Ecole Supérieure d'Ingénieurs de Luminy
Université de la Méditerranée

Support de cours pour l'apprentissage du langage JAVA
ESIL - GBM 2

Claudine Chaouiya
2003/2004

chaouiya@esil.univ-mrs.fr
http ://www.esil.univ-mrs.fr/~chaouiya

Chapitre 1

Introduction

`Java` est un langage de programmation orienté objets adapté à la distribution d'aplications sur Internet et qui s'intègre au Web. Nous verrons avant tout les différentes approches de programmation.

Avant d'écrire un programme, il faut poser le problème que ce programme devra résoudre. La formulation du problème influe sur l'écriture du programme, on parle de *paradigmes* de programmation. S'il est à peu près possible d'implémenter tous les paradigmes avec tout langage de programmation, chaque langage est quand même plus adapté à un paradigme donné. Ainsi, `C` est un langage dit procédural, `C++` et `Java` sont *orientés objets*.

1.1 Programmation procédurale

C'est l'approche que vous connaissez. Le langage `C` est adapté à la programmation procédurale. Dans ce style de programmation, l'accent porte sur l'algorithme mis en oeuvre. Chaque étape de l'algorithme peut elle même être découpée. C'est la programmation structurée qui indique qu'il faut isoler et clairement identifier les différentes opérations. On utilise ainsi des fonctions auxquelles on fournit des arguments et qui retournent des résultats. Ces fonctions peuvent éventuellement être rangées dans des bibliothèques, pour que l'on puisse les réutiliser. On retrouve ici les notions de modules (voir plus loin), et de compilation séparée vues l'an dernier.

Exemple du calcul du pgcd

```
int pgcd(int a, int b){
   int r;
   if (a<b){r=a;a=b;b=r;}
   do {
      r=a%b;
      a=b;
     b=r;
   } while (r!=0);
   return a;
}
...
```

```
void fonction1(....) {
   ....
   x=pgcd(1990,y);
   ....
}
```

Exemple de la pile

```
#include <stdio.h>
typedef struct elt {
  char info;
  struct elt *suiv;
} Maillon, *Pile;

Pile empiler(char c,Pile P) {
  Pile q;
  q=(Pile)malloc(sizeof(Maillon));
  q->info=c;
  q->suiv=P;
  return(q);
}
char depiler(Pile *P) {
  Pile q;
  char c;
  q=*P;
  *P=q->suiv;
  c=q->info;
  free(q);
  return(c);
}
int vide(Pile P){
  return (P==NULL);
}
int  main() {
  char c;
  Pile P=NULL;
  for (c='a';c<'e';c++) P=empiler(c,P);
  while(!vide(P)) {
    printf("%c \n",depiler(&P));
  }
}
```

1.2 Programmation modulaire

L'encapsulation des données est fondamentale dès que la taille des programmes est importante. Cela permet de se concentrer sur l'essentiel. Ainsi, l'ensemble des procédures ou fonctions et les données qu'elles manipulent sont regroupées dans un `module`. Un programme est alors constitué de différents modules, et la communication entre modules se fait à travers une interface, les détails d'implémentation de chaque module étant cachés aux autres modules. On a vu ce principe dans le cours de `C` avec les fichiers d'entêtes.

Les types abstraits de données (TAD) sont basés sur deux idées :
- L'encapsulation : c'est la définition d'un type et d'un ensemble d'opérations pour le manipuler à l'intérieur d'une unité syntaxique (un contenant : fichier, classe, module, package, etc.)
- La dissimulation de l'information : c'est le masquage des détails d'implémentation qui ne concernent pas l'utilisateur de l'abstraction.

L'encapsulation est utilisée pour la compilation séparée : on regroupe dans un fichier les parties d'un programme qui sont sémantiquement liées. Lors d'une modification, on n'a pas à recompiler tout le programme mais seulement le module qui a changé. Les TAD sont définis par :
- La **spécification** du type tel que vu de l'extérieur, elle définit comment utiliser le type (données et opérations accessibles). Elle décrit aussi l'interface, ce qui est exporté. Les programmes utilisant le TAD importent l'interface pour pouvoir utiliser le type.
- La **représentation** des objets de ce type (structure de donnée du type), elle décrit comment les objets du TAD sont construits.
- L'**implémentation** des opérations qui manipulent les objets de ce type. Il y a parfois deux opérations particulières : constructeur et destructeur qui spécifient comment créer un objet du type et quoi faire quand on veut le détruire.

Un bon TAD ne devrait exporter que des opérations, pas de données (champs). Eventuellement, les données sont accédées au travers d'opérations très simples appelées fonctions d'accès (*getters* et *setters*) pour donner une valeur ou extraire la valeur d'une donnée.

Remarque : Les classes du modèle à objets sont des TAD.

Exemple de TAD : Type Polynome - **Interface**

Constructeurs

ZERO → polynome

PLUS(Polynome, entier, reel) → Polynome

Autres générateurs

ADD(Polynome,Polynome) → Polynome

SOUSTR(Polynome,Polynome) → Polynome

MULT(Polynome,reel) → Polynome

PROD(Polynome,Polynome) → Polynome

DERIV(Polynome) → Polynome

Fonctions d'accès et d'interrogation

NUL(Polynome) → booleen

DEGRE(Polynome) → entier

COEF(Polynome, entier) → reel

VAL(Polynome,reel) → reel

Dans l'exemple de la pile, on aurait pu implémenter cette structure avec un tableau, ou encore une liste doublement chaînée... Si, pour un problème donné, on a besoin de cette structure de donnée, la façon dont elle est mise en oeuvre nous importe peu. Ce qui compte, c'est son comportement (caractérisations des fonctions `empiler`, `depiler`,...). Or, tel que le module `pile` a été écrit, on a le fichier `pile.h` suivant pour décrire l'interface. Notez que rien n'empêche l'utilisateur du module `pile` d'accéder directement aux données (champ `info` par exemple) :

```
// interface du module Pile de caracteres (pile.h)
#include <stdio.h>
typedef struct elt {
  char info;
  struct elt *suiv;
} Maillon, *Pile;
Pile empiler(char c,Pile P);
char depiler(Pile *P);
int vide(Pile P);
```

Exemple des formes géométriques

```
enum type{cercle,triangle,carre}
typedef struct{
    float l;
    point c;
    type f;
} forme;
float surface(forme x) {
   switch(x.f) {
   case cercle :
      return(PI*l*l);
      break;
   case triangle :
        .....
        break;
   case carre :
        .....
        break;
   }
}
```

L'ajout ou la suppression d'une nouvelle forme amènent à reprendre l'ensemble des fonctions et à les adapter. De plus, un cercle a un rayon, un carré un côté, un rectangle une longueur et une largeur...

1.3 Programmation orientée objets

Dans l'exemple des piles de caractères, comment faire lorsqu'on veut utiliser des piles d'entiers, et non plus de caractères ? Il faudrait réécrire un module (alors que, fondamentalement, une pile a toujours le même comportement, que ce soit une pile d'entiers, de caractères, ou de n'importe quoi). L'approche orientée objets permet de

résoudre ce problème. La POO permet également de résoudre de façon élégante le problème posé par le petit exemple des formes géométriques, et ceci grâce à la notion d'héritage que l'on verra au chapitre 4.

Les objets sont au coeur de la POO... Un objet a deux caractéristiques : son état courant et son comportement. Dans l'exemple de la pile de caractères, l'état est représenté par le contenu de la pile, le comportement par les fonctions que l'on peut appliquer à la pile.

L'état d'un objet est représenté par des **attributs**, son comportement par des **méthodes**. On ne peut modifier l'état d'un objet que par l'utilisation de ses méthodes ; l'encapsulation des données permet de cacher les détails d'implémentation d'un objet. Ceci dit, cette vue idéale est trop rigide pour des applications informatiques. Ainsi, en POO, on pourra avoir des attributs privés (modifiables uniquement via une méthode appropriée) ou publics (accessibles directement par l'extérieur).

En POO, on utilise le concept de **classe**, qui permet de regrouper des objets de même nature. Par exemple, une pile de caractères n'est qu'une pile de caractères parmi d'autres. En POO, on dira que notre pile particulière est une **instance** de la classe des objets connus sous le nom de piles de caractères. Toutes les piles de caractères ont des caractéristiques communes mais peuvent être dans un état différent.

- **Une classe est un moule (on dira prototype) qui permet de définir les attributs (ou champs) et méthodes communs à tous les objets de cette classe.**
- **Les types abstraits de données dans le modèle à objets s'appellent des classes.**
- **Les instances des classes sont des objets (ou instances).**
- **Les opérations d'une classe sont ses méthodes.**

attributs	surface, couleur...
méthodes	afficher, detruire, changerCouleur...

TAB. 1.1 – La classe Forme géométrique

L'héritage Une notion fondamentale en POO est la notion d'héritage. Si l'on reprend notre exemple des formes géométriques, une façon de procéder est de définir la classe `FormeGeometrique` avec les attributs et comportements communs à **toutes** les formes géométriques. La sous-classe `Cercle` hérite alors de la classe `FormeGeometrique` et a ses propres spécificités.

1.4 Java, qu'est-ce-que c'est ?

Java est composé de 4 éléments :
- un langage de programmation
- une machine virtuelle (JVM)

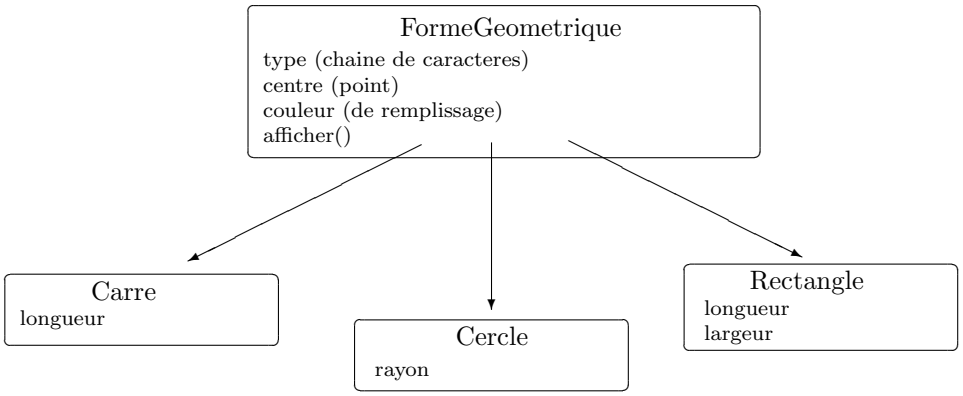

FIG. 1.1 – Classe mère et ses sous-classes

- un ensemble de classes standards réparties dans différentes API (Application Programming Interface)
- un ensemble d'outils (jdb, javadoc,...).

Le langage Java est connu et est très souvent associé aux `applets` que l'on peut voir sur certaines pages *WEB*, généralement de jolies applications graphiques... Il ne faut pas réduire Java à un langage dédié aux pages *WEB*. Il s'agit bien d'un langage à part entière, qui vous permettra de réaliser de vraies applications !

Dans ce qui suit, on reprend les adjectifs associés à Java par leurs concepteurs (voir http ://java.sun.com/docs/white/langenv/Intro.doc2.html).

1.4.1 Simple et familier

Java est simple et familier car il n'utilise qu'un nombre restreint de concepts nouveaux. Sa syntaxe est très proche du langage `C`. Toutes les embûches sur lesquelles butte le programmeur en `C` ou `C++` sont éliminées, par exemple :

- seul existe le concept de classe, plus de `struct`, `union` et `enum`,
- plus de pointeurs et leur manipulation (avec parfois des pointeurs adressant des emplacements non maîtrisés !),
- plus de préoccupation de gestion de la mémoire, Java a un "ramasse-miettes" (*garbage collector*) qui se charge (presque) de restituer au système les zones mémoires inaccessibles,
- plus de préprocesseur :
  - comme Java est indépendant de la plateforme (voir plus loin), il n'est plus nécessaire d'écrire du code dépendant de la plateforme,
  - les fichiers d'entête `.h` n'ont plus lieu d'être, le code produit contient toutes les informations sur les types de données manipulés.
  - ...

1.4.2 Orienté objets

Enfin, Java est orienté objets, car un programme Java est complètement centré sur les objets. Mis à part les types primitifs et les tableaux, en Java **tout est objet**,

autrement dit, toutes les classes dérivent de `java.lang.Object`. L'héritage en Java est simple, mais il existe l'héritage multiple pour les *interfaces*. Les objets se manipulent via des *références*. Enfin une librairie standard fournit plus de 500 classes au programmeur (l'API).

1.4.3 Interprété, portable et indépendant des plateformes

C'est justement pour une question de portabilité que les programmes Java ne sont pas compilés en code machine. Le compilateur génère un code appelé *bytecode*, code intermédiaire qui est ensuite interprété par la JVM (cf figure 1.2). De plus il n'y a pas de phase d'édition de liens ; les classes sont chargées dynamiquement en fonction des besoins, au cours de l'exécution, de manière incrémentale. La taille des types primitifs est indépendante de la plateforme.

La *Java Virtual Machine* (JVM) est présente sur Unix, Windows, Mac, Netscape, Internet Explorer, ...

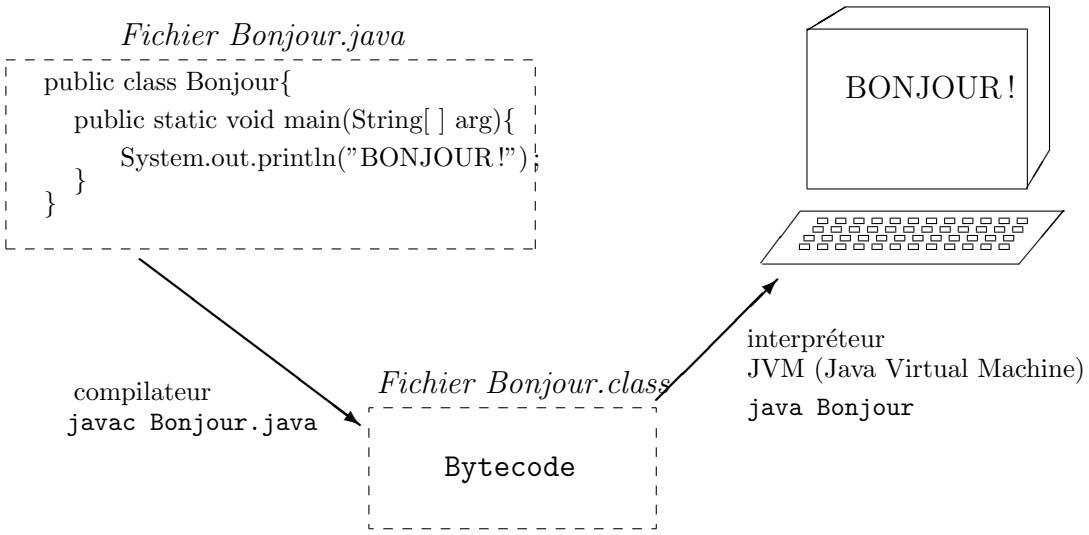

FIG. 1.2 – Compilation et exécution d'un programme Java

1.4.4 Robuste et sûr

Il s'agit d'un langage fortement typé, beaucoup d'erreurs sont donc éliminées à la compilation. Le *ramasse-miettes* assure une bonne gestion de la mémoire et il n'y a pas d'accès direct à la mémoire. Le mécanisme des levées d'exceptions permet une bonne gestion des erreurs d'exécution. Le compilateur est contraignant.

La sécurité est prise en charge par l'interpréteur avec trois niveaux :

- *Verifier* qui vérifie le *code byte*
- *Class Loader* qui est responsable du chargement des classes
- *Security Manager* qui vérifie les accès ressources

1.4.5 Dynamique et *multithread*

Un programme Java est constitué de plusieurs classes. Lorsqu'un classe inconnue dans un programme est requise par celui-ci, la JVM la recherche et la charge dynamiquement.

Un *thread* (appelé aussi "processus léger") est une partie de code s'exécutant en concurrence avec d'autres threads dans un même processus. Cela permet donc à un programme unique d'effectuer plusieurs tâches "simultanément". La notion de thread est intégrée au langage et aux API.

1.4.6 Distribué

1.5 Environnement de développement

Le JDK (Java Developer Kit) contient l'ensemble des librairies standards de java (`java.lang, java.util, java.awt,` ...), le compilateur (`javac`), un interpréteur d'applets (`appletviewer`), un interpréteur (`java`), un générateur de documentation (`javadoc`) et quelques autres outils... C'est le minimum pour développer des applications en Java.

Par ailleurs, il existe de nombreux environnement de développement. On en cite deux ci-dessous (cf. http ://java.developpez.com/outils/edi/) pour plus de détails...

JBuilder de Bordland est très bien placé parmi les environnements professionnels pour le développement d'applications Java (http ://www.borland.fr/jbuilder/index.html).

Anciennement connu sous le nom de Forte for Java, **Sun ONE** (Open Net Environnement) Studio s'appuie sur le noyau de NetBeans, projet initié par Sun (http ://developers.sun.com/prodtech/devtools/).

GNU/Emacs est un éditeur polyvalent. Pour l'édition du code, il possède de nombreux "modes" : C, C++, HTML, Java, qui vont adapter le fonctionnement d'Emacs. Il dispose d'un grand nombre de fonctions, couramment utilisées (en programmation) : recherche/remplacement (supporte les expressions régulières), indentation automatique du code, coloration syntaxique, (Re)définition des raccourcis claviers, auto-complétion, gestion du multifenêtrage, etc... (plus de 1600 fonctions assurées). Nous choisirons de travailler avec cet éditeur et de compiler et exécuter en ligne de commande.

Chapitre 2

Syntaxe de base

Dans ce chapitre, on introduit la syntaxe de base du langage. Vous verrez qu'elle est assez proche de celle du langage `C`, avec quelques ajouts et différences. Ce qui change radicalement, c'est l'approche orientée objets, ce sera l'objet du chapitre suivant.

2.1 Unités lexicales

Le compilateur Java reconnait cinq types d'unités lexicales : les identificateurs, les mots réservés, les littéraux, les opérateurs et les séparateurs.

