key:listKeyToRemove)
       annu.remove(key);
    // la liste auxiliaire sera supprimée
    listKeyToRemove.removeAll(listKeyToRemove);
    //affichage
    for(Map.Entry<String ,Client>
      set:annu.entrySet())
         System.out.println(set.getValue());
}
```

```
// Supprimer de l'annuaire les entrées dont les
// numéros téléphoniques commencent par 06
// (version avec Map)
// Le nom de la personne sert de clé.
struct Client {
    Client(char *nom="",char *prenom="",char *tel=""):
       _nom(nom),_prenom(prenom),_tel(tel){}
    string _nom;
    string _prenom;
    string _tel;
    bool operator<(const Client & c) {
       return _nom < c._nom;
    }
};
ostream &operator<<(ostream &out,const Client &c) {
    return out<<c._nom<<" "<<c._prenom<<" "<<c._tel<<endl;
}
struct TelTerminator{
    TelTerminator(string prefix):_prefix(prefix) {}
    string _prefix;
    bool operator()(const Client &c) {
       return c._tel.find ( _prefix , 0 )==0;
    }
};
// typedef list <Client> Annuaire;
typedef map <char *,Client> Annuaire;
Annuaire annu;
void main() {
    // init
    annu["durand"]=
      Client("durand","jean","0425146898");
    annu["dupont"]=
      Client("dupont","anne","0325146128");
    annu["martin"]=
      Client("martin","luc","0458146898");
    annu["ducha"]=
      Client("ducha","leon","0625416898");
    annu["val"]=
      Client("val","jules","0963256898");
    annu["hu"]=
      Client("hu","charles","0625446898");
    // suppression entrée telephone debutant par 06
    TelTerminator criteria("06");
    map <char *,Client >::iterator p;
    {
       vector <char *> vkey; // vecteur auxiliaire
       // reperage des noeuds a supprimer
       for(p=annu.begin();p!=annu.end();p++)
           if (criteria(p->second))
                       vkey.push_back(p->first);
       // suppression
       for(vector <char *>::iterator p=vkey.begin();
                                p!=vkey.end();p++)
           annu.erase(*p);
    }
    // le vecteur auxiliaire est supprimé
    // affichage
    for(p=annu.begin();p!=annu.end();p++)
       cout<<p->second<<endl;
}
```

Etude de cas - Index - Version 1

La classe *Index* maintient une liste de mots triés par ordre alphabétique. Chacun d'entre eux est suivi du ou des numéro(s) de page où ce mot a été rencontré. La classe *EntryWord* représente l'objet associé à un mot et à l'ensemble des numéros de page correspondant aux occurrences de ce mot dans le document considéré. La classe Index représente la classe de plus haut niveau. Elle présente essentiellement la méthode *addWord* permettant d'y insérer un mot et un numéro de page donné. Dans le cas où ce mot est absent il est ajouté (avec le numéro de page en question), sinon le numéro de page passé en argument est ajouté à la liste des numéros de page déjà associés à ce mot.

Java Collection Classes

```
public class Launcher {
    static public class Index {
       public void addWord(String word, int lineNum) {
           boolean found = false;
           for (EntryWord entryWord : _list)
              if (word.equals(entryWord.getWord())) {
                  entryWord.addNum(lineNum);
                  found = true;
                  break;
              }
           if (!found) {
              _list.add(new EntryWord(word, lineNum));
              Collections.sort(_list);
           }
       }
       private ArrayList<EntryWord>
                     _list=new ArrayList<EntryWord>();
       @Override
       public String toString() {
           String result="";
           for (EntryWord entryWord:_list)
              result+=entryWord+"\n";
           return result;
       }
    }
    static public class EntryWord
                  implements Comparable<EntryWord> {
       public EntryWord(String word, int lineNum) {
           _word = word;
           _lineList.add(lineNum);
       }
       public void addNum(int lineNum) {
           _lineList.add(lineNum);
       };
       public final String getWord() {
           return _word;
       }
       public final ArrayList<Integer> getNumLine() {
           return _lineList;
       }
       private String _word;
       public int compareTo(EntryWord entryWord) {
           return _word.compareTo(entryWord._word);
       }

       private ArrayList<Integer>
                  _lineList=new ArrayList<Integer>();
       @Override
       public String toString() {
           String result=_word+" ";
           for(Integer num:_lineList)
              result+=num+" ";
           return result;
       }
    }

    public static void main(String[] args) {
       Index index=new Index();
       index.addWord("chat",3);
       index.addWord("soupe",7);
       index.addWord("chat",11);
       index.addWord("chocolat",13);
       index.addWord("sapin",30);
       index.addWord("chat",37);
       index.addWord("chocolat",37);
       // affichage
       System.out.println(index);
```

C++-STL

```
class EntryWord {
public:
    EntryWord(char *nom,int lineNum):
        _nom(nom){_lineList.push_back(lineNum);}
    void addNum(int lineNum)
{ _lineList.push_back(lineNum); };
    friend ostream &operator<<(ostream &,const EntryWord
&);
    friend class Index;
    bool operator<(const EntryWord ew) { return
_nom<ew._nom; }
    string getWord() const { return _nom; }
    const vector <int> &getNumLine() const { return
_lineList; }
protected:
    string _nom;
    vector < int> _lineList;
};
ostream &operator<<(ostream &out,const EntryWord &ew) {
    out<<ew._nom<<" : ";
    vector <int>::const_iterator p;
    for(p=ew._lineList.begin();p!=ew._lineList.end();p++)
out<<*p<<",";
    return out<<endl;
}
class Index {
public:
    void addWord(char *nom,int lineNum) {
       for(list <EntryWord>::iterator p=_list.begin();
                        p!=_list.end();p++)
           if (p->_nom==nom) break;
       if (p!=_list.end())p->addNum(lineNum);
       else {
           _list.push_back(EntryWord(nom,lineNum));
           _list.sort();
       }
    }
    const list <EntryWord> &getList() const { return _list;
}
protected:
    list <EntryWord> _list;
    friend ostream &operator<<(ostream &,const Index &);
};
ostream &operator<<(ostream &out,const Index &index) {
    list <EntryWord>::const_iterator p;
    for(p=index._list.begin();p!=index._list.end();p++)
out<<*p;
    return out;
}
Index index;

void main() {
    // init
    index.addWord("chat",3);
    index.addWord("soupe",7);
    index.addWord("chat",11);
    index.addWord("chocolat",13);
    index.addWord("sapin",30);
    index.addWord("chat",37);
    index.addWord("chocolat",37);
    // affichage
    cout<<index<<endl;
    // idem avec parcours explicite des entrees de
    // l'index
    const list <EntryWord> &listEntry=index.getList();
    list <EntryWord>::const_iterator p;
    for ( p=listEntry.begin();p!=listEntry.end();p++)
```

```
       // idem avec parcours explicite des entrees de
       // l'index
       for(EntryWord entryWord:index._list)
           System.out.println(entryWord);
      // idem avec parcours explicite de chaque ligne
      for(EntryWord entryWord:index._list) {
           System.out.print(entryWord.getWord()+" ");
           for(Integer num:entryWord.getNumLine())
              System.out.print(num+" ");
           System.out.println("");
       }

    }

}
```

```
cout<<*p;
    cout<<endl;
    // idem avec parcours explicite de chaque ligne
    for ( p=listEntry.begin();p!=listEntry.end();p++) {
       const vector<int> &line=p->getNumLine();
       cout<<p->getWord()<<':';
       for (vector<int>::const_iterator q=line.begin();
           q!=line.end();q++) cout<<*q<<',';
       cout<<endl;
    }

}
```

Etude de cas Index - Version 2

On se propose ci-dessous de résoudre le même problème en utilisant cette fois le conteneur associatif map. Cette structure de données permet d'associer une valeur à un clé. Dans une map cette clé est unique : un mot constitue donc un candidat naturel pour une telle clé ; de plus l'ordonnancement offert par le type map sur la clé correspond à notre besoin. La valeur associée à chaque mot sera l'ensemble des numéros de page représenté ici par un vecteur d'entiers. C'est donc le type TreeMap<String, ArrayList<Integer>> (ou map< string,vector<int>> en C++) qui offrira ici la structuration naturelle des données requises pour l'objet Index.

Globalement cette structure map s'avère mieux adaptée au besoin de la classe Index que la structure de liste choisie précédemment. A l'inverse de l'étude précédente on ne ressent pas ici le besoin d'utiliser une classe auxiliaire, analogue à EntryWord, exprimant le couple (mot,vecteur).

Java Collection Classes

```
import java.util.LinkedList;
import java.util.List;
import java.util.Set;
import java.util.TreeMap;
public class Launcher {
  static public class Index {
     public void addWord(String word, int lineNum) {
       List<Integer>  listNum=treeMap.get(word);
         if (listNum!=null) {
            listNum.add(lineNum);
         }
         else {
           ArrayList<Integer> list=new
                                 ArrayList<Integer>();
              list.add(lineNum);
              treeMap.put(word,list );
           }
       }
       private TreeMap<String, ArrayList<Integer>>
    treeMap=new TreeMap<String, ArrayList<Integer> >();
     public String toString() {
       String result="";
       Set<String> set=treeMap.keySet();
       for(String key:set) {
           result+=key;
           for(Integer num:treeMap.get(key))
                                  result+=" "+num;
           result+="\n";
       }
       return result;
     }
    }

    public static void main(String[] args) {
           Index index=new Index();
           index.addWord("chat",3);
           index.addWord("soupe",7);
           index.addWord("chat",11);
           index.addWord("chocolat",13);
           index.addWord("sapin",30);
           index.addWord("chat",37);
           index.addWord("chocolat",37);
           //affichage
           System.out.println(index);
    }
}
```

C++-STL

```
#include <map>
#include <vector>
#include <string>
#include <iostream>
using namespace std;
class Index {
public:
    void addWord(char *word,int num)
                  { _map[word].push_back(num); }
    typedef map<string,vector<int> > Map;
    typedef map<string,vector<int> >::iterator
                                       Map_iterator;
    typedef map<string,vector<int> >::const_iterator
                                    Map_const_iterator;
    friend ostream &operator<<(ostream &,const Index &);
    const Map &getMap() const { return _map; }
protected:
    Map _map;

};
ostream &operator<<(ostream &out,const Index &index) {
    Index::Map_const_iterator pLine;
    vector<int>::const_iterator p;
for(pLine=index._map.begin();
          pLine!=index._map.end();
pLine++) {
       out<<pLine->first<<" : ";
       for(p=pLine->second.begin();
                           p!=pLine->second.end();p++)
           out<<*p<<",";
       out<<endl;
    }
    return out<<endl;
}

Index index;
int main() {
    index.addWord("chat",3);
    index.addWord("soupe",7);
    index.addWord("chat",11);
    index.addWord("chocolat",13);
    index.addWord("sapin",30);
    index.addWord("chat",37);
    index.addWord("chocolat",37);
    // affichage
    cout<<index<<endl;
    return 0;
```

	}

Exceptions

Le concept d'exception a été introduit à l'origine par le langage *ADA*. La notion d'exception vise à permettre à un programme de contrôler la survenue de situation ou d'évènements inattendus ou anormaux.

Les événements anormaux, qu'on appellera *exception* dans la suite, peuvent avoir de nombreuses origines : erreur de programmation (appel d'une méthode sur une variable *null* par exemple), erreur de l'utilisateur (dans une saisie par exemple), erreur mathématique (division par zéro par exemple), connexion perdue dans une *socket*, erreur d'entrée/sortie etc...

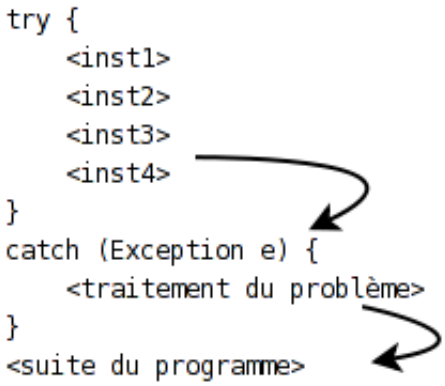
Pour une application robuste le code traitant ces situations anormales doit bien entendu exister, qu'il y ait ou non un mécanisme d'exception prévu au niveau du langage support. Ce qui rend l'affaire difficile c'est que, sans prise en compte spécifique de la part du langage, le code traitant de la situation anormale va se trouver souvent étroitement mêlé avec le code traitant de la situation normale. La conséquence est une complexité accrue du code de l'application et une difficulté accrue dans sa lisibilité, sa maintenance et son évolution.

Le mécanisme d'exception proposé par Java ne permet pas de faire l'économie du code traitant la situation anormale. Il va permettre en revanche de distinguer clairement la portion de code *normal* de celle en charge de la situation *anormale*. En outre, et grâce là aussi à l'héritage, une hiérarchisation des exceptions va être possible. Ainsi, par exemple, une exception d'entrée/sortie (*IOException*) est un cas particulier d'exception (*Exception*). Cette capacité à discriminer les cas d'exception est précieuse car elle permettra aussi de traiter l'exception au niveau adéquat. La vie réelle fourmille d'exemples : ainsi dans une maison, par exemple, on peut traiter localement l'exception constituée par un fusible qui grille par suite d'un court-circuit en le remplaçant ; c'est un situation où l'exception est alors traitée au moyen des compétences locales. Si, par suite de cet incident, la maison prend feu à un tel point que localement on ne peut résoudre ou traiter le problème, il faudra certainement relayer l'exception à une entité d'un autre niveau, à un traitant d'exception spécialisé, les pompiers en l'occurrence. Le mécanisme d'exception de Java permet une mise en oeuvre de telles métaphores : traitement local, ou propagation de l'exception à un niveau où elle pourra être traitée.

Le mécanisme d'exception

Le principe général est le suivant : un bloc de code (section *try*) traite la situation *normale*. Lors de l'exécution de ce bloc, dans le cas où une exception est *levée*, le flux d'exécution quitte le bloc et passe directement vers un autre bloc de code (section *catch*) chargé de traiter le problème.

En cas de survenue d'exception le schéma général est donc le suivant :



Dans cet exemple, l'exception a été levée pendant l'exécution de *<inst3>* , le flux quitte alors le bloc (*<inst4>* n'est pas exécutée) et le contrôle est passé au traitant d'exception. Ce déroutement du flux n'est pas un simple *goto* car il faut auparavant fermer correctement le contexte du bloc *try* (variables locales au bloc par exemple). Un fois le bloc *catch* exécuté, le flux exécute normalement la suite du programme.

Si aucune exception ne survient le flux d'exécution est le suivant : la section *catch* est ignorée.

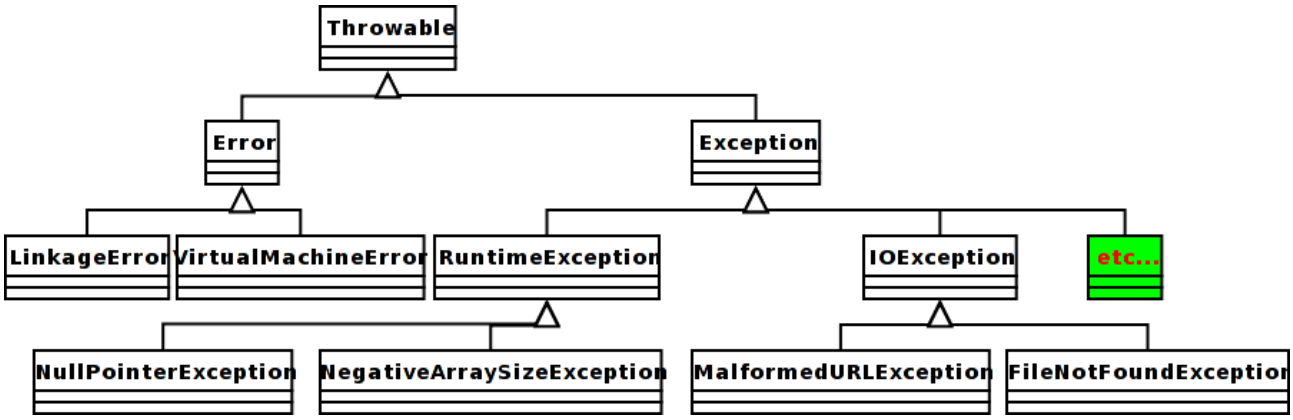
- Le mot réservé static

```
try {
    <inst1>
    <inst2>
    <inst3>
    <inst4>
}
catch (Exception e) {
    <traitement du problème>
}
<suite du programme>
```

Si une exception survient et qu'aucun traitant n'a été défini pour cette exception le comportement par défaut est l'arrêt du programme.

Hiérarchie des exceptions

Le mot réservé qui permet de lancer une exception est le mot *throw*. Au plus haut niveau de la hiérarchie des exceptions on trouve la classe *Throwable*. L'arbre, très incomplet, ci-dessous vise simplement à fournir les grandes catégories d'exceptions.



La branche *Error* de cet arbre d'héritage correspondent à des évènements sur lesquels en général le logiciel n'a aucun contrôle, et qui ne font pas normalement l'objet de section *try*/*catch*. Il s'agit d'évènements qui surviennent dans les couches inférieures comme l'illustre les deux classes fournies à titres d'exemple (*VirtualMachineError* et *LinkageError*). Un programme Java n'a normalement pas à invoquer ces classes.

La branche *Exception* en revanche est extrêmement utilisée. Parmi ses sous-branches, dont deux seulement sont représentées ici, celle initiée par *RunTimeException* correspond *grosso-modo* à des erreurs provoquées par le programme lui-même et dont le programmeur porte a priori la responsabilité. Les deux sous-classes retenues ici pour illustrer cela sont assez explicites : *NullPointerException* est levée lorsqu'on tente d'invoquer une méthode sur une variable *null* : c'est donc typiquement une erreur de programmation qui est à l'origine de ce problème. *NegativeArraySizeException*, dont le nom est suffisamment explicite, correspond au même type d'erreur.

Par contre *IOException* est un des nombreux représentants d'exceptions qui correspondent a priori à une cause externe : avec *FileNotFoundException* un utilisateur du logiciel a peut-être tenté d'ouvrir un fichier en fournissant un nom de fichier erroné, ou fourni une URL incorrecte (*MalformedURLException*).

Il est possible de définir des classes spécifiques d'exception propre à l'applicatif par greffe au niveau approprié de la nouvelle classe dans la hiérarchie d'exception. Une classe d'exception ne peut pas être générique.

Génération d'exceptions

Une exception survient lorsque quelque part dans le code des bibliothèques ou du programme applicatif une méthode *throw* est invoquée, par exemple :

```
throw new FileNotFoundException();
```

Dans ce cas une instance de *FileNotFoundException* a donc été créée et sera ou non traitée spécifiquement selon le contexte de récupération de cette exception.

Si rien n'a été prévu pour la traiter, le comportement par défaut est l'arrêt du programme. Si un traitant a été prévu c'est le code de la section *catch* correspondante qui constituera la réaction du programme à cet événement.

Le lancement d'une telle exception par *throw* peut constituer une décision explicite du programmeur de l'application. Mais le plus souvent la levée de l'exception sera le résultat de l'invocation d'une méthode appliquée à une classe ou à une instance et qui, directement ou non, invoque elle même *throw* dans son code, voir à ce sujet *Relayage d'exception avec ou sans chaînage.*

Déclaration d'exception

Le comportement par défaut d'une exception non traitée étant l'arrêt du programme il peut être intéressant d'être informé des exceptions qu'est susceptible de produire une portion de code donnée, plus précisément une méthode.

Java, tout comme C++, propose un moyen très puissant, non seulement d'informer l'utilisateur d'une méthode des exceptions que celle-ci risque de générer, mais aussi de l'obliger à prendre en compte par un traitant spécifique la survenue éventuelle de l'exception. Ce moyen consiste à enrichir le typage d'une méthode en y faisant figurer le type des exceptions que cette méthode peut générer.

Considérons l'exemple d'un constructeur de la classe *Socket* : *public Socket(String host,int port)*

Ce constructeur est invoqué pour créer une connection vers une machine hôte et le port passé en paramètres. Il est évident que cette création peut échouer pour plusieurs raisons, par exemple la machine hôte n'existe pas, ou elle n'a pas installé de service sur le port considéré.

Dans ce cas il semble normal que ce constructeur signale la survenue potentielle de ce type de problème et l'*oblige* par là même à en tenir compte.

Voici la signature complète de ce constructeur.

```
public Socket(String host, int port) throws UnknowHostException, IOException;
```

Remarquer l'usage du mot *throws* (avec un *s* à l'inverse celui déjà rencontrer, *throw*, qui permet de créer effectivement une exception) pour cette sémantique purement déclarative. La signature se lit ainsi :

"Le constructeur *Socket* attend un paramètre de type *String* et un paramètre de type *int* et est susceptible de lever l'exception *UnknownHostException* ou *IOException* "

L'aide en ligne concernant cet appel nous informe également que ce constructeur peut générer une autre catégorie d'exception :

```
public Socket(String host, int port)
       throws UnknownHostException,IOException
Creates a stream socket and connects it to the specified port number on the named host.
< etc... >
Throws:
UnknownHostException - if the IP address of the host could not be determined.
IOException - if an I/O error occurs when creating the socket.
SecurityException - if a security manager exists and its checkConnect method doesn't allow
the operation.
```

Le commentaire indique la survenue possible d'une exception n'apparaissant pourtant pas dans la signature de la méthode : c'est qu'il s'agit d'une exception héritière de *RuntimeException.* On a précisé que ce type d'exception (les exceptions héritières de *RunTimeException*) trouve son origine généralement dans une erreur du programme lui-même (par exemple un débordement de tableau). Pour ce type d'erreur, à l'inverse des autres anomalies qui ont une cause externe, on ne cherche pas à modifier le comportement du programme quand l'anomalie survient : en effet la seule solution à ce type d'exception consiste à corriger le programme lui même qui est seul responsable de sa survenue...

Autrement dit le mécanisme d'exception n'est pas une technique permettant d'accommoder un programme comportant des erreurs : c'est au contraire une technique qui permet de réagir sous contrôle à la survenue d'évènements externes anormaux.

On ne trouvera donc pas de déclaration du type :

```
public void method(String word) throws ArrayOutOfBoundException; // ERREUR
```

Une méthode ne déclarera dans sa section exception :

- ni la classe *Error* ou une héritière de *Error* ; on a vu que ces entités n'étaient de toute façon pas *catchable*.
- ni la classe *RunTimeException* ou une héritière de *RunTimeException* ; on a vu que la seule façon de parer à ces erreurs est de déboguer le programme lui même.

Traitement Hiérarchique des exceptions

La spécification d'une clause d'exception dans la signature d'une méthode oblige l'appelant à une prise en compte de sa survenue éventuelle. Le traitement peut être pris en charge par l'appelant directement ou pris en charge par une entité de plus haut niveau que l'appelant.

Traitement de l'exception par l'appelant

Reprenons l'exemple du constructeur de *Socket* et considérons que l'appel de ce contructeur est émis depuis une méthode appelée *openConnection* située dans une classe quelconque.

Situation 1 ; pas de prise en compte d'exception

```
void openConnection(String host, int port) {
   socket=new Socket(host,port);   // ERREUR : Unhandled exception type UnknownHostException
}
```

Cette situation génère une erreur à la compilation. Le système nous oblige bel et bien à tenir compte de l'exception éventuelle.

Situation 2 ; prise en compte locale d'exception incomplète

```
void openConnection(String host, int port) {
   try {
      socket=new Socket(host,port); ERREUR : Unhandled exception type IOException
   }
   catch (UnknownHostException e) {
      System.out.println("Hôte inconnu");
   }
}
```

Le système cette fois nous oblige à tenir compte du second type d'exception susceptible d'être levée par *Socket*.

Situation 3 ; prise en compte locale des exceptions

```
void openConnection(String host, int port) {
   try {
      socket=new Socket(host,port);
   }
   catch (UnknownHostException e) { System.out.println("Hôte inconnu"); }
   catch (IOException e) { e.printStackTrace(); }
}
```

La prise en compte des exceptions est cette fois suffisante. Le premier traitant affiche simplement ici *Hôte inconnu* . Il s'agit typiquement d'une exception sous contrôle de l'application. La seconde (qui résulte du fait que les sockets sont d'un point de vue système rattachées à la logique globale des fichiers) est un moyen terme typique d'une phase de mise au point de l'application : on y demande l'affichage sur la sortie d'erreur de la pile d'appels qui ont mené à cette situation. Cela illustre la capacité qu'offre le système à interroger une instance d'exception pour en tirer des informations sur l'anomalie en question.

Situation 4 ; prise en compte locale hiérarchisée des exceptions

```
void openConnection(String host, int port) {
   try {
      socket=new Socket(host,port);
   }
   catch (Exception e) { e.printStackTrace(); }
}
```

Dans une situation où on ne souhaiterait pas discriminer les deux situations d'exceptions il est possible d'utiliser la hiérarchisation des exceptions résultant de l'arborescence d'héritage. En raison des relations de conformité qui en résulte une *IOException* et une *UnknownHostException* sont toutes deux des *sortes d'Exception*. Une clause unique *catch (Exception e)* sera donc en mesure de capturer l'une et l'autre.

Situation 5 ; prise en compte locale avec une hiérarchie incorrecte des exceptions

Ce qui précède implique en particulier un usage logique de la hiérarchie des traitants d'exceptions.