2.1.1 Jeu de caractères

Java utilise le jeu de caractères **Unicode**. Les caractères sont codés sur 16 bits (au lieu de 7 pour le code ASCII). Ce code a été introduit pour tenir compte de tous (ou presque !) les alphabets.

2.1.2 Commentaires

Java reconnait trois types de commentaires :
- les commentaires sur une ligne : tous les caractères suivants `//...` jusqu'à la fin de la ligne sont ignorés
- les commentaires multilignes : tous les caractères entre `/* ... et...*/` sont ignorés
- les commentaires de documentation : quand ils sont placés juste avant une déclaration, les caractères entre `/** ...et...*/` sont inclus dans une documentation générée automatiquement par l'utilitaire `javadoc`.

2.1.3 Identificateurs

Les identificateurs ne peuvent commencer que par une lettre, un souligné ('_') ou un dollar ('$'). Les caractères suivants peuvent être des lettres ou des chiffres ou tout caractère du jeu Unicode de code supérieur à H00C0.

Exemples : `x` `Bidule` `_Bidule` `$Bidule`

Note : on convient de réserver des noms commençant par une majuscule aux classes, les noms composés sont sous la forme "NomComposé" ou bien "nomComposé", et de façon générale, on conseille de nommer les variables et méthodes de façon parlante.

2.1.4 Mots réservés

Les identificateurs du tableau suivant sont des mots clés du langage et sont à ce titre des mots réservés que vous ne pouvez en aucun cas utiliser comme identificateurs.

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

* indique un mot clé qui n'est pas utilisé dans les versions actuelles

Il y a encore trois mots réservés du langage qui ne sont pas des mots clés mais des littéraux :`true` `false` et `null`.

2.1.5 Types primitifs simples

Toute variable ou expression a un type qui permet de définir l'ensemble des valeurs et des actions légales. Java a un petit nombre de types prédéfinis appelés aussi types primitifs, le mécanisme des classes et interfaces permet ensuite de définir d'autres types.

Java a deux sortes de types : les types simples, atomiques (entiers, réels, booléens et caractères) et les types composites (tableaux, classes et interfaces).

Caractères

Le type caractère est `char`. Il est représenté sur 16 bits (jeu de caractères Unicode).

Booléens

Le type booléen est `boolean`. Les deux seules valeurs qu'il peut prendre sont `true` et `false`. Il s'agit du type retourné par les opérateurs relationnels (cf.2.3.4).

Entiers

Ils sont très similaires à ceux de `C`, sinon qu'ils sont indépendants de la plateforme. Les 4 types d'entiers sont :
- `byte` $\Longrightarrow$ entier sur 8 bits (complément à 2)
- `short` $\Longrightarrow$ entier sur 16 bits (complément à 2)
- `int` $\Longrightarrow$ entier sur 32 bits (complément à 2)
- `long` $\Longrightarrow$ entier sur 64 bits (complément à 2)

Réels

Il n'y a que deux types de réels en Java :
- `float`$\Longrightarrow$ représenté sur 32 bits
- `double`$\Longrightarrow$ représenté sur 64 bits

2.1.6 Constantes littérales

Constantes booléennes

On l'a vu, les seules possibles sont `true` et `false`

Constantes caractères

Elles sont constituées d'un caractère ou une séquence d'échappement entre des guillemets simples :

`'a'`, `'b'`,...

`'\''`, `'\"'`, `'\\'`

`'\n'` nouvelle ligne

`'\t'` tabulation

Constantes entières

Elles peuvent s'écrire
- en notation décimale : `123`, `-123`
- en notation octale avec un zéro en première position : `0123`

– en notation hexadécimale, avec les caractères `0x` ou `0X` au début : `0xDead`, `0XbaBA`

Le type d'une constante est toujours `int`, pour préciser qu'une constante est de type `long`, la faire suivre de `l` ou `L` (par exemple, `1L`, `0x7FFL`,...).

Constantes réelles

Elles se présentent sous la forme d'une partie entière suivie d'un point (`.`), suivi d'une partie décimale, d'un exposant et un suffixe de type. L'exposant est un `E` ou `e` suivi d'un entier.

`3.1415`, `3.1E12`, `.1E-4`

`2.0d` (ou `2.0D`) est un réel `double`

`2.0f` (ou `2.0F`) est un réel `float`

Constantes chaînes de caractères

Une chaîne de caractères est une suite de caractères délimitée par des guillemets. Attention, en Java les chaînes de caractères sont des **objets** et forment à ce titre un type à part entière, il ne s'agit pas d'un tableau de caractères. On verra la classe `String` plus loin (6)

2.2 Les variables

Toute variable doit avoir été déclarée. La déclaration d'une variable consiste à lui donner un nom et un type, éventuellement une valeur initiale et un qualificatif. La déclaration alloue la place mémoire nécessaire au stockage, la taille dépendant du type. Java distingue différentes natures de variables (pas nécessairement incompatibles) :

– les variables d'instances,
– les variables de classe,
– les paramètres de méthodes,
– les paramètres de constructeurs,
– les variables de type exception,
– les variables locales.

Le qualificatif `final` permet d'interdire la modification ultérieure d'une variable.

La déclaration des variables locales se fait dans le bloc où elles sont utilisées, elles sont alors visibles à partir de leur déclaration jusqu'à la fin du bloc.

```
...
for(int i=0;i<10;i++) {
       ...
       // i est visible dans ce bloc
}
...
```

Il n'y a pas vraiment de notion de variable globale en Java. Toute déclaration de variable se trouve nécessairement dans la déclaration d'un classe. Seules les variables qualifiées de `static` (dites de classe) et `public` peuvent ressembler aux variables globales de `C` et `C++`.

2.3 Expressions et opérateurs

Les expressions en Java ressemblent beaucoup à celles que l'on écrit en `C`. Elles sont composées de constantes, variables et opérateurs, assemblés "correctement".

L'essentiel d'un programme consiste à évaluer des expressions pour produire ce qu'on appelle des effets de bord, ou pour calculer des valeurs. Une expression avec effet de bord est une expression qui, lorsqu'elle est évaluée, produit un changement de l'état du système. Par exemple, l'affectation est une expression qui a pour effet de bord la modification du contenu de l'opérande gauche.

Le résultat de l'évaluation d'une expression est soit une valeur, soit une variable (une *lvalue*, (pour *left* ou *location value*) une adresse, membre gauche d'une affectation, à opposer à une *rvalue* qui est une valeur), soit `void`. Ce dernier cas apparaît lors de l'invocation d'une méthode qui ne retourne rien.

2.3.1 Priorité des opérateurs et ordre d'évaluation

Les opérateurs, de la priorité la plus forte à la plus faible sont donnés dans le tableau 2.3. Introduire des parenthèses rend souvent l'expression plus lisible, et dans tous les cas lève les ambiguités. L'ordre d'évaluation est important. A part dans le cas des opérateurs && || et ? : les opérandes de chaque opération sont complètement évaluées avant d'effectuer l'opération. Java garantit également que les opérandes sont évaluées de gauche à droite (dans l'expression $x + y$, x est évalué avant y). Ceci est important lorsqu'on a des expressions avec effets de bord. Les opérateurs de Java sont détaillés plus loin (2.3.4).

2.3.2 Type d'une expression

Toute expression a un type, connu dès la compilation. Ce type est déterminé par celui des opérandes et par la sémantique des opérateurs. Si le type d'une expression n'est pas approprié, cela conduit à une erreur de compilation. Par exemple, si l'expression dans une structure de test `if` n'est pas de type `boolean`, le compilateur produit une erreur. Dans d'autres cas, plutôt que de demander au programmeur d'indiquer une conversion de type explicite, Java produit une conversion implicite du type de l'expression en un type qui convient au contexte. Il existe ainsi plusieurs sortes de conversions implicites.

Les conversions d'un type primitif à un type primitif plus *large* pour lesquelles on ne perd pas d'information :

- `byte` → `short`,`int`,`long`,`float`, ou `double`
- `short` → `int`, `long`, `float`, ou `double`

- `char` → `int`, `long`, `float`, ou `double`
- `int` → `long`, `float`, ou `double`
- `long` → `float` ou `double`
- `float` → `double`

Les conversions d'un type primitif à un type primitif plus *restreint* pour lesquelles on perd de l'information, ou de la précision :

- `byte` → `char`
- `short` → `byte` ou `char`
- `char` → `byte` ou `short`
- `int` → `byte`, `short`, ou `char`
- `long` → `byte`, `short`, `char`, `int`
- `float` → `byte`, `short`, `char`, `int`,ou `long`
- `double` → `byte`, `short`, `char`, `int`,`long`, `float`

Pour les conversions sur les références, nous verrons cela plus loin...

2.3.3 Erreur d'évaluation d'une expression

L'évaluation d'une expression peut conduire à une erreur, dans ce cas Java *lance* une *exception* qui précise la nature de l'erreur (voir chapitre 7).

`OutOfMemoryError`	espace mémoire requis insuffisant
`ArrayNegativeSizeException`	une dimension de tableau est négative
`NullPointerException`	valeur de référence à `null`
`IndexOutOfBoundsException`	valeur d'indice de tabelau hors des bornes
`ClassCastException`	opŕation de *cast* interdite
`ArithmeticException`	division par zéro
`ArrayStoreException`	affectation à un élément de tableau d'une référence de type incompatible
	des exceptions générées par l'invocation d'une méthode
	des exceptions générées par les constructeurs
	bien d'autres !...

TAB. 2.1 – Exemples de levées d'exceptions

2.3.4 Opérateurs

Le tableau 2.2 présente tous les opérateurs du langage, avec leur ordre d'évaluation et leur sémantique. Il manque dans ce tableau les opérateurs :

- les opérateurs d'affectation (+=, -=, *=, ...) dont l'évaluation est faite de droite à gauche,
- les opérateurs de manipulation de bits :

  & (ET bit à bit), | (OU bit à bit), ∧ (OU exclusif bit à bit) et ∼ (complémentation bit à bit),

<< >> >>> de décalage des bits ;

– l'opérateur ternaire conditionnel (si-alors-sinon) : `cond ?expr1 :expr2`

Opérateur(s)	Ordre	Type	Description
=	D/G	variable	affectation
* / %	G/D	arithmétique	multiplication, division, reste
+ -	G/D	arithmétique	addition, soustration
+ -	G/D	arithmétique	plus, moins unaires
++ --	G/D	arithmétique	pré et post incrément, décrément[1]
< > ≤ ≥	G/D	arithmétique	comparaison arithmétique
== !=	G/D	objet, type primitif	comparaison égal et différent
+	G/D	chaînes de caractères	concaténation
!	D/G	booléen	non booléen
& ∧ |	G/D	booléen	ET, OU exclusif, OU (les 2 opérandes sont évaluées)
&& ||	G/D	booléens	ET, OU conditionnels (l'opérande de droite n'est pas nécessairement évaluée)

TAB. 2.2 – Opérateurs de Java

2.4 Structures de contrôle

2.4.1 Instructions et blocs d'instructions

Un programme Java est constitué de déclarations de classes dans lesquelles figurent des méthodes. Ces dernières sont construites à l'aide d'instructions combinées entre elles par des structures de contrôle.

Une instruction est une expression suivie d'un point virgule. Les instructions composées ou blocs d'instructions sont des suites d'instructions simples ou composées délimitées par des accolades { et }. L'accolade fermante n'est pas suivie de point virgule.

Exemple :

```
{ int i;
  i=4;
  System.out.println("coucou ! ");
  System.out.println("i vaut "+i);
}
```

[1] la valeur d'une expression de post-incrément est la valeur de l'opérande et a pour effet de bord le stockage de la valeur de l'opérande incrémentée de 1, la valeur d'une expression de pré-incrément est la valeur de l'opérande incrémentée de 1 et a pour effet de bord le stockage de cette valeur. C'est similaire pour le décrément.

opérateurs postfixes	[] . (params) expr++ expr–
opérateurs unaires	++expr –expr +expr -expr ∼ !
création ou *cast*	new (type)expr
opérateurs multiplicatifs	* / %
opérateurs additifs	+ -
opérateurs de *shift*	<< >> >>>
opérateurs relationnels	< > <= >= instanceof
opérateurs d'égalité	== !=
ET bit à bit	&
OU exclusif bit à bit	∧
OU inclusif bit à bit	|
ET logique	&&
OU logique	||
opérateur conditionnel	? :
affectations	= += -= *= /= %= &= ∧= |= <<= >>= >>>=

TAB. 2.3 – Opérateurs dans l'ordre décroissant de priorité

L'objet de cette section est de passer brièvement en revue toutes les structures de contrôle (vous les connaissez déjà).

2.4.2 Instruction conditionnelle : `if`

Elle permet d'exécution des instructions de manière sélective, en fonction du résultat d'un test.

```
if (expression) instruction1
if (expression) instruction1 else instruction2
```

2.4.3 Etude de cas : `switch`

Elle permet de choisir un bloc d'instruction selon la valeur d'une expression **entière** :

```
switch (expression) {
   case cste1 :
              instruction1
   case cste2 :
              instruction2
   ...
   case csteN :
              instructionN
   default :
```

```
            instructionDefaut
}
```

Attention, si la valeur de `expression` vaut `csteI`, `instructionI` sera exécutée ainsi que toutes les suivantes (`instructionI+1`...`instructionDefaut`) sauf si une instruction `break` a été rencontrée. L'exemple suivant illustre le fonctionnement de cette structure de contrôle :

Exemple :

```
char c;
...
switch (c) {
   case '1':
   case '2':
   case '3':       // notez l'absence d'intruction
   case '5':
   case '7':
       System.out.println(c+"est un nombre premier\n");
       break;     // notez l'instruction break
   case '6':
       System.out.println(c+"est un multiple de 3\n");
                   // notez l'absecnce de break
   case '4':
   case '8':
       System.out.println(c+"est un multiple de 2\n");
       break;
   case '9':
       System.out.println(c+"est un multiple de 3\n");
       break;
   default :
       System.out.println(c+"n\'est pas un chiffre non nul\n");
}
...
```

2.4.4 Itérations : `while`, `do...while` et `for`

La structure de contrôle `while` évalue une condition et exécute l'instruction tant que cette condition reste vraie.

```
while (condition)
  instruction
```

Exemple :

```
int i=10;
while (i>=0) {
       System.out.println(i);
       i=i-1;
}
```

L'instruction `do...while` est une variante de la précédente. Une itération est toujours exécutée. Il faut la traduire en français par *Faire... tant que*. Attention de ne pas confondre avec la structure *répéter...jusqu'à ce que*!

```
do
  instruction
while (condition)
```

Exemple :

```
int i=-1;
do {
       System.out.println(i);
       i=i-1;
} while (i>=0);
```

Enfin, l'instruction `for` qui comporte une initialisation, une condition d'arrêt, et une ou des instructions de fin de boucle :

```
for (instruction1;condition_de_poursuite;instruction2) instruction3
```

est équivalente à :

```
instruction1;
while (condition_de_poursuite) {
       instruction3
       instruction2
}
```

La virgule (,) est utilisée pour combiner plusieurs initialisations et plusieurs instructions de fin de boucle.

2.4.5 Etiquettes, `break`, `continue` et `return`

Toute instruction peut être étiquetée.

```
label : instruction
```

L'instruction `break` déjà vue avec le `switch` est utilisée aussi dans les structures de boucle et permet la sortie immédiate de la boucle, sans tenir compte des conditions d'arrêt de cette dernière. Une variante permet d'associer une étiquette à l'instruction `break`.

```
label : instruction1
while(...){
...
break label;
...
}
```

Ceci dit, l'usage des étiquettes et du `break` est fortement déconseillé, ce n'est pas élégant, cela nuit à la lisibilité du programme, c'est contraire aux principes de la programmation structurée ! La plupart du temps, on peut s'en passer.

L'instruction `continue` apparaît dans les structures de boucles. Elle produit l'abandon de l'itération courante et, si la condition d'arrêt n'est pas satisfaite, le démarrage de l'itération suivante.

L'instruction `return` quant à elle est indispensable ! Elle provoque l'abandon de la fonction en cours et le retour à la fonction appelante. Quand elle est suivie d'une expression, le résultat de cette expression est la valeur que la fonction appelée renvoie à la fonction appelante. Mais attention, il est déconseillé de placer une instruction `return` dans le corps d'une boucle, cela signifie que vous n'avez probablement pas bien écrit la condition de sortie de la boucle !

2.5 Structure d'un programme autonome Java

Un programme Java est constitué d'une ou plusieurs classes. Parmi ces classes, il doit y en avoir au moins une qui contienne la méthode statique et publique `main` qui est le point d'entrée de l'exécution du programme. Voici un exemple (l'inévitable !) :

Exemple :

```
// Fichier Bonjour.java
public class Bonjour {
   public static void main(String[] arg) {
       System.out.println("Bonjour !\n");
   }
}
```

On a défini une classe `Bonjour` qui ne possède qu'une seule méthode. La méthode `main` **doit** être déclarée `static` et `public` pour pouvoir être invoquée par l'interpréteur Java. L'argument `arg` est un tableau de chaînes de caractères qui correspond aux arguments de la ligne de commande lors du lancement du programme.

Avant tout, il faut compiler ce programme avec la commande `javac` :

```
javac Bonjour.java
```

La compilation traduit le code source en *byte code*. Le compilateur produit autant de fichiers que de classes présentes dans le fichier source. Les fichiers compilés ont l'extension `.class`.

Enfin, pour exécuter le programme, il faut utiliser l'interpréteur de code Java et lui fournir le nom de la classe `public` que l'on veut utiliser comme point d'entrée :

```
java Bonjour
```

Chapitre 3

Classes et Objets

En `C` on utilise des *structures* pour crér des TAD (Types Abstraits de Données), ou structures de données complexes. Dans les langages orientés objets, on utilise le concept de *classes*. Elle permettent de définir de nouveaux types de données qui se comportent comme des types prédéfinis et dont les détails d'implémentation sont cachés aux utilisateurs de ces classes. Seule l'*interface* fournie par le concepteur peut être utilisée.

Un *objet* est une *instance* d'une classe (qui peut être vue comme un moule). Les objets communiquent entre eux par des messages qui sont évalués par des méthodes. Ces messages évalués par des méthodes de l'objet, induisent des modification de son état ou de son comportement. Les objets vivent en famille, et peuvent donc hériter des caractéristiques de leurs ancêtres, en affinant (spécialisant) ces caractéristiques. Un objet est caractériqé par :

- un ensemble d'attributs, typés et nommés représentant des propriétés statiques. L'ensemble des valeurs des attributs consitue l'état de l'objet,
- un ensemble de méthodes, définissant son comportement et ses réactions à des stimulations externes. Ces méthodes implémentent les algorithmes que l'on peut invoquer sur ces objets,

En Java, on ne peut accéder à un objet que par une *référence* vers celui-ci. Une référence est une sorte de pointeur vers la structure de données, avec la différence qu'il est interdit de manipuler les références comme les pointeurs en `C` ou `C++`. On ne peut pas connaître la valeur d'une référence, ni effectuer d'opérations arithmétiques. La seule manipulation possible consiste à changer la valeur de la référence pour qu'elle "fasse référence" à un autre objet.

Une classe est un moule d'objets, elle en décrit la partie privée (structure de données interne ou attributs et corps des méthodes), et la partie publique (nom et paramètres des méthodes). C'est un générateur d'objets, on peut ainsi créer un ensemble d'objets rentrant dans ce moule.

3.1 Déclaration des classes

Basiquement, une classe définit :

- les structures de données associées aux objet de la classe, les variables désignant ces données sont appelées *champs* ou *attributs*,
- les services ou comportements associés aux objets de la classe qui sont les *méthodes*, définies dans la classe.

3.1.1 Champs ou attributs

Java possède trois mots clés pour l'encapsulation des données (les droits d'accès) : `public`, `private` et `protected`. Nous les reverrons plus en détail, mais retenez ici que les données et méthodes déclarées `public` sont accessibles par toutes les classes. Inversement, les données et méthodes déclarées `private` ne sont accessibles que par les méthodes de cette classe. Enfin, le mot clé `protected` institue une notion de "famille". Supposons que nous voulions déclarer une structure de données `Date` constituée de trois entiers codant le jour, le mois et l'année :

```
class Date{
   private int mois;
   private int jour;
   private int annee;
   ...
}
```

Les données `mois`, `jour` et `année` ont été déclarées privées. Elles ne seront accessibles que par des méthodes définies de la classe `Date` dans la section qui suit.

3.1.2 Méthodes

Elles sont définies par un identificateur, des paramètres formels, un type de retour, un corps et éventuellement un qualificatif (comme pour les champs) `public`, `private` ou `protected`.

```
class Date{
   private int mois;
   private int jour;
   private int annee;
   ...
   public void affecter(int m, int j, int a) {
       mois=m;  jour=j; annee=a;
   }
   public int quelJour(){return jour;}
   public int quelMois(){return mois;}
   public int quelleAnnee(){return annee;}
   public void imprimer(){
     System.out.println(jour+"/"+mois+"/"+annee);
   }
}
```

La méthode `affecter` fait partie de la classe `Date`, il lui est donc permis d'accéder à ses champs privés. Et cette méthode, puisqu'elle est déclarée `public`, permet de modifier les champs d'un objet de la classe `Date`. Les méthodes publiques d'une classe constituent ce que l'on appelle son *interface publique*.

Contrairement au langage `C++` la définition effective des méthodes de la classe doit se faire dans la définition de la classe.

Une méthode est un message envoyé à un objet. Ainsi, pour afficher la date contenue dans l'objet `d`, on lui envoie le message `imprimer` :

```
d.imprimer();
```

De telles méthodes sont appelées *méthodes d'instances*, elles sont évoquées via un objet. Nous verrons plus loin qu'il existe des *méthodes de classes*. La méthode `imprimer` n'est utilisable que parce qu'elle fait partie des méthodes publiques. Par contre, il ne sera pas possible d'accéder aux champs `d.jour`, `d.mois` et `d.annee` car ce sont des données *privées*.

3.1.3 Création d'objets

Une fois la classe déclarée, pour pouvoir utiliser un objet de cette classe, il faut définir une *instance* de cette classe. La déclaration suivante ne fait que définir une référence vers un objet éventuel de la classe `Date` :

```
Date d;
```

La variable `d` représente une référence vers un objet de type `Date` qui doit être instancié (créé) explicitement avec le mot clé `new` et le constructeur (cf. section3.1.4) de la classe `Date` :

```
Date d;          //déclaration de la référence d
d = new Date();  // instanciation de l'objet référencé par d
```

3.1.4 Constructeurs

On a dit que pour définir un objet d'une classe, il fallait faire appel à son constructeur. En l'absence de constructeur(s) explicite(s), un constructeur implicite, sans argument, est invoqué par défaut.

Lorsque l'on veut définir un objet, il est souvent utile de pouvoir initialiser cet objet. Dans notre exemple de la classe `Date`, il est possible d'utiliser la méthode `affecter` pour donner une valeur aux champs `d.jour`, `d.mois` et `d.annee`.

```
Date aujourdhui=new Date();
aujourdhui.affecter(8,25,1961);
```

Mais ce n'est pas très agréable. Le constructeur est une méthode spécifique qui est automatiquement appelée lors de la création d'un objet. Elle a la particularité de porter le même nom que la classe, d'être publique et n'a pas de valeur de retour.

```
class Date {
          ...
       public Date(int j, int m, int a) {
              jour=j; mois=m; annee=a;}
       ...
}
```

Maintenant, pour créer un objet de type `Date` il faudra fournir impérativement le jour, le mois et l'année. On peut contourner ce problème en fournissant plusieurs constructeurs :

```
class Date {
          ...
       public Date(int j, int m, int a) {
              jour=j; mois=m; annee=a;}
       public Date(int j, int m) {
              jour=j; mois=m; annee=2000;}
       public Date(int j) {
              jour=j; mois=1; annee=2000;}
       public Date() {
              jour=1; mois=1; annee=2000;}
       ...
}
```

3.1.5 Destructeurs

En général, en Java, on n'a pas à se soucier de la restitution de l'espace mémoire occupé par un objet qui n'est plus référencé. On a déjà évoqué le "ramasse-miettes" (*garbage collector*) qui est un système de récupération de mémoire automatique. Par défaut, ce système tourne en arrière-plan pendant l'exécution de vos programmes. Il repère les objets qui ne sont plus référencés, et libère l'espace en mémoire alloué à ceux-ci. Vous pouvez désactiver le ramasse-miettes (option `-noasyngc` sur la ligne de commande de lancement de la JVM).

Selon les applications, un objet peut bloquer d'autres types de ressources que la mémoire (descripteur de fichiers, socket, ...), il est alors bon d'utiliser un destructeur pour libérer ces ressources. De plus, vous pouvez ne pas vouloir attendre que le ramasse-miettes libère des ressources critiques. Il existe une méthode spécifique `finalize` qui est un destructeur et redéfinit la méthode `protected void finalize` de la classe `Object`. Une classe peut donc implémenter une méthode `finalize` qui est déclarée de la façon suivante :

```
protected void finalize() throws Throwable {
   super.finalize();
   ...
}
```

Ce code s'éclaircira plus tard, avec les notions d'héritage et d'exceptions.

3.2 Définitions de champs

3.2.1 Champs de classe

Si l'on définit trois objets de type `Date`, chacun aura évidemment son propre jeu de valeurs pour les champs `jour`, `mois`, `annee`. De tels champs sont appelés variables (ou attributs) d'instances. Il est des cas où il est souhaitable d'avoir une donnée commune à tous les objets d'une même classe. Un champ d'une classe est dit *static* (ou de classe) ; il n'y a qu'un seul exemplaire de ce champ pour tous les objets de cette classe. Ce champ existe même s'il n'y a aucune instance de la classe.

Exemple :

```
class Date{
    private int mois;
    private int jour;
    private int annee;
    public static int nbDate=0;

    public Date(int j, int m, int a){
       mois=m;  jour=j; annee=a;
       nbDate++;
    }
    public int quelJour(){return jour;}
    public int quelMois(){return mois;}
    public int quelleAnnee(){return annee;}
    public void imprimer(){
       System.out.println(jour+"/"+mois+"/"+annee);
    }
}

class Programme{
    public static void main(String[] arg){
       Date aujourdhui=new Date(25,9,2000);
       Date noel=new Date(25,12,2000);
       aujourdhui.imprimer();
       noel.imprimer();
       System.out.println(noel.nbDate);
       System.out.println(Date.nbDate);
    }
}
```

Voici le résultat obtenu :

```
chaouiya@pccc:~/coursJava/Notes_cours$ javac Programme.java
chaouiya@pccc:~/coursJava/Notes_cours$ java Programme
```

```
25/9/2000

25/12/2000

2
2
```

Initialisation des champs de classe

Les champs `static` sont initialisés une fois lors du chargement de la classe qui les contient. Une erreur de compilation se produit lorsque :

- un champ de classe est initialisé relativement à un champ de classe défini plus loin

```
class X{
  static int x = y+1; // erreur, y est declare apres x
  static int y =0;
  static int z=z+1;   // erreur
}
```

- un champ de classe est initialisé relativement à un champ d'instance

```
class X{
   public int x=120;
   static int y=x+10; // erreur, x variable d'instance
}
```

Initialisation des champs d'instance

Les champs d'instance sont initialisés lors de l'instanciation (à la création) des objets de la classe. Contrairement aux champs de classe, chaque instanciation provoque l'intialisation des champs de l'objet créé. Une erreur de compilation se produit si un champ d'instance est initialisé par référence à un champ d'instance défini plus loin. On peut utiliser les valeurs des champs de classe pour initialiser des champs d'instance.

3.2.2 Mot clé `this`

Il désigne l'objet sur lequel la méthode est invoquée. On peut par exemple réécrire la méthode `affecter` comme suit :

```
public void affecter(int m, int j, int a) {
    this.mois=m;  this.jour=j; this.annee=a;
}
```

Dans l'exemple qui suit, l'intérêt du mot clé `this` est certainement mieux illustré. On crée une liste chaînée de tous les objets de type `Date` qui ont été instanciés :

```
class Date{
```

```
   private int mois;
   private int jour;
   private int annee;
   private Date suivant;
   public static Date listeDates=null;
   public Date(int j, int m, int a){
       jour=j; mois=m; annee=a;
       suivant=listeDates;
       listeDates=this;
   }
    public void imprimer(){
       System.out.println(jour+"/"+mois+"/"+annee);
    }
}
class Test {
   public static void main(String[] arg){
       Date noel=new Date(25,12,2000);
       Date aujourdhui=new Date(25,9,2000);
       for (Date d=Date.listeDates; d!=null; d=d.suivant) d.imprimer();
   }
}
```

3.2.3 Champs `final`

Un champ peut être déclaré `final` pour indiquer qu'il ne peut pas être modifé, et gardera donc une valeur constante. Leur initialisation doit se faire de la même façon que pour les champs de classe.

3.3 Définition de méthodes

3.3.1 Le passage des paramètres

Tous les paramètres sont passés *par valeur*. Les seuls types possibles de paramètres sont les types primitifs et les références. Autrement dit :

- les types primitifs sont passés par valeur. Une méthode ne peut donc jamais modifier la valeur d'une variable de type primitif,
- les références également sont passées par valeur (valeur de la référence vers l'objet). Si la méthode modifie un champ de l'objet référencé, c'est l'objet qui est modifié, et le code appelant voit donc l'objet référencé modifié.

3.3.2 Signature et polymorphisme

Contrairement à ce que vous connaissez en `C`, un même identificateur peut être utilisé pour désigner deux méthodes **à condition** que leur signature soit différente.

On appelle *signature* d'une méthode, la donnée de son nom, du nombre de ses paramètres formels et de leurs types.

```
int methode1(int i){...}          // erreur, type retour de la methode ne
float methode1(int i){...}        // fait pas partie de sa signature
int methode2(int i){...}
float methode2(float f){...}        //OK
int methode3(int i) {...}
int methode3(int i, int j) {...} //OK
```

3.3.3 Variables locales

Les *variables locales* sont allouées lors de l'invocation de la méthode et sont détruites à la fin de celle-ci. Ces variables ne sont visibles qu'à l'intérieur de la méthode ou du bloc d'instructions où elles sont déclarées.

3.3.4 Méthodes de classe

Les méthodes vues juqu'à présent s'appliquent toujours à une référence sur un objet. Les méthodes qualifiées de `static` sont celles qui n'ont pas besoin d'une instance pour être invoquées.

Comme toute méthode, une méthode de classe est membre d'une classe. Elle est invoquée en lui associant, non pas un objet mais la classe à laquelle elle appartient. Par exemple, la méthode `sqrt` qui calcule la racine carrée appartient à la classe `Math`. Pour l'invoquer on écrit : `Math.sqrt(x) ;`

Une méthode `static`, puisqu'elle ne s'applique pas sur un objet, ne peut accéder aux variables d'instances. De même, le mot clé `this` n'a pas de sens dans une méthode `static`.

```
class Date{
   private int mois;
   private int jour;
   private int annee;
   private Date suivant;
   public static Date listeDates=null;
   public Date(int j, int m, int a){
       jour=j; mois=m; annee=a;
       suivant=listeDates;
       listeDates=this;
   }
   ...
   public void imprimer(){
       System.out.println(jour+"/"+mois+"/"+annee);
   }
   public static void listerDate(){
```

```
        for (Date d=Date.listeDates; d!=null; d=d.suivant)
              d.imprimer();*
   }
}
class Test {
   public static void main(String[] arg){
          Date noel=new Date(25,12,2000);
        Date aujourdhui=new Date(25,9,2000);
        Date.listerDate();
   }
}
```

Chapitre 4

Héritage

4.1 Introduction

La notion d'héritage est fondamentale en POO. Elle permet de spécialiser des classes. Reprenons l'exemple de la classe `Date`, et supposons que nous devions maintenant définir une classe `DateAnniversaire`, qui associe à une date donnée le nom et le prénom d'une personne née à cette date. Une première solution consisterait à définir complètement la nouvelle classe :

```
class DateAnniversaire{
   private int mois;
   private int jour;
   private int annee;
   private String nom;
   private String prenom;
   public DateAnniversaire(int j,int m,int a,String n,String p) {
      jour=j; mois=m; annee=a;
      nom=n; prenom=p;
   }
   public affecter(int m,int j,int a,String n,String p) {
      jour=j; mois=m; annee=a;
      nom=n; prenom=p;
   }
   ...
   public void imprimer(){
      System.out.println(prenom+" "+nom+" est ne le "+jour+"/"+mois+"/"+annee);
   }
}
```

Cette approche va à l'encontre de l'esprit de la POO. Dans la mesure où l'on a déjà écrit une classe `Date`, il s'agit de la réutiliser, en la spécialisant. C'est l'idée de l'héritage. Une `DateAnniversaire` est une `Date` avec des fonctionnalités supplémentaires.

L'héritage est une caractéristique des langages orientés objets. Une classe obtenue par héritage possède la totalité des champs et méthodes de la classe de base (dont elle hérite). Une classe `B` peut donc se définir par rapport à une classe `A` dont elle hérite. On dit que la classe `B` est une sous classe de la classe de base `A`. Une

sous classe doit évidemment compléter (enrichir) la classe de base, on parle aussi de spécialisation. Elle définit donc des champs et comportements supplémentaires, et peut, éventuellement, modifier une ou des méthodes de la classe de base.

Notre exemple de classe `DateAnniversaire` possède beaucoup de caractéristiques de la classe `Date` (évidemment, c'est une date !). Elle comporte deux champs supplémentaires, et les méthodes (constructeur, méthodes d'accès et de modification) doivent être complétées et/ou adaptées en fonction de l'ajout de ces nouveaux champs. On définira la classe `DateAnniversaire` comme une sous classe de la classe `Date`. Cela se fait en Java grâce au mot clé `extends`.

Voici l'exemple complet de la classe `DateAnniversaire`. Nous y reviendrons par la suite :

```
class Date {
    protected int mois;
    protected int jour;
    protected int annee;
    public Date(int j, int m, int a) {
     jour=j; mois=m; annee=a;
    }
    public void affecter(int j, int m, int a) {
     mois=m;  jour=j; annee=a;
   }

}

class DateAnniversaire extends Date{
   private String nom;
   private String prenom;
   public DateAnniversaire(int j,int m,int a,String n,String p) {
    super(j,m,a);
    nom=n; prenom=p;
   }
   public void affecter(int j,int m,int a,String n,String p) {
    super.affecter(j,m,a);
    nom=n; prenom=p;
   }
    public void imprimer(){
     System.out.println(prenom+" "+nom+" est ne(e) le "+super.jour+"/"+\
super.mois+"/"+super.annee);
    }
}

class TestDate{
    public static void main(String[] arg){
     DateAnniversaire d=new DateAnniversaire(0,0,0,"","");
     d.affecter(10,3,1920,"Boris","Vian");
     d.imprimer();
    }
}
```

4.2 Retour sur les qualificatifs de classes et champs

Il existe trois qualificatifs (on dit aussi modifieurs) pour les classes :
- `public` : **une seule classe** ou interface peut être déclarée public par fichier source `.java`, et par convention, le fichier porte le nom de la classe déclarée `public`. Une telle classe est accessible depuis l'extérieur (nous reverrons ces notions avec les paquetages).
- `final` : une classe déclarée `final` ne peut être dérivée (et ne peut donc jamais suivre la clause `extends`).
- `abstract` : une classe déclarée `abstract` ne peut jamais être instanciée. Nous verrons l'intérêt de telles classes un peu plus loin. Disons simplement pour le moment que leur intérêt est de fournir une espèce de modèle pour les classes dérivées.

Pour les champs, voici les qualificatifs possibles :
- `public` : pour signifier que le champ est accessible partout où est accessible la classe dans laquelle il est déclaré,
- `protected` : pour signifier que le champ est accessible par les classes du même paquetage et les classes dérivées de la classe où il est déclaré,
- `package` : pour signifier que le champ est accessible par les classes du même paquetage (c'est le qualificatif par défaut),
- `private` : pour signifier que le champ n'est accessible qu'à l'intérieur de la classe où il est déclaré,
- `static` : pour signifier qu'il s'agit d'un champ de classe, un seul exemplaire est créé,
- `final` : pour signifier qu'il s'agit d'une constante,
- `transient` : que nous verrons plus tard... lorsque nous aborderons les notions de persistance,
- `volatile` : que nous verrons plus tard... lorsque nous aborderons les notions de processus (threads).