```
void openConnection(String host, int port) {
   try {
      socket=new Socket(host,port);
   }
   catch (Exception e) { e.printStackTrace();}
   catch (IOException e) { // ERRREUR: Unreachable catch block for IOException.
                           // It is already handled by  the  catch block for Exception
   }
}
```

Cette organisation illogique des traitants est détectée par le compilateur. avec un message particulièrement clair.

Situation 6 ; prise en compte combinée locale et déléguée

La méthode *openConnection* peut être en mesure de traiter localement l'anomalie si c'est une *UnknownHostException*, mais se déclarer incompétente pour l'anomalie *IOException* . Dans ce cas si elle ne prévoit en conséquence pas de traitant pour *IOException* elle a l'obligation d'annoncer qu'elle est susceptible de renvoyer elle-même cette exception à l'entité qui l'a elle-même appelé. Ce qui reporte l'obligation de prévoir ou déléguer un traitement à cette entité appelante.

```
void openConnection(String host, int port) throws IOException {
   try {
      socket=new Socket(host,port);
   }
   catch (UnknownHostException e) { System.out.println("Hôte inconnu"); }
}
```

Relayage d'exception

Relayage sans chaînage

Une clause *catch* peut relayer, en l'enrichissant éventuellement, une exception. Son traitant va consister alors à lever une nouvelle exception. Une exception de couche basse, par exemple, pourra être réemballée pour la rendre compréhensible à l'entité de couche haute qui aura en charge de la traiter.

```
try {
   ... code ...
}
catch (IOException e) {
   throw new AccessSocketException("Accès à la socket interdit");
}
```

Cette classe *AccessSocketException* (qui n'existe pas dans la bibliothèque standard Java) peut avoir été créée en tant que nouvelle catégorie d'exception et pourvue d'informations qui seront utiles au traitant. Elle doit bien sûr hériter directement ou non de la classe *Exception* pour être utilisable dans un tel contexte.

La classe *Exception* présente un certain nombre de fonctionnalités :

```
// constructeurs
Exception(),Exception(String message), Exception(String message, Throwable cause),
Exception(Throwable cause).
// méthodes
fillInStackTrace, getCause, getLocalizedMessage, getMessage, getStackTrace, initCause,
printStackTrace, printStackTrace, printStackTrace, setStackTrace, toString
```

On voit qu'il est possible d'interroger une exception (*getStackTrace* etc.) pour s'informer sur les conditions d'exécutions qui ont mené à l'exception. Lorsqu'aucun traitant d'exception n'est spécifié le système appelle par défaut *printStackTrace* et interrompt le programme.

Relayage avec chaînage

Quelques versions du constructeur (ainsi que les méthodes *initCause getCause*) permettent en outre, lors de la création d'une exception ou après, de la chaîner avec l'éventuelle exception qui a provoqué sa levée.

L'exemple précédent peut être repris pour illustrer ce mécanisme de chaînage :

```
try {
   ... code ...
}
catch (IOException e) {
   throw new AccessSocketException("Accès à la socket interdit",e);
}
```

Cette fois on a pris soin d'indiquer à l'instance de *AccessSocketException* l'exception originelle *e*. Le traitant pourra éventuellement consulter la cause (et les éventuelles exceptions chaînées par ce mécanisme) ayant mené à la situation actuelle.

La métaphore précédente peut aider à comprendre l'intérêt de ce mécanisme : l'exception *fusible grillée* peut être traitée localement sur la base de cette seule information.

Si ce *fusible en grillant* a provoqué un incendie une nouvelle exception peut être générée à l'adresse des pompiers (contenant l'adresse de l'habitation où a eu lieu le sinistre, l'ampleur du sinistre etc.) à laquelle on pourra chaîner l'exception fusible grillé en tant qu'exception originelle.

La clause finally

La clause *finally* permet de rendre compatible le mécanisme d'exception avec certaines situations relevant d'une problématique de libération de ressource. Une application peut avoir à s'appuyer sur un contexte système qui possède sa propre logique d'allocation ou de libération de ressource. La ressource constituée par une connexion TCP, ou celle constituée par la possession d'un contexte de périphérique (en raison du mapping opéré alors sur les objets natifs du système Windows par exemple) ou encore la ressource d'accès à une base de données.

Considérons l'exemple suivant : la classe *Connexion* contient la méthode *openConnexion* combinant traitement local et délégué de deux catégories d'exceptions, où nous avons mis en commentaire un tel exemple d'allocation/libération de ressource (contexte SWT).

```
public class Connexion {
   private Object socket;
   void openConnexion(String host, int port) throws UnknownHostException {
      try {
         System.err.println("Ressource à libérer");//ex: Display
display=PlatformUI.createDisplay();
         socket=new Socket(host,port);
         System.err.println("Ressource libérée");  // ex : display.dispose();
      }
      catch (IOException e) {
         System.err.println(e);
      }
   }
}
```

Le programme de test est le suivant :

```
public class Launcher {
      public static void main(String[] args) {
         try {
            new Connexion().openConnection("www.enic.fr", 80);
         }
         catch (UnknownHostException e) {
            System.err.println(e);
         }
      }
}
```

La trace de cette application donne :

```
Ressource à libérer
Ressource libérée
```

La ressource a donc bien été libérée.

Supposons maintenant qu'une *IOException* survienne, la trace devient:

```
Ressource à libérer
java.io.IOException
```

on s'aperçoit cette fois que la ressource n'a pas été libérée. La solution peut consister ici à ajouter cette libération au traitant *catch* de *IOException*, et nécessite au passage de changer la localité de la variable *display* pour que cette variable soit visible dans le bloc *catch*.

```
public class Connexion {
   private Object socket;
   void openConnexion(String host, int port) throws UnknownHostException {
      // ex : Display display;
      try {
         System.err.println("Ressource à libérer");//ex: display=PlatformUI.createDisplay();
         socket=new Socket(host,port);
         System.err.println("Ressource libérée");        // ex : display.dispose();
      }
      catch (IOException e) {
         System.err.println(e);
         System.err.println("Ressource libérée");  // ex : display.dispose();
      }
   }
}
```

La trace devient alors :

```
Ressource à libérer
java.io.IOException
Ressource libérée
```

Cette fois la ressource est libérée.

Supposons maintenant que l'exception *UnknownHostException* soit maintenant levée par *new Socket(...)* . La trace devient :

```
Ressource à libérer
java.net.UnknownHostException
```

La ressource n'est donc pas libérée, mais cette fois la solution qui consisterait à ajouter le code de libération de la ressource est problématique puisqu'il faudrait le faire dans le traitant de *UnknownHostException* qui est situé en dehors du bloc local qui a procédé à la réservation de la ressource et possiblement dans un contexte ou l'évocation de la ressource en question n'est pas pertinente.

```
public class Launcher {
      public static void main(String[] args) {
         try {
            new Launcher().openConnection("www.enic.fr", 80);
         }
         catch (UnknownHostException e) {
            e.printStackTrace();
            System.err.println("Ressouce libérée"); // ?? display.dispose() ??;
         }
   }
}
```

La variable *display* n'est ici pas disponible pour opérer la libération de la ressource.

La clause *finally* permet de résoudre élégamment ce problème. Cette clause qui suit la section *try* ou *catch* spécifie une section de code qui sera toujours exécutée (même en cas d'exception dans les blocs qui précèdent) et qui sera exécutée *après* le traitement normal des exceptions (si elles ont eu lieu) des blocs qui précèdent.

La version qui suit fournit une allocation et la libération correspondante de la ressource dans tous les cas d'exécution :

```
public class Connexion {
   private Object socket;
   void openConnexion(String host, int port) throws UnknownHostException {
      // ex : Display display;
      try {
         System.err.println("Ressource à libérer");//ex: display=PlatformUI.createDisplay();
         socket=new Socket(host,port);
         }
```

```
      catch (IOException e) {
         System.err.println(e);
      }
      finally {
         System.err.println("Ressource libérée");  // ex : display.dispose();
      }
   }
}
```

Le programme de test est le suivant :

```
public class Launcher {
      public static void main(String[] args) {
         try {
            new Connexion().openConnection("www.enic.fr", 80);
         }
         catch (UnknownHostException e) {
            System.err.println(e);
         }
      }
}
```

Trace sans exception	Trace avec IOException	Trace avec UnknownHostException
Ressource à libérer Ressource libérée	Ressource à libérer java.io.IOException Ressource libérée	Ressource à libérer java.net.UnknownHostException Ressource libérée

Entrée/Sortie - Gestion de fichier

Introduction

Comme pour le système Unix, la métaphore du *flux* est un concept unificateur pour ce qui concerne la gestion de fichier. Un clavier est ainsi vu comme une entité depuis laquelle on peut recevoir un flux de code ASCII, un écran en mode console comme une entité vers laquelle on peut produire un flux de codes ASCII. Un fichier au sens traditionnel du terme est vu comme une entité vers ou depuis lequel un flux d'octets circulera. Une *socket* (c'est à dire un point d'accès à Internet) est vue au niveau du système comme un fichier, tout comme le *pipe* de communication entre deux processus etc.

Java reprend à son compte cette métaphore venue des systèmes d'exploitation. Ainsi en Java les fichiers seront essentiellement manipulés à travers des entités issues directement ou indirectement des classes abstraites *InputStream* ou *OutputStream*.

Ces flux se déclineront concrètement par la suite en fonction de leur nature plus ou moins affinée : la bibliothèque comporte un grand nombre de classes dédiées aux différents types de flux (flux d'octets, flux d'entiers, flux compressés, flux bufférisés, flux filtrés, flux issus de fichier, flux sérialisé etc...).

Certaines classes se distinguent pas la fourniture d'un niveau de service supplémentaire (par exemple détection de ligne), d'autres par la nature des entités constituant le flux.

Les objets susceptibles de produire ou consommer un flux peuvent être de nature très diverses (une socket, un fichier traditionnel, un tube de redirection, une URL etc...) avec leur propre jeu spécifique de méthodes. Néanmoins ces objets fourniront généralement deux méthodes permettant de leur adosser un flux en lecture ou en écriture, généralement une réalisation spécialisée de *InputStream* ou *OutputStream*. C'est à travers ces flux qu'on procédera aux opérations de circulations d'informations.

Exemple avec un objet de type *socket* .

```
socket=new Socket(host,port);
InputStream is=socket.getInputStream();
byte [] tab=new byte[1000];
is.read(tab, 0, 1000);
```

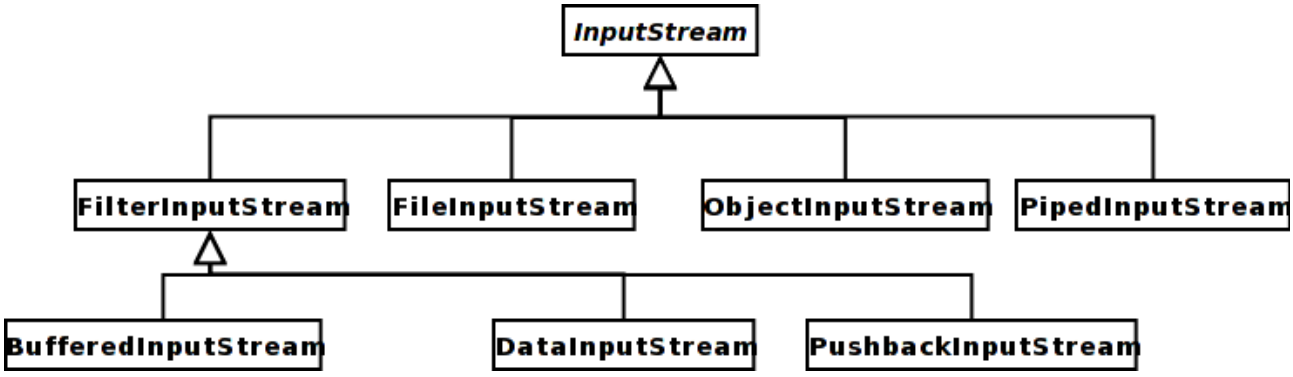
is est un flux instancié par l'appel *getInputStream*. Le type dynamique de *is* dans cet exemple est *SocketInputStream*, classe concrète héritière de *InputStream*.

Hiérarchie de quelques classes *stream*

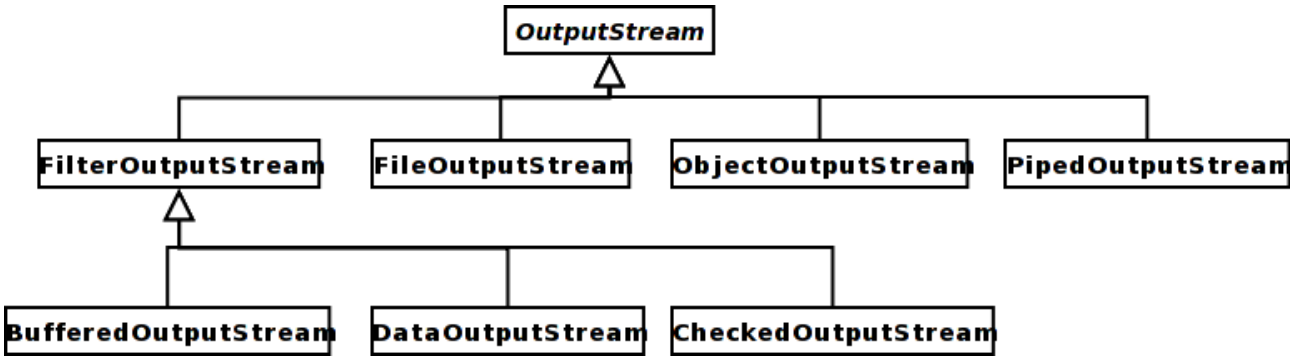
Les classes fournies dans le package *java.io* en matière de *stream* sont surabondantes. On ne présentera que quelques unes d'entre elles.

La classe *InputStream* est abstraite. La sous arborescence initiée par *FilterInputStream* définit un ensemble de classes fournissant chacune un niveau de service spécifique concernant les flux (par exemple capacité à gérer une bufférisation, capacité à lire des données numériques *natives int,* double etc., (*DataInputStream*), capacité à lire un octet sans le consommer (*PushbackInputStream*) etc... Ces services sont combinables en utilisant une technique d'empilement de filtres décrite dans le paragraphe suivant.

FileInputStream est un flux plus spécialement dédié aux fichiers, *ObjectInputStream* à la lecture d'objets sérialisés dans un flux, *PipedInputStream* est spécialisé dans les flux issus d'un tube de redirection etc...



De la même façon les flux d'écriture sont organisés hiérarchiquement.



Fichiers classiques

Les fichiers, au sens traditionnel du terme, peuvent être manipulés au moyen de la classe *File,* qui fournira les méthodes correspondant aux services et propriétés classiques de la notion de fichier (ouverture, cloture, nom de fichier etc...)

Pour lire ou écrire dans un tel fichier il faut faire appel à l'objet *stream* correspondant, c'est à dire *FileInputStream* ou *FileOutputStream.* Ces deux classes ne fournissent que les services d'écriture et de lecture élémentaires d'octets, ou de tableau d'octets.

```
public static void main(String[] args) {
   System.out.println(System.getProperty("user.dir")); // affichage repertoire courant
   try {
      // ecriture
      File file1=new File("exemple1");
      FileOutputStream f0s1=new FileOutputStream(file1);
      byte[] data="Hello World".getBytes();
      f0s1.write(data, 0, data.length);
      System.out.println(file1.length());
      // lecture
      FileInputStream fis=new FileInputStream(file1);
      while (fis.read(data)>0) {
         System.out.println(new String(data));
      }
   } catch (FileNotFoundException e) { e.printStackTrace();}
   catch (IOException e) { e.printStackTrace(); }
```

La sortie est :

```
/home/colin/workspacesEclipse/workspaceEclipsePeda/Peda11_FileStream
11
Hello World
```

Les filtres et leurs empilements

Le niveau de services offert par *FileOutputStream* (ou sa version *Input*) est à peu près analogue à ce qu'offre le macro-type *File ** en C classique. Le flux y est vu comme un flux d'octets sans sémantique ajoutée.

La bibliothèque d'entrée/sortie de Java (*java.io*) offre un moyen élégant de construire et manipuler, au dessus des flux basiques, des flux présentant du point de vue sémantique ou opératoire une valeur ajoutée : flux bufférisé, flux compressé, flux typé, flux avec remise, flux de texte etc...

Filtre de données

La technique consiste à construire ces instances de flux spécialisés à partir d'autre flux. Par exemple la classe *DataOutputStream* est pourvue, en autres choses, de méthodes permettant l'ajout de données natives numériques. La création d'un fichier de valeurs numériques *double* peut ainsi prendre la forme suivante :

```
File file2=new File("exemple2");
FileOutputStream fos2=new FileOutputStream(file2);
```

```
DataOutputStream dos2=new DataOutputStream(fos2);
for(int i=0;i<10000000;i++)
  dos2.writeDouble(i);
System.out.println(file2.length());
```

La sortie est :

```
80000000
```

La lecture de ce (gros) fichier peut être effectuée de la même façon:

```
File file2=new File("exemple2");
FileInputStream fis2=new FileInputStream(file2);
DataInputStream dis2=new DataInputStream(fis2);
double somme=0;
for(int i=0;i<10000000;i++) somme+=dis2.readDouble();
System.out.println("Somme = "+somme);
```

La sortie est la somme des 10 000 000 premiers entiers, qui s'affiche (sur la plate-forme utilisée) au bout de 18 secondes:

```
Somme = 4.9999995E13
```

Les classes *Data(Input/Output)Stream* permettent donc d'extraire depuis un flux les données relatives au principaux types de base en java : *int, short, long, double, float, char, byte, boolean*. Ce format de données est généralement efficace en terme de traitement puisqu'il consiste finalement à manipuler les données sous leur format natif mais il souffre d'un manque de portabilité. Alors qu'en C ce format natif est déterminé essentiellement par la plateforme système sous-jacente (déterminant par exemple le codage poids fort/poids faible pour les entiers), en java la plate-forme système sous-jacente est, du point de vue du compilateur, la machine virtuelle. En conséquence le format natif utilisé par les applications java sera toujours le même (*big-endian* par exemple pour les entiers, normalisé IEEE pour les flottants) et garantit ainsi une certaine interopérabilité des données natives entre applications java. Cette compatibilité ne s'étend bien entendu pas aux autres applications pour lesquelles on va retrouver les difficultés créées par la variété des formats natifs de données.

Les flux ASCII sont, on le sait, massivement utilisés dans les protocoles normalisés de l'IETF. Les méthodes *readByte*, *readLine* pourront par exemple être mise en oeuvre pour la lecture de ce type de flux (*readLine* notamment permet de commodément découper le flux incident par ligne de texte ASCII).

Les méthodes *readChar ou writeChar* ou *writeChars* sont relatives à la manipulation du type natif *char*. N'oublions pas que ce type de base est en java différent du type *char* du langage C. En java le type *char* est dédié au codage UTF Unicode qui généralement constitué d'unités de 16 bits. Ce sujet est traité également dans le paragraphe *Flux de texte*.

Filtre de bufférisation

La bufférisation est une technique qui permet d'optimiser les opérations de lecture et écriture en favorisant les transferts de bloc mémoire dans une zone tampon depuis ou vers un fichier.

Nous reprenons l'exemple précédent et pour mettre en oeuvre cette technique il suffit ici d'interposer entre l'objet *FileInputStream* et l'objet *DataInputStream* une instance de *BufferedInputStream* comme ci-dessous :

```
FileInputStream fis2=new FileInputStream(file2);
DataInputStream dis2=new DataInputStream(new BufferedInputStream(fis2));
System.out.println("File size = "+file2.length());
double somme=0;
for(int i=0;i<10000000;i++)     somme+=dis2.readDouble();
System.out.println("Somme = "+somme);
```

La somme des 10 000 000 premiers entiers s'affichent cette fois au bout de 4 secondes environ (au lieu de 18 secondes sans filtre de bufférisation)

Filtre de compression

La portion de programme suivante crée un fichier compressé au moyen d'un filtre spécialisé (*java.util.zip*). Trois filtres successifs sont empilés : le filtre fournissant le flux natif d'octets à partir du fichier, le filtre de bufférisation, le filtre de compression.