Maintenant, vous devez mieux comprendre les qualificatifs donnés aux champs de la classe `Date`.

4.3 Constructeur de la sous-classe

4.3.1 Invocation du constructeur

Lors de la définition d'une classe dérivée, il faut s'assurer que, lors de l'instanciation des objets de cette nouvelle classe, les champs propres à cette classe mais aussi les champs de la classe de base seront bien initialisés. Souvent, les champs de la classe de base sont *privés* et la classe dérivée ne peut donc se charger de leur initialisation.

Ainsi le constructeur de la classe dérivée devra faire appel à celui de la classe de base pour l'initialisation de ces champs. Dans notre exemple de dates, on dira que pour créer une `DateAnniversaire`, il faut d'abord créer une `Date`.

Voici quelques points essentiels :

- Le constructeur est appelé au moment de la création de l'objet (instanciation). Il initialise cet objet en fonction des paramètres fournis.
- Si la classe ne comporte pas de constructeur, Java en crée un de façon implicite, sans paramètre. Mais attention, si la classe a au moins un constructeur avec paramètre(s) et aucun sans paramètre, elle n'a alors plus de constructeur par défaut.
- Si, la **première instruction** du constructeur n'est pas un appel explicite d'un constructeur de la classe de base (utilisation de `super(...)`, voir plus loin), **le constructeur par défaut de la classe de base est appelé.**
- Si la classe de base n'a pas de constructeur par défaut (ou de constructeur sans paramètre), on a une erreur de compilation (j'ai repris l'exemple des dates, et enlevé l'appel explicite au constructeur de la classe Date dans celui de la classe DateAnniversaire) :

```
Date2.java:20: No constructor matching Date2() found in class Date2
   public DateAnniversaire(int j,int m,int a,String n,String p) {
          ^
1 error
```

4.3.2 Enchaînement des constructeurs

Rappelons que la classe `Object` est la mère de toutes les classes : toute classe est dérivée directement ou non de la classe `Object`. Pour tout objet instancié, le constructeur de sa classe est invoqué, lequel,à son tour, invoque le constructeur de sa classe de base et ainsi de suite. Cette cascade d'appels s'arrête évidemment lorsqu'on atteint le constructeur de la classe `Object`.

4.4 Redéfinition et surcharge

4.4.1 Redéfinition des champs

Les champs déclarés dans la classe dérivée sont toujours des champs **supplémentaires**. Si l'on définit un champ ayant le même nom qu'un champ de la classe de base, il existera alors deux champs de même nom. Le nom de champ désignera celui déclaré dans la classe dérivée. Pour avoir accès à celui de la classe de base, il faudra changer le type de la référence pointant sur l'objet, ou utiliser `super`. Voici un exemple :

```
class A {
   public int i;
   ...
}
class B extends A {
   public int i;
```

```
   ...
   public void uneMethode(){
      i=0;                    // champ defini dans la classe B
      this.i=0;               // champ defini dans B
      super.i=1;              // champ defini dans A
      ((A) this).i=1;         // champ defini dans A
      ...
   }
}
class C extends B {
   public int i;
   ...
   public void uneMethode(){
      i=0;                    // champ defini dans la classe C
      this.i=0;               // champ defini dans C
      super.i=1;              // champ defini dans B
      ((B) this).i=1;         // champ defini dans B
      ((A) this).i=1;         // champ defini dans A
      ...
   }
}
```

Mais attention l'instruction suivante est incorrecte ! `super.super.i=1 ;`

De plus, souvenez-vous que comme l'utilisation du mot-clé `this`, le mot-clé `super` ne peut pas être utilisé dans les méthodes qualifiées de `static`.

4.4.2 Redéfinition des méthodes

On n'est bien sûr pas tenu de déclarer de nouveaux champs dans une classe dérivée, il se peut que seuls les comportements (méthodes) changent avec de nouvelles méthodes ou des méthodes redéfinies.

La redéfinition d'une méthode consiste à fournir une implémentation différente de la méthode **de même signature** fournie par la classe mère.

Exemple :

```
class Fruit{
    public String nom;
    public Fruit(String n) {
    nom=n;
    }
    public void imprimer() {
    System.out.println("je suis un(e) "+nom);
    }
    public String getNom(){
    return nom;
    }
}
class Pomme extends Fruit{
    public Pomme(){
```

```
    super("pomme");
    }
    public void imprimer() {
    System.out.println("je suis une "+nom);
    }
}
class Test{
    public static void main(String[] arg){
    Fruit f=new Fruit("ananas");
    Pomme p=new Pomme();
    f.imprimer();
    System.out.println(f.getNom());
    p.imprimer();
    f=(Fruit)p;
    f.imprimer();
    System.out.println(p.getNom());
    System.out.println(f.getNom());
    }
}
/*** exemple d'execution :
je suis un(e) ananas
ananas
je suis une pomme
je suis une pomme
pomme
pomme
*/
```

Quelques précisions supplémentaires :

1. Pour avoir accès à une méthode redéfinie de la classe de base, il faudra utiliser le mot clé `super`.
2. Une méthode `static` peut aussi être redéfinie par une autre méthode `static` (mais pas par une méthode non `static`).
3. Les destructeurs (cf. 3.1.5) ne sont pas invoqués en chaîne comme les constructeurs, c'est au programmeur, s'il le juge utile de réaliser cette chaîne de destructeurs (à l'aide du mot clé `super`).

4.5 Méthodes et classes finales

Une méthode est `final` si elle ne peut être redéfinie par des classes dérivées. Ainsi, on peut figer l'implémentation d'une méthode. On peut aussi décider de figer la définition d'une classe en la déclarant `final`. Cela signifie qu'il ne sera pas possible d'en dériver une nouvelle classe.

4.6 Conversions entre classes et sous-classes

Une opération de `cast` permet de modifier le type d'une référence. Ces modifications ne sont permises que dans des cas précis. On peut ainsi *affiner* le type d'une référence. Par exemple une référence vers un objet de type `Date` peut être changée en une référence vers un objet de type `DateAnniversaire`. L'objet référencé est toujours le même, on essaie juste de faire croire à la JVM que l'objet référencé est d'une autre nature. En aucun cas, l'objet référencé n'est modifié par le `cast`. On ne peut changer le type d'une référence en une référence vers un objet d'une classe dérivée que si cet objet est effectivement du type prétendu. Une référence vers un objet de type `Date` peut être changé en une référence vers un objet de type `DateAnniversaire` que si l'on s'est assuré que l'objet référencé est réellement de la classe `DateAnniversaire`. Sinon, l'exception `ClassCastException` est générée.
Exemple :

```
Date d;
DateAnniversaire da;
...
da=d;     // erreur compilation !
d=da;     // OK
da=(DateAnniversaire) d  // OK
```

4.7 Classes et méthodes abstraites

Une méthode est qualifiée `abstract` lorsqu'on la déclare sans donner son implémentation (on n'a que son prototype). Une classe **doit** être déclarée `abstract` dès lors qu'elle contient une méthode abstraite. Il est interdit de créer une instance d'une classe abstraite (souvenez-vous que son implémentation n'est pas complète). Puisqu'une classe abstraite ne peut pas être instanciée, il faudra évidemment la dériver pour pouvoir l'utiliser. Une sous-classe d'une classe abstraite sera encore abstraite si elle ne définit pas toutes les méthodes abstraites de la classe mère.

Une méthode `final` ne peut être déclarée abstraite, puisqu'on ne peut pas redéfinir une telle méthode.

Une classe abstraite peut être utilisée pour regrouper des classes. C'est l'exemple de la classe abstraite `Polygone` que l'on verra en TD.

4.8 Interfaces

`Java` ne permet pas l'héritage multiple. Il pallie ce manque par l'introduction des interfaces. Les interfaces peuvent être vues comme des *modèles*, sortes de classes ne possédant que des champs `static final` (c'est-à-dire des constantes) et des méthodes abstraites. On pourrait dire que les interfaces sont des classes abstraites dont **toutes** les méthodes sont abstraites et publiques et **tous** les champs sont publics et constants.

Les interfaces servent à :
- garantir aux *clients* d'une classe que ses instances peuvent assurer certains services
- faire du polymorphisme avec des objets dont les classes n'appartiennent pas à la même hiérarchie d'héritage.

4.8.1 Déclarer des interfaces

Comme les classes, les interfaces sont constituées de champs (ou attributs) et de méthodes. Il existe néanmoins de très fortes contraintes dans la définition d'une interface :
- toutes les méthodes qui sont déclarées sont abstraites : aucune implémentation n'est donnéee. Toutes les méthodes étant publiques et abstraites, les mots clés `public` et `abstract` sont implicites et n'apparaissent pas,
- aucune méthode n'est `static`,
- tous les champs sont `public`, `static` et `final`, il définissent des constantes. Les mots clés `static` et `final` sont implicites.

Qualificatifs pour une interface : Une interface peut être qualifiée de `public`, auquel cas elle sera utilisable par n'importe quelle classe. En l'absence de ce qualificatif, elle ne peut être utilisée que par les classes du même paquetage. Contrairement aux classes, on ne peut qualifier une interface de `private` ni `protected`.

Attributs d'une interface : Il sont `static` et donc les règles d'initialisation de tels attributs s'appliquent ici. On ne peut pas qualifier les attributs d'une interface de `transient`, `volatile`, `synchronized`, `private` ni `protected`.

Dériver une interface : Comme pour les classes, on peut organiser les interfaces de façon hiérarchique.Mais contrairement aux classes, une interface peut dériver plusieurs autres interfaces.

4.8.2 Implémenter des interfaces

Les interfaces définissent des promesses de services. Mais seule une classe peut rendre effectivement ces services. Une interface seule ne sert à rien ! Il faut une classe qui implémente l'interface. Une telle classe déclare dans son entête qu'elle implémente une interface :

```
interface Service {
 ...
}
class X implements Service {
 ...
}
```

Par l'utilisation du mot clé `implements`, la classe promet d'implémenter **toutes** les méthodes déclarées dans l'interface. La signature d'une méthode implémentée doit évidemment être identique à celle qui apparait dans l'interface, sinon la méthode est considérée comme une méthode de la classe et non de l'interface.

4.8.3 Utiliser des interfaces

Comme pour des classes, on peut définir des références ayant le type d'une interface. Par contre, il ne sera pas possible de définir un objet de ce type ! Si l'on déclare par exemple `Service s`, `s` est une référence qui contient soit la valeur `null`, soit une reférence à un objet d'une classe implémentant l'interface `Service`.

Chapitre 5

Chapitre 6

Tableaux et chaînes de caractères

6.1 Tableaux

Ce sont des suites d'objets **de même type**. Le nombre d'éléments est fixe et est appelé *taille* du tableau. Les tableaux sont des objets et leurs éléments sont soit de type primitif, soit des références. Pour utiliser un objet de type tableau, il faut donc définir une variable de type référence :

```
int [] tab1;
int tab2[];
```

Ces variables sont des références ; l'espace mémoire nécessaire pour coder la suite des objets est réservé avec le mot clé `new` et l'opérateur `[]` :

```
tab1 = new int[5];
tab2 = new int[2 * nbre +5];
```

Contrairement au langage `C`, il n'est pas nécessaire que la taille du tableau soit textuellement une constante. Comme il s'agit d'une allocation dynamique, la taille peut être une expression dont la valeur est un entier positif ou nul.

6.1.1 Type des éléments

Les éléments d'un tableau peuvent être de n'importe quel type : primitif ou référence. On peut tout à fait définir un tableau de références vers une classe abstraite, ou vers des objets implémentant une interface :

```
Dirigeable [] tab=new Dirigeable[10];
VehiculeARoues tab2=new VehiculesARoues[10];
Voiture [] tab3 = new Voiture[4];
```

On initialisera le premier tableau avec tout objet implémentant l'interface `Dirigeable` (cf. chapitre 8), le deuxième tableau avec des objets de classe dérivée de la classe abstraite `VehiculeARoues`, enfin le troisième tableau avec des objets de la classe `Voiture`.

6.1.2 Accès aux éléments

On accède aux éléments d'un tableau grâce à l'opérateur `[]`. On notera `tab[0]`, `tab[1]`,...,`tab[n-1]` les `n` premiers éléments du tableau (notez l'indice du premier élément).

Les indices peuvent être de type `int`, `short`, `byte` ou char.

Lors de l'accès à un élément d'un tableau, `Java` vérifie s'il y a débordement. Si l'indice est en dehors des limites du tableau l'exception `IndexOutOfBoundsException` est lancée.

6.1.3 Taille des tableaux

A chaque tableau est associée sa taille `length` qui est un champ `public final` de la classe des tableaux. On connait donc la taille du tableau par ce champ.

```
for (int i=0;i<tab1.length;i++) tab1[i]=i;
```

6.1.4 Initialisation

Lors de la création d'un tableau, ses éléments sont initialisés à une valeur par défaut. Pour les tableaux de nombres (entiers et flottants), la valeur initiale est zéro, pour les tableaux de références, la valeur initiale est `null`.
Attention ! Définir un tableau d'objets ne définit qu'un tableau de références. Les objets devront être alloués ultérieurement.

```
Date[] tabDate = new Date[3];
tabDate[0] = new Date(15,9,59);
tabDate[1] = new Date(1,1,0);
tabDate[2] = new Date(31,3,94);
```

L'initialisation d'un tableau peut se faire au moment de sa définition, comme en `C`, à l'aide d'accolades :

```
int [] tab ={1,2,3,4,5,6,7,8,9,0};
Date [] tabDate = {new Date(15,9,59),new Date(1,1,0),new Date(31,3,94))};
```

6.1.5 Tableaux multidimensions

Les tableaux multidimensions sont des tableaux de tableaux. La syntaxe pour définir une matrice 5x5 d'entiers est

```
int[][]mat = new int[5][5 ];
```

Comme pour les tableaux à une dimension, on peut initialiser les tableaux multidimensions au moment de leur définition :

```
int[][]mat = {{1,0,0,{0,1,0},{0,0,1}};
int [][]pascal ={{1},{1,1},{1,2,1},{1,3,3,1}{1,4,6,4,1}};
```

Notez que dans le cas du tableau `pascal`, les sous-tableaux sont tous de taille différente. Les noms de ces sous-tableaux sont `pascal[0], pascal[1], ....` On doit toujours spécifier la première dimension quand on crée le tableau, on peut ne spécifier les dimensions suivantes qu'au moment de la création des sous-tableaux.

```
public class TableauDeTableau {
   public static void main(String[] arg) {
      int [][] uneMatrice=new int[4][];
      // remplir la matrice
      for (int i=0;i<uneMatrice.length;i++) {
          uneMatrice[i] = new int[5]; // creation d'un sous-tableau
          for (int j=0;j<uneMatrice[i].length;j++) {
            uneMatrice[i][j]=i+j;
          }
      }
      // imprimer la matrice
      for (int i=0;i<uneMatrice.length;i++) {
          for (int j=0;j<uneMatrice[i].length;j++) {
            System.out.print(uneMatrice[i][j]+" ");
          }
          System.out.println();
      }
   }
}
```

6.2 Chaînes de caractères

En Java, les chaînes de caractères sont des objets d'une classe spécifique. Il ne s'agit pas de tableaux de caractères. Le paquetage `java.lang` contient deux classes de chaînes de caractères : `String` et `StringBuffer`. On a déjè rencontré la classe `String`. On l'a utilisée quand on avait besoin de chaînes de caractères qui n'étaient pas modifiées (chaînes constantes). La classe `StringBuffer` est utilisée pour travailler avec des chaînes dont le contenu est modifié.

6.2.1 Classe `String`

Dans beaucoup de cas, les chaînes de caractères qu'on utilise ne sont pas destinées à être modifiées, il s'agit d'objets constants. Le compilateur `Java` transforme automatiquement les constantes de type chaînes en objets de type `String`. On peut aussi créer explicitement un objet de type `String` avec un des constructeurs de la classe.

En plus de ses constructeurs (il y en a 11 !), la classe fournit des méthodes de comparaisons, de recherches, d'extractions et de copies. Voici celles qui me paraissent les plus utilisées. Pour les autres, n'hésitez pas à consulter la documentation.

Prototype	Rôle
`public String()`	constructeur
`public String(String str)`	constructeur
`public int length()`	longueur de la chaîne
`public char charAt(int index)`	caractère à la position `index`
`public String substring(int dbt,int fin)`	extrait la chaîne entre les positions `dbt` et `fin`
`public boolean equals(Object o)`	test d'égalité
`public boolean startsWith(String pref)`	test si le début de la chaîne est égal à `pref`
`public boolean endsWith(String suf)`	test si la fin de la chaîne est égal à `suf`
`public int compareTo(String str)`	comparaison des 2 chaînes,(0 si `str` est égale, négatif si elle est inférieure, positif sinon)
`public int indexOf(int ch)`	position du caractère `ch`
`public int lastIndexOf(int ch)`	dernière position du caractère `ch`
`public int indexOf(int ch, int i)`	position de `ch` à partir de `i`
`public int indexOf(String str)`	position de la ss-chaîne `str`
`public String replace(char c,char d)`	remplace toute occurrence de c par d
`public String toLowerCase()`	conversion en minuscules
`public String toUpperCase()`	conversion en majuscules
`public char[] toCharArray()`	conversion en tableau de caractères
`public String trim()`	suppression des espace en début et fin
`public static String valueOf(char t[])`	conversion d'un tableau de caractères en String

Exemple :

```
class chaines{
 public static void main(String [] arg){
  String a="Coucou";
  String b=new String(", c'est moi !\n");
  String c=a+b;
  System.out.println(c);
  System.out.println("longueur de a : "+a.length()); //6
  System.out.println("caractere en position 2 : "+a.charAt(2)); //u
  System.out.println("a est Coucou : "+a.equals("Coucou")); //true
  System.out.println("a est b : "+a.equals(b)); //false
  System.out.println("position de o dans a? "+a.indexOf('o')); //1
  System.out.println("position du dernier o dans a? "+a.lastIndexOf('o')); //4
  System.out.println("position de \"cou\" dans a? "+a.indexOf("cou")); //3
  System.out.println("position de \"moi\" dans a? "+a.indexOf("moi")); //-1
  System.out.println("a en majuscules : "+a.toUpperCase()); //COUCOU
  System.out.println("a en minuscules : "+a.toLowerCase()); //coucou
  System.out.println("a > b ? "+a.compareTo(b)); //23
```

```
  System.out.println("a < b ? "+b.compareTo(a)); //-23
 }
}
```

6.2.2 Classe `StringBuffer`

On a vu que l'on avait recours à la classe `String` pour les chaînes que l'on n'est pas amené à modifier. Mais dans les programmes, certaines chaînes sont amenées à être modifiées, dans ce cas, il faut utiliser des objets de la classe `StringBuffer`. Typiquement, on utilise des *Strings* pour les arguments et les résultats des méthodes. Pour construire une chaîne, on utilisera le type *StringBuffers*. Notez que, justement parce qu'il s'agit de constantes, les *Strings* sont moins onéreuses (en mémoire) que les *StringBuffers*. L'exemple qui suit est typique de l'utilisation de ces deux classes[1].

Exemple :

```
class ReverseString{
  public static String reverseIt(String source) {
     int i, len=source.length();
     StringBuffer dest=new StringBuffer(len);
     for (i=(len-1);i>=0;i--) {
        dest.append(source.charAt(i));
     }
     return dest.toString();
  }
}
```

Un objet de type `StringBuffer` a un espace de stockage à la création, automatiquement redimensionné en fonction des besoins. Pour créer un objet de cette classe, on peut utiliser un des 3 constructeurs :

- `StringBuffer()` : construit un *string buffer* ne contenant pas de caractères et avec une capacité initiale de 16 caractères.
- `StringBuffer(int)` : construit un *string buffer* ne contenant pas de caractères et avec une capacité initiale spécifiée par l'argument.
- `StringBuffer(String)` : construit un *string buffer* contenant la même séquence de caractères que la chaîne constante passée en argument, avec une capacité de 16 caractères plus la longueur de la chaîne passée en argument.

Comme pour la classe `String` il existe un certain nombre de méthodes pour les `StringBuffer`. En voici quelques unes :

[1] tiré du tutorial Java Sun

Prototype	Rôle
`public int length()`	longueur de la chaîne
`public char charAt(int index)`	caractère à la position `index`
`public void getChars(int dbt, int fin, char dst[],int index)`	recopie la ss-chaîne entre les positions `dbt` et `fin`, dans le tableau `dst`, à partir de l'indice `index`
`public int capacity()`	capacité courante
`public void setCharAt(int index, char c)`	met le caractère `c` à l'indice `index`
`public StringBuffer append(Object obj)`	concatène la représentation textuelle de l'obj. `obj`

Chapitre 7

Exceptions

7.1 Introduction

Dans un programme, il faut soigner la gestion des erreurs. Ce n'est pas toujours facile avec les langages classiques. `Java` propose une approche très différente des approches traditionnelles, à travers le mécanisme des *exceptions*. Une *exception* est une sorte de signal indiquant qu'une erreur ou une situation anormale a eu lieu. On dit qu'une méthode ayant détecté une situation anormale déclenche (*throws*) une exception. Cette exception pourra être capturée (*catch*) par le code.

On peut distinguer deux types de situations anormales : les *exceptions* et les *erreurs*. Les *erreurs* sont en principe des erreurs fatales et le programme s'arrête à la suite de ce type de situation (classe `java.lang.Error`). Les *exceptions* ne sont pas uniquement des *erreurs système*. Le programmeur peut définir des erreurs (non fatales) pour assurer que son programme est robuste (classe `java.lang.Exception`). Par exemple, le débordement d'un tableau est une exception.

Lorsqu'une méthode déclenche une exception la JVM remonte la suite des invocations des méthodes jusqu'à atteindre une méthode qui capture cette exception. Si une telle méthode n'est pas rencontrée, l'exécution est arrêtée.

L'uilisation des exceptions permet de :

- séparer le code correspondant au fonctionnement normal d'un programme, du code concernant la gestion des erreurs,
- propager de proche en proche les exceptions d'une méthode à la méthode appelante jusqu'à atteindre une méthode capable de gérer l'exception. Il n'est donc pas nécessaire que la gestion d'une exception figure dans la méthode qui est susceptible de déclencher cette exception. Une méthode peut ignorer la gestion d'une exception à condition qu'elle transmette l'exception à la méthode appelante,
- regrouper par types la gestion des exceptions.

7.2 Qu'est-ce qu'une exception

C'est un objet de la classe `java.lang.Throwable` qui est la classe mère de toutes les erreurs et exceptions du langage Java. Seuls les objets qui sont des instances de cette classe (ou d'une classe dérivée) sont déclenchés par la JVM et apparaissent comme arguments d'une clause `catch`. Nous allons voir ci-après les sous-classes principales de la classe `java.lang.Throwable`.

- `java.lang.Error` est la classe des erreurs, qui indiquent un problème grave qui doit conduire à l'arrêt de l'application en cours. On ne demande pas aux méthodes de déclarer une telle erreur dans la clause `throws`, puisqu'elle n'est pas susceptible d'être capturée. Un certain nombre d'erreurs dérivent de cette classe, par exemple `OutOfMemoryError`, et d'autres...
- `java.lang.Exception` est la classe des exceptions qui indiquent qu'une application devrait raisonnablement les capturer, c'est-à-dire traiter ces cas de situations anormales, sans arrêter le programme. Voici des exceptions classiques qui dérivent de cette classe : `java.io.IOException`, `FileNotFoundException`, et bien d'autres... A chaque objet de la classe `java.lang.Exception` (ou d'une classe dérivée) est associé un message que l'on peut récupérer avec la méthode `getMessage()` de la classe `java.lang.Throwable`
- `RuntimeException` est une classe dérivée de la précédente, et c'est la classe mère des exceptions qui peuvent être déclenchées au cours de l'exécution d'un programme. Supposons qu'une méthode soit susceptible de lever une exception de type `RuntimeException`, il n'est pas obligatoire de le signaler dans sa clause `throws`. En effet, les exceptions de type `RuntimeException` peuvent être levées mais ne pas être capturées, générant ainsi un arrêt du programme. Voici quelques exemples de sous-classes de la classe `RuntimeException` :
  - `ArrayStoreException`,
  - `ArithmeticException`,
  - `NullPointerException`,
  - `NumberFormatException`...

7.2.1 Capturer une exception

On l'a dit précédemment, lorsqu'une exception est lancée, elle se propage dans la pile des méthodes jusqu'à être capturée. Si elle ne l'est pas, elle provoque la fin du programme, et la pile des méthodes traversées est indiquée à l'utilisateur.

Supposons qu'une instruction `instr` d'une méthode `uneMethode` lance une exception, alors :

- si `instr` se trouve dans un bloc `try`, suivi d'un bloc `catch` alors,
  1. les instructions du bloc `try` suivant `instr` ne sont pas exécutées,
  2. les instructions du bloc `catch` sont exécutées,
  3. le programme reprend son cours normalement avec l'instruction suivant le bloc `catch`.

– si `instr` ne se trouve pas dans un bloc `try` comme décrit précédemment, alors la méthode `uneMethode` est terminéee. Si `uneMethode` est la méthode `main`, le programme se termine, et l'exception n'a pas été capturée. Sinon, on se retrouve dans une méthode qui a appelé la méthode `uneMethode` via une instruction `instr2` qui lance à son tour l'exception.

Une méthode susceptible de lancer une exception sans la capturer doit l'indiquer dans son entête avec la clause `throws`. Cependant, comme précisé précédemment, on est dispensé de déclarer le lancement des erreurs les plus courantes, comme par exemple :

– `ArrayOutOfBoundsException`,
– `ArrayStoreException`,
– `ArithmeticException`,
– `NullPointerException`,
– `NumberFormatException`...

Exemple :[1]

```
class AttrapExcep{
  static int moyenne(String[] liste) {
   int somme=0, entier, nbNotes=0;
   for (int i=0;i<liste.length;i++) {
      try{
          entier=Integer.parseInt(liste[i]);
          somme+=entier;
          nbNotes++;
      }
      catch(NumberFormatException e) {
          System.out.println("La "+(i+1)+"ième note pas entière");
      }
   }
   return somme/nbNotes;
 }
 public static void main(String [] arg) {
   System.out.println("La moyenne est :"+moyenne(arg));
 }
}
```

Voici quelques exemples d'exécution du programme précédent :

```
chaouiya/GBM2/coursJava/Notes_cours$ java AttrapExcep 5 b 10
La 2ième note n'est pas un entier
La moyenne est :7

chaouiya@pccc:~/GBM2/coursJava/Notes_cours$ java AttrapExcep 5 10 15
```

[1] emprunté à I.Charon

```
La moyenne est :10

chaouiya@pccc:~/GBM2/coursJava/Notes_cours$ java AttrapExcep 5 10 15 n
La 4ième note n'est pas un entier
La moyenne est :10
chaouiya@pccc:~/GBM2/coursJava/Notes_cours$ java AttrapExcep 10.5 xx
La 1ième note n'est pas un entier
La 2ième note n'est pas un entier
java.lang.ArithmeticException: / by zero
        at AttrapExcep.moyenne(AttrapExcep.java:14)
        at AttrapExcep.main(AttrapExcep.java:17)
```

7.2.2 Définir de nouveaux types d'exceptions

Les exceptions sont des objets d'une classe dérivée de `java.lang.Exception`. Si l'on veut signaler un événement inattendu, non prévu par l'API de `Java`, il faut dériver la classe `Exception` et définir une nouvelle classe qui ne contient en général pas d'autre champ qu'un ou plusieurs constructeur(s) et éventuellement une redéfinition de la méthode `toString`. Lors du lancement d'une telle exception, on crée une instance de cette nouvelle classe.
Exemple :

```
class ExceptionRien extends Exception {
   public String toString() {
     return("Aucune note n'est valide'\n");
   }
}
```

7.2.3 Lancer et capturer une exception

Rien ne vaut un exemple, reprenons celui de I.Charon[2] :

```
class ExceptionThrow {
  static int moyenne(String[] liste) throws ExceptionRien {
    int somme=0,entier, nbNotes=0;
    int i;
    for (i=0;i < liste.length;i++) {
        try{
            entier=Integer.parseInt(liste[i]);
            somme+=entier;
            nbNotes++;
        }
        catch (NumberFormatException e){
            System.out.println("La "+(i+1)+" eme note n'est "+
```

[2]http ://www.infres.enst.fr/ charon/coursJava

```
                                "pas entiere");
        }
    }
    if (nbNotes==0) throw new ExceptionRien();
    return somme/nbNotes;
  }
  public static void main(String[] argv)  {
    try {
        System.out.println("La moyenne est "+moyenne(argv));
    }
    catch (ExceptionRien e) {
         System.out.println(e);
    }
  }
```

7.2.4 Blocs `finally`

La clause `finally` est en général utilisée pour "faire le ménage" (par exemple fermer les fichiers, libérer les ressources, ...). Un bloc `finally` est utilisée en association avec un bloc `try`. On sort d'un bloc `try` par une instruction `break` ou `return` ou `continue` ou par une propagation d'exception. Un bloc `finally` suit un bloc `try` suivi, en général, d'un bloc `catch`. Dans tous les cas, quelque soit la façon dont on est sorti du bloc `try`, les instructions du bloc `finally` sont exécutées.

Voici un exemple, toujours tiré du support de cours d'Irène Charon, qui n'a d'autre objectif que d'illustrer l'effet du bloc `finally` :

```
class MonException extends Exception {
  MonException() {
    System.out.println("me voila");
  }
}
class Propagation {
  static boolean probleme=true;

  static void methodeBasse() throws MonException {
    try {
        if (probleme) throw new MonException();
        System.out.println("et moi ?");
    }
    finally {
        System.out.println("hauteur basse : il faudrait etre ici");
    }
    System.out.println("pas mieux");
  }
  static void methodeMoyenne() throws MonException {
```

```
    try {
        methodeBasse();
        System.out.println("et ici ?");
    }
    finally {
        System.out.println("moyenne hauteur : ou bien etre la");
    }
  }
  static void methodeHaute() {
    try {
        methodeMoyenne();
    }
    catch(MonException e) {
        System.out.println("attrape...");
    }
 }
 static public void main(String[] argv) {
    methodeHaute();
 }
}
```

Chapitre 8

Un exemple : des véhicules

Dans ce chapitre, nous allons détailler une application qui utilise des classes décrivant différents types de véhicules.

8.1 Une classe `Direction`

Elle définit les 4 directions et est utilisée dans la suite.

```
class Direction {
    int valeur;
    String nom;
    public Direction(int b,String s) {
      valeur=b;
      nom=s;
    }
}
```

8.2 Une interface : `Dirigeable`

Cette interface annonce les caractéristiques et services communs à tout système de "dirigeable", c'est-à-dire que l'on peut conduire... Elle définit aussi 4 constantes de la classe `Direction`.

```
interface Dirigeable {
    Direction Sud=new Direction(1,"Sud");
    Direction Est=new Direction(2,"Est");
    Direction Nord=new Direction(3,"Nord");
    Direction Ouest=new Direction(4,"Ouest");

    void accelerer(int facteur)throws VitesseExcessive;
                                //pour accélerer d'un facteur donné
    void ralentir(int facteur);  // pour ralentir
    int quelleVitesseCourante();       // vitesse courante
    void tournerDroite();        // pour tourner à droite
```

```
    void tournerGauche();        // pour tourner à gauche
    void faireDemiTour();         // pour faire demi-tour
    Direction quelleDirectionCourante();
}
```

8.3 Une classe abstraite :VehiculeARoues

Cette classe définit ce qu'est un véhicule à roues (en opposition aux autres véhicules que l'on écrira en TD). Elle implémente (quand cela est possible) l'interface `Dirigeable`. Elle reste abstraite car les vitesses excessives, le nombre roues sont (notamment) différentes selon les véhicules à roues considérés.

```
abstract class VehiculeARoues implements Dirigeable  {
    protected String couleur;
    protected int nbRoues;
    protected int vitesseCourante;
    protected Direction directionCourante;
    protected boolean etat; // marche ou panne
    public int quelleVitesseCourante(){
      return vitesseCourante;
    }
    public Direction quelleDirectionCourante(){
      return directionCourante;
    }
    public void tournerDroite(){
      switch (directionCourante.valeur) {
      case 1 : //sud
          directionCourante=Ouest;
          break;
      case 2 ://est
          directionCourante=Sud;
          break;
      case 3 : //nord
          directionCourante=Est;
          break;
      case 4 :  // ouest
          directionCourante=Nord;
          break;
      }
    }
    public void tournerGauche(){
      switch (directionCourante.valeur) {
      case 1 : //sud
          directionCourante=Est;
          break;
      case 2 ://est
          directionCourante=Nord;
```

```
            break;
        case 3 : //nord
            directionCourante=Ouest;
            break;
        case 4 : //ouest
            directionCourante=Sud;
            break;
        }
    }
    public void faireDemiTour(){
        switch (directionCourante.valeur) {
        case 1 : //sud
            directionCourante=Nord;
            break;
        case 2 ://est
            directionCourante=Ouest;
            break;
        case 3 ://nord
            directionCourante=Sud;
            break;
        case 4://ouest
            directionCourante=Est;
            break;
        }
    }

    public boolean getEtat(){
        return etat;
    }
    abstract public void accelerer(int param) throws VitesseExcessive;
    public void ralentir(int param) {
        vitesseCourante-=param;
        if (vitesseCourante<0) vitesseCourante=0;
    }
    public void changerEtat(){
        if (etat) vitesseCourante=0;
        etat=!etat;
    }
    abstract public void afficher();
    public int combienDeRoues(){return nbRoues;}
}
```

8.4 Des classes Voiture, Camion et Velo

```
class Voiture extends VehiculeARoues{
```

```
    private static final  int vitesseMax=130;
    private static final int nbRoues=4;
    public Voiture(int v,boolean e, String c) throws VitesseExcessive {
      if (v>vitesseMax) throw new VitesseExcessive(vitesseMax);
      else vitesseCourante=v;
      etat=e;
      if (!etat) vitesseCourante=0;
      couleur=c;
      directionCourante=Sud; // par défaut
    }
    public Voiture(int v,boolean e, String c,String d) throws VitesseExcessive{
      if (v>vitesseMax) throw new VitesseExcessive(vitesseMax);
      else vitesseCourante=v;
      etat=e;
      if (!etat) vitesseCourante=0;
      couleur=c;
      if (d.equalsIgnoreCase("Nord")) directionCourante=Nord;
      else if (d.equalsIgnoreCase("Est")) directionCourante=Est;
      else if (d.equalsIgnoreCase("Ouest")) directionCourante=Ouest;
      else directionCourante=Sud;
    }

    public void accelerer(int param) throws VitesseExcessive{
      int nouvelleVitesse = vitesseCourante+param;
      if (nouvelleVitesse <0 ) vitesseCourante=0;
      else if (nouvelleVitesse >vitesseMax) throw
                              new VitesseExcessive(this.vitesseMax);
      else vitesseCourante=nouvelleVitesse;
    }
    public void afficher() {
      if (etat)
          System.out.println("Voiture "+couleur +" en état de marche roulant
                      à "+vitesseCourante+"km/h plein "+directionCourante.nom);
      else
          System.out.println("Voiture "+couleur +" en panne");
    }
}

class Camion extends VehiculeARoues{
    private static final  int vitesseMax=90;
    private int nbRoues;

    public Camion(int r,int v,boolean e,String c)
                                throws VitesseExcessive,NbRouesImpossible {
      if (r!=4 && r!=6 && r!=8) throw new NbRouesImpossible(r);
      else nbRoues=r;
      if (v>vitesseMax) throw new VitesseExcessive(vitesseMax);
```

```
      else vitesseCourante=v;
      etat=e;
      if (!etat) vitesseCourante=0;
      couleur=c;
      directionCourante=Sud;
    }
    public void accelerer(int param) throws VitesseExcessive{
      int nouvelleVitesse = vitesseCourante+param;
      if (nouvelleVitesse <0 ) vitesseCourante=0;
      else if (nouvelleVitesse >vitesseMax) throw
                                 new VitesseExcessive(this.vitesseMax);
      else vitesseCourante=nouvelleVitesse;
    }
    public void afficher() {
      if (etat)
          System.out.println("Camion "+couleur +" en état de marche, roulant
                     à "+vitesseCourante+"km/h, sur "+nbRoues+" roues,
                     plein "+directionCourante.nom);
      else
          System.out.println("Camion "+couleur +" en panne");
    }
}

class Velo extends VehiculeARoues{
    private static final  int vitesseMax=20;
    private static final int nbRoues=2;

    public Velo(int v,boolean e,String c) throws VitesseExcessive{
      if (v>vitesseMax) throw new VitesseExcessive(vitesseMax);
      else vitesseCourante=v;
      etat=e;
      if (!etat) vitesseCourante=0;
      couleur=c;
      directionCourante=Sud;
    }
    public void accelerer(int param) throws VitesseExcessive{
      int nouvelleVitesse = vitesseCourante+param;
      if (nouvelleVitesse <0 ) vitesseCourante=0;
      else if (nouvelleVitesse >vitesseMax) throw
                                    new VitesseExcessive(this.vitesseMax);
      else vitesseCourante=nouvelleVitesse;
    }
    public void afficher() {
      if (etat)
          System.out.println("Vélo "+couleur +" en état de marche, roulant à "
                       +vitesseCourante +"km/h, plein "+directionCourante.nom);
      else System.out.println("Vélo "+couleur +" en panne");
```

```
    }
}
```