```
File file3=new File("exemple3");
FileOutputStream fos3=new FileOutputStream(file3);
ZipOutputStream zos3=new ZipOutputStream(fos3);
DataOutputStream dos3=new DataOutputStream(zos3);
```

```
ZipEntry ze=new ZipEntry("test");
zos3.putNextEntry(ze);
for(int i=0;i<2000000;i++) dos3.writeDouble(i);
zos3.closeEntry();
System.out.println("File size= "+file3.length());
```

La sortie montre que le fichier est effectivement comprimé d'un facteur 4 environ; sans compression sa taille serait de 16 000 000 d'octets.

File size= 4227402

Entrées-sorties de base

Les entrées-sorties de base concernent les opérations élémentaires de saisie d'informations ou d'affichage. Elles concernent à la fois la notion de flux, puisque l'information échangée avec le clavier ou l'écran est constituée d'un flux de codes ASCII, et des notions systèmes permettant de référencer les périphériques en question.

Des méthodes spécifiques, proposées par la classe *System* sont un moyen léger de mettre en place les différentes opérations logiques impliquées normalement dans l'opération, par exemple pour la saisie clavier : désignation du périphérique, acquisition d'un *InputStream* adossé à ce dispositif, adjonction d'un filtre adapté à la nature ASCII du flux (héritier de *Reader*) et enfin lecture proprement dite dans le flux.

Classe *System*

La classe *System* fournit donc ces méthodes d'entrée sortie standard accompagnées de différentes commodités. Mais comme son nom l'indique cette classe permet d'accéder à certaines informations concernant la plateforme d'exécution : accès aux variables d'environnement, accès à un ensemble d'informations système, gestion du *garbage-collector,* accès à des informations de temps, redirection d'entrée-sortie, accès aux périphériques standards d'entrée/sortie (*in, out, err*), accès à la console java, terminaison de programme, sécurité etc...

```
static Console console()
static long   currentTimeMillis()
static void   exit(int status)
static void gc()
static Map<String,String> getenv()
static String getenv(String name)
static Properties  getProperties()
static String getProperty(String key)
static SecurityManager     getSecurityManager()
static void   setErr(PrintStream err)
static void   setIn(InputStream in)
static void   setOut(PrintStream out)
static void   setSecurityManager(SecurityManager s
```

La portion de programme suivant expose les couples *variable/value* des différentes variables d'environnement:

```
Map<String,String> env=System.getenv();
for(String name:env.keySet()) System.out.println(name+"="+env.get(name));
```

Une petite portion de la sortie est la suivante :

```
LANG=fr_FR.UTF-8
TMP=/home/colin/tmp
HOME=/home/colin
LANGUAGE=fr_FR.UTF-8:fr
USER=colin
SHELL=/bin/bash
JAVA_HOME=/usr/lib/jdk-1_5_0_11/jre
<etc...>
```

La portion de programme suivant expose les couples *variable/value* des différentes propriétés du système courant:

```
Properties p=System.getProperties();
for(Object prop:p.entrySet()) System.out.println(prop);
```

Une petite portion de la sortie est la suivante :

```
java.runtime.name=Java(TM) 2 Runtime Environment, Standard Edition
sun.boot.library.path=/usr/lib/jdk-1_5_0_11/jre/lib/i386
java.vm.version=1.5.0_11-b03
```

```
user.dir=/home/colin/workspacesEclipse/workspaceEclipsePeda/Peda12_ReaderWriter
java.library.path=/usr/lib/jdk-1_5_0_11/jre/lib/i386/client:/usr/lib/jdk-
user.home=/home/colin
user.timezone=
```

La portion de programme suivant retrouve le répertoire courant d'exécution:

```
String path=System.getProperty("user.dir");
```

La sortie est la suivante :

```
PATH=/usr/lib/jdk-1_5_0_11/jre/bin/:/usr/bin:/bin:/usr/local/bin:/usr/X11R6/bin/
```

Entrée/sortie avec System

Les trois entrée-sorties standards apparaissent comme trois attributs statiques de la classe *System*.

```
public static PrintStream  err  ; The "standard" error output stream.
public static InputStream  in   ; The "standard" input stream.
public static PrintStream  out  ; The "standard" output stream.
```

La classe *PrintStream* hérite de *FilterOutputStream* et propose un certains nombres de méthodes d'accès commode au flux concerné.

Exemple :

```
System.out.print("Hello");
System.out.flush();
System.out.println("Hello");
System.out.printf("Exemple de formatage : %s\n","Hello");
System.out.printf("Exemple de formatage : %d + %d = %g\n",2,3,5.);
System.out.printf("Exemple de formatage : %+05d + %6d = %f\n",2,3,5.);
System.out.println(System.out);
System.err.print("Hello");
```

Sortie :

```
HelloHello
Exemple de formatage : Hello
Exemple de formatage : 2 + 3 = 5.00000
Exemple de formatage : +0002 +      3 = 5,000000
java.io.PrintStream@e5b723
Hello
```

L'entrée standard est classiquement un peu plus complexe dans la mesure où il faut prévoir une structure d'accueil pour les informations incidentes. L'exemple qui suit est calqué sur son équivalent en C et permet de saisir un texte terminé par un retour chariot (et non un autre séparateur)

```
byte [] tab=new byte[500];
try {
   System.in.read(tab);
} catch (IOException e) { e.printStackTrace(); }
System.out.println(new String(tab));
```

Heureusement la version 5 de java a apporté une commodité appréciable pour cette opération de saisie au moyen de la classe *Scanner*. Cette classe est en même temps un analyseur lexical simple capable de découper dans le flux qu'on lui soumet un certain nombre de types natifs ou plus élaborés.

L'exemple précédent est (à la limitation de la taille du tableau près) analogue à celui-ci :

```
Scanner scanner=new Scanner(System.in);
String name=scanner.nextLine();
System.out.println(name);
```

La saisie d'autres types de base s'effectue par :

```
int n=scanner.nextInt();              // saisie d'un entier
double x=scanner.nextDouble();  // saisie d'un double
String word=scanner.next();     // saisie d'un mot
```

Filtre de sérialisation

La sérialisation est l'opération consistant à sauvegarder un objet (c'est à dire une instance) vers un support de masse, incarné par un fichier par exemple.

La difficulté de l'opération a plusieurs origines :

- les références d'objets sont par nature polymorphes : il faut donc disposer d'un mécanisme qui tienne compte du type dynamique de l'objet.

- un objet peut référencer, par ses variables membres, d'autres objets. Si ces objets sont partagés entre plusieurs instances composant le flux courant d'objets à sérialiser il faut en éviter la duplication.
- toutes les données internes d'un objet ne doivent pas forcément faire l'objet d'une sérialisation.
- Une classe peut définir plus finement la façon dont elle se sérialise

Par ailleurs les types de bases sont pris en charge nativement (par implémentation de *DataInput* et *DataOutput* déjà utilisées par *DataOutputStream* par exemple).

Pour qu'une classe soit sérialisable il faut qu'elle implémente l'interface *Serializable*. Cette interface est un pur jeu d'écriture dans la mesure où elle n'expose aucune méthode. L'exemple simple qui suit sérialise un tableau de 10 compteurs, composés aléatoirement de *Compteur* ou de *CompteurDeb*. Il faut remarquer la clause d'héritage concernant *Serializable* et l'usage des filtres *ObjectOutputStream* (pour créer le flux sérialisé) et *ObjectInputStream* pour lire ce flux. Les méthode *toString* de ces deux classes ont été redéfinies pour mettre en évidence le support du polymorphisme dans le mécanisme de sérialisation. Il faut remarquer que l'entité sérialisée est le tableau lui-même ; il en résulte une grande simplicité de ce code.

```
class Compteur implements Serializable {
   private int value;
   public Compteur() { value=0; }
   public void up() { value++; }
   public void raz() { value = 0; }
   public int getValue() { return value; }
   public String toString() { return "" + value; }
}
class CompteurDeb extends Compteur {
   public static final int MAX=999;
   boolean deb;
   CompteurDeb() { super(); deb=false; }
   boolean getDeb() { return deb; }
   public void up() {
      if (getValue()==MAX) deb=true;
      else super.up();
   }
   public void raz() { super.raz();      deb=false; }
   @Override
   public String toString() { return "["+super.toString()+" "+deb+"]"; }
}

public class Launcher {
   public static void main(String[] args) {
      try {
         // Le fichier sérialisé est file4
         File file4=new File("exemple4");
         FileOutputStream fos4=new FileOutputStream(file4);
         ObjectOutputStream oos4=new ObjectOutputStream(fos4);
         Compteur [] tab=new Compteur[10];
         // création aléatoire de Compteur ou CompteurDeb
         System.out.println("Création aléatoire d'un tableau de Compteur ou CompteurDeb");
         for(int i=0;i<tab.length;i++) {
            if (Math.random()>0.5) tab[i]=new Compteur();
            else tab[i]=new CompteurDeb();
            for(int k=0;k<=i;k++) tab[i].up(); // donner une valeur à chaque compteur
            System.out.print(tab[i]+" ");
         }
         System.out.println("\n" +"Ecriture du flux sérialisé");
         oos4.writeObject(tab);
         System.out.println("File size : "+file4.length());
         // lecture
         ObjectInputStream ois=new ObjectInputStream(new FileInputStream(file4));
         Compteur [] readCompteur=(Compteur [] )ois.readObject();
         System.out.println("Lecture du flux sérialisé");
         for(Compteur c:readCompteur) System.out.print(c);
      } catch (FileNotFoundException e) {
      } catch (IOException e) {
      } catch (ClassNotFoundException e) {
      }
```

```
   }
}
```

La sortie permet de constater que le tableau d'objets a été reconstitué à l'identique :

```
Création aléatoire d'un tableau de Compteur ou CompteurDeb
1 2 3 4 [5 false] 6 [7 false] 8 9 [10 false]
Ecriture du flux sérialisé
File size : 198
Lecture du flux sérialisé
1234[5 false]6[7 false]89[10 false]
```

Donnée transient

On peut signifier qu'une donnée interne ne doit pas faire l'objet de la sérialisation en employant le mot réservé *transient* : il s'agit généralement de données qui n'ont de sens que dans le contexte complet d'exécution (une variable référençant une fenêtre d'une l'interface graphique par exemple).

Nous l'illustrons ci-dessous en déclarant *transient* la variable de débordement *deb* du compteur à débordement (et en positionnant la valeur par défaut de *deb* à *true*)

```
class CompteurDeb extends Compteur {
   public static final int MAX=999;
   transient boolean deb;
   CompteurDeb() { super(); deb=true; } < etc...>
```

Le même programme de test que précédemment donne la sortie suivante :

```
Création aléatoire d'un tableau de Compteur ou CompteurDeb
1 [2 true] 3 4 5 6 7 [8 true] [9 true] 10
Ecriture du flux sérialisé
File size : 189
Lecture du flux sérialisé
1[2 false]34567[8 false][9 false]10
```

Cette fois les valeurs booléennes de *deb* n'ont pas été sauvegardées : initialement à *true* pour les trois compteur à débordement, elles ont été restaurées, en l'absence de valeurs enregistrées dans le fichier, à leur valeur par défaut, c'est à dire *false*, lors de la désérialisation du flux. On peut remarquer également (à égalité de nombre de compteurs à débordement) que la longueur du fichier est plus petite.

La balise *transient* est à utiliser à chaque fois que la variable membre se révèle non pertinent pour une sérialisation, ou qu'elle référence une classe qui n'implémente pas elle-même l'interface *Serializable*.

```
class A { double x=45.23; }
class B implements Serializable {
   int n=5;
   A a=new A();
}
```

Dans cet exemple la sérialisation d'un flux de *B* génère une erreur d'exécution (l'incohérence n'est pas révélée à la compilation) avec l'exception "*java.io.NotSerializableException: A" .*

Il y a donc deux façons de régler ce problème :

- faire de *A* une classe implémentant *Serializable* et la valeur *x* est alors sérialisée
- faire de *a* une variable *transient* et la valeur de *x* n'est pas sérialisée

Format de sérialisation

La sérialisation permet de sauvegarder non seulement les données des objets qu'on lui soumet mais aussi leurs informations symboliques. On peut ainsi interroger un objet restitué par désérialisation et utiliser les outils de la *reflection* pour interroger l'objet.

```
ObjectInputStream ois=new ObjectInputStream(new FileInputStream(file4));
Object [] tab=(Object [] )ois.readObject();
// reflection sur une instance (la quatrieme)
for(Object o:tab[3].getClass().getMethods()) System.out.println(o);
```

La portion de code ci-dessus lit le fichier *exemple4* précédemment garni avec 10 compteurs polymorphes. Le type *Compteur* lui-même n'est pas évoqué explicitement dans ce code puisqu'on ne manipule les entités reçues du flux qu'avec des pointeurs polymorphes *Object* . On interroge, par exemple sur les méthodes de l'objet, celui placé en position d'index 3. Le résultat en sortie montre que le processus de sérialisation concerne à la fois les données et la sémantique de l'objet:

```
public java.lang.String Compteur.toString()
public int Compteur.getValue()
public void Compteur.up()
public void Compteur.raz()
public native int java.lang.Object.hashCode()
public final native java.lang.Class java.lang.Object.getClass()
public final native void java.lang.Object.wait(long) throws java.lang.InterruptedException
public final void java.lang.Object.wait(long,int) throws java.lang.InterruptedException
public final void java.lang.Object.wait() throws java.lang.InterruptedException
public boolean java.lang.Object.equals(java.lang.Object)
public final native void java.lang.Object.notify()
public final native void java.lang.Object.notifyAll()
```

Surcharge du mécanisme de sérialisation

Une classe implémentant l'interface *Serializable* peut surcharger le mécanisme de sérialisation proposé par défaut. Il lui faut pour cela définir les méthodes *readObject* et *writeObject* en respectant la signature ci-dessous:

```
private void writeObject(java.io.ObjectOutputStream out)
                throws IOException;
private void readObject(java.io.ObjectInputStream in)
                  throws IOException, ClassNotFoundException;
```

La classe Compteur ci-dessous ajoute à la sérialisation par défaut la sauvegarde de la date courante :

```
class Compteur implements Serializable {
   private int value;
   public Compteur() {value=0;}
   public void up() {value++;}
   public void raz() {      value = 0;    }
   public int getValue() {  return value; }
   public String toString() {      return "" + value;   }
   private void writeObject(java.io.ObjectOutputStream out) throws IOException {
      out.defaultWriteObject();
      Date date=new Date();
      out.writeObject(date);
   }
   private void readObject(java.io.ObjectInputStream in)
      throws IOException, ClassNotFoundException {
      in.defaultReadObject();
      Object o;
      System.out.println("saved on "+in.readObject());
   }
}
```

Il faut noter l'usage ci-dessus de *defaultReadObject* et de *defaultWriteObject* (voir l'interface *Serializable* dans l'aide en ligne) pour rappeler le comportement par défaut. S'y ajoute ici l'apposition d'une date de sauvegarde dans le flux.

La portion de code de test qui suit sérialise un compteur et le désérialise. La sortie montre

```
FileOutputStream fos=new FileOutputStream(file4);
ObjectOutputStream oos=new ObjectOutputStream(fos);
ObjectInputStream ois=new ObjectInputStream(new FileInputStream(file4));
Compteur c1=new Compteur();
oos.writeObject(c1);
Object o=ois.readObject();
```

La sortie montre que les versions surchargées de *readObject* et *writeObject* ont bien été invoquées:

```
saved on Tue Jun 26 16:10:20 CEST 2007
```

Ce mécanisme de sérialisation se révèle donc très puissant. La plupart des classes de la *Java Collection Framework* , pour ne citer qu'elle, implémente *Serializable*. L'utilisation combiné de ces outils permet de sérialiser des collections,voire des architectures complexes de données. Ces outils pourront être complétés également d'outils complémentaires permettant un certaine gestion des versions,voir à ce sujet l'outil du JDK *serialver*.

Flux de texte

Flux textuels

Ces deux classes, déjà présentées dans le paragraphe *Filtre de données* permettent la lecture et écritures des données natives dans un flux, notamment numériques.

Ces classes se prêtent également à la lecture de données textuelles et comportent des méthodes *pour DataInputStream* telles que *readByte*, *read (byte [] b)*, *readLine*, *readChar*, *readUTF*. Les méthodes équivalentes pour *DataOutputStream* existent également. Ces méthodes sont dédiées à la lecture et à l'écriture d'informations textuelles natives sous leur forme ASCII ou UTF par exemple.

Flux ASCII

Un flux *ASCII* est un flux constitué d'octets qui ont vocation à être interprétés en tant que caractères alphabétiques en suivant une correspondance qui est celle du codage ASCII. Celui-ci conçu au départ pour les caractères de l'alphabet latin non accentués, correspond à un jeu de valeur compris entre 0 et 255. L'unité de codage est donc l'octet et le pouvoir de représentation (256 valeurs) offert par cette implémentation réduit l'usage de ce codage à ce seul alphabet (quelques aménagements existent pour tenir compte des accents notamment et qui utilisent la plage de valeur plus ouvert comprise entre 128 et 255).

Le type *byte* sera en Java le candidat naturel à l'exploitation des codes ASCII. Néanmoins il s'agit d'un type numérique signé, équivalent du type *char* en *C*, et qui comme lui devra être manipulé convenablement pour le rendre capable de manipuler les codes ASCII.

Les méthodes *readByte* ou *writeByte* permettent la lecture-écriture des octets en général, et en particulier des codes ASCII.

Ci-dessous la lecture *byte* à *byte* du fichier source java courant avec *readByte.* Pour obtenir la représentation ASCII du code l'affichage formattée *printf* est utilisé. La fin de fichier est détectée par exception.

```
public class Launcher {
   public static void main(String[] args) {
      try {
         System.out.println(System.getProperty("user.dir"));
         File f=new File("src/Launcher.java");
         FileInputStream fis=new FileInputStream(f);
         DataInputStream dis=new DataInputStream(fis);
         while (true) {
            System.out.printf("%c",dis.readByte());
         }
      }
      catch (EOFException e) {
         System.out.print("\nFin");
      }
      catch (Exception e) {
         e.printStackTrace();
      }
   }
}
```

La lecture de bloc de *byte* est également possible au moyen de la méthode *read* (dans ce cas l'usage de *DataInputStream* est facultatif)

```
try {
         System.out.println(System.getProperty("user.dir"));
         File f=new File("src/Launcher.java");
         FileInputStream fis=new FileInputStream(f);
         DataInputStream dis=new DataInputStream(fis);
         byte [] buf=new byte[100];
         int n=0;
         while ((n=dis.read(buf))>0) {
            System.out.print(new String(buf,0,n));
         }
      }
      catch (FileNotFoundException e) {
         e.printStackTrace();
      } catch (IOException e) {
```

```
            e.printStackTrace();
        }
```

La méthode *readLine* proposée par *DataInputStream* est *deprecated*. L'aide en ligne indique qu'il faut employer celle proposée par la classe *BufferedIReader* dédiée à la lecture de texte, voir plus loin.

Codage *UTF*

UTF (*Unicode Transformation Format*) est un codage des caractères créé pour permettre la prise en compte d'autres alphabets (cyrillique, chinois, arabe etc...).

Le codage des caractères alphabétiques est un monde évolutif et complexe qui ne peut pas être traité correctement dans le cadre de ce document. Le type *char* du langage Java oblige néanmoins à une prise en compte minimale de l'existence du codage UTF parce qu'il s'agit du type de base qui permet la gestion du codage UTF-16 (représentation du 16 bits). La classe *Character* est une classe de *wraping* du type de base *char* et c'est elle qui intervient dans les situations d'*autoboxing* concernant les variables de type *char*. Ses méthodes, dont beaucoup sont statiques, donnent une indication sur la complexité du codage UTF.

Extrait :

```
public final class Character {
    public static final int SIZE = 16;
    public Character(char value)
    public static Character valueOf(char c)
    public char charValue()
    public int hashCode()
    public boolean equals(Object obj)
    public String toString()
    public static boolean isValidCodePoint(int codePoint)
    public static boolean isSupplementaryCodePoint(int codePoint)
    public static boolean isHighSurrogate(char ch)
    public static boolean isLowSurrogate(char ch)
    public static boolean isSurrogatePair(char high, char low)
    public static int charCount(int codePoint)
    public static int toCodePoint(char high, char low)
    public static int codePointAt(CharSequence seq, int index)
    public static int codePointBefore(CharSequence seq, int index)
    public static int toChars(int codePoint, char[] dst, int dstIndex)
    public static int codePointCount(CharSequence seq, int beginIndex, int endIndex)
    public static int offsetByCodePoints(CharSequence seq, int index,
    public static boolean isLowerCase(int codePoint)
    public static boolean isUpperCase(int codePoint)
    public static boolean isTitleCase(int codePoint)
    public static boolean isDigit(int codePoint)
    public static boolean isDefined(int codePoint)
    public static boolean isLetter(int codePoint)
    public static boolean isLetterOrDigit(int codePoint)
    public static boolean isUnicodeIdentifierStart(int codePoint)
    public static boolean isIdentifierIgnorable(int codePoint)
    public static int toLowerCase(int codePoint)
    public static int toUpperCase(int codePoint)
    public static int toTitleCase(int codePoint)
    public static int digit(int codePoint, int radix)
    public static int getNumericValue(int codePoint)
    public static boolean isSpaceChar(int codePoint)
    public static boolean isWhitespace(int codePoint)
    public static boolean isISOControl(int codePoint)
    public static int getType(int codePoint)
    public static char forDigit(int digit, int radix)
    public static byte getDirectionality(int codePoint)
    public static boolean isMirrored(int codePoint)
    public int compareTo(Character anotherCharacter)
    public static char reverseBytes(char ch)
    <etc...>
}
```

La prise en compte de ces codes s'effectue sur la station hôte en fonction de la table de codage active localement. Les normes locales les plus utilisées en France (sur les systèmes Unix, Windows, Mac) sont *ISO 8859-1*, *ISO 8859-15*, *Windows-1252* et *MacRoman*. *ISO 8859-15* est une évolution de *ISO 8859-1* caractérisée notamment par l'introduction du symbole €.

La portion de code suivante permet de mettre en correspondance les caractères employés dans une chaîne et le codage interne de ceux-ci utilisé par la classe *String* (le code ASCII octal de A est 101)

```
String s="AA";
System.out.print(s+" -> ");
for(byte b:s.getBytes()) System.out.printf(" %o ",b);
```

Sortie :

AA -> 101 101

Appliquée aux différentes chaînes suivantes cela donne :

```
A€A ->  101  342  202  254  101
AéA ->  101  303  251  101
AêA ->  101  303  252  101
AöA ->  101  303  266  101
AçA ->  101  303  247  101
```

Lecture écriture au format UTF

Les classes *DataInputStream* et *DataOutputStream* permettent la lecture et écriture au format natif UTF d'informations textuelles. Ce format présente des caractéristiques analogues aux formats natifs numériques : les fichiers ainsi constitués ne seront pas lisible pour un lecteur humain et ne seront pas considérés, à l'inverse des fichiers ASCII, comme des fichiers texte par le gestionnaire de fichier du système d'exploitation.

Les méthodes *readChar, writeChars* et *writeChar* permettent la lecture écriture de fichier de données au format UTF-16.

Exemple :

```
DataOutputStream dos=new DataOutputStream(new FileOutputStream(new File("exemple2.txt")));
dos.writeChars("Hello world");
```

Le fichier créé *exemple2.txt* comporte 24 octets soit davantage, en raison du codage UTF, que les 11 octets nécessaires pour l'encodage ASCII de *Hello world.* La lecture de ce fichier non ASCII ne pourra s'effectuer que par *readChar.*

```
DataInputStream dis=new DataInputStream(new FileInputStream(new File("exemple2.txt")));
try {   while (true) System.out.print(dis.readChar()); }
catch(EOFException e) {System.out.println(); }
dis.close();
```

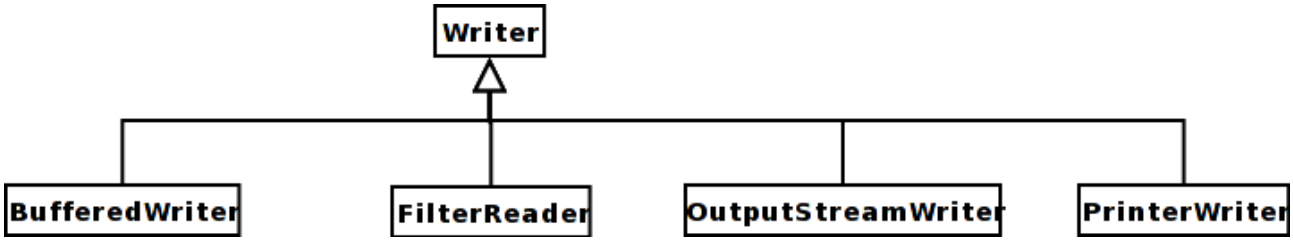
Sortie :

```
Hello world
```

Ces deux classes proposent également les méthodes *readUTF* et *writeUTF*. Ces méthodes ne sont présentes que pour des raisons de compatibilité avec l'existant et ne doivent normalement pas être utilisées pour l'usage mentionné.

Ecriture d'un flux de texte

Les classes issues de *Writer* prennent en charge les conversions nécessitées pour l'écriture d'informations textuelles ASCII.



La classe *PrintWriter* est un filtre permettant de fournir les fonctionnalités permettant la conversion des données natives Unicode en un flux textuel. Le résultat est la production d'un flux d'octets adapté au codage de la machine hôte (et notamment aux fins de lignes utilisées par celle-ci).