8.5 Des exceptions : VitesseExcessive et NbRouesImpossible

```
class NbRouesImpossible extends Exception {
    private String msg;
    public NbRouesImpossible(int r) {
msg="pas de camion à "+r+" roues !";
    }
    public String toString(){return msg;}
}
class VitesseExcessive extends Exception {
    private String msg;
    public VitesseExcessive(int r) {
        msg="interdit de dépasser "+r+"km/h !";
    }
    public String toString(){return msg;}
}
```

8.6 L'application : AppliVehicules

```
public class AppliVehicules{
    public static void main(String arg[])
                              throws NbRouesImpossible,VitesseExcessive {
Voiture maVoiture=new Voiture(0,true,"verte");
Voiture taVoiture=new Voiture(30,true,"bleue","est");
Velo monVelo=new Velo(10,false,"rouge");
Camion monCamion=new Camion(8,50,true,"jaune");
try {
Camion tonCamion=new Camion(7,50,true,"jaune");
}
catch(NbRouesImpossible e){
    System.out.println(e);
}
try{ maVoiture.accelerer(140);}
catch (VitesseExcessive e){
    System.out.println(e);
}
maVoiture.afficher();
maVoiture.faireDemiTour();
maVoiture.accelerer(40);
maVoiture.afficher();
maVoiture.changerEtat();
maVoiture.afficher();
monVelo.changerEtat();
```

```
taVoiture.afficher();
taVoiture.faireDemiTour();

monCamion.accelerer(-10);
monCamion.afficher();
monCamion.changerEtat();
monCamion.afficher();
    }
}
```

Vous pouvez récupérer le fichier complet avec toutes les classes :

http ://www.esil.univ-mrs.fr/ chaouiya/Java/cours/AppliVehicules

Chapitre 9

Paquetage `java.lang`

Ce paquetage définit un ensemble de classes et exceptions (voir le tableau à la fin du chapitre, le paquetage définit aussi des erreurs qui ne sont pas listées dans ce tableau) qui constituent le noyau du langage `Java`. On y retrouve surtout ce que l'on appelle les enveloppes (*wrappers* en anglais) qui définissent des classes spéciales correspondant à un certain nombre de types primitifs. Encore une fois, n'hésitez pas à consulter la documentation de l'API !

9.1 Enveloppes

9.1.1 Classe `java.lang.Number`

C'est une classe abstraite mère des classes `Byte`, `Short`, `Integer`, `Long`, `Float`, `Double`.

```
public abstract class Number
public Number() // constructeur
public abstract int intValue()
public abstract long longValue()
public abstract float floatValue()
public abstract double doubleValue()
public short shortValue()
```

9.1.2 Classe `java.lang.Integer`

Elle permet de représenter un entier sous la forme d'un objet. Ce qui me parait le plus utilisé (pour le reste se référer à la documentation) :

```
public final class Integer extends Number
 public static final int MAX_VALUE // la valeur max d'un int
 public static final int MIN_VALUE
 public static final Class TYPE  // l'objet classe représentant le type int
 public Integer(int value) // constructeur
 public Integer(String s) throws NumberFormatException
```

```
public static String toString(int i, int radix) //radix est la base
.........
public static String toString(int i)
public static int parseInt(String s) throws NumberFormatException
........
public static Integer valueOf(String s) throws NumberFormatException
........
public int intValue()
```

9.1.3 Classe `java.lang.Boolean`

Elle permet de représenter un booléen sous la forme d'un objet. Voici ce qui me parait le plus utilisé (pour le reste se référer à la documentation) :

```
public final class Boolean extends Object
public static final Boolean TRUE
public static final Boolean FALSE
public static final Class TYPE
public Boolean(boolean value)
........
public boolean booleanValue()
........
public String toString()
........
```

9.1.4 Classe `java.lang.Character`

Elle permet de représenter un caractère sous forme d'objet. Voici ce qui me parait le plus utilisé (pour le reste se référer à la documentation) :

```
public final class Character extends Object
.........
public static final Class TYPE
.........
public static final byte SPACE_SEPARATOR
public static final byte LINE_SEPARATOR
public static final byte PARAGRAPH_SEPARATOR
public static final byte CONTROL
.........
public Character(char value)
public char charValue()
public String toString()
public static boolean isLowerCase(char ch)
public static boolean isUpperCase(char ch)
public static boolean isDigit(char ch)
public static boolean isLetter(char ch)
public static boolean isLetterOrDigit(char ch)
public static char toLowerCase(char ch)
```

```
public static char toUpperCase(char ch)
public static boolean isSpace(char ch)
...........
public static int getNumericValue(char ch)
..........
```

9.2 Classe java.lang.Math

C'est la bibliothèque mathématique de Java.Toutes ses méthodes sont publiques et statiques. Voici ce qui me parait le plus utilisé (pour le reste se référer à la documentation) :

```
public final class Math extends Object
public static final double E
public static final double PI
public static native double sin(double a)
public static native double cos(double a)
public static native double tan(double a)
public static native double asin(double a)
public static native double acos(double a)
public static native double atan(double a)
public static native double exp(double a)
public static native double log(double a)
public static native double sqrt(double a)
public static native double ceil(double a) // partie entiere sup
public static native double floor(double a) // partie entiere inf
public static native double pow(double a,double b) Throws: ArithmeticException
                                                  // a puissance b
public static int round(float a)
public static synchronized double random()
public static int abs(int a)
....... et les surcharges de abs .......
public static int max(int a,int b)
....... et les surcharges de max .......
public static int min(int a,int b)
....... et les surcharges de min .......
```

Nom	Descriptif
Interfaces :	
`Clonable`	indique qu'un objet peut être *cloné*
`Runnable`	cf chapitre sur les processus légers
Classes :	
`Boolean`	cf. ce chapitre
`Byte`	pour le type primitif `byte`
`Character`	cf. ce chapitre
`Class`	les classes et interfaces d'une application Java
`ClassLoader`	
`Compiler`	
`Double`	pour le type primitif `double`
`Float`	pour le type primitif `float`
`Integer`	cf ce chapitre
`Long`	pour le type primitif `float`
`Math`	cf ce chapitre
`Number`	cf ce chapitre
`Object`	mère de toutes les classes !
`Process`	
`Runtime`	
`SecurityManager` hline `Short`	pour le type primitif `short`
`String`	cf chapitre 6
`StringBuffer`	cf chapitre 6
`System`	voir chapitre 11
`Thread`	voir chapitre sur les processus légers
`ThreadGroup`	
`Throwable`	cf chapitre 7
`Void`	pour le type primitif `void`
Liste des exceptions :	`ArithmeticException`,`ArrayIndexOutOfBoundsException` `ArrayStoreException`, `ClassCastException` `ClassNotFoundException`, `CloneNotSupportedException` `Exception`, `IllegalAccessException` `IllegalArgumentException`, `IllegalMonitorStateException` `IllegalStateException`, `IllegalThreadStateException` `IndexOutOfBoundsException`, `InstantiationException` `InterruptedException`, `NegativeArraySizeException` `NoSuchFieldException`, `NoSuchMethodException` `NullPointerException`, `NumberFormatException` `RuntimeException`, `SecurityException` `StringIndexOutOfBoundsException`

Chapitre 10

Le paquetage `java.util`

Le paquetage `java.util` contient des classes utilitaires telles que `Vector`, `Stack` (pour stocker un nombre variable d'objets), `Dictionnary` et `HashTable` (pour associer deux objets, clé/valeur), `StringTokenizer` (pour découper des chaînes de caractères), et bien d'autres... que je vous liste ci-dessous avec un bref descriptif (pour certains seulement !). Pour plus d'"information, n'hésitez pas à consulter la documentation de l'API ! Nous ne détaillerons ici que les classes `Vector` et `Stack`. La classe `StringTokenizer`, utile pour l'analyse de chaîne de caractères, sera décrite dans le chapitre sur les entrées sorties.

10.1 Classe `java.util.Vector`

Rappel sur les tableaux :

- déclaration : `int [] tab` ou `int tab[]`, on indique quil s'agit d'un tableau qui contiendra ici des entiers, le type des éléments d'un tableau est unique,
- instanciation : `tab = new int[dimension]`, la taille du tableau est fixe, mais l'allocation est dynamique.

La classe `java.util.Vector` permet d'avoir :

- une taille dynamique,
- la possiblité de stocker des objets (`Object`) hétérogènes. Pour les types primitifs, on devra utiliser les enveloppes (*wrappers*) du paquetage `java.lang` (`Integer, Double, ...`).

Constructeurs

- `public Vector()`
- `public Vector(int capaciteInitiale)`, il est conseillé d'indiquer une taille initiale
- `public Vector(int capaciteInitiale,int incrementCapacite)`, par défaut, le vecteur double sa taille à chaque dépassement.

Ajout d'un élément

1. à la fin :
   ```
   public final synchronized void addElement(Object nvElt) ;
   ```

2. entre 2 élélements :
   ```
   public final synchronized void insertElement(Object nvElt,int indice)
   throws ArrayIndexOutOfBoundsException ;
   ```

Nom	Descriptif
Interfaces	
Enumeration	génère une série d'éléments, l'un après l'autre (cf.plus loin)
EventListener	cf. chapitre java.awt
Observer	pour être informer du changement d'objets Observable
Classes	
BitSet	vecteurs redimensionnables de bits
Calendar	
Date	
Dictionary	(abstraite) pour des tableaux associatifs
EventObject	
GregorianCalendar	
Hashtable	(hérite de Dictionary) pour des tables de hachage
ListResourceBundle	
Locale	
Observable	objets *observables* par des Observer
Properties	
PropertyResourceBundle	
Random	pour générer un flot de nombres pseudo-aléatoires
ResourceBundle	
SimpleTimeZone	
Stack	pile décrite plus loin
StringTokenizer	p/la décomposition de chaînes de caractères en unités lexicales
TimeZone	
Vector	vecteurs redimensionnables, décrite ci-après
Exceptions	
EmptyStackException	
MissingResourceException	
NoSuchElementException	
TooManyListenersException	

Remplacer un élément Il s'agit de remplacer un objet situé à la position `indice` par un autre :

```
public final synchronized void setElementAt(Object nvElt,int indice)
        throws ArrayIndexOutOfBoundsException ;
```

Accéder à un élément Attention, contrairement aux tableaux, il n'y a pas de mécanisme d'indexation pour les `Vector`, il faut passer par les méthodes suivantes :

1. élément à la position `indice` :
   ```
   public final synchronized Object elementAt(int indice)
           throws ArrayIndexOutOfBoundsException ;
   ```

2. premier élément :
   ```
   public final synchronized Object firstElement()
            throws NoSuchElementException ;
   ```
3. dernier élément :
   ```
   public final synchronized Object LastElement()
            throws NoSuchElementException ;
   ```

Vérifier si la liste est vide `public final boolean isEmpty() ;`

Déterminer la taille `public final int size() ;`

Changer la taille Si la nouvelle taille est supérieure à la taille courante, le vecteur est complété par **null**, sinon, le vecteur est tronqué :
```
public final synchronized void setSize(int taille) ;
```

Recopier un vecteur dans un tableau
```
public final synchronized void copyInto(Object[] tab) ;
```

Obtenir une liste des éléments La méthode `elements()` renvoie un objet de type `Enumeration` qui permet d'accéder aux élément de manière séquentielle (mais dans un ordre à priori indéterminé). C'est l'interface `Enumeration` du paquetage `java.util` qui permet d'énumérer une liste d'objets (qui peuvent être hétérogènes). Cette interfaces a deux méthodes :

1. pour savoir s'il reste des élémentsà énumérer,
   ```
   public abstract booleman hasMoreElements()
   ```
2. pour obtenir le prochain objet et avancer d'un cran,
   ```
   public abstract Object nextElement()
                  throws NoSuchElementException
   ```

Exemple :
```
Enumeration e = monVecteur.elements() ;
while (e.hasMoreElements())
Object objSuivant = e.nextElement() ;
// faire ce qu'il y a lieu de faire sur objSuivant
```

Note : attention, `objSuivant` est de type `Object`. Il est souvent nécessaire de faire un *cast* quand le vecteur contient des objets de classe spécifique.

Rechercher un élément On pourrait utiliser une `Enumeration` et parcourir tous les éléments. Mais c'est fastidieux et peu efficace, on a des méthodes spécifiques !

1. tester la présence d'un objet :
   ```
   public final boolean contains(Object obf) ;
   ```
2. trouver la première position :
   ```
   public final synchronized int indexOf(Object obj)
                  throws ArrayIndexOutOfBoundsException ;
   ```

3. trouver la première position à partir de `index` :
```
public final synchronized int indexOf(Object obj,int index)
          throws ArrayIndexOutOfBoundsException ;
```
4. trouver la dernière position :
```
public final synchronized int lastIndexOf(Object obj)
                    throws ArrayIndexOutOfBoundsException ;
```

5.
```
public final synchronized int lastIndexOf(Object obj)
                    throws ArrayIndexOutOfBoundsException ;
```

Supprimer un élément

1. supprimer tous les éléments :
```
public final synchronized void removeAllElements() ;
```
2. supprimer la première occurrence d'un objet :
```
public final synchronized boolean removeElement(Object obj) ;
```
3. supprimer un élément à une position donnée :
```
public final synchronized boolean removeElementAt(int index)
throws ArrayIndexOutOfBoundsException ;
```

10.2 Classe `java.util.Stack`

L'image traditionnelle de la pile est celle de la pile d'assiettes, on parle aussi de liste LIFO (*Last In First Out*). L'insertion et la suppression d'éléments se fait par la même extrêmité. Les opérations classiques sur les piles sont *PileVide*, *Empiler*, *Depiler*.
La classe `java.util.Stack` hérite de la classe `java.util.Vector`, et implémente les piles. Voici les méthodes de cette classe :

1. un constructeur sans argument : `public Stack() ;`
2. pour empiler :
```
public Object push(Object item) ;
```
3. pour dépiler :
```
public synchronized Object pop() ;
```
4. pour accéder au sommet de la pile (sans le supprimer) :
```
public synchronized Object peek() ;
```
5. pour tester si la pile est vide :
```
public boolean empty() ;
```
6. pour rechercher un objet dans la pile :
```
public synchronized int search(Object o) ;
```

10.3 Classe `java.util.StringTokenizer`

Elle permet le découpage de chaînes de caractères en *tokens* ou unités lexicales. La classe `java.io.StreamTokenizer` est similaire, mais plus sophistiquée.
Voici l'essentiel de ce qu'on trouve dans la classe `java.util.StringTokenizer` :

```
public StringTokenizer(String) // constructeur (delimiteurs \t\n\r)
public StringTokenizer(String str, String delim, boolean renvDelim)
        // avec la specification des caracteres de separation,
// renvDelim==true les delimiteurs sont consideres comme des tokens
public StringTokenizer(String str, String delim)

public boolean hasMoreTokens()
public String nextToken() throws NoSuchElementException
public boolean hasMoreElements() // equivalent hasMoreTokens
public Object nextElement() // equivalent nextToken
public int countTokens()
```

Chapitre 11

Entrées-sorties, paquetage `java.io`

Le paquetage `java.io` contient un grand nombre de classes, chacune pourvue d'attributs et méthodes, dont l'objectif est de vous permettre de réaliser des entrées et sorties de données. Le tableau à la fin du chapitre donne les interfaces, classes et exceptions définies dans ce paquetage. Nous ne détaillerons que les classes et méthodes qui me paraissent essentielles, pour le reste, vous référer à la documentation...

11.1 Paquetage `java.io`

Le principe des entrées/sorties est basé sur le concept des *flots* de données (*streams*). Un flot est un canal de communication dans lequel les données sont écrites ou lues de manière séquentielle. Deux groupes de classes manipulent des flots :

- les classes qui manipulent des octets (`InputStream, OutputStream` et leurs classes dérivées)
- celles qui manipulent des caractères (depuis la version 1.1) (`Reader, Write` et leurs classes dérivées).

Le paquetage `java.io` contient également :

- la classe `File` qui permet de gérer tous les accès aux informations relatives au système de fichiers,
- la classe `RandomAccessFile` qui permet la lecture et l'écriture dans des fichiers,
- la classe `StreamTokenizer` qui permet de faire de l'analyse syntaxique de base,
- des interfaces comme `DataInput` et `DataOutput` (e/s de types primitifs), `ObjectInput` et `ObjectOuput` (e/s d'objets), `Serialisable`...
- des exceptions, dont la plus classique `IOException`.

11.2 Classe `File`

Un objet de cette classe représente le nom d'un fichier ou répertoire de la machine hôte. Un fichier ou répertoire est spécifié par un chemin relatif (au répertoire courant). Cette classe tend à s'affranchir des particularités des différentes plateformes. La classe `RandomAccessFile` permet des manipulations plus complexes, mais ce n'est pas l'objet de ce cours.