```
try {
     PrintWriter pw=new  PrintWriter(new FileWriter("exemple.txt"));
     pw.print("Prix HT = ");
     pw.print(5.20);
     pw.println(" €");
     pw.close();
  } catch (IOException e) {
     e.printStackTrace();
  }
```

Cette portion de code construit le fichier *exemple.txt* dans le répertoire courant. L'affichage de ce fichier texte avec l'outil par défaut proposée par la plate-forme fournit un affichage conforme à l'attente :

Prix HT = 5.2 €

A l'inverse l'utilisation de la méthode *writeChars* de la classe *DataOutputStream* produit un flux Unicode qui ne peut pas être directement utilisé pour garnir un fichier texte.

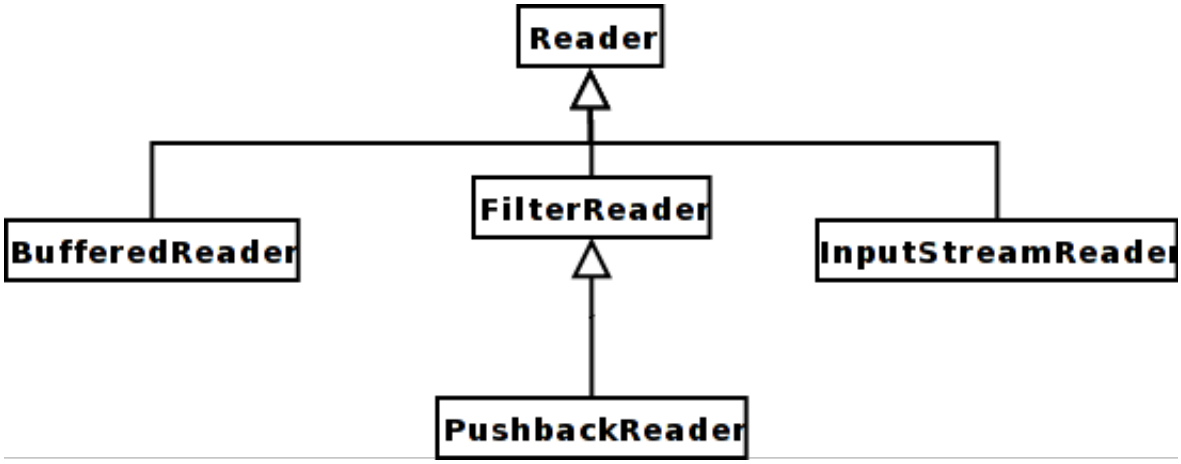
Le code suivant (nous avons ôté la production du flottant au format binaire pour la démonstration) :

```
DataOutputStream dos=new DataOutputStream(new FileOutputStream(new File("exemple1.txt")));
dos.writeChars("Prix HT = ");
// dos.writeDouble(5.20);
dos.writeChars(" €");
```

conduit à la création d'un fichier de codes UTF, plus gros car les caractères y sont développés sur 2 octets ou plus, et non reconnu en tant que fichier texte par le système.

Lecture d'un flux de texte

Les classes issues de *Reader* prennent en charge les conversions nécessitées pour la lecture d'informations textuelles ASCII.



La classe *BufferedReader* propose la très commode *méthode readLine*. Elle doit être adossée à une classe de lecture et permet de réaliser l'encodage des caractères provenant de la plateforme courante vers le format Unicode requis pour les chaînes de caractères manipulant les informations lues.

```
BufferedReader br=new BufferedReader(new FileReader("exemple.txt"));
String line;
while ((line=br.readLine())!=null) System.out.println(line);
```

La sortie restitue bien le résultat attendu à partir du fichier *exemple.txt* construit dans le paragraphe précédent :

```
Prix HT = 5.2 €
```

L'application de ce code au fichier *exemple1.txt* ne produirait pas un résultat convenable puisque ce dernier est implémenté nativement sous la forme des codes Unicode (sa lecture nécessiterait la méthode *readChar* d'un *DataInputStream*).

Lecture et parsing d'un texte

La classe *Scanner* (package *java.util)* permet également de lire un texte et de l'analyser.

```
Scanner scan=new Scanner(new FileReader("exemple.txt"));
String line;
while (scan.hasNext()) {
      System.out.print(scan.next());
}
```

Sortie :

```
PrixHT=5.2€
```

Clonage

Le clonage est l'opération par laquelle on tente de construire la copie d'une instance. Il s'agit d'une opération relativement peu fréquente en Java, à l'inverse du C++ pour lequel cette opération est très facilement implicite, et qui doit être définie soigneusement pour ne pas créer de problème.

Cette opération est délicate : recopier un objet ne signifie pas seulement recopier le bloc mémoire incarnant physiquement cet objet. Un objet, surtout lorsqu'il atteint un certain niveau de complexité, peut référencer, par ses variables membres, d'autres objets : il faut alors prendre la décision de savoir si cette recopie doit concerner ces sous-objets. Bien entendu ce problème va se poser de façon récurrente pour ces derniers. Une copie d'objet peut donc être plus ou moins profonde. La copie par défaut proposée dans *Object* est la plus légère (*shallow copy*) : une copie des données et des références de l'objet à recopier. Cela signifie qu'à travers les références recopiées l'original et la copie peuvent être en situation de partage d'objets.

Entre la copie légère et la copie intégrale Java ne tranche pas mais nous propose le schéma général suivant:

- Si une classe ne spécifie rien de particulier en ce sens, elle n'offre a priori pas le service de *clonage*.
- La capacité à se cloner est donc une décision qui revient au concepteur de la classe. C'est fort logiquement au niveau de cette classe, en conséquence, que se choisiront exactement les modalités de recopie d'instance.
- Dans le cas où le concepteur choisit de définir un comportement de recopie il procède par surcharge de la méthode *clone* , extension de sa visibilité, et par l'implémentation de l'interface *Clonable*.

La méthode *clone* de *Object*

La capacité d'un objet à se cloner est héritée de la surclasse absolue *Object* au moyen de la méthode *clone*.

```
package java.lang;
public class Object {
    public final native Class<? extends Object> getClass();
    public boolean equals(Object obj) { return (this == obj); }
    protected native Object clone() throws CloneNotSupportedException;
    public String toString() {
        return getClass().getName() + "@" + Integer.toHexString(hashCode());
    }
    etc...
}
```

L'utilisation visée par *clone* serait la suivante pour une classe *A* qui, comme toute classe, hérite de *Object :*

```
A a1=new A();
A a2=a1.clone(); // ERREUR : clone est protected
```

Néanmoins dans *Object c*ette méthode est *protected* : cela signifie que la modalité d'utilisation mentionnée ne peut fonctionner directement avec la méthode clone héritée *de Object*.

Cela signifie, comme on l'a dit, que le comportement par défaut d'une classe est de ne pas être *clonable*, c'est à dire de ne pas offrir de support pour la copie de ses instances.

La première condition pour que la classe *A* soit clonable est donc de surcharger *clone* en étendant la visibilité de cette méthode au niveau *public*.

La classe *A* ci-dessous surcharge *clone* et définit un comportement par super invocation, qui est celui d'*Object,* consistant à procéder à une recopie de mémoire a minima.

```
class A {
   private int n;
   private B b;
   public A() { b=new B(); }
   @Override
   public A clone() throws CloneNotSupportedException {return (A) super.clone(); }
}
public class Launcher {
   public static void main(String[] args) {
      A a1=new A(),a2=null;
      try {
         a2=a1.clone();     // ERREUR à l'exécution : java.lang.CloneNotSupportedException: A
      }
      catch (CloneNotSupportedException e) {
         e.printStackTrace();
      }
   }
}
```

Dans ce schéma de recopie, qui est celui d'une *shallow* copie, l'attribut *n* est bien recopié, mais seule la référence *b* est recopié ce qui fait que l'objet désigné par *b* sera de fait partagé entre l'instance et sa copie.

Bien qu'étant une surcharge de la méthode clone d'*Object* la méthode *clone* de la classe *A* ne possède pas tout à fait la même signature : son type de retour a été affiné de *Object* vers *A.* Cette évolution dite *covariante* de la signature est acceptée, compatible avec les règles de la surcharge, et cohérente avec celles du polymorphisme.

L'exécution du programme de test conduit néanmoins à une erreur, *CloneNotSupportedException,* dont la cause est exposée ci-après.

L'interface *Cloneable*

Il manque en effet un élément aux considérations qui précèdent pour que le schéma de clonage soit complet. Dans un langage fortement typé tel que Java le schéma de clonage présenté ne semble pas avoir de traduction sur le plan du typage. Il semble pourtant important de permettre à un programme de tester la capacité d'une instance à se cloner, avec une expression du genre : *if (obj instanceof Cloneable)* etc...

C'est la raison pour laquelle l'ultime opération à réaliser pour permettre à la classe *A* de se cloner est l'adjonction d'une clause d'implémentation vis à vis de l'interface *Cloneable*. Cette interface n'expose aucune méthode abstraite pas même *clone (*qui d'ailleurs n'est pas une méthode abstraite). Il s'agit au contraire d'une pure balise (qui peut se comparer à cet égard avec *Serializable*) et dont l'objectif est d'inscrire sémantiquement la recopie dans le typage Java.

Le programme ci-dessous complète le précédent et donne un exemple de recopie partielle.

```
class B  {
   public B(String info) { this.info=info; }
   public String info;
}
class A implements Cloneable {
   public int n=5;
   public B b;
   public A() { b=new B("Hello"); }
   @Override
   public A clone() throws CloneNotSupportedException {
         return (A) super.clone();
   }
   public String toString() { return "["+n+","+b.info+"]";    }
}

public class Launcher {
   public static void main(String[] args) {
     A a1=new A(),a2=null;
     try {   a2=a1.clone(); }
     catch (CloneNotSupportedException e) {
        e.printStackTrace();
     }
     System.out.println(a1);
     System.out.println(a2);
     a1.b.info="Salut";
     a1.n=17;
     System.out.println(a1);
     System.out.println(a2);
   }
}
```

Le programme de test semblable au précédent est le suivant, et montre que *a1 et* a2 sont deux instances partageant *b.*

```
[5,Hello]
[5,Hello]
[17,Salut]
[5,Salut]
```

L'exemple est repris ci-dessous mais avec cette fois une recopie profonde. *B* est *Cloneable* et la méthode *clone* de *A* procède également à la recopie de son membre *b*.

```
class B  implements Cloneable {
   public B(String info) { this.info=info; }
   public String info;
   @Override
   public B clone() throws CloneNotSupportedException { return (B) super.clone(); }
```

```
}
class A implements Cloneable {
   public int n=5;
   public B b;
   public A() { b=new B("Hello"); }
   @Override
   public A clone() throws CloneNotSupportedException {
      A a=(A) super.clone();
      a.b=b.clone();
      return a;
   }
   public String toString() { return "["+n+","+b.info+"]"; }
}
```

Le programme de test semblable au précédent est le suivant, et montre que *a1 et* a2 sont deux instances disjointes.

```
[5,Hello]
[5,Hello]
[17,Salut]
[5,Hello]
```

Mort d'un objet - Garbage collector

Le cycle de vie d'un objet doit aboutir naturellement à la disparition de celui-ci lorsqu'il n'est plus utile dans l'application. Cette disparition doit permettre de restituer au système les ressources mobilisées, en particulier la mémoire allouée pour cet objet.

Lors de l'exécution au sein de la machine virtuelle, Java utilise un modèle d'allocation mémoire assez différent de celui utilisé par les langages C ou C++.

En C++ les objets peuvent être alloués dans la zone de données globale, dans la pile ou dans le tas.

C++	Zone d'allocation
`A a1;` `int main() {` `  A a2;` `  A *pa=new A;` `  delete pa;` `}`	`// objet a1 dans la zone de données globale` `// objet a2 dans la pile` `// objet *pa dans le tas.(pointeur pa dans la pile)` `// objet *pa détruit` `// objet a2 détruit (sortie de bloc)` `// objet a1 détruit (sortie de programme)`

En Java, en dehors des types de bases et des adresses, les objets sont alloués uniquement dans le tas.

Java	Zone d'allocation
`A a1;` `int main() {` `  A a2;` `  a2=new A();` `}`	`// erreur  : pas de variable globale en Java` `// a2 est une référence (donc une adresse)` `// stockée dans la pile. Aucun objet créé.` `// l'objet a2 est dans le tas`

C'est ce choix d'une allocation systématique des objets dans le tas qui confère au langage une certaine simplicité. Néanmoins l'allocation dans le tas pose le problème de la libération du bloc mémoire utilisé lorsque l'objet ne doit (ou ne peut) plus être utilisé. En C++ un objet alloué dans le tas doit être explicitement libéré par le programme: un opérateur nommé *delete* est dédié à cette opération. Il est donc à la charge du programmeur d'application de repérer la situation qui conduit à cette libération de mémoire. Cela est souvent complexe, il s'agit de la principale source d'erreur dans les applications écrites en C++. Ces fuites de mémoire (*memory leaks*) peuvent conduire des applications à longue durée d'exécution (comme une application serveur) vers l'engorgement progressif. En outre il est souvent difficile de maintenir la cohérence des différents pointeurs utilisés par un programme C++ au travers des éventuelles multiples et complexes allocations et libérations d'objets.

Le modèle unique d'allocation mémoire des objets proposé par Java ne peut tenir que parce que, par ailleurs, ce langage nous propose un mécanisme, appelé *garbage collector*, qui automatise complètement ce travail de désallocation des objets inutilisés. Grâce à lui, les multiples scories mémoire générées même par un programme très simple (voir à ce sujet l'exemple des chaînes de caractères) sont libérées sans intervention explicite du programmeur.

Le *garbage collector*

En C++ lorsqu'un *new* (ou un *malloc* en *C*) *est* généré pour demander l'allocation dynamique d'un objet il y a mise en oeuvre d'une requête adressée au gestionnaire de tas. Celui-ci parcourt la liste des blocs chaînés de mémoire libre jusqu'à trouver un bloc de taille suffisante pour y loger le nouveau venu.

En Java les demandes d'allocation sont en réalité adressées à la machine virtuelle. Cette dernière, parce qu'elle s'interpose entre la demande d'exécution du programme Java et l'exécution proprement dite, peut mettre en oeuvre des algorithmes sophistiqués d'allocation et de libération de mémoire. C'est un point sur lequel les JVM ont beaucoup évolué. Différents mécanismes peuvent être mis en oeuvre, voire optionnellement, et différents éditeurs de JVM peuvent proposer des solutions différentes.

D'une façon générale la technique mise en oeuvre par le gestionnaire d'allocation mémoire de la JVM vise à rendre l'allocation aussi efficace que possible. A la coûteuse recherche d'un bloc de mémoire libre de taille suffisante mise en oeuvre classiquement par le gestionnaire de tas, la JVM substitue un modèle d'organisation contiguë des blocs mémoire représentant les objets. Ainsi l'allocation d'un nouveau bloc

relève de la simple mise à jour du pointeur interne représentant le sommet de la zone concernée. Il s'agit d'un modèle d'allocation analogue à celui utilisé par la pile.

Ce modèle a un coût : c'est celui engendré par les destructions d'objets et les libérations de blocs mémoire qu'elles impliquent.

Lorsque la JVM détecte le besoin de procéder à un nettoyage de la mémoire elle procède par vol de cycles : elle stoppe le programme applicatif et lance le nettoyage. Cette phase appelée *garbage collection* est coûteuse et complexe.

Ce nettoyage consiste à repérer les objets non utilisés, à libérer l'espace mémoire correspondant et à garder une organisation contiguë des blocs.

Il est très simple de produire un objet non utilisé :

```
{
A a1=new A();
a1=null;
..... // à partir de ce point l'objet qui était repéré par a1 a perdu cette référence.
      // il n'est plus utilisable par le programme, donc définitivement inutile
      // il devra être détruit le moment venu par le garbage collector
}
```

Dans cet exemple la référence (ou pointeur) *a1* est elle-même allouée dans la pile. L'objet référencé par *a1* est alloué dans le tas comme c'est la règle en Java. La référence aurait pu également être créée dans la zone globale de données s'il s'était agit d'une variable statique.

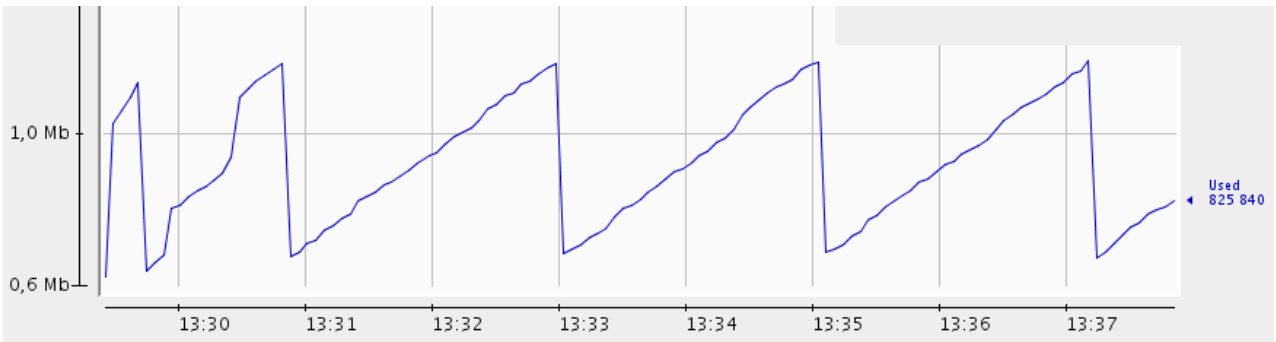
C'est la raison pour laquelle le principe utilisé par le garbage collector pour repérer les objets sans référence consiste à parcourir tous les objets accessibles en prenant comme points de départ toutes les adresses glanées dans la zone de données globales ou dans la pile (en réalité dans les piles car nous sommes dans un contexte multithreadé et chaque thread utilise sa propre pile). De la même façon ces blocs, représentant autant d'objets, auxquels mènent ces adresses font l'objet d'une recherche d'adresses menant elles-mêmes à d'autres objets. Ce mécanisme récursif conduira à une arborescence dont les noeuds seront les adresses des objets potentiellement accessibles de l'application. Le garbage collector construit alors un nouveau bloc d'objets contigus par recopie des seuls objets référencés: ce nouveau bloc deviendra la nouvelle zone d'allocation de la JVM.

Comme on le voit ce mécanisme ignore les objets non référencés ainsi que les éventuelles références circulaires. D'autre part des optimisations existent de façon à rendre cette recopie en blocs contigus moins systématique.

Le déclenchement du *garbage collector* peut être forcé par une méthode de la classe *System* ce qui peut être utile en phase de mise au point, par exemple.

```
System.gc();     // force le garbage collector
```

En régime de croisière une application Java sera l'objet de déclenchements réguliers du *garbage collector.* Le graphe ci-dessous résulte du monitoring d'une application serveur Java. La courbe représente l'évolution au cours du temps de l'espace mémoire occupé au sein du tas. On peut constater que la minute est un ordre de grandeur caractéristique du déclenchement périodique du *garbage collector*.



La bibliothèque *JMX* permet de mettre en place un appareillage du code java permettant une surveillance en temps réel de l'état de la mémoire, des threads, des objets des applications.

La méthode *finalize*

En C++ le cycle de vie d'un objet, naissance vie et mort, s'accompagne respectivement des notions de constructeurs, méthodes et destructeurs. Le destructeur, en particulier, est invoqué par le *runtime* préalablement à la destruction de l'objet. La définition d'un destructeur est fréquente dans ce langage en raison du fait que la destruction d'un objet y est un événement déterministe: la destruction d'un objet survient systématiquement en sortie de bloc pour un objet local, en fin de programme pour un objet global ou lors de l'invocation de *delete* pour un objet dynamique.

En Java il en est tout autrement: la destruction d'un objet est un événement qui est, du point de vue de l'application, indéterministe. C'est l'état de la machine virtuelle et lui seul qui provoquera la mise en oeuvre du *garbage collector*. En conséquence une méthode précédant immédiatement la destruction de l'objet (analogue pour cela au destructeur du C++) existe en Java mais son usage est beaucoup plus rare que le destructeur du C++.

Cette méthode s'appelle *finalize,* et une classe qui souhaiterait faire précéder la destruction de ses instances d'une action spécifique procéderait par surcharge de cette méthode.

L'usage de *finalize* est souvent réservé à la mise au point de logiciel. Pour se convaincre de la destruction effective d'un objet la surcharge de cette méthode peut être utile. D'une façon générale *finalize* pourra être utilisée pour mettre en oeuvre une libération de ressource associéesà l'objet mais extérieure à la logique de la JVM : un exemple typique est l'invocation d'une méthode *native* C++ procédant elle-même à une allocation de mémoire devant être libérée.

L'exemple suivant créé un tableau de 5 instances de la classe *A*, dont on a surchargé la méthode *finalize* et pour laquelle on procède à une invocation explicite du *garbage collector* avec *System.gc()*.

```
class A {
   private String name;
   static private int num=0;
   public A() {
      name="A"+num++;
      System.out.println("Creation de "+this);
   }
   @Override
   public String toString() { return name; }
   @Override
   public void finalize() {
      System.out.println("Destruction de "+this);
   }
}

public class Launcher {
    public static void main(String[] args) {
      A [] ta=new A[5];
      for(A a:ta) a=new A();
      for(A a:ta) a=null;
      System.gc();
  }
}
```

La sortie est :

```
Creation de A0
Creation de A1
Creation de A2
Creation de A3
Creation de A4
Destruction de A4
Destruction de A3
Destruction de A2
Destruction de A1
Destruction de A0
```

Les Threads et la synchronisation

Introduction

Apparue à la suite des travaux de l'université de Berkeley à la fin des années 80, la notion de threads est devenue aujourd'hui un élément incontournable des systèmes d'exploitation et des applications.

Les threads permettent de définir plusieurs flux d'exécution à l'intérieur d'une même application. Chaque flux d'exécution est alors appelé un *thread* . La plupart des applications sont aujourd'hui multithreadées: il est banal aujourd'hui d'attendre d'une application qu'elle fasse plusieurs choses à la fois.

Lorsque Java est apparu les threads étaient déjà généralisés au niveau des systèmes d'exploitation. Aussi, dès son origine, Java a incorporé cette notion. Les threads y sont donc intégrés de façon élégante et sécurisée. Un programme Java, même très simple, peut être rapidement confronté à la nécessité d'incorporer un nouveau thread. Il faut noter que si aucune création de thread n'intervient dans un programme il en existe au moins un, celui qui assure l'exécution de ce programme.

Un thread apparaît d'abord en Java comme une instance de la classe *Thread*. Celle-ci possède l'interface publique très incomplète suivante :

```
public class Thread {
   public static final int MIN_PRIORITY, NORM_PRIORITY,MAX_PRIORITY
   public Thread(Runnable target)
   public Thread(Runnable target,String name)
   public static Thread currentThread()
   public int getPriority()
   public Thread.State getState()
   public void interrupt()
   public static boolean interrupted()
   public boolean isAlive()
   public void run()
   public void setPriority(int newPriority)
   public static void sleep(long millis)
   public void  start()
   public static void yield()
   public static void sleep(long millis) throws InterruptedException;
   etc...
}
```

Un thread se crée donc par une instanciation de la classe *Thread* : le constructeur attend un objet de type *Runnable* qui expose, comme son nom l'indique, la méthode *run*.

```
public interface Runnable {
   void run();
}
```

Un objet qui hérite de l'interface *Runnable* définit donc une méthode *run.* C'est cette méthode qui sera invoquée par le thread pour commencer son exécution.

Classiquement, en *C* par exemple, la création d'un thread nécessite de lui passer un pointeur de la fonction qui sera exécutée par le thread. La mécanique Java, basée sur l'interface *Runnable*, vient se substituer avantageusement à cette manipulation de pointeur.

Création d'un Thread

Pour créer un *thread* il faut donc disposer d'un objet *Thread* et d'un objet *Runnable*. L'objet *Runnable* possède donc une méthode *run* qui définira l'exécution du thread. L'objet *Runnable* est typiquement passé au thread à sa création. puis le thread est effectivement lancé par *start*. Les exemples ci-dessous présentent trois modalités très fréquentes de ce schéma de création de thread.

Exemple 1

La classe Launcher lance le thread en lui passant une instance de la classe *A,* qui est *Runnable*

```
class A implements Runnable {
   public void run() {
     while (true) {        System.err.println("Hello World");}
   }
}
```

```
public class Launcher {
   public static void main(String[] args) {
      Thread thread=new Thread(new A());
      thread.start();
   }
}
```

Sortie (boucle) :

```
Hello World
Hello World
etc...
```

Exemple 2

La classe *A* lance le thread et est *Runnable* : elle lance le thread sur elle-même par *this*.

```
class A implements Runnable {
   public A() {
      Thread thread=new Thread(this);
      thread.start();
   }
   public void run() {
      while (true) { System.err.println("Hello World");}
   }
}

public class Launcher {
   public static void main(String[] args) {
      new A();
   }
}
```

Sortie (boucle):

```
Hello World
Hello World
etc...
```

Exemple 3

La classe *Launcher* lance le thread sur un objet *Runnable* créé à la volée à l'aide d'une classe locale anonyme.

```
public class Launcher {
   public static void main(String[] args) {
      Thread thread=new Thread(new Runnable(){
         public void run() {
            while (true) { System.err.println("Hello World"); }
         }
      });
      thread.start();
   }
}
```

Sortie (boucle):

```
Hello World
Hello World
etc...
```

Exemple 4

Comme précédemment mais le thread reçoit un nom ("*Premier*") à sa création. Cette possibilité est utile dans la mesure où cela facilite l'identification de ce thread en situation de débogage.

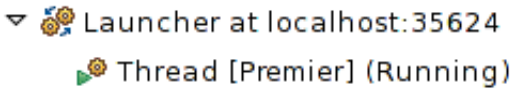
```
public class Launcher {
   public static void main(String[] args) {
      Thread thread=new Thread(new Runnable(){
         public void run() {
            while (true) { System.err.println("Hello World"); }
         }
      },"Premier");
      thread.start();
```

```
   }
}
```

Sortie (boucle):

```
Hello World
Hello World
etc...
```

Vue du débogueur (Eclipse) :



Thread courant

Un thread peut s'auto-désigner en tant qu'instance de *Thread* par la méthode statique *currentThread.*

Dans l'exemple ci-dessous le thread modifie son propre niveau de priorité.

```
public class Launcher {
   public static void main(String[] args) {
      Thread thread=new Thread(new Runnable(){
         public void run() {
            while (true) {
               Thread.currentThread().setPriority(Thread.MIN_PRIORITY);
               System.err.println("Hello World");
            }
         }
      });
      thread.start();
   }
}
```

La méthode *sleep* permet de mettre en sommeil le thread courant pendant une durée définie en millisecondes.

```
public class Launcher {
   public static void main(String[] args) {
      Thread thread=new Thread(new Runnable(){
         public void run() {
            while (true) {
               try {
                  Thread.sleep(1000);
               } catch (InterruptedException e) {
                  e.printStackTrace();
               }
               System.err.println("Hello World");
            }
         }
      });
      thread.start();
   }
}
```

En sortie le mot *Hello World* s'affiche maintenant toutes les secondes.

Sur le même mode que *sleep* la méthode statique *yield* permet de signaler à l'ordonnanceur que le thread courant (celui qui invoque *Thread.yield()*) souhaite passer la main à un autre thread. Cette méthode est parfois utile pour donner une meilleure illusion de simultanéité de l'exécution de plusieurs threads concurrents.

Dans l'exemple qui suit 10 threads sont lancés concurremment. Un objet *Runnable* distinct est passé à chacun pour permettre de les distinguer.

```
class A implements Runnable {
   private String info;
   public A(String info) {
      this.info=info;
   }
   public void run() {
      while (true) {
```

```
            Thread.yield(); // ligne facultative
            System.err.println(info);
      }
   }
}

public class Launcher {
   public static void main(String[] args) {
      Thread [] tabThread=new Thread[10];
      for(int i=0;i<10;i++) tabThread[i]=new Thread(new A("Hello "+i));
      for(Thread thread:tabThread) thread.start();
   }
}
```

Sans la ligne *yield* la sortie est la suivante :

```
hello 7
hello 7
hello 7 répété 255 fois... suivi de  hello 6 répété 252 fois etc...
```

Avec la ligne *yield* la sortie est la suivante :

```
Hello 9
Hello 7
Hello 4
Hello 8
Hello 1
Hello 0
Hello 3
Hello 5
Hello 6
```

Terminaison d'un thread

Un thread peut se terminer naturellement lorsqu'il a terminé l'exécution de la méthode *run* sur laquelle il était installé. Il peut se terminer également anormalement parce qu'une exception non traitée a été levée lors de son exécution.

Il existe dans la classe *Thread* une méthode *stop* qui a pour effet d'arrêter définitivement (à l'inverse du couple *suspend/resume*) l'exécution d'un thread. Cette méthode est aujourd'hui classée *deprecated*, ainsi que les méthodes *suspend* et *resume*. Leur usage est donc fortement déconseillé.

La méthode *isAlive* permet comme son nom l'indique de tester si un thread est actif, c'est à dire démarré et non stoppé.

Il faut distinguer à cet égard l'objet *Thread* lui même de l'entité physique qu'il représente (c'est à dire un concept du système d'exploitation qui incarne un flux d'exécution). Lorsque l'instance de *Thread* est créé, le thread en tant qu'entité du système d'exploitation n'existe pas encore. Ce dernier n'acquiert une existence qu'après l'invocation de la méthode *start* sur l'objet *Thread*. Un objet *Thread* est donc davantage une entité orienté objet par la médiation de laquelle on accède à un certain contrôle sur le thread réel.

Les services d'exécution

La version 5 de Java a été enrichie d'outils permettant d'affiner l'exécution et le contrôle des threads d'une application. Le lancement d'un ensemble de threads peut être mieux contrôlé, les ressources internes allouées à chaque thread peuvent être optimisées, un support pour les appels asynchrones et pour les variables *future* est fourni, etc...

L' interface *Executor*

L'interface *Executor* expose la méthode *void execute(Runnable o).* Il s'agit donc d'une interface qui permet de définir une étape intermédiaire, consistant en la fourniture d'un niveau de service, entre l'appel de l'exécution d'une tâche dans un thread et le lancement effectif de ce dernier.

L'exemple ci-dessous montre 2 utilisations immédiates de cette interface tirées des exemples fournis par le JDK. Il s'agit de deux implémentations minimales d'*Executor* données à titre d'exemples.

La classe *DirectExecutor* se contente d'invoquer directement la méthode *run* de l'objet *Runnable* passé : celui-ci est donc exécuté dans le thread courant, et donc sans création de nouveau thread.

La classe *ThreadPerTaskExecutor* lance un thread par objet *Runnable* passé.

Cet exemple montre donc 3 versions du lancement d'un ensemble de 5 threads :

- Version 1 : classique (avec création explicite des threads et leur lancement par *start*),
- Version 2 : par la médiation de *DirectExecutor* (c'est à dire sans création de thread)
- Version 3 : par la médiation de *ThreadPerTaskExecutor* (qui ne fait rien de plus que la méthode classique de la version 1)

Pour éviter que les sorties des différentes versions ne se mélangent un *sleep* a été posé lorsque c'est utile.

```
import java.util.concurrent.Executor;
import java.util.concurrent.ExecutorService;

class Task implements Runnable {
   private String info="t";
   private int num;
   public Task(int num) { this.num=num; }
   public void run() {
      for(int i=0;i<5;i++) {
         System.err.print(info+num+" ");
         Thread.yield();
      }
      System.err.println(info+num+" a fini ");
   }
}

class DirectExecutor implements Executor {
    public void execute(Runnable r) {
        r.run();
    }
}

class ThreadPerTaskExecutor implements Executor {
    public void execute(Runnable r) {
        new Thread(r).start();
    }
}

public class Launcher {
   public static void main(String[] args) {
      final int MAX=5;
      // Lancement de threads version 1
      System.err.println("\nVersion 1 : lancement classique sans executor");
      Thread []   tabThread=new Thread[MAX];
      for(int i=0;i<MAX;i++) tabThread[i]=new Thread(new Task(i));
      for(Thread thread:tabThread) thread.start();
      sleep();
      // Version 2
      System.err.println("\nVersion 2 : lancement avec un executor travaillant dans le thread courant");
      DirectExecutor exec2=new DirectExecutor();
      for(int i=0;i<MAX;i++) exec2.execute(new Task(i));
      // Version 3
      System.err.println("\nVersion 3 : lancement avec un executor allouant un thread par Task ");
      ThreadPerTaskExecutor exec3=new ThreadPerTaskExecutor();
      for(int i=0;i<MAX;i++) exec3.execute(new Task(i));
   }

   private static void sleep() {
      try { Thread.sleep(1000); } catch (InterruptedException e) {}
   }
}
```

Exemple de sortie :

```
Version 1 : lancement classique sans executor
t0 t2 t1 t3 t0 t1 t0 t0 t1 t1 t0 t1 t0 a fini
t1 a fini
t3 t2 t4 t4 t4 t4 t4 t4 a fini
```

```
t2 t3 t2 t3 t2 t3 t2 a fini
t3 a fini

Version 2 : lancement avec un executor travaillant dans le thread courant
t0 t0 t0 t0 t0 t0 a fini
t1 t1 t1 t1 t1 t1 a fini
t2 t2 t2 t2 t2 t2 a fini
t3 t3 t3 t3 t3 t3 a fini
t4 t4 t4 t4 t4 t4 a fini

Version 3 : lancement avec un executor allouant un thread par Task
t0 t0 t0 t0 t0 t0 a fini
t2 t1 t2 t1 t2 t1 t3 t2 t1 t3 t2 t4 t3 t4 t3 t2 a fini
t4 t1 t3 t1 a fini
t4 t3 a fini
t4 t4 a fini
```

Executors et *ExecutorService*

Executors est une classe qui permet de fabriquer des objets offrant un niveau de service plus intéressant que les exemples très simples qui précèdent (*DirectExecutor* ou *ThreadPerTaskExecutor*).

Executors offre un certain nombre de méthodes statiques permettant de construire, sur le modèle des *factory*, des objets de contrôle de l'exécution d'un ou plusieurs threads :

```
static ExecutorService newCachedThreadPool()
static ExecutorService newFixedThreadPool(int nThreads)
static ScheduledExecutorService newScheduledThreadPool(int corePoolSize)
static ExecutorService newSingleThreadExecutor()
static ScheduledExecutorService newSingleThreadScheduledExecutor()
```

Les types de retour, *ExecutorService* ou *ScheduledExecutorService*, sont des interfaces héritées de *Executor*, et qui en conséquence présentent la méthode *execute*. Mais ces interfaces présentent également un certain nombre de services supplémentaires relatifs au contrôle des threads dont leurs instances ont la charge. Ainsi l'interface E*xecutorService (*et *ScheduledExecutorService* qui en hérite*)* propose par exemple les services suivants :

```
boolean awaitTermination(long timeout, TimeUnit unit)
<T> List<Future<T>> invokeAll(Collection<? extends Callable<T>> tasks)
<T> List<Future<T>> invokeAll(Collection<? extends Callable<T>> tasks, long timeout, TimeUnit
unit)
<T> T invokeAny(Collection<? extends Callable<T>> tasks)
<T> T invokeAny(Collection<? extends Callable<T>> tasks, long timeout, TimeUnit unit)
boolean isShutdown()
boolean isTerminated()
void shutdown()
List<Runnable>  shutdownNow()
<T> Future<T> submit(Callable<T> task)
Future<?> submit(Runnable task)
<T> Future<T> submit(Runnable task, T result)
```

L'utilisation de ces méthodes sera illustrée dans quelques exemples.

Mises en oeuvre d'*ExecutorService*

Dans cet exemple nous poursuivons les déclinaisons de lancement de threads avec la classe *Task* déjà utilisée ci-dessus (ainsi que la méthode *sleep*). Les deux *ExecutorService* créés ici au moyen de *newCachedThreadPool* et *newFixedThreadPool* sont deux des instances de la classe *ThreadPoolExecutor*. Cette classe, qui implémente bien *sûr ExecutorService* mais aussi la classe abstraite plus *précise AbstractExecutorService,* propose de nombreux paramètres qui vont déterminer son mode de fonctionnement. A cet égard les méthodes statiques *newCachedThreadPool* et *newFixedThreadPool,* ainsi que *newScheduledThreadPool* et *newSingleThreadExecutor* proposées également dans *Executors,* apparaissent comme des commodités mettant en place un paramétrage qui correspond aux utilisations communes de ces pools de threads.

```
public class Launcher {
   public static void main(String[] args) {
      final int MAX=5;
      // suite..
      // Version 4 :newCachedThreadExecutor
```

```
        System.err.println("\nVersion 4 : lancement avec un cached ThreaPoolExecutor ");
        ExecutorService exec4=Executors.newCachedThreadPool();
        for(int i=0;i<MAX;i++) exec4.execute(new Task(i));
        exec4.shutdown();
        sleep();
        // Version 5 :newFixedThreadExecutor
        System.err.println("\nVersion 5 : lancement avec un fixed ThreaPoolExecutor ");
        ExecutorService exec5=Executors.newFixedThreadPool(2);
        for(int i=0;i<MAX;i++) exec5.execute(new Task(i));
        exec5.shutdown();
    }
```

La sortie est par exemple :

```
Version 4 : lancement avec un cached ThreaPoolExecutor
t0 t0 t0 t1 t0 t0 t0 a fini
t2 t1 t2 t1 t2 t3 t1 t2 t3 t1 t2 t3 t1 a fini
t2 a fini
t3 t3 t3 a fini
t4 t4 t4 t4 t4 t4 a fini

Version 5 : lancement avec un fixed ThreaPoolExecutor
t0 t0 t0 t0 t0 t0 a fini
t1 t2 t1 t2 t1 t2 t1 t2 t1 t2 t1 a fini
t2 a fini
t3 t4 t3 t4 t3 t4 t3 t4 t3 t4 t3 a fini
t4 a fini
```

Voir les commentaire ci-après.

L'appel newCachedThreadPool

Cet appel crée une instance de *ThreadPoolExecutor* préparée pour l'exécution d'un pool de threads sur des modalités assez courantes. Il est fréquent par exemple que cet appel constitue l'unique invocation d'un *ExecutorService* dans une application. Pour comprendre son rôle il faut dans la suite distinguer le thread en tant que flux d'exécution et l'objet, instance de *Thread*, qui en permet le contrôle en Java.

Ce gestionnaire de threads active un thread par objet *Runnable* passé. En réalité le mot *cached* signifie que le *threadPoolExecutor* gère en interne une ensemble d'instances de *Thread* et permet d'optimiser la consommation de ressource nécessitée par la mise oeuvre d'une nouveau thread qu'on lui soumet par *execute*; si aucun objet *Thread* n'est disponible dans le cache un nouvel objet *Thread* est créé, sinon le système alloue au nouveau thread l'instance de *Thread* du cache existante. Lorsqu'un thread meurt son instance reste disponible pendant un certain temps (typiquement une minute) pour un éventuel autre thread.

Lorsque la méthode *shutdown* est invoquée aucun objet *Runnable* ne peut être soumis au pool pour exécution (déclenchement de *java.util.concurrent.RejectedExecutionException*)

L'usage de ce pool de threads est typiquement caractéristique des situations où une application doit lancer un grand nombre de petit threads, à courte durée de vie par exemple. Il peut s'agir d'appels asynchrones par exemple. Ce mécanisme permet d'optimiser la consommation des ressources impliquées.

L'appel newFixedThreadPool

Ce pool de threads permet de mettre en oeuvre une certaine politique d'ordonnancement des threads : à sa création on choisit le nombre de threads maximum pris en charge par le pool (2 dans l'exemple ci-dessus). Les demandes surnuméraires sont mises en file d'attente, et servies au fur et à mesure de la libération des instances de threads internes du pool.

La sortie précédente permet d'illustrer le mécanisme :

```
Version 5 : lancement avec un fixed ThreaPoolExecutor (2 instances réservées)
// 2 instances disponibles : t0 et t1 les prennent
t0 t1 t0 t1 t0 t1 t0 t1 t0 t1 t0 a fini
// 1 instance disponible (t1 en cours) : t2 la prend
t2 t1 a fini
// 1 instance disponible (t2 en cours) : t3 la prend
t3 t2 t3 t2 t3 t2 t3 t2 t3 t2 a fini
// 1 instance disponible (t3 en cours) : t4 la prend
t4 t3 a fini
// 1 instance disponible (t4 en cours)
t4 t4 t4 t4 t4 a fini // 2 instances disponibles
```

Lorsqu'il est invoqué avec le paramètre 1 l'appel *newFixedThreadPool* est équivalent à *newSingleThreadExecutor :* il permet de sérialiser une ensemble de tache dans un thread distinct.

```
// Version 6 :newSingleThreadExecutor
System.err.println("\nVersion 6 : lancement avec un single ThreaPoolExecutor ");
ExecutorService exec6=Executors.newSingleThreadExecutor();
for(int i=0;i<MAX;i++) exec6.execute(new Task(i));
```

La sortie correspond à une sérialisation de ces exécutions mais cette fois (à l'inverse de la version 2 précédente) dans un thread distinct:

```
Version 6 : lancement avec un fixed ThreaPoolExecutor
t0 t0 t0 t0 t0 t0 a fini
t1 t1 t1 t1 t1 t1 a fini
t2 t2 t2 t2 t2 t2 a fini
t3 t3 t3 t3 t3 t3 a fini
t4 t4 t4 t4 t4 t4 a fini
```

Appel asynchrone- variable *Future*

Un appel asynchrone correspond à la mise en oeuvre d'un appel de fonction (ou de méthode dans un contexte objet) pour lequel l'appelant n'est pas bloqué en attente de la valeur retournée par l'appel.

L'appel fonctionnel classique est donc implicitement synchrone. Pour ne pas bloquer l'appelant la mise en oeuvre d'un appel asynchrone nécessite l'exécution de la partie appelée dans un thread distinct de celui de l'appelant.

L'appelant, même s'il continue son exécution après l'appel, devra toutefois attendre la disponibilité de la valeur retournée pour l'exploiter : mais ce n'est qu'au moment de l'exploitation de cette dernière que le blocage est nécessaire. Il se peut même qu'entre le moment de l'appel et le moment de l'exploitation de la valeur retournée celle-ci soit déjà disponible : aucun blocage n'aura alors été généré pour l'exécution du service asynchrone.

Ce mécanisme relève d'une problématique de synchronisation entre l'appelant et l'appelé. Avec le type *Future et* l'interface *Callable,* Java fournit des éléments permettant une expression élégante de ces contraintes de synchronisation.

L'interface générique *Callable* est définie ainsi :

```
public interface Callable<V> {
     V call() throws Exception;
}
```

Elle permet de spécifier la notion de thread retournant une *valeur* de type *V*. Un appel asynchrone consistera en un lancement de thread sur un objet *Callable*.

Une variable de type *Future*<*V*> servira de réceptacle pour le résultat retourné. Cette entité sera en plus pourvu des méthodes permettant une synchronisation entre l'entité exploitant le résultat et sa disponibilité effective.

L'interface *Future* se présente ainsi:

```
public interface Future<V> {
    boolean cancel(boolean mayInterruptIfRunning);
    boolean isCancelled();
    boolean isDone();
    V get() throws InterruptedException, ExecutionException;
    V get(long timeout, TimeUnit unit)
        throws InterruptedException, ExecutionException, TimeoutException;
}
```

La méthode *get* permet d'accéder au résultat du calcul, mais cet accès est bloquant sur la disponibilité effective de la valeur. Une version avec temps de garde est également fournie. D'autre part les méthodes *isDone* ou *isCancelled* permettent de s'informer sans blocage de la disponibilité du résultat.

Exemples utilisant Callable et *Future*

L'exemple qui suit lance des tâches qui consistent chacune à décomposer en facteurs premiers la valeur passée en paramètre. Ce service de décomposition est fourni sous la forme d'un service asynchrone avec synchronisation déportée sur une variable *Future*.

La variable *Future* contient la liste chaînée des facteurs premiers correspondant à l'entier *num* passé *à ComputeTask* lors d'une instanciation : cette liste est implémentée sous la forme d'une *LinkedList*<*Long*>.