1. Les variables les plus utiles :

- `separator` : chaîne de séparation entre répertoires d'un chemin (dépend de la plateforme)

2. Les méthodes les plus utiles :
   - `File(File, String)` constructeur qui crée une instance de `File` représentant le fichier ayant le nom (précédé du chemin) donné par la chaîne, dans le répertoire donné par l'argument de type `File`,
   - `File(String)` constructeur qui crée une instance de `File` qui représente le fichier dont le chemin est spécifié par la chaîne,
   - `File(String, String)` constructeur qui crée une instance de `File` qui représente le fichier dont le chemin est spécifié par le premier argument et le nom est spécifié par le second,
   - `boolean canRead()` teste si l'application peut lire le fichier correspondant à l'objet courant,
   - `boolean canWrite()` teste si l'application peut écrire dans le fichier correspondant à l'objet courant,
   - `boolean delete()` supprime le fichier correspondant à l'objet courant,
   - `boolean exists()` teste l'existence du fichier correspondant à l'objet courant,
   - `String getAbsolutePath()` retourne le nom absolu du fichier correspondant à l'objet courant,
   - `String getName()` retourne le nom du fichier correspondant à l'objet courant,
   - `String getPath()` retourne le chemin du fichier correspondant à l'objet courant,
   - `boolean isDirectory()` teste si le fichier correspondant à l'objet courant est un répertoire,
   - `boolean isFile()` teste si le fichier correspondant à l'objet courant est un fichier,
   - `long lastModified()` retourne la date de dernière modification du fichier (peut servir à comparer deux fichiers et dire lequel est le plus récent),
   - `long length()` retourne la taille (en nombre d'octets) du fichier correspondant à l'objet courant,
   - `String[] list()` retourne une liste des fichiers du répertoire correspondant à l'objet courant.

Exemple 1 : lister le contenu des répertoires donnés sur la ligne de commande :

```
import java.io.File;
public class Ls {
   public static void main(String arg[]){
     File dir;
     for (int i=0;i<arg.length;i++) {
       dir = new File(arg[i]);
       System.out.println(dir.getAbsolutePath()+":");
       String [] r=dir.list();
       for (int j=0;j<r.length;j++)
           System.out.println("\t"+r[j]);
     }
  }
}
```

Exemple 2 : lister les attributs d'un fichier donné sur la ligne de commande (type (répertoire/fichier), taille, droits d'accès) :

```
import java.io.File;
public class AttributsFichier {
  public static void main(String arg[]){
    File dir;
    if (arg.length!=1) {
       System.out.println("usage : java AttributsFichier <nom_fichier>");
    }
    else {
       dir = new File(arg[0]);
       if (dir.isFile()) {
         System.out.println(dir.getAbsolutePath()+" est un fichier");
       }
       else {
         System.out.println(dir.getAbsolutePath()+" est un repertoire");
       }
       System.out.println("taille : "+dir.length()+" octets");
       System.out.println("lecture autorisee : "+(dir.canRed()?"oui":"non"));
       System.out.println("ecriture autorisee : "+(dir.canWrite()?"oui":"non"));
   }
 }
}
```

11.3 Classes InputStream, Reader, OutputStream, Writer

Il s'agit de classes abstraites qui servent de classes de base pour toutes les autres autres classes de flots d'entrées et de sorties. `InputStream` et `OutputStream` permettent la lecture et l'écriture de flots d'octets (fichiers binaires).`Reader` et `Writer` correspondent à des flots de caractères. Le tableau suivant donne les principales classes dérivées, et une description succincte :

Flot d'octets	Flot de caractères	Description
`InputStream`	`Reader`	Classe abstraite avec les méthodes de base
`BufferedInputStream`	`BufferedReader`	*bufférise* les entrées (utilisation d'une mémoire-tampon)
`FileInputStream`	`FileReader`	lit dans un fichier
`LineNumberInputStream`	`LineNumberReader`	garde la trace du nombre de lignes lues
`ByteArrayInputStream`	`CharArrayReader`	lit un tableau
`FilterInputStream`	`FilterReader`	(classe abstraite) filtre l'entrée
`DataInputStream`	pas d'équivalent	lit des données de types primitifs
`StringInputReader`	`StringReader`	lit depuis une chaîde caractères
pas d'équivalent	`InputStreamReader`	transforme un flot d'octets en caractères
`OutputStream`	`Writer`	Classe abtraite avec les méthodes de base
`PrintStream`	`PrintWriter`	écrit des valeurs et objets dans un flot de sortie
`BufferedOutputStream`	`BufferedReader`	*bufférise* les sorties
`FileOutputStream`	`FileWriter`	écrit dans un fichier de sortie
`DataOutputStream`	pas d'équivalence	écrit des données de types primitifs
pas d'équivalent	`OutputStreamWrite`	transforme un flot d'octets en un flot de caractères

11.3.1 Méthodes des classes `InputStream`, `Reader`, `OutputStream`, `Writer`

1. les méthodes de lecture :
   - `int read()` lit simplement un octet dans le flot et le retourne sous forme d'entier (-1 si on a atteint la fin du flot),
   - `int read(byte b[])`lit plusieurs octets à la fois et les stocke dans le tableau (lance une exception `IOException` si le tableau n'est pas assez grand), retourne le nombre d'octets lus (-1 si on a atteint la fin du flot),
   - `int read(byte b[],int off, int len)` est la même que la précédente mais on précise ici où placer les octets dans le tableau (à partir de l'indice `off` et sur une longueur `len`).
2. les autres méthodes de `InputStream`
   - `long skip(long n)` est utilisé pour sauter `n` octets dans le flot, retourne le nombre d'octets réellement sautés (-1 si on a atteint la fin du flot),
   - `int available()` indique le nombre d'octets présents dans le flot d'entrée, cette méthode permet d'éviter le blocage résultant d'un ordre de lecture dans un flot où il n'y a pas de données disponibles,
   - `synchronized void mark(int limit)` place une marque à la position courante dans le flot (on peut ensuite revenir à cette marque avec la méthode `reset()` qui

suit), l'entier `limit` spécifie le nombre maximum d'octets que l'on peut lire avant que la marque devienne invalide,
- `synchronized void reset()` retourne dans le flot à la dernière position marquée (par la méthode`mark()`),
- `boolean markSupported` indique si le flot peut être marqué ou non,
- `void close()` ferme un flot d'entrée et libère toutes les ressources associées, même si cela n'est pas vraiment nécessaire, c'est une bonne habitude de faire appel explicitement à cette méthode.

3. Les méthodes d'écriture :
   - `void write(int b)` écrit l'octet donné sur le flot de sortie,
   - `void write(byte b[])` écrit les octets du tableau donné sur le flot de sortie,
   - `public void write(byte b[],int off,int len)` écrit `len` octets du tableau donné à partir de la position `off`.
4. Les autres méthodes de `OutputStream` :
   - `void flush()`, sert plutôt pour les flots *bufférisés* et force l'écriture du contenu du *buffer*,
   - `void close()` ferme un flot de sortie et libère les ressources associées.

Les méthodes des classes `Reader` et `Writer` sont similaires à celles de leur correspondantes `InputStream` et `OutputStream`. Nous ne les décrivons donc pas en détail.

11.4 Classe `java.lang.System`

Vous la connaissez déjà, elle vous permet de réaliser des entrées-sorties standards. Cette classe constitue une interface avec le système d'exploitation. Trois flots standards : `System.in`, `System.out` et `System.err`. Voici l'essentiel de cette classe :

```
public static final InputStream in // l'entree standard
public static final PrintStream out // la sortie standard
public static final PrintStream err // la sorite standard (erreur)
public static void setIn(InputStream in) // pour rediriger l'entree std
public static void setOut(PrintStream out) // pour rediriger la sortie std
public static void setErr(PrintStream err)
.......
public static Properties getProperties()     Throws: SecurityException
   // les proprietes courantes du systeme :
   //  java.version, java.home,..., os.name,file.separator ("/" sur UNIX)
   //  path.separator(":" sur UNIX), line.separator("\n" sur UNIX),
   //  user.name,user.home,
........
public static String getProperty(String key)  Throws: SecurityException
   // recupere les prop selon la cle (cf ci-dessus)
........
public static void exit(int status)
public static void gc() // lance le ramasse miettes
.........
```

La classe `java.lang.System` a trois attributs : `in` le flot d'entrée standard, `out` le flot de sortie standard et `err` le flot d'erreur standard (la plupart du temps identique à `System.out`). `System.in` est une instance de `InputStream` et correspond (la plupart du

temps) au clavier. On peut donc utiliser les méthodes de cette classe. On peut aussi le convertir en un `Reader`. Pour `System.out`, qui est une instance de `PrintStream`, vous l'avez déjà utilisé, avec ses deux méthodes `print` et `println`. Nous n'y reviendrons pas. L'exemple ci-dessous illustre la lecture au clavier de caractères :

```
import java.io.*;
class Lecture {
  public static void main(String arg[]){
    char c;
    try {
        System.out.print("Saisie :");
        c=(char)System.in.read();
        System.out.println(" c= "+c);
    }
    catch (IOException e) {
        System.out.println(e.toString());
    }
    try {
        Reader in = new InputStreamReader(System.in);
        c=(char)in.read();
        System.out.println(" c= "+c);
    }
    catch (IOException e) {
        System.out.println(e.toString());
    }
  }
}
/*** exemple d'execution ***
Saisie :abcdef
 c= a
 c= b
```

Voici un exemple de lecture au clavier d'une chaîne de caractères, utilisant la classe `Reader`.

```
import java.io.*;
class Lecture2 {
 public static void main(String arg[]){
   char buf[]=new char[10];
   try {
       Reader in = new InputStreamReader(System.in);
       in.read(buf,0,5);
       String s = new String(buf);
       System.out.println("chaine lue :"+s);
   }
   catch (IOException e) {
       System.out.println(e.toString());
   }
 }
}

/*** exemple d'execution ***
gbm-server:~/coursJava/Td/ES> java Lecture2
abcdefghijk
chaine lue :abcde
*/
```

11.4.1 Classes PrintStream et PrintWriter

Elles sont bien utiles puisqu'elles permettent d'écrire des valeurs ou des objets sur des flots de sortie. En fait, on conseille d'utiliser la seule classe `PrintWriter` (`PrintStream` n'est encore là que pour des raisons de compatibilité entre versions). Le constructeur de `PrintWriter` attend un `OutputStream`, ou bien un `Writer`. Ensuite, vous l'utilisez comme vous utilisiez `System.out`, avec essentiellement les méthodes `print` et `println` !

11.4.2 Classes BufferedInputStream (BufferedReader) et BufferedOutputStream (BufferedWriter)

Comme leur nom l'indique, ces classes fournissent un tampon (ou *buffer*) d'entrée (ou de sortie). Cela implique une meilleure performance car les accès disque sont plus lents que les accès mémoire.

1. Quelques méthodes de la classe `BufferedInputStream` :
   - le constructeur attend un paramètre de type `InputStream`,
   - on utilise principalement les méthodes `read()` et `read(byte[],int,int)`, similaires à celles de la classe mère.
2. Quelques méthodes de la classe `BufferedOutputStream` :
   - le constructeur attend un paramètre de type `OutputStream` et,éventuellement une taille de *buffer*,
   - on utilise les méthodes `write(int)`, `write(byte[],int,int)` et `flush()` similaires à celles de la classe mère.
3. Quelques méthodes de la classe `BufferedReader` :
   - le constructeur attend un paramètre de type `Reader`,
   - on utilise les méthodes `read()` et `read(char[],int,int)`, similaires à celles de la classe mère et surtout la méthode `readLine()`qui permet la lecture d'une ligne de texte.
4. Quelques méthodes de la classe `BufferedWriter` :
   - le constructeur attend un paramètre de type `Writer`,
   - on utilise les méthodes `flush()`, `write(int)`, `write(char[],int,int)` similaires à celles de la classe mère et la méthode `newLine()`qui écrit un "retour chariot".

Exemple de lecture d'une ligne de texte :

```
import java.io.*;
class Lecture3 {
 public static void main(String arg[]){
   String s;
   BufferedReader in = new BufferedReader( new InputStreamReader(System.in));
   try {
       s=in.readLine();
       System.out.println("chaine lue :"+s);
   }
   catch (IOException e) {
       System.out.println(e.toString());
   }
 }
}
```

```
/*** exemple d'execution ***
gbm-server:~/coursJava/Td/ES> java Lecture3
bonjour, comment va ?
chaine lue :bonjour, comment va ?
```

11.4.3 Classes FileInputStream (FileReader) et FileOutputStream (FileWriter)

Ce sont les classes pour l'écriture et la lecture dans un fichier. On instancie un objet de la classe `FileInputStream` avec un nom ou avec un objet de la classe `File`, mais attention il s'agit de fichiers binaires. De même on instancie un `FileReader`. Les méthodes sont les mêmes que celles des classes mères. De même pour les flots de sortie `FileOutputStream` et `FileWriter`.
Dans l'exemple qui suit, on crée un fichier binaire dans lequel sont écrits les entiers de 0 à 9. Essayez d'exécuter ce programme et de visualiser le fichier obtenu (il n'est guère lisible !).

```
import java.io.*;
class EcritureBinaire {
   public static void main(String arg[]){
      try {
         FileOutputStream fichier = new FileOutputStream(new File("fichier.dat"));
         for (int i=1;i<10;i++) fichier.write(i);
      }
      catch (IOException e) {
         System.out.println(e.toString());
      }
   }
}
```

11.4.4 Classes DataInputStream et DataOutputStream

Elle est utile pour lire des données de types primitifs dans des fichiers binaires. Voici les méthodes que vous serez le plus susceptibles d'utiliser :

`boolean readBoolean()`	lit un booléen
`byte readByte()`	lit un octet
`int readInt()`	lit un entier
`char readChar()`	lit un caractère
`float readFloat()`	lit un *float*
`double readDouble()`	lit un *double*
...	

Bien sûr, la classe `DataInputStream` a sa classe correspondante `DataOutpuStream` pour écrire des données de types primitifs. Les méthodes sont similaires, mais se nomment `writeBoolean`, `writeInt`, etc.
L'exemple qui suit illustre l'écriture de données typées dans un fichier binaire, puis leur lecture.

```
import java.io.*;
class LectEcriture {
```

```
  public static void main(String arg[]){
    try{
       DataOutputStream out= new DataOutputStream(new FileOutputStream(arg[0]));
       out.writeInt(3);
       out.writeDouble(3.2);
       out.writeChar('c');
       out.writeUTF("bye");
       out.close();
    }
    catch (IOException e) {
       System.out.println(e.toString());
    }
    try {
       DataInputStream in=new DataInputStream(new FileInputStream(arg[0]));
       System.out.println("contenu du fichier :");
       System.out.println(in.readInt()+" "+in.readDouble()+" "+in.readChar()+" "+in.readUTF());
       in.close();
    }
    catch (IOException e) {
       System.out.println(e.toString());
    }
  }
}

/*** exemple d'execution ***
gbm-server:~/coursJava/Td/ES> java LectEcriture sortie
contenu du fichier :
3 3.2 c bye
```

11.5 Classe StreamTokenizer

Cette classe fournit les méthodes permettant de faire de l'analyse syntaxique rudimentaire sur les données en entrée. La lecture se fait d'unité lexicale (*token*) en unité lexicale. Le constructeur d'un `StreamTokenizer` prend en argument un objet `StreamInput` ou `Reader` Voici les variables et méthodes les plus utiles :

`double nval`	si le *token* est un nombre, c'est sa valeur
`String sval`	si le *token* est un mot, c'est la chaîne correspondante
`int TT_EOF`	constante indiquant la fin du flot
`int TT_EOL`	constante indiquant la fin d'une ligne
`int TT_NUMBER`	constante indiquant un nombre
`int TT_WORD`	constante indiquant un mot
`int ttype`	contient le type du *token* et prend une des valeurs constantes définies précédemment
`int nextToken()`	lit le *token* suivant, son type est rangé dans `ttype` et sa valeur éventuellement dans `nval` ou `sval`
`int lineno()`	est une méthode qui retourne le numéro de ligne,
`void parseNumbers()`	spécifie que ce sont des nombres qui sont analysés

Exemple d'analyse de l'entrée lue au clavier :

```
import java.io.*;
class Analyse {
  public static void main (String[] argv) throws IOException  {
    int somme=0,nb=0;
    int type;
    String stop=new String("fin");
    System.out.println("Donnez le nom de l'etudiant,
                                        et ses notes, terminez avec fin");
    StreamTokenizer entree= new StreamTokenizer
                                        (new InputStreamReader(System.in));
    type=entree.nextToken();
    if ((type!=StreamTokenizer.TT\_WORD)) {
       System.out.println("donnez d'abord le nom !");
    }
    else {
       String nom=entree.sval;
       while(true) {
         type=entree.nextToken();
         if (type==StreamTokenizer.TT\_NUMBER) {
             somme+=(int)entree.nval;
             nb++;
         }
         else
            if ((type==StreamTokenizer.TT_WORD)&&(stop.equals(entree.sval)))
            break;
       }
       System.out.println("La moyenne de "+nom+" est "+((double)somme/nb));
    }
  }
}
/******** exemple d'execution
Donnez le nom de l'etudiant, et ses notes, terminez avec fin
machin 10 12
10
sgdb
12
fin
La moyenne de machin est 11.0
```

11.6 Autres exemples utiles

Vous pourrez consulter le site de I.Charon pour d'autres exemples (http ://www.infres.enst.fr/ charon/coursJava/fichiersEtSaisies/index.html)

La classe `java.net.URL` représente une *URL* (i.e. une adresse de ressource sur le Web). On peut, à partir d'un tel objet, ouvrir une connection et obtenir un flot à partir de l'*URL*. L'exemple ci-dessous vous donne une illustration :

```
import java.net.*;
import java.io.*;
```

```
class LectURL {
    public static void main(String arg[]) throws MalformedURLException{
      String s;
      URL monUrl = new URL("http://www.esil.univ-mrs.fr/~chaouiya/Java/Individu.java");
      try {
          InputStream in1=monUrl.openStream();
          BufferedReader in2 = new BufferedReader(new InputStreamReader(in1));
          s=in2.readLine();
          in2.close();
          System.out.println("chaine lue :"+s);
      }
      catch (IOException e) {
          System.out.println(e.toString());
      }
    }
}
/*** exemple d'execution ***
gbm-server:~/coursJava/Td/ES> java LectURL
chaine lue :class Individu \{
*/
```