```
class ComputeTask implements Callable<LinkedList<Long>> {
   private long num;
   public ComputeTask(long num) { this.num=num; }
   public LinkedList<Long> call() throws Exception {
      LinkedList<Long> list=new LinkedList<Long>();
      long n=num,d=2L;
      while (d*d<=num) {
         if (n % d ==0) {
            n/=d;list.add(d);// ajout du facteur premier à la liste
         }
         else d++;
      }
      if (n!=1) list.add(n);
      return list;
   }
}
```

Le programme d'exploitation ci-dessous saisit (au clavier) le nombre *num* à décomposer, lance un thread de calcul et récupère tout de suite le résultat (une liste chaînée de facteurs premiers) dans la variable *Future* listPrimes. Sans précaution le programme tente tout de suite d'afficher cette liste par *println(listPrimes.get())*. L'appel *get* bloque jusqu'à la disponibilité de la liste.

```
long num=new Scanner(System.in).nextLong();
ExecutorService exec8=Executors.newSingleThreadExecutor();
Future<LinkedList<Long>> listPrimes=exec8.submit(new ComputeTask(num));
try { System.out.println(listPrimes.get()); }
catch (InterruptedException e) {}
catch (ExecutionException e) {}
finally {   exec8.shutdown(); }
```

Exemple de sortie :

```
123456789123456789 (valeur saisie; attente de 30 s environ)
[3, 3, 7, 11, 13, 19, 3607, 3803, 52579]
```

Dans l'exemple ci-dessous on tente la même décomposition et on abandonne le calcul s'il est supérieur à 20 secondes:

```
long num=new Scanner(System.in).nextLong();
      ExecutorService exec8=Executors.newSingleThreadExecutor();
      Future<LinkedList<Long>> listPrimes=exec8.submit(new ComputeTask(num));
      try {
         System.out.println(listPrimes.get(20,TimeUnit.SECONDS));
      }
      catch (InterruptedException e) {}
      catch (ExecutionException e) {}
      catch (TimeoutException e) { System.out.println("Trop long : abandon !"); }
      finally { exec8.shutdown(); }
```

Exemple de sortie :

```
123456789123456789
Trop long : abandon !
```

Le programme de test qui suit affiche un décompte des secondes écoulées jusqu'à la résolution du calcul (au passage il illustre une autre méthode *sleep*).

```
long num=new Scanner(System.in).nextLong();
ExecutorService exec8=Executors.newSingleThreadExecutor();
Future<LinkedList<Long>> listPrimes=exec8.submit(new ComputeTask(num));
int delai=0;
try {
   do {
      System.out.print(" "+delai++);
      TimeUnit.SECONDS.sleep(1);
   }
   while (!listPrimes.isDone());
   System.out.println("\n"+listPrimes.get());
}
catch (InterruptedException e) {} catch (ExecutionException e) {}
finally {  exec8.shutdown();}
```

Exemple de sortie :

- Enumération

```
123456789123456789
 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28
[3, 3, 7, 11, 13, 19, 3607, 3803, 52579]
```

Synchronisation des threads

La notion de threads complique la représentation mentale que l'on peut avoir de l'exécution d'une application.

Dans un contexte mono-thread une application orientée objet peut se percevoir comme une interaction d'objet : dans cette vision le flux unique d'exécution passe d'un objet à l'autre.

Dans un contexte multi-thread deux paradigmes se chevauchent : celui d'objets qui interagissent et celui de threads multiples qui traversent ces objets.

Deux threads peuvent traverser simultanément un même objet pour tenter de le modifier : cette situation de partage impose évidemment la mise en place d'une politique de synchronisation permettant à ces threads de garder la cohérence des entités qu'ils modifient.

La nécessité de la synchronisation résulte du partage d'une ressource entre plusieurs activités. Les threads, par construction, partagent, à l'inverse des processus, presque tout : le code, les données globales, le tas (*heap*). Seule la pile, et donc ses variables locales, appartient en propre à chaque thread.

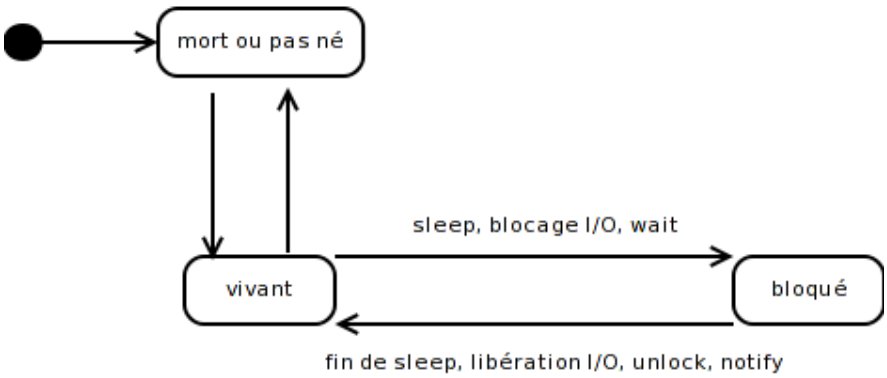
Les contraintes de synchronisation sont donc indissolublement liées à la notion de threads et ont été incorporées dès l'origine dans le langage.

Si la création d'un thread est une chose simple en java, la maitrise d'une architecture multithreadée et des contraintes de synchronisation qui s'y attachent sont des choses complexes.

Etats d'un thread

Les états d'un thread sont une affaire de système d'exploitation. Les états que nous allons considérer ici sont une version simplifiée de ces derniers :

- l'objet *Thread* n'existe pas.
- l'objet *Thread* existe et le thread peut être en cours d'exécution (il possède le processeur ou il l'aura dès que son tour viendra dans le tourniquet des threads éligibles).
- l'objet *Thread* existe et il est soit mort parce qu'il a terminé son exécution, soit pas encore né (méthode *start* non appelée)
- l'objet *Thread* existe et le thread est bloqué sur une ressource dont il attend la libération.



Les évènements concernant les transitions *vivant/bloqué* peuvent être des évènements purement système (accès aux entrées sorties par exemple) ou des événements de l'application (synchronisation avec *wait, notify* etc...)

Exemple de problème de synchronisation

Un arbre binaire de recherche est une structure chaînée où les éléments sont ordonnés. Le parcours simultané d'une telle structure par plusieurs threads réalisant par exemple l'insertion d'un élément est une opération à haut risque si aucun moyen de synchronisation est prévu : il suffit qu'un thread soit interrompu

par l'ordonnanceur au milieu d'une phase de mise à jour des pointeurs de cette arbre, et qu'à cet instant un autre thread parcourt la même zone pour qu'un risque d'incohérence apparaisse.

C'est ce que tente d'illustrer a minima la portion de code ci-dessous qui consiste à mettre en oeuvre *MAX* threads d'insertion qui réalisent chacun *MAX* insertions. Le thread principal se charge, après avoir attendu suffisamment longtemps pour être sûr que les taches d'insertion sont finies, d'afficher le nombre d'éléments qui devrait être MAX² (il s'agit d'un arbre autorisant les doublons)

```
class Tree<T extends Comparable> {
   Node <T> root;
   private static class Node <T> {
      Node(T info) { this.info=info; }
      private T info;
      private Node<T> lNode,rNode;
   }
   void insert(T info) { root=_insert(new Node<T>(info),root);       }
   static private <T extends Comparable> Node<T> _insert(Node<T> node, Node<T> nodeTo) {
      if (nodeTo==null)  return node;
      else if (nodeTo.info.compareTo(node.info)<0) nodeTo.rNode=_insert(node,nodeTo.rNode);
      else nodeTo.lNode=_insert(node,nodeTo.lNode);
      return nodeTo;
   }
   @Override
   public String toString() { return _toString(root); }
   static private <T extends Comparable> String _toString(Node<T> node) {
      return node==null?"":_toString(node.lNode)+" "+node.info+_toString(node.rNode);
   }
   public int length() { return _length(root); }
   static private int _length(Node<?> root) {
           return root==null?0:1+_length(root.lNode)+_length(root.rNode);
   }
}

public class Launcher {
   public static void main(String[] args) {
      final Tree<Integer> tree=new Tree<Integer>();
      final int MAX=80;
      Thread[] tabThread=new Thread[MAX];
      Runnable r=new Runnable() {
         public void run() {
            for(int i=0;i<MAX;i++) tree.insert((int)(Math.random()*100));
         }
      };
      for(int i=0;i<MAX;i++) tabThread[i]=new Thread(r);
      for(Thread thread:tabThread) thread.start();
      try { Thread.sleep(30000); } // attente 30 s
      catch (InterruptedException e) { }
      System.out.println(tree.length());
   }
}
```

Sur la plateforme de test la sortie peut être 6400 (valeur correcte), mais aussi 6396, 6397 etc... Ces dernières valeurs indiquent que des insertions ont été perdues faute de la mise en oeuvre d'une synchronisation qui protège les noeuds de cet arbre d'insertions simultanées.

Java propose plusieurs outils pour résoudre ce type de problème. Dans cet exemple la simple adjonction du mot réservé *synchronized* devant la déclaration de *insert* suffit à régler le problème.

```
synchronized void insert(T info) { root=_insert(new Node<T>(info),root); }
```

Les outils de synchronisation sont étudiés ci-après.

Outils de synchronisation

L'exemple typique pour illustrer la nécessité d'outils de synchronisation est celui d'une réservation. Nous prenons le cas très simple d'une réservation de places dans un avion ci-dessous.

```
class Avion {
   private final int MAX=300;
   private int free=MAX;
```

```
    void reserve(int num) {
      if (num<=free) {
         Thread.yield();
         free-=num;
         System.out.print(this);
      }
   }
    void libere(int num) {
      if (num+free<=MAX) {
         Thread.yield();
         free+=num;
         System.out.print(this);
      }
   }
   @Override
   public String toString() { return " Free="+free; }
}
```

Cette classe permet la réservation par *reserve* d'un certain nombre de places. La méthode s'assure que le nombre de places demandées n'excède pas la capacité (300) de l'avion. De même en cas de désistement la méthode *libere* permet de libérer un certain nombre de places, tout en faisant là aussi une vérification de cohérence. Dans son état normal la variable interne *free* représentant le nombre de places restant disponibles devrait être dans l'intervalle [0-300].

Si un avion est sollicité par plusieurs threads simultanés des incohérences peuvent survenir : lorsqu'un thread est interrompu , par exemple juste après l'évaluation de *num<=free* dans *reserve (*supposée *true* par exemple), un autre thread peut s'engager dans la même portion de code, faire le même constat (*num<=free* est *true*) et s'engager dans sa propre réservation. Le thread initial lorsqu'il reprendra la main sur la base de son évaluation précédente de l'expression manipulera en réalité une variable *free* qui n'a plus la valeur qu'elle avait lors de cette évaluation.

Pour rendre plus probable une telle situation nous avons interposé un *Thread.yield()* qui force le passage de main vers un autre thread.

Enfin une trace dans *libere* et *reserve* permet de suivre l'évolution de la variable *free*.

Dans le programme de test ci-dessous on a supposé qu'un certain nombre (100) d'agences procédaient à des réservations et des libérations aléatoires de places.

```
class Agence implements Runnable {
   private Avion avion;
   private final int MAX=300;
   public Agence(Avion avion) { this.avion=avion; }
   public void run() {
      for (int i=0;i<MAX;i++) {
         int rnd=(int)(Math.random()*500);
         if (rnd%2==0) avion.reserve(rnd);
         else avion.libere(rnd);
      }
   }
}
public class launcher {
   public static void main(String[] args) {
      final int MAX_AGENCES=100;
      Agence [] tabAgences=new Agence[MAX_AGENCES];
      Avion avion=new Avion();
      for(int i=0;i<MAX_AGENCES;i++) tabAgences[i]=new Agence(avion);
      ExecutorService exec=Executors.newFixedThreadPool(MAX_AGENCES);
      for(int i=0;i<MAX_AGENCES;i++) exec.execute(tabAgences[i]);
      try {
         TimeUnit.SECONDS.sleep(3);
      } catch (InterruptedException e) {}
      System.out.println(avion);
      exec.shutdown();
   }
}
```

La sortie met en évidence l'incohérence qui résulte de l'absence de protection de la classe *Avion*

```
 Free=198 Free=-30 Free=-90 Free=-57 Free=66 Free=357 Free=446 Free=803 Free=870 Free=1183
etc...
```

Nous allons examiner dans la suite différentes façons de résoudre le problème.

Verrous

La solution des problèmes de cohérence évoqués passe par la possibilité de rendre le couple d'instructions des méthodes *libere* ou *reserve* indivisible, d'en faire, autrement dit, une section critique.

Les verrous sont un des moyens d'y parvenir.

Le package *java.util.concurrent.locks* fournit la classe *ReentrantLock.* On dispose avec cette classe de l'outil le plus général de synchronisation fournit par Java, qui fournit par ailleurs, avec les blocs synchronisés par exemple, d'autres outils plus simples mais aussi plus limités.

Le principe du verrouillage est simple : lorsqu'un thread pénètre dans la section à protéger il verrouille (*lock*) une instance de cette classe. Si cette instance avait déjà été verrouillée (par un autre thread ou par le même, d'où le nom *Reentrant*) le thread est bloqué en attente de la libération du verrou. En sortie de section le thread déverrouille (*unlock*) l'instance, libérant ainsi un des threads en attente.

Le programmeur doit s'assurer de la complétude de ces cycles *verrouillage/déverrouillage* . Un schéma typique et simple de mise en oeuvre met à contribution la clause *finally* pour ses propriétés interessantes concernant la prise en compte de la libération de ressource.

```
class X {
   private final ReentrantLock lock = new ReentrantLock();
   // ...
   public void m() {
     lock.lock();  // verrouillage
     try {
       // ... method body
     } finally {
       lock.unlock() // deverrouillage
     }
   }
 }
```

En appliquant ce schéma à la classe *Avion* nous obtenons :

```
class Avion {
   private final int MAX = 300;
   private int free = MAX;
   ReentrantLock lock = new ReentrantLock();
   void reserve(int num) {
      lock.lock();
      try {
         if (num <= free) {
            Thread.yield();
            free -= num;
            System.out.print(this);
         }
      } finally {   lock.unlock(); }
   }

   void libere(int num) {
      lock.lock();
      try {
         if (num + free <= MAX) {
            Thread.yield();
            free += num;
            System.out.print(this);
         }
      } finally {   lock.unlock(); }
   }
   @Override
   public String toString() { return " Free="+free; }
}
```

La sortie est cette fois cohérente, les valeurs obtenues sont dans l'intervalle [0-300]:

```
 Free=200 Free=76 Free=163 Free=87 Free=136 Free=10 Free=107 Free=192 Free=74 Free=289
Free=145 etc...
```

L'interface *Lock* dont hérite la classe *ReentrantLock* expose clairement les principales méthodes de cette classe.

```
public interface Lock {
   void lock();
   void lockInterruptibly() throws InterruptedException;
   boolean tryLock();
   boolean tryLock(long time, TimeUnit unit) throws InterruptedException;
   void unlock();
   Condition newCondition();
}
```

En plus du couple *lock* et *unlock* on trouve *tryLock*, sa version avec temps de garde, et *lockInterruptibly* qui sont des variantes permettant d'accéder à plus de souplesse dans l'utilisation de cette outil. La méthode *newCondition* est exposée ci-après.

La classe *ReentrantLock* ajoute à ce jeu de méthodes héritées de nombreuses méthodes spécifiques.

Les schémas de mise en oeuvre de cette classe dans des problèmes de synchronisation plus délicats peuvent être bien sûr plus complexes, c'est ce qui fait l'intérêt de cette classe. Toutefois il faut veiller à une mise en oeuvre cohérente des cycles verrouillage/déverrouillage (qui peuvent être contrariés par la survenue d'exception, d'où l'intérêt de la clause *finally)*, sous peine de générer des situations d'interblocage.

Conditions

Dans l'exemple de l'avion nous avons défini le verrou *lock* en tant que variable membre de la classe *Avion.* Cela signifie que les opérations *libere* et *reserve* seront exclusives (et exclusives l'une de l'autre) pour une instance d'avion donnée (mais deux threads distincts gardent la possibilité d'activer *libere* (ou *reserve*) simultanément lorsque qu'il s'agit de deux instances d'*Avion* distinctes).

Lorsqu'on demande la réservation d'un nombre de places supérieur à la capacité disponible la méthode actuelle ne fait rien. On pourrait néanmoins envisager une autre politique de synchronisation : lorsqu'on demande la réservation d'un nombre de places supérieur à la capacité disponible, la méthode *reserve* attend qu'un ou plusieurs autres threads activent *libere*, jusqu'à ce que le nombre de places libérées rende possible sa propre transaction.

Les *conditions* rendent possibles l'implémentation d'une telle synchronisation. Le *nom* condition vient d'une structure de synchronisation appelée *moniteur de* Hoare, du nom de son créateur, qui a inspiré beaucoup de travaux autour du thème de la synchronisation.

L'interface *Condition* expose les méthodes suivantes:

```
public interface Condition {
    void await() throws InterruptedException;
    void awaitUninterruptibly();
    long awaitNanos(long nanosTimeout) throws InterruptedException;
    boolean await(long time, TimeUnit unit) throws InterruptedException;
    boolean awaitUntil(Date deadline) throws InterruptedException;
    void signal();
    void signalAll();
}
```

Les deux principales méthodes sont *await* et *signalAll*. La première met le thread qui l'invoque dans la file d'attente des threads qui attendent que la condition soit réalisée (dans notre cas que le nombre de places libres soit suffisant pour procéder à la réservation du nombre de places souhaité par le thread courant). *signalAll* invite le plus vieux thread en attente sur cette condition à réexaminer la situation (dans notre cas *num+free<=MAX*) : si c'est un échec le thread reprend sa place dans la file d'attente sinon il poursuit son exécution normalement et procède donc à la réservation.

Ce schéma implique que le thread qui souhaite conditionner son action dans une section critique le fasse typiquement dans une boucle *while (not condition) condition.await()*. Une variable de type *condition* est sémantiquement liée à une expression booléenne, d'où le nom choisi pour ce type de variable. Une condition se crée toujours à partir d'un verrou (en lui appliquant *newCondition()*) et il est possible de définir plusieurs conditions sur un même verrou.

La version d'*Avion* avec condition est la suivante. Il faut noter dans cet exemple une certaine dissymétrie de *reserve* et *libere* : on peut tenter de réserver plus de place qu'il y en a de disponibles, mais, dans une

véritable logique de réservation, on ne doit pas normalement se trouver dans une situation où on demande une libération de places supérieures aux nombre de places occupés. Notre programme de simulation d'agence ne se soucie pas de cette cohérence, mais pour préserver cette dissymétrie le programme ignore les tentatives de libération incohérentes. Donc les seuls threads susceptibles de *signaler* les threads en attente d'un nombre de places suffisant pour leur propre demande sont les threads traversant *libere*.

```
class Avion {
   private final int MAX = 300;
   private int free = MAX;
   ReentrantLock lock = new ReentrantLock();
   Condition freeEnough=lock.newCondition();
   void reserve(int num) {
      lock.lock();
      try {
         System.err.println("demande:"+num+" free:"+free);
         while (num > free) {
            System.err.println("await");
            freeEnough.await();
         }
         Thread.yield();
         free -= num;
         System.out.print(this);

      } catch (InterruptedException e) {
      } finally {   lock.unlock(); }
   }

   void libere(int num) {
      lock.lock();
      try {
         System.err.println("libere:"+num+" free:"+free);
         if (num + free <= MAX) {
            Thread.yield();
            free += num;
            System.out.print(this);
            freeEnough.signalAll();
         }
      } finally {
         lock.unlock();
      }
   }
   @Override
   public String toString() { return " Free="+free; }
}
```

Exemple de sortie (hors commentaires) :

```
demande:336 free:300 // cette demande ne pourra jamais etre satisfaite
await         // elle est mise en attente de condition
libere:95 free:300 // ignoré
libere:233 free:300// ignoré
demande:222 free:300 // OK pas d'attente
 Free=78libere:487 free:78  // ignoré
libere:303 free:78  // ignoré
libere:207 free:78  // ignoré
 Free=285demande:2 free:285 // OK pas d'attente
 Free=283libere:1 free:283
 Free=284libere:209 free:284
demande:442 free:284
await
demande:232 free:284
 Free=52demande:204 free:52 // attente de condition
await
etc...
```

Méthode synchronized

Le mot réservé *synchronized* permet de déclarer une méthode en tant que section critique, comme ci-dessous:

```
public synchronized  void reserve(int num)
```

La première version d'*Avion* (celle ou on ignore les demandes de réservations qui ne peuvent pas être satisfaites tout de suite) peut être écrite très simplement ainsi :

```
class Avion {
   private final int MAX = 300;
   private int free = MAX;
   synchronized void reserve(int num) {
     if (num <= free) {
        Thread.yield();
        free -= num;
        System.out.print(this);
     }
   }
   synchronized void libere(int num) {
     if (num + free <= MAX) {
        Thread.yield();
        free += num;
        System.out.print(this);
     }
   }
   @Override
   public String toString() { return " Free="+free; }
}
```

La déclaration *synchronized* rend implicite la pose d'un verrou, son verrouillage lors de l'entrée dans la méthode et son déverrouillage en sortie.

```
public synchronized  void reserve(int num) { ... }
```

est donc équivalent à ce qui suit (*où* lock désigne une variable d'instance de type *ReentrantLock*)

```
public  void reserve(int num) {
   lock.lock();
   try {
      ...
   }
   finally {
     lock.unlock();
   }
}
```

Dans la classe *Avion, libere* et *reserve* sont donc en accès exclusif et en accès exclusif l'un de l'autre.

Méthode *synchronized* avec *Condition*

De même un support élégant de la notion de condition est proposé dans le cas de méthode *synchronized*.

La construction typique est la suivante :

```
   synchronized void reserve(int num) {
     while (num > free) wait();   // equivaut à lock.await()
     free -= num;
   }
   synchronized void libere(int num) {
     if (num + free <= MAX) {
        free += num;
        notifyAll();              // equivaut à lock.signalAll()
     }
   }
```

Une reformulation de la classe *Avion*, dans une version où on donne une chance aux réservation qui ne sont pas possibles tout de suite mais pourrait le devenir si entre temps des libérations surviennent, peut être la suivante :

```
class Avion {
   private final int MAX = 300;
   private int free = MAX;
   synchronized void reserve(int num) {
```

```
      System.err.println("demande:"+num+" free:"+free);
      while (num > free) {
         System.err.println("wait");
         try {
            wait();
         } catch (InterruptedException e) {
         }
      }
      Thread.yield();
      free -= num;
      System.out.print(this);
   }
   synchronized void libere(int num) {
      System.err.println("libere:" + num + " free:" + free);
      if (num + free <= MAX) {
         Thread.yield();
         free += num;
         System.out.print(this);
         notifyAll();
      }
   }
   @Override
   public String toString() {
      return " Free=" + free;
   }
}
```

Exemple de sortie :

```
demande:58 free:300
 Free=242demande:88 free:242
 Free=154demande:6 free:154
 Free=148libere:225 free:148
demande:202 free:148
wait
libere:471 free:148
demande:268 free:148
wait
demande:332 free:148
wait
libere:453 free:148
libere:223 free:148
demande:448 free:148
wait
etc...
```

Bloc synchronized

La classe *Object*, au sommet de l'arbre des classes Java, possède en tant que variable d'instance un verrou simple. Cela signifie que tout objet Java possède ce verrou. De même sur un objet quelconque il est possible d'invoquer les méthodes *wait* ou *notifyAll*.

Java propose une construction spécifique fondée sur la notion de bloc (délimité par les classiques accolades *{* et *}*) pour associer un verrou à une section quelconque de code. Le schéma est le suivant :

```
synchronized(obj) {
   ... bloc de code en accès exclusif
}
```

La version simple de l'*Avion* (avec renoncement en cas de réservation impossible immédiatement) peut s'exprimer ainsi avec des blocs synchronisés (ici on a retenu *this* comme objet de synchronisation)

```
class Avion {
   private final int MAX = 300;
   private int free = MAX;

   void reserve(int num) {
      synchronized (this) {
         if (num <= free) {
            Thread.yield();
```

```
            free -= num;
            System.out.print(this);
         }
      }
   }
   void libere(int num) {
      synchronized (this) {
         if (num + free <= MAX) {
            Thread.yield();
            free += num;
            System.out.print(this);
         }
      }
   }
   @Override
   public String toString() {
      return " Free=" + free;
   }
}
```

Bloc synchronized avec *condition* implicite

De même qu'avec *synchronized* utilisé pour marquer une méthode, un bloc synchronisé possède une condition implicite (une seule) qui peut être invoquée par *wait* et *notifyAll*.

La version de l'*Avion* donnant une chance aux demandes de réservations qui ne peuvent pas être satisfaites immédiatement peut être exprimée ainsi avec des blocs synchronisés munis d'une condition.

```
class Avion {
   private final int MAX = 300;
   private int free = MAX;
   void reserve(int num) {
      synchronized (this) {
         System.err.println("demande:" + num + " free:" + free);
         while (num > free) {
            System.err.println("wait");
            try {
               wait();
            } catch (InterruptedException e) {
            }
         }
         Thread.yield();
         free -= num;
         System.out.print(this);
      }
   }

   void libere(int num) {
      synchronized (this) {
         System.err.println("libere:" + num + " free:" + free);
         if (num + free <= MAX) {
            Thread.yield();
            free += num;
            System.out.print(this);
            notifyAll();
         }
      }
   }

   @Override
   public String toString() {
      return " Free=" + free;
   }
}
```

On invoque *wait* et *notify* sur l'objet courant *this* (c'est à dire un *Avion*) Ces méthodes sont disponibles par l'héritage implicite de la classe *Object*.

La sortie est cohérente :

```
demande:422 free:300
wait
demande:448 free:300
wait
libere:103 free:300
libere:205 free:300
demande:80 free:300
 Free=220demande:400 free:220
wait
demande:320 free:220
wait
demande:212 free:220
 Free=8libere:359 free:8
demande:68 free:8
wait
etc...
```

Service d'exécution et variable future

Les services d'exécution en *pool* déjà étudiés ne sont pas à proprement parler des services de synchronisation car leur vocation première est l'optimisation des ressources.. Mais ils peuvent procéder à une certaine forme de synchronisation le cas échéant : ainsi un pool à nombre fixe de threads est capable de mettre en attente les threads excédentaires. Dans sa version où le nombre de threads est un (*single*) ce pool procède à une sérialisation de l'exécution des taches qu'on lui soumet.

Les variables futures, étudiées par ailleurs, sont un élégant outil de synchronisation : elles permettent de mettre en attente une activité sur la disponibilité du résultat produit par une autre activité. Le cas typique d'utilisation est donc la mise en oeuvre d'un appel fonctionnel asynchrone.

Le rendez-vous entre threads avec *join*

La classe *Thread* présente la méthode *join* et deux variantes permettant l'apposition d'un temps de garde.

```
public class Thread {
public final synchronized void join(long millis) throws InterruptedException
public final synchronized void join(long millis, int nanos) throws InterruptedException
public final void join() throws InterruptedException
etc...
}
```

Lorsque le thread courant exécute *t.join()*, il suspend sa propre exécution en attente de la terminaison du thread *t*. Cette terminaison correspond pout le thread identifié par *t* par l'atteinte d'un état pour lequel *t.isAlive()* retourne la valeur *false*.

Ainsi le programme mentionné dans exemple de problème de synchronisation lançait *MAX* threads d'insertion aléatoire dans un arbre binaire. Pour afficher le nombre total d'insertion réalisées (normalement MAX^2 dans ce programme) nous avions mis le thread courant en sommeil pendant un temps suffisant pour permettre à l'ensemble des threads de faire leur travail. Cette méthode peut être avantageusement remplacée par une boucle de *join* sur chacun des threads d'insertions.

```
public class Launcher {
   public static void main(String[] args) {
      final Tree<Integer> tree=new Tree<Integer>();
      final int MAX=200;
      Thread[] tabThread=new Thread[MAX];
      Runnable r=new Runnable() {
         public void run() {
            for(int i=0;i<MAX;i++) tree.insert((int)(Math.random()*100));
         }
      };
      for(int i=0;i<MAX;i++) tabThread[i]=new Thread(r);
      for(Thread thread:tabThread) thread.start();
      try {
         for(Thread thread:tabThread)thread.join(); // attente de la terminaison des 200
threads
      } catch (InterruptedException e1) { }      // se substitue à Thread.sleep(30000);
      System.out.println(tree.length());  // A l'affichage le travail des threads est terminé
   }
```

Autres objets de synchronisation

A côté des mécanismes de bas niveaux implémentés dès la conception du langage, la librairie Java s'est enrichie depuis d'un ensemble de composants de plus haut niveau d'abstraction, avec la version 1.5 notamment. Ils sont regroupés dans le package *java.util.concurrent*.

ArrayBlockingQueue, ConcurrentHashMap, ConcurrentLinkedQueue, ConcurrentSkipListMap, ConcurrentSkipListSet, CopyOnWriteArrayList, CopyOnWriteArraySet, CountDownLatch, CyclicBarrier, DelayQueue, Exchanger, ExecutorCompletionService, Executors, FutureTask, LinkedBlockingDeque, LinkedBlockingQueue, PriorityBlockingQueue, ScheduledThreadPoolExecutor, Semaphore, SynchronousQueue, ThreadPoolExecutor

Certaines de ces classes sont des versions des classes de la *JCF* rendues résistantes au multithreading (*ArrayBlockingQueue, ConcurrentHashMap, ConcurrentLinkedQueue, ConcurrentSkipListMap, ConcurrentSkipListSet*, ...)

D'autres sont des classes adaptées à quelques problèmes génériques et récurrents de synchronisation. Nous n'évoquerons dans la suite que certains d'entre eux.

La classe Semaphore

Cette classe est une implémentation d'un concept de synchronisation souvent pris en charge au plus bas niveau par les systèmes d'exploitation. Imaginé par *Djisktra*, les sémaphores permettent, dans leur forme la plus simple, de contrôler l'accès à une ressource critique à un nombre fixé et maximal de threads. Un sémaphore initialisé à une valeur maximale de un se comporte donc comme un verrou.

Lorsqu'un thread souhaite accéder à la ressource il acquiert (*acquire*) un ticket auprès du sémaphore. Lorsqu'il quitte la ressource il rend le ticket (*release*), signalant ainsi un éventuel thread en attente.

Le nombre maximal de threads autorisés à pénétrer simultanément dans le ressource est fixé à l'intialisation du sémaphore.

Le programme suivant illustre une utilisation typique de sémaphore. La classe *Runnable PrinterPool se* charge de fournir une ressource d'impression au thread qui l'exécute. Trois imprimantes sont à la disposition des 10 threads utilisateurs. L'accès exclusif à trois threads au plus est réglé par le sémaphore. Chaque job d'impression a une durée aléatoire.

```
 class Printer {
   private int ident;
   private boolean busy=false;
   Printer(int n) { this.ident=n; }
   void doPrint() throws InterruptedException {
      busy=true;
      System.err.println("Printer "+ident+" begin job at "+new Date());
      TimeUnit.SECONDS.sleep((int)(Math.random()*10));
      busy=false;
   }
   public boolean isBusy() {       return busy;}
}
class PrinterPool implements Runnable{
   final int MAX_PRINTERS=3;
   Printer [] tabPrinter=new Printer[MAX_PRINTERS];
   Semaphore semaphore=new Semaphore(MAX_PRINTERS,true);
   PrinterPool() { for(int i=0;i<MAX_PRINTERS;i++) tabPrinter[i]=new Printer(i); }
   void print() throws InterruptedException {
      semaphore.acquire();
      Printer printer=getPrinter();
      printer.doPrint();
      semaphore.release();
   }
   private Printer getPrinter() {
      for(Printer printer:tabPrinter) if (!printer.isBusy()) return printer;
      return null;
   }
   public void run() {
      try { print();
      } catch (InterruptedException e) { }
   }
}
```

```
public class Launcher {
   public static void main(String[] args) {
      final int MAX_JOBS=10;
      PrinterPool printerPool=new PrinterPool();
      ExecutorService exec=Executors.newCachedThreadPool();
      for(int i=0;i<MAX_JOBS;i++) exec.execute(printerPool);
      exec.shutdown();
   }
}
```

La sortie datée permet de vérifier que la ressource est utilisée au maximum de ses capacités :

```
Printer 1 begin job at Thu Jul 05 15:30:28 CEST 2007
Printer 2 begin job at Thu Jul 05 15:30:28 CEST 2007
Printer 0 begin job at Thu Jul 05 15:30:28 CEST 2007 // les 3 premières taches accèdent tout de suite
Printer 1 begin job at Thu Jul 05 15:30:31 CEST 2007
Printer 2 begin job at Thu Jul 05 15:30:32 CEST 2007
Printer 2 begin job at Thu Jul 05 15:30:35 CEST 2007
Printer 0 begin job at Thu Jul 05 15:30:36 CEST 2007
Printer 1 begin job at Thu Jul 05 15:30:39 CEST 2007
Printer 0 begin job at Thu Jul 05 15:30:42 CEST 2007
Printer 2 begin job at Thu Jul 05 15:30:42 CEST 2007
```

La classe *Semaphore* permet également à un thread de prendre ou de rendre plusieurs tickets.

Le compte à rebours CountDownLatch

La classe *CountDownLatch* permet de mettre en attente le thread invoquant *await* de l'expiration du compte à rebours interne mise oeuvre par l'objet *CountDownLatch.* Les méthodes principales exposées par cette classe sont, outre le constructeur, *await et countDown.* La méthode *countDown* sera mise en oeuvre chaque fois qu'il s'agira de de décrémenter la valeur interne du compte à rebours.

L'exemple suivant met en oeuvre 100 threads qui ont chacun pour tache de déposer une valeur aléatoire dans une des cases du tableau *tabValue*. Quand le job est fait, chaque thread invoque *countDown* sur le compte à rebours. Le thread principal attend sur *await* et sera débloqué lorsque le travail de tous les threads sera fini.

```
final int MAX_THREADS=100;
      final CountDownLatch cdl=new CountDownLatch(MAX_THREADS);
      final Integer [] tabValue=new Integer[MAX_THREADS];
      ExecutorService exec1=Executors.newCachedThreadPool();
      for (int i=0;i<MAX_THREADS;i++) {
         final int num=i;
         exec1.execute(new Runnable() {
            public void run() {
               tabValue[num]=(int)(Math.random()*100);
               cdl.countDown();
            }
         });
      }
      try {
         cdl.await();
      } catch (InterruptedException e) {}
      for(Integer n:tabValue) System.out.print (n+",");
      exec1.shutdown();
```

Sortie :

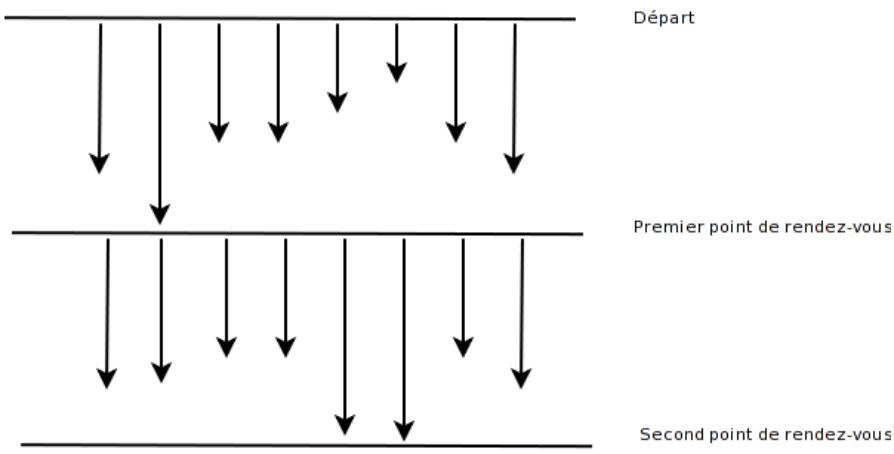
```
74,94,27,67,28,4,59,74,82,97,87,51,43,64,21,15,80,75,67,28 etc...
```

Cyclic Barrier

Un objet de type *CyclicBarrier* permet de poser un point de synchronisation entre plusieurs threads, exécuter une action spécifique en ce point éventuellement, et être réutilisé pour une autre synchronisation (d'où le nom *cyclic*).

Dans cet exemple chaque travailleur dépose dans une case donnée d'un tableau de numérique 6 valeurs successives, et l'objet *CyclicBarrier* calcule par l'objet anonyme qu'on lui passe la somme des 100 valeurs, ceci 6 fois.

- Enumération




```
class Worker implements Runnable {
   private CyclicBarrier cb;
   private int index;
   private Integer[] tabValue;
   public Worker(CyclicBarrier cb,Integer [] tabValue,int index) {
      this.cb=cb;this.tabValue=tabValue;this.index=index;
   }
   public void run() {
      for(int i=1;i<=6;i++) {
         tabValue[index]=(int)(Math.random()*i*100);
         try { TimeUnit.SECONDS.sleep((int)(Math.random()*5));}
            catch (InterruptedException e1) { }
         try {
            cb.await();
         } catch (InterruptedException e) {
         } catch (BrokenBarrierException e) {
         }
      }
   }
}

public class Launcher {
   public static void main(String[] args) {
      int MAX_THREADS=100;
      final Integer [] tabValue=new Integer[MAX_THREADS];
      CyclicBarrier cb=new CyclicBarrier(MAX_THREADS,new Runnable() {
         public void run() {
            int somme=0;
            for(Integer num:tabValue) somme+=num;
            System.out.println("Somme des éléments = "+somme+ " at"+ new Date());
         }
      });
      ExecutorService exec1=Executors.newCachedThreadPool();
      for (int i=0;i<MAX_THREADS;i++) exec1.execute(new Worker(cb,tabValue,i));
      exec1.shutdown();
   }
}
```

La sortie montre qu'à chaque fois on attend l'atteinte du point de synchronisation des 100 threads (appel de *await*) dure 4 secondes, sachant que pour chacun des 6 cycles chaque tache va avoir une durée aléatoire entre 0 et 4 s. Il est en effet très probable que sur les 100 threads parallèles il y en ait au moins un qui tire au hasard une durée de 4 secondes environ.

```
Somme des éléments = 4879 atFri Jul 06 15:36:04 CEST 2007
Somme des éléments = 9494 atFri Jul 06 15:36:08 CEST 2007
Somme des éléments = 14058 atFri Jul 06 15:36:12 CEST 2007
Somme des éléments = 20292 atFri Jul 06 15:36:16 CEST 2007
Somme des éléments = 26040 atFri Jul 06 15:36:20 CEST 2007
Somme des éléments = 29121 atFri Jul 06 15:36:24 CEST 2007
```

TimerTask et ScheduledThreadExecutor

TimerTask est un outil simple qui permet de lancer une tâche à intervalle régulier. ScheduledThreadExecutor est dédiée au même objectif, mais, comme son nom l'indique, il s'agit d'un des *ExecutorService* disponible au moyen de la classe *Executors* : en conséquence il s'agit d'un *thread pool executor* muni des capacités à gérer un pool de threads, mais muni également d'outils de déclenchement divers. L'exemple ci-dessous utilise les deux classes sur un exemple très simple :

```
ScheduledThreadPoolExecutor stpe=new ScheduledThreadPoolExecutor(5);
stpe.scheduleAtFixedRate(new Runnable() {
   public void run() { System.out.println("Hello from ScheduledThreadPoolExecutor"); }
   }, 1L,2L,TimeUnit.SECONDS);
Timer timer=new Timer();
timer.schedule(new TimerTask() {
   public void run() {
         System.out.println("Hello from TimerTask");
   }
   }, 100L, 1500L);
```

La sortie donne :

```
Hello from TimerTask
Hello from ScheduledThreadPoolExecutor
Hello from TimerTask
Hello from ScheduledThreadPoolExecutor
Hello from TimerTask
Hello from TimerTask
Hello from ScheduledThreadPoolExecutor
```

Sécurisation des Conteneurs

Les conteneurs de la *Java Collection Framework* ne sont pas par défaut sécurisés. La classe *Collections*, qui a vocation à proposer une ensemble de méthodes statiques pour les collections, présente également des méthodes permettant d'immuniser ces collections vis à vis du multithreading et des problèmes que posent l'accès simultané à une même structure.

L'exemple ci-dessous illustre les dangers du multithreading lorsqu'il est mis en oeuvre sans précaution sur un conteneur, il s'agit ici d'un *TreeMap*. La *CyclicBarrier* est là simplement pour tenter d'afficher un état final de l'arbre lorsque tous les threads ont terminé leur travail.

```
class Worker implements Runnable {
   private Map<Integer, String> tm;
   private CyclicBarrier cb;
   public Worker(Map<Integer, String> dtm, CyclicBarrier cb) { this.tm=dtm;this.cb=cb; }
   public void run() {
      for(int i=0;i<10000;i++) tm.put(i, Integer.toString(i, 16));
      try {   cb.await(); // pour la synchronisation de terminaison avec CyclicBarrier
      } catch (InterruptedException e) {} catch (BrokenBarrierException e) {}
   }
}
public class Launcher {
      // Attention version non sécurisée sans synchro !
      final int MAX_THREADS=100;
      final Map<Integer,String> tm2=new TreeMap<Integer, String>();
      ExecutorService exec2=Executors.newCachedThreadPool();
      CyclicBarrier cb=new CyclicBarrier(MAX_THREADS, new Runnable() {
         public void run() {
            System.out.println(tm2.size());
            System.out.println(tm2.toString().substring(0, 100));
         }
      });
      for(int i=0;i<MAX_THREADS;i++) exec2.execute(new Worker(tm2,cb));
      exec2.shutdown();
   }
}
```

La sortie révèle, de façon non déterministe, la survenue de problème pendant une opération d'insertion, au milieu d'une opération de rééquilibrage de l'arbre, conduisant à l'arrêt du programme :

```
Exception in thread "pool-1-thread-9" java.lang.NullPointerException
   at java.util.TreeMap.rotateLeft(TreeMap.java:1261)
```

```
   at java.util.TreeMap.fixAfterInsertion(TreeMap.java:1328)
   at java.util.TreeMap.put(TreeMap.java:483)
```

La version sécurisée de ce programme est la suivante (la classe *Worker* est inchangée, elle ne figure pas ci-dessous) :

```
public class Launcher {
   public static void main(String [] args) {
      final TreeMap<Integer,String> tm1=new TreeMap<Integer, String>();
      Map<Integer,String> dtm = Collections.synchronizedMap(tm1); // securisation
      ExecutorService exec1=Executors.newCachedThreadPool();
      final int MAX_THREADS=100;
      CyclicBarrier cb=new CyclicBarrier(MAX_THREADS, new Runnable() {
         public void run() {
            System.out.println(tm1.size());
            System.out.println(tm1.toString().substring(0, 100));
         }
      });
      for(int i=0;i<MAX_THREADS;i++) exec1.execute(new Worker(dtm,cb));
      exec1.shutdown();
   }
}
```

Cette fois le programme se déroule jusqu'à sa fin normale et un état de l'arbre est affiché :

```
10000   // nombre d'éléments insérés
{0=0, 1=1, 2=2, 3=3, 4=4, 5=5, 6=6, 7=7, 8=8, 9=9, 10=a, 11=b, 12=c, 13=d, 14=e, 15=f, 16=10,
17=11,  // 100 premiers caractères
```

La classe *Collections* propose un certain nombre de méthodes dédiés à la sécurisation des conteneurs.

```
class Collections {
   public static <T> Collection<T> synchronizedCollection(Collection<T> c)
   public static <T> Set<T> synchronizedSet(Set<T> s)
   public static <T> SortedSet<T> synchronizedSortedSet(SortedSet<T> s)
   public static <T> List<T> synchronizedList(List<T> list)
   public static <K,V> Map<K,V> synchronizedMap(Map<K,V> m)
   public static <K,V> SortedMap<K,V> synchronizedSortedMap(SortedMap<K,V> m)
   etc...
}
```

Reflection et contrôle dynamique de type

Contrôle dynamique de type (RTTI)

Le contrôle dynamique de type (*Run time Type Identification*) permet d'interroger une instance à l'exécution pour connaître son type dynamique, qui peut être différent de son type déclaratif dans un contexte où le polymorphisme est actif.

Dans un langage comme C++, où la notion même de classe n'existe plus à l'exécution, l'accès dynamique suppose la mise en oeuvre d'une option de compilation qui préserve un minimum d'information symbolique, celle concernant le nom des classes particulièrement.

En Java cela n'est pas nécessaire: il est toujours possible d'interroger une instance avec un opérateur tel que *instanceof* ou avec une méthode telle que *getClass*. L'information concernant le type est conservée à l'exécution, et cela va même plus loin comme cela sera exposé dans la partie *Reflection*.

Exemple :

```
Integer n=5; // autoboxing : équivaut à n=new Integer(5)
System.out.println(n.getClass());
if (n instanceof Integer) System.out.println("n est un Integer");
if (n instanceof Number) System.out.println("n est un Number");
```

Sortie :

```
class java.lang.Integer
n est un Integer
n est un Number
```

Comme on le voit *instanceof* tient compte de la relation de conformité existant entre *Integer* et *Number* dans la mesure où ces classes sont liées par une relation d'héritage.

La classe *Class*

La méthode *getClass* est héritée de *Object* et est déclarée ainsi dans la classe *Object* :

```
class Object {
   public final Class<? extends Object> getClass()
   ...
}
```

C'est donc l'appel implicite à *n.getClass().toString()* qui génère finalement le nom de la classe. Cela montre que le compilateur préserve à l'exécution non seulement le nom de la classe (ici *Integer*) mais aussi un objet associé à cette classe et auquel on accède par *getClass()*. Cet objet est une instance de la classe *Class* et incarne à l'exécution la classe *Integer* elle-même. Il existe d'abord physiquement sous la forme d'un fichier éponyme de la classe et portant l'extension *.class*. Lors de l'exécution du programme, dès que la classe est invoquée dans le programme, mais à ce moment là seulement, ce fichier est chargé (par une entité gérée par *ClassLoader*) et devient une instance représentative de la classe. C'est cette instance de la classe générique *Class* qui prendra en charge, par exemple, la création des instances, au sens habituel, de cette classe. C'est aussi cette instance qui prendra en charge l'accès aux informations concernant le type dynamique d'une instance.

Il s'agit d'un modèle qui présente des similtudes avec celui de SmallTalk et pour lequel les classes sont elles mêmes des instances de métaclasses. Ce modèle est aux antipodes de celui du C++ pour lequel, par défaut, aucune information de type n'existe à l'exécution.

Tout comme l'accès à l'instance courante s'effectue en invoquant *this*, l'accès à l'instance unique représentant une classe donnée *A* s'effectue en invoquant *A.class.* La ligne de code qui suit compare par leurs adresses deux instances qui ne sont qu'une seule instance, celle incarnant la classe *Integer*.

```
if (n.getClass()==Integer.class) System.out.println("n est bien unInteger");
```

Sortie :

```
n est bien un integer
```

Integer.class est une instance de la classe *Class<Integer>* (ou de la classe *Class*, l'écriture non générique étant encore tolérée) comme le montre la portion de code suivante :

```
Class<Integer> inst1=Integer.class;
if (n.getClass()==inst1) System.out.println("n est bien un Integer");
System.out.println(inst1.getClass());
```

La sortie est :

```
n est bien un Integer
class java.lang.Class
```

L'objet *inst1*, interrogé lui-même par *getClass()*, révèle sa nature qui est d'être une instance de *Class.*

Tout comme les classes les types de base ont également une représentation à l'exécution sous forme d'instance.

```
Class inst2=int.class; // seule la version non générique de Class est utilisable ici
System.out.println(int.class.getClass());
```

La sortie est :

```
class java.lang.Class
```

La classe *Integer* et le type de base *int* ont des rapports privilégiés, pris en compte par l'autoboxing notamment. Il en est de même des couples *double/Double*, *float/Float*, *char/Character*, *byte/Byte*, *short/Short*, *long/Long*, *void/Void.* Ces classes sont autant de classes de *wrapping* que le type de base correspondant.

Integer.class représente l'objet incarnant la classe *Integer*, *int.class* représente l'objet incarnant le type *int.* Ces deux entités sont différentes. Toutefois il est possible de référencer depuis la classe *Integer* le type de base correspondant avec *Integer.TYPE.*

```
if (int.class==Integer.class) System.out.println("int.class=Integer.class");
if (int.class==Integer.TYPE) System.out.println("int.class=Integer.TYPE");
```

La sortie ne contient que :

```
int.class=Integer.TYPE
```

Cette façon de déterminer une information de typage en utilisant directement les instances qui représentent les classes n'est pas équivalente aux résultats qu'on obtient avec *instanceof*.

```
Integer n=5; // autoboxing : équivaut à n=new Integer(5)
System.out.println(n.getClass());
if (n instanceof Integer) System.out.println("n est un Integer");
if (n.getClass()==Integer.class) System.out.println("n est un Integer: confirmation");
```

```
    if (n instanceof Number) System.out.println("n est aussi un Number");
    if (n.getClass()==Number.class) System.out.println("surprenant !");
```

La sortie est :

```
n est un Integer
n est un Integer: confirmation
n est aussi un Number
```

instanceof tient compte de la relation d'héritage ce qui n'est pas le cas lorsqu'on compare directement les adresses de deux instances représentatives de deux classes différentes. Même si elles sont liées par l'héritage ces instances sont différentes donc leurs adresses aussi.

Reflection

La *RTTI* permet d'interroger un objet pour avoir accès à son type. La *reflection* est un mécanisme qui permet d'aller plus loin: il est possible également d'interroger une instance et de connaître le nom des méthodes présentées par sa classe, d'avoir accès aux relations d'héritage, de connaître le nom et le type de ses attributs, ses constructeurs, ses types génériques, et même ses annotations pour celles d'entre elles qui sont conservées à l'exécution.

Ce type de fonctionnalités est utile lorsque qu'on est conduit à utiliser des objets dont on ne sait rien : contexte *RMI* (*Remote Method Invocation)* par exemple, ou contexte *JavaBean*, contexte graphique où on incorpore dynamiquement des composants dont on explore les propriétés dynamiquement au moyen de la *reflection* etc...

L'information conservée par le compilateur pour l'exécution va donc au delà de l'information symbolique : c'est la logique opératoire de la classe qui est ainsi préservée. Dans certains contextes contraints, par exemple l'environnement *J2ME*, Java pour les mobiles, la reflection n'est pas proposée en raison de son coût en ressource machine.

La portion de code suivante interroge une instance de liste chaînée d'entiers pour connaître, par exemple, son nom, celui de ses constructeurs, de ses méthodes, des ses champs, des interfaces implémentées etc...

```
LinkedList<Integer> li=new LinkedList<Integer>();
Class cli= li.getClass();
System.out.println(Arrays.deepToString(cli.getDeclaredConstructors()));
System.out.println(Arrays.deepToString(cli.getClasses()));
System.out.println(Arrays.deepToString(cli.getDeclaredFields()));
System.out.println(Arrays.deepToString(cli.getInterfaces()));
System.out.println(Arrays.deepToString(cli.getDeclaredMethods()));
```

Sortie :

```
java.util.LinkedList
[public java.util.LinkedList(), public java.util.LinkedList(java.util.Collection)]
[]
[private transient java.util.LinkedList$Entry java.util.LinkedList.header, private transient
int java.util.LinkedList.size, private static final long
java.util.LinkedList.serialVersionUID]
[interface java.util.List, interface java.util.Queue, interface java.lang.Cloneable,
interface java.io.Serializable]
[public boolean java.util.LinkedList.add(java.lang.Object), public void
java.util.LinkedList.add(int,java.lang.Object), public int
java.util.LinkedList.indexOf(java.lang.Object), public java.lang.Object
java.util.LinkedList.clone(), public void java.util.LinkedList.clear(), public boolean
java.util.LinkedList.contains(java.lang.Object) etc...
```

La class *Class* fournit l'essentiel de la boite à outils. Les noms des méthodes indiquent clairement leur fonction.

```
public final class Class<T> {
    public String toString()
    public static Class<?> forName(String className)
    public T newInstance()
    public native boolean isInstance(Object obj);
    public native boolean isAssignableFrom(Class<?> cls);
    public native boolean isInterface();
    public native boolean isArray();
     public native boolean isPrimitive();
    public boolean isAnnotation()
    public boolean isSynthetic()
    public String getName()
    public ClassLoader getClassLoader()
    public TypeVariable<Class<T>>[] getTypeParameters()
    public native Class<? super T> getSuperclass();
```

```
    public Type getGenericSuperclass()
    public Package getPackage()
    public native Class[] getInterfaces();
    public Type[] getGenericInterfaces()
    public native Class<?> getComponentType();
    public native int getModifiers();
    public native Object[] getSigners();
    public Method getEnclosingMethod()
    public Constructor<?> getEnclosingConstructor()
    public native Class<?> getDeclaringClass()
    public Class<?> getEnclosingClass()
    public String getSimpleName()
    public String getCanonicalName()
    public boolean isAnonymousClass()
    public boolean isLocalClass()
    public boolean isMemberClass()
    public Class[] getClasses()
    public Field[] getFields() throws SecurityException
    public Method[] getMethods() throws SecurityException
    public Constructor[] getConstructors() throws SecurityException
    public Field getField(String name)
    public Method getMethod(String name, Class ... parameterTypes)
    public Constructor<T> getConstructor(Class ... parameterTypes)
    public Class[] getDeclaredClasses() throws SecurityException
    public Field[] getDeclaredFields() throws SecurityException
    public Method[] getDeclaredMethods() throws SecurityException
    public Constructor[] getDeclaredConstructors() throws SecurityException
    public Field getDeclaredField(String name)
    public Method getDeclaredMethod(String name, Class ... parameterTypes)
    public Constructor<T> getDeclaredConstructor(Class ... parameterTypes)
    public InputStream getResourceAsStream(String name)
    public java.net.URL getResource(String name)
    public java.security.ProtectionDomain getProtectionDomain()
    public boolean desiredAssertionStatus()
    public boolean isEnum()
    public T[] getEnumConstants()
    public T cast(Object obj)
    public <U> Class<? extends U> asSubclass(Class<U> clazz)
    public <A extends Annotation> A getAnnotation(Class<A> annotationClass)
    public boolean isAnnotationPresent()
    public Annotation[] getAnnotations()
    public Annotation[] getDeclaredAnnotations()
}
```

La méthode *cast* permet de formuler sans construction syntaxique spécifique l'opération de forçage de type :

```
Number num=5;
Integer n2=(Integer) num;
Integer n3=Integer.class.cast(num);
```

Annotations

Les annotations sont un outil d'appareillage du code Java. Elles permettent de poser, en plus des règles qui régissent la compilation, d'autres règles qui vont intervenir par exemple sur le processus de documentation du code, sur le contrôle de certains éléments sémantiques, sur la production automatisée de fichier XML de description nécessaire dans certains contextes (J2EE, RCP etc...).

L'environnement introduit un petit nombre d'annotations prédéfinies, mais il est possible au moyen d'un langage d'annotation spécifique de créer ses propres annotations.

Les annotations peuvent exercer une action au niveau du code source, mais aussi du code compilé (*.class*) jusqu'au moment de l'exécution. Il est possible par *reflection* d'interroger à l'exécution une classe (*Class*) pour accéder aux annotations qui la concernent.

Annotations prédéfinies

@Override

L'annotation *@override* permet d'annoncer que la méthode qui la suit est une surcharge de méthode héritée. C'est en soit un élément documentaire intéressant pour la lisibilité du programme, mais son apposition permet de s'assurer que la méthode redéfinie est effectivement une surcharge et non l'introduction d'une méthode nouvelle.

Exemple 1 :

```
   class MyDate extends Date {
      String tostring() { // aucune erreur déclenchée : le programmeur pense avoir surchargé toString
         return "date";
      }
   }
```

Exemple 2 :

```
   class MyDate extends Date {
      @Override
      String tostring() { // Erreur : The method tostring() of type Launcher.MyDate must
                                                                override a  superclass method
         return "date";
      }
   }
```

L'annotation *@Override* met en oeuvre un contrôle fin des paramètres et type retournés par la méthode concernée pour s'assurer que sa déclaration est compatible avec la signature de la méthode héritée, voir à ce sujet *Polymorphisme et Covariance*.

@Deprecated

Cette annotation purement documentaire permet d'annoncer qu'une méthode est abandonnée. Un *warning* sera émis si cette méthode est utilisée.

Exemple :

```
class MyDate extends Date {
   @Override
   public  String toString() {
      return "date";
   }
   @Deprecated
   public void display() {
      System.out.println("Hello");
   }
}
...
   MyDate myDate=new MyDate();
   myDate.~~display~~(); // L'environnement Eclipse annonce la depreciation en barrant l'appel
```

@SuppressWarnings

Comme son nom l'indique cette annotation permet de supprimer les éventuels messages d'avertissement générés par la compilation de l'entité concernée.

Création d'annotation

La syntaxe de création d'un nouveau type d'annotation ressemble à celle d'une interface. Lors de la définition de l'annotation il faut préciser la cible (une classe, une méthode, un constructeur, un package, un paramètre etc...) et la portée (code source, code source + fichier *class*, code source + fichier *class* + *runtime*)

Chaque nouvelle définition d'annotation prend place dans son propre fichier *.java* et générera son propre fichier *.class.*

Exemple de définition d'annotation pour une méthode (fichier *MyDisplayAnnotation.java*):

```
import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;
import java.util.Date;

@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
public @interface MyDisplayAnnotation {
   public String title() default "MyAnnotation";
   public int id() default 0;
   int num=8;
   String name="MyAnnotation is there";
   Date date=new Date();
```

```
   enum Size { SMALL, LARGE, HUDGE };
   Size size=Size.LARGE;
}
```

Exemple de définition d'annotation pour une classe (fichier *MyDateAnnotation.java),* sans répétition des *import* :

```
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
public @interface MyDateAnnotation {
     public String title() default "MyAnnotationDate";
     public int id() default 0;
}
```

Exemple d'apposition de ces annotations dans un code source java:

```
@MyDateAnnotation(title="There MyAnnotationDate",id=8)
class MyDate extends Date {
   @Override
   public  String toString() {
      return "date";
   }
   public void display() {
      System.out.println("Hello");
   }
   @MyDisplayAnnotation(id=5,title="Hello!")
   void testDisplay() { display(); }

}
```

Exemple d'exploitation par une application externe :

```
public class Launcher {
   public static void main(String [] args) {
      Class<MyDate> cDate=MyDate.class;
      // recuperation de l'annotation de la classe MyDate
      MyDateAnnotation myDateAnnot=cDate.getAnnotation(MyDateAnnotation.class);
      System.out.println(myDateAnnot);
      // recuperation de l'annotation de la methode display de MyDate
      Method method=null;
      method = cDate.getDeclaredMethods()[2];
      System.out.println(method);
      MyDisplayAnnotation annot=method.getAnnotation(MyDisplayAnnotation.class);
      System.out.println(annot.title());
      System.out.println(annot.id());
      System.out.println(annot.name);
      System.out.println(annot.num);
      System.out.println(annot.date);
      System.out.println(annot.size);
}
```

Sortie :

```
@MyDateAnnotation(title=There MyAnnotationDate, id=8)
void MyDate.testDisplay()
Hello!
5
MyAnnotation is there
8
Tue Jul 10 11:42:46 CEST 2007
LARGE
```

Mots clés de Java

Le langage Java comporte un jeu d'une cinquantaine de mots réservés.

Mot réservé	Commentaire	**Exemple**
abstract	Une méthode ou une classe peut être déclarée *abstract*.	`abstract public class A { ... }` `abstract public void m(...) { ... }`
assert	Permet de positionner une assertion sous la forme d'une expression booléenne. Le contrôle d'assertion est activable à l'exécution par *java -ea etc...*	`assert x>1;`
boolean	Le type booléen comporte deux valeurs : *true* ou *false.* Taille non spécifié.	`boolean b=true;`
break	Sortie immédiate d'une boucle ou d'un switch	`while (x<100) {` `  x++;` `  if (x>50) break;` `  f(x);` `}` `switch (n) {` `  case 0:  x=1; break;` `  case 1 : x=11; break;` `  default : x=50; break` `}` `label1:` `while (true) {` `  x++;` `  break label1;` `  x--;` `}`
byte	Type entier signé sur 8 bits. Intervalle [-128,+127]. Taille 1 octet. Equivalent du type *char* en *C*.	`byte b=0;`
case	Clause de discrimination d'un switch	`voir` ***break***
catch	Interception d'une exception après un bloc *try.*	`try {` `  openConnection();` `  _isConnected=true;` `}` `catch (UnknownHostException e) {` `  e.printStackTrace();` `  _isConnected=false;` `}`
char	Type caractère (codage Unicode UTF-16). Taille 16 bits.	`char TM='\u2122';` `String java="JAVA"+TM;;` `System.out.println(java);`
class	Définition d'une classe --- Accès à l'instance représentative d'une classe à l'exécution --- Accès à l'instance représentative d'un type de base à l'exécution	`public class A {...}` --- `Class<Double> cd=Double.class;` --- `Class cb=boolean.class;`
const	Mot réservé inutilisé	

continue	Termine immédiatement l'itération en cours et reprend à l'itération suivante	`while (x<100) {` `  x++;` `  if (x>50) continue;` `  f(x);` `}`
default	Cas par défaut dans un switch	voir ***break***
do	Initiation d'une boucle do while	`do {` `  x++;` `}` `while (x<100)`
double	Type numérique flottant sur 8 octets. 15 chiffres significatifs. Amplitude 10^{300}	`double x=0.12345678987E+302`
else	Clause sinon d'une alternative	`if (x>100) {` `  x++;` `}` `else {` `  x=-x;` `}`
enum	Type énuméré	`public enum DAY` `{MONDAY,TUESDAY,WEDNESDAY};`
extends	Extension d'une classe par héritage --- Construction d'un type générique contraint	`public class CompteurDeb extends Compteur { ... }` --- `< X extends T >` `< ? extends T >`
false	Valeur de vérité de type *boolean*	`boolean b=false;`
final	Apposé devant une variable ou un paramètre signifie que cette variable ne peut recevoir d'autre valeur ou référencer d'autre objet. --- Devant une classe ou une méthode final bloque la possibilité d'héritage ou de redéfinition. --- Devant un paramètre ou une variable locale marque la possibilité d'utilisation par une classe interne locale	`final double PI=3.14159;` `static final double E=2.78;` --- `final class A { ... }` `final public void m() { ... }` --- `void m(final boolean b) {` `  final num=0;` `  new Runnable() {` `  if (b) num++;` `  ...` `  }` `}`
finally	Une section *finally* complète les sections *try* et *catch* par une section qui sera toujours exécutée qu'il y ait déclenchement d'exception on non.	`try {` `...` `} catch (Exception e) {` `...` `}` `finally {` `....` `}`
float	Type numérique flottant sur 4 octets. 7 chiffres significatifs. Amplitude 10^{38}	`float x=3.895642E-35`
for	Itération classique : *for (init;condition de continuation; iteration)* --- ou formalisme *foreach* : f*or(TypeElement element:tableau)*	`for(int i=0;i<100;i++)` `System.out.println(i);` `for(int i=0;i<tab.length();i++)` `n+=tab[i];` `----------` `for(int t:tab) n+=t;`

goto	Non utilisé	
if	Alternative. Toujours suivie d'une expression booléenne entre parenthèses	`if (x>3) {` `  x++;` `}` `else x--;`
implements	Clause d'implémentation d'interface	`public class A implements Runnable {` `  ...` `  void run() { ... }` `}`
import	Importation de package. --- Avec *static importation* de membres statiques	`import java.util;` `---` `import static java.lang.System.out;` `println("Hello");`
instanceof	Teste du type dynamique d'un objet	`if (o.instanceof(Button)) {` `  ...` `}`
int	Entier signé sur 32 bits. De -2 milliard à +2 milliard environ	`int n=348;`
interface	Classe totalement abstraite ne déclarant, donc, que des méthodes non définies et aucune variable membre.	`public interface Runnable {` `  void run();` `}`
long	Entier long signé sur 64 bits. De -10^{20} à $+10^{20}$	`long n=456123789741;`
native	Accès à une fonction native (c'est à dire spécifique) du système d'exploitation courant	`public static native void hello();`
new	Opérateur d'instanciation	`A a=new A();`
null	Une variable *null* ne référence aucune instance	`A a=null;`
package	Positionnement d'une classe dans un package	`package composant;` `public class Ampoule extends Dipole {` `  ...` `}`
private	Restriction de la visibilité d'une méthode ou d'un membre à la classe seule. S'applique à une méthode ou à un membre, ou à une classe interne.	`public class A {` `  private void m() { ... }` `}`
protected	Restriction de la visibilité d'une méthode ou d'un membre à la classe seule et aux classes héritières. S'applique à une méthode ou à un membre ou à une classe interne.	`public class A {` `  protected void m() { ... }` `}`
public	Accès à tous. S'applique à une classe, une méthode ou un membre	`public class A {` `  public int n;` `  public void m() { ... }` `}`
return	Spécifie le retour de valeur dans une méthode	`public class A {` `  public int f() { return 2+3; }` `}`

short	Entier signé sur 16 bits. De -65536 à +65535	`short n=564;`
static	Appliqué à une variable membre : allocation unique (globale) --- Appliqué à une méthode : méthode sans objet courant (sans *this)* --- Appliqué à une classe interne : classe imbriquée Après *import* , voir *import* Bloc d'initialisation *static* dans une classe.	`static int n=5;` --- `class Math {` `  public static double abs(double x)` `{` `  return x<0?-x:x;` `  }` `}` `public class A {` `  static public class B {` `  ...` `  }` `  ...` `}` `import static java.lang.Math;` --- `public class A {` `  static {` `  System.out.println("Hello");` `  }` `  A() { ... }` `}`
strictfp	Optimisation des calculs en virgule flottante sur registres processeurs (rare).	`public static strictfp void m()` `{ ... }`
super	Invocation d'un constructeur de la surclasse depuis un constructeur. Invocation d'une méthode de la surclasse depuis une méthode de même nom. Construction d'un type générique contraint	`public class B extends A {` `  private int n;` `  public B() {` `  super();` `  n=0;` `  }` `- - - - - - - - - -` `public void m() {` `  super.m();` `  n++;` `  }` `}` `< ? super T >`
switch	Instruction de choix. Voir *break*.	
synchronized	Marque une méthode en section critique --- Marque un bloc de code en section critique	`synchronized void insert(T t)` `synchronized (this) {` `  <bloc>` `}`
this	Accès à l'objet courant Accès à l'objet courant de la classe enveloppant en cas d'imbrication	`this.num=num;` --- `A.this.num=7;`
throw	Instruction de lancement d'une exception	`throw new FileException();`
throws	Déclaratif : indique qu'une méthode est susceptible de lever l'exception qui suit	`int await() throws` `  InterruptedException,` `  BrokenBarrierException`
transient	Marque un attribut pour qu'il ne fasse pas partie d'un flux de sérialisation	`public transient int num;`

true	Valeur booléenne vraie	`boolean b=true;`
try	ouverture d'un bloc de contrôle d'exception	`try {` `  ...` `}` `catch {` `}`
void	type void	`public void m();`
volatile	Empêche le compilateur de procéder à certaines optimisations lorsqu'il s'agit d'une variable partagée par plusieurs threads en lecture/écriture	`public volatile int num;`
while	itération générale	`while(n<100) {` `}`

Index alphabétique

Index alphabétique