Les exemples qui suivent montrent l'utilisation de `StringTokenizer` et des méthodes de conversion de chaînes en objets *enveloppes* comme `Integer`, `Double`...

```
import java.io.*;
import java.util.*;
class SaisieClavier {
  public static void main (String[] argv) throws IOException, NumberFormatException {
    int somme=0;
    String ligne;
    StringTokenizer st;
    BufferedReader entree =new BufferedReader(new InputStreamReader(System.in);
    ligne=entree.readLine();
    while(ligne.length() > 0) {
        st=new StringTokenizer(ligne);
        while(st.hasMoreTokens())
           somme+=Integer.parseInt(st.nextToken());
        ligne=entree.readLine();
    }
    System.out.println("La somme vaut : "+somme);
  }
}
/******* exemple d'execution
gbm-server:~/coursJava/Td/ES> java SaisieClavier
1 2
3
4
La somme vaut : 10
```

Autre exemple :

```
import java.io.*;
import java.util.*;
class SaisieClavier {
```

```
  public static void main (String[] argv) throws IOException, NumberFormatException {
    int entier;;
    BufferedReader entree =new BufferedReader(new InputStreamReader(System.in));
    StringTokenizer st = new StringTokenizer(entree.readLine());
    entier=Integer.valueOf(st.nextToken()).intValue();
  }
}
```

Nom	Descriptif
Interfaces :	
DataInput	lecture de types primitifs Java
DataOutput	écriture de types primitifs Java
Externalizable, FilenameFilter ObjectInput	(hérite de DataInput lecture d'objets
ObjectInputValidation ObjectOutput	(hérite de DataOutput) écriture d'objets
Serializable	pour la sauvegarde d'objets
Classes :	
BufferedInputStream,BufferedOutputStream	dans ce chapitre
BufferedReader,BufferedWriter	dans ce chapitre
ByteArrayInputStream,ByteArrayOutputStream	
CharArrayReader,CharArrayWriter	
DataInputStream, DataOutputStream	dans ce chapitre
File	dans ce chapitre
FileDescriptor	dans ce chapitre
FileInputStream, FileOutputStream	dans ce chapitre
FileReader, FileWriter	dans ce chapitre
FilterInputStream, FilterOutputStream	
FilterReader, FilterWriter	
InputStream	
InputStreamReader	
LineNumberInputStream, LineNumberReader	
ObjectInputStream ,ObjectOutputStream	
ObjectStreamClass	
OutputStream, ObjectOutputStream	
OutputStreamWriter	
PipedInputStream,PipedOutputStream	
PipedReader, PipedWriter	
PrintStream, PrintWriter	
PushbackInputStream,PushbackReader	
RandomAccessFile	
Reader	cf plus loin
SequenceInputStream	
StreamTokenizer	dans ce chapitre
StringBufferInputStream	
StringReader, StringWriter	
Writer	dans ce chapitre
Exceptions :	
CharConversionException,EOFException FileNotFoundException,IOException InterruptedIOException, InvalidClassException InvalidObjectException, NotActiveException NotSerializableException, ObjectStreamException OptionalDataException, StreamCorruptedException SyncFailedException, UTFDataFormatException UnsupportedEncodingException WriteAbortedException	

Chapitre 12

Applets, les bases

On présente dans ce chapitre l'essentiel de ce qu'il faut savoir pour développer des *applets* qui sont des applications Java qui tournent dans un navigateur. Nous verrons ensuite comment composer une interface graphique, puis comment la faire fonctionner.

12.1 Introduction

Jusque là, nous avons travaillé avec des applications indépendantes. Il s'agit de programmes, comparables à ceux écrits dans d'autres langages, dont le point d'entrée est la méthode `main` de la classe donnée en argument de la commande `java` (une application indépendante est exécutée directement par la JVM) et déclarée comme suit :
`public static void main(String [] arg)`

Les *applets*[1] sont des applications Java très particulières :
- elles s'exécutent dans un *browser* (navigateur) ou un *appletviewer* et non pas sous le contrôle direct de la JVM,
- elles ne peuvent accéder à toutes les ressources du système, pour des raisons évidentes de sécurité,
- elles possèdent différents points d'entrée, en fonction de leur cycle de vie (`init, start, stop,...`),
- elles sont des applications graphiques particulières qui dérivent de la classe `Applet`.

Une *applet* est donc un composant graphique (`Component`), qui peut contenir d'autres composants (`Component`) que l'on pourra placer (`Panel`). Nous reverrons les composants graphiques plus loin dans ce chapitre. La figure 12.1 donne l'arbre d'héritage de la classe `java.applet.Applet`.

12.2 Créer une applet

Pour créer une applet, vous devez créer une sous-classe de la classe `Applet` qui fait partie du paquetage `java.applet`. Cette classe fournit l'essentiel du comportement requis

[1] la terminologie française serait **appliquette** ou encore **appelette**, mais ne semble pas souvent adoptée

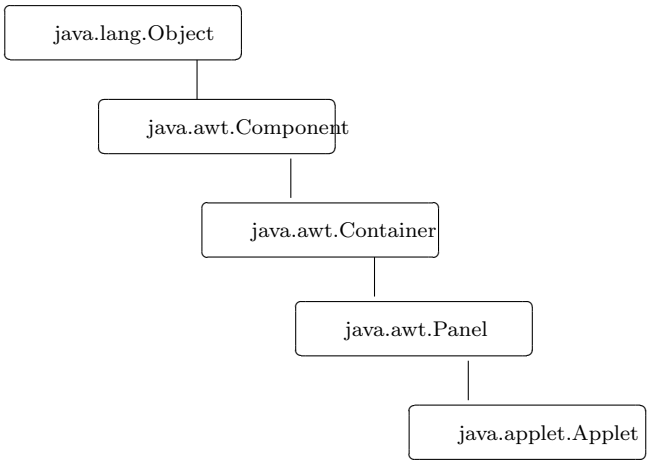

FIG. 12.1 – arbre héritage de la classe `Applet`

pour une applet pour qu'elle puisse être exécutée dans un navigateur supportant Java. Elle permet aussi d'utiliser toutes les fonctionnalités d'un composant du paquetage `awt`.

Bien sûr, votre applet peut faire appel à d'autres classes décrites par ailleurs, mais la classe initiale de l'applet doit avoir une signature comme suit :

```
public class monApplet extends java.applet.Applet ...
```

Notez que le mot clé `public` est ici indispensable pour la classe principale de votre applet, par contre les classes utilisées que vous créez ne sont pas nécessairement déclarées publiques. Quand un navigateur rencontre votre applet dans une page Web, il va charger votre classe initiale à travers le réseau, ainsi que les autres classes utilisées au fur et à mesure des besoins. Cela peut entraîner des pertes de performances (pour éviter cette perte de temps en chargement à travers le réseau, une solution consiste à utiliser des archives...)

Cycle de vie d'une applet :

initialisation phase où le navigateur charge l'applet et lui demande d'effectuer les initialisations nécessaires (méthode `public void init()`), cette phase n'a lieu qu'une seule fois,

visible chaque fois que l'applet devient ou redevient visible à l'utilisateur, il est demandé à l'applet de se redessiner (méthode `public void start()`),

invisible chaque fois que l'applet devient ou redevient invisible à l'utilisateur, il lui est demandé de procéder à la libération éventuelle des ressources utilisées, ou à l'interruption de certaines de ses activités comme la suspension d'une animation graphique (méthode `public void stop()`), cette phase est liée à la précédente,

arrêt définitif lorsque l'applet doit s'arrêter, il lui est demandé de se terminer proprement (méthode `public void destroy()`),

dessiner est la phase durant laquelle l'applet dessine effectivement quelque chose (texte, lignes, figures,...) cette phase peut avoir lieu des centaines de fois durant le cycle de vie de l'applet (méthode `public void paint(Graphics g)`, définie dans la classe `Container` du paquetage `java.awt`). Notez que l'argument de la méthode `paint` est une instance de la classe `Graphics` (du paquetage `java.awt`), nous y reviendrons.

C'est la classe `Applet` qui donne les prototypes des méthodes `init`, `start`, `stop`,

`destroy`.

Exemple : fichier `Bonjour.java`

```
import java.applet.*;
import java.awt.*;
public class Bonjour extends Applet {
    public void paint(Graphics g) {
     g.setFont(new Font("TimesRomman",Font.BOLD,30));
     g.setColor(Color.blue);
     g.drawString("Bonjour !",50,50);
    }
}
```

12.3 Introduction d'une applet dans une page Web

Pour exécuter une applet, comme cela se fait dans un navigateur, il faut créer une page HTML qui contiendra le nom de la classe principale, la taille et les paramètres de l'applet. Cela se fait par la balise (forme minimale) :

```
<APPLET
  CODE="MonApplet.class"
  WIDTH = taille_en_pixels
  HEIGHT = taille_en_pixels
</APPLET>
```

- l'attribut `CODE` donne le nom du fichier (extension `.class`) contenant l'applet. S'il ne se trouve pas dans le même répertoire que le fichier HTML, il faudra utiliser l'attribut `CODEBASE` décrit plus loin,
- les attributs `WIDTH` et `HEIGHT` sont nécessaires et donnent la taille de la boîte réservée à l'affichage de votre applet sur la page Web,
- le texte entre les balises `<APPLET>` et `</APPLET>` est affiché par les navigateurs qui ne comprennent pas ces balises (notamment ceux qui ne supportent pas les applets), il est donc bon de prévoir un petit message pour que vos lecteurs qui n'ont pas de navigateurs supportant Java, voient autre chose qu'une ligne blanche muette...

La balise `<APPLET>` a été présentée ci-dessus dans sa forme minimale. Ellea en fait plus de fonctionnalités (pour beaucoup, identiques à celles de la balise `<IMG>`) :

```
<APPLET ... liste d'attributs ..>
[<PARAM NAME={\it nomparam1} VALUE={\it valeurparam1}>]
[<PARAM NAME={\it nomparam1} VALUE={\it valeurparam1}>]
...
[texte pour les navigateurs ne comprenant pas Java]
</APPLET>
```

- l'attribut `ALIGN` définit comment l'applet doit être alignée dans la page, il peut prendre une des valeurs suivantes : `LEFT, RIGHT, TOP, TEXTTOP, MIDDLE, ABSMIDDLE, BASELINE, BOTTOM, ABSBOTTOM`,

- les attributs `HSPACE` et `VSPACE` sont utilisés pour définir un espace entre l'applet et le texte qui l'entoure,
- l'attribut `CODEBASE` permet de spécifier le répertoire ou l'URL où trouver l'applet, si elle n'est pas au même endroit que le fichier HTML de la page la contenant.

Enfin, il est possible de passer des paramètres à l'applet. Cela se fait avec la balise `<PARAM NAME=nom_parametre VALUE=valeur_parametre>` placée dans le champ de la balise `>APPLET>`.

Exemple : une applet qui affiche le texte qu'on lui passe en paramètre (fichier Texte.java), et le fichier HTML permettant de charger cette applet

- fichier `Texte.java` :

```
import java.applet.*;
import java.awt.*;
public class Texte extends Applet {
    String leTexte;
    public void init(){
leTexte=getParameter("le_texte");
    }
    public void paint(Graphics g) {
g.setFont(new Font("mafonte",Font.ITALIC,30));
g.setColor(Color.blue);
g.drawString(leTexte,50,50);
    }
}
```

- fichier `Texte.html`

```
<HTML>
<HEAD>
<TITLE>Applet simple qui affiche un texte</title>
</HEAD>
  <BODY>
    <h1>Applet simple qui affiche un texte</h1>
    <APPLET
      CODE = "Texte.class"
      WIDTH=300 HEIGHT = 200>
      <PARAM NAME=le_texte VALUE="Bonjour !">
      Votre navigateur ne supporte pas Java ?! Bonjour quand meme !
    </APPLET>
  </BODY>
</HTML>
```

Autres exemples :

- L'exemple ci-dessous illustre le cycle de vie d'une applet :

```
/******* le fichier java *******/
import java.awt.*;
import java.applet.*;
import java.util.*;
public class Affiche extends Applet {
    public Vector lesAppels;
    public void init() {
       lesAppels=new Vector();
       lesAppels.addElement("init");
```

```
    }
    public void start(){
       lesAppels.addElement("start");
    }
    public void stop(){
       lesAppels.addElement("stop");
    }
    public void paint(Graphics g) {
       lesAppels.addElement("paint");
       Enumeration e=lesAppels.elements();
       int y=15;
       while(e.hasMoreElements()) {
           g.drawString((String)e.nextElement(),20,y);
           y+=10;
       }
    }
}
/******* le fichier HTML ******/
<html>
  <head>
    <title>Premi&eacute;re Applet</title>
  </head>

  <body>
    <h1>Premi&eacute;re Applet</h1>
    <applet
      code = "Affiche.class" width=300 height = 200>
    </applet>
  </body>
</html>
```

– un petit exemple qui illustre quelques unes des méthodes de la classe `java.awt.Graphics` :

```
import java.awt.*;
import java.applet.*;
public class ParamGraphic extends Applet {
    public void init() {
      String couleur = getParameter("couleur");
    }
    public Color rendCouleur(String s)
    {
      if (s==null)return Color.black;
      else if (s.equals("rouge")) return Color.red;
      else if (s.equals("vert")) return Color.green;
      else if (s.equals("bleu")) return Color.blue;
      else if (s.equals("magenta")) return Color.magenta;
      else if (s.equals("pink")) return Color.pink;
      else if (s.equals("orange")) return Color.orange;
      else if (s.equals("cyan")) return Color.cyan;
      else if (s.equals("yellow")) return Color.yellow;
      return Color.black;
    }
    public void paint(Graphics g) {
      g.drawString((String)getParameter("titre"),20,10);
      g.setColor(rendCouleur((String)getParameter("couleur")));
```

```
      g.fillRect(30,30,100,100);
      g.fillOval(130,130,40,40);
    }
}
/******* le fichier HTML  *********/
<html>
  <head>
    <title>Un exemple tout simple</title>
  </head>
  <body>
    <h1>Un exemple tout simple</h1>
    <applet
      code = "ParamGraphic.class" width=400 height = 300>
      <PARAM NAME=couleur VALUE="bleu">
      <PARAM NAME=titre VALUE="comment passer des paramètres">
    </applet>
  </body>
</html>
```

12.3.1 Classe `Applet`, plus de détails

Comme il a été dit précédemment, la classe `Applet` hérite de la classe `Panel` qui elle même hérite de la classe `Container`. Ainsi beaucoup de méthodes sont décrites dans les classes mères (comme les méthodes que vous utiliserez beaucoup, `add`, `setLayout`, `setBackground`, et bien d'autres). Ici on ne donne que les méthodes propres à la classe `Applet` qui sont très utilisées :

- `void destroy()` : appelée par le browser pour informer l'applet qu'elle doit se *détruire* et libérer toutes les ressources utilisées,
- `AppletContext getAppletContext()` : détermine l'`AppletContext` (cf. 12.4),
- `URL getCodeBase()` : récupère l'URL de base de l'applet (où se trouve son code)
- `URL getDocumentBase()` : c'est l'adresse précédente suivie du nom du fichier
- `String getParameter(String name)` : retourne la valeur du paramètre ayant le nom passé en paramètre et qui est défini dans la balise HTML,
- `void init()` : appelée par le browser pour charger l'applet,
- `boolean isActive()` : détermine si l'applet est active,
- `void resize(Dimension d)` : demande que l'aplet soit redimensionnée (cf.classe `java.awt.Dimension` dans la documentation)
- `void resize(int width, int height)` : même chose que précédemment,
- `void start()` : appelée par le browser pour que l'applet démarre son exécution
- `void stop()` : appelée par le browser pour que l'applet stoppe son exécution,

12.4 Interface `AppletContext`

Elle permet d'obtenir des informations sur l'environnement dans lequel l'applet est exécutée (Navigateur, visualisateur d'applet...). Les méthodes de cette interface peuvent être utilisées par une applet pour obtenir ces informations. Voici les deux plus utilisées (pour le reste, consulter la documentation) :

- `Enumeration getApplets()` : trouve toutes les applets contenues dans le document correspondant à cet `AppletContext`,

– `void showDocument(URL url)` : remplace la page Web par celle référencée par l'URL donnée en paramètre.

Chapitre 13

Paquetage `java.awt` et les interfaces graphiques

Vous avez franchi la première étape qui consiste à comprendre comment fonctionnent les applets. Il faut maintenant se familiariser avec les outils fournis par Java pour dessiner sur la page, actualiser le contenu de la page, gérer les événements souris et clavier, créer des éléments d'interface utilisateur.

Attention, toutes les interfaces que vous serez amenés à écrire ne seront pas nécessairement incluses dans une applet, elles peuvent très bien être liées à une application indépendante. On pourra consulter dans l'annexe **??** des diagrammes représentant la hiérarchie des classes du paquetage `java.awt`.

13.1 Classe `java.awt.Graphics`

Elle définit la plupart des méthodes graphiques de Java. On n'a pas à créer explicitement une instance de `Graphics` (c'est de toute façon une classe abstraite). Dans la méthode `paint` vue pour les applets, on reçoit un objet de cette classe. Dessiner sur cet objet revient à dessiner sur la portion de fenêtre graphique allouée à l'applet. Le système des coordonnées place l'origine dans le coin supérieur gauche du composant graphique. Le tableau ci-après donne l'essentiel des méthodes trouvées dans la classe `Graphics`.

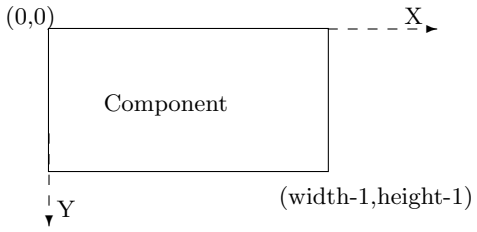

FIG. 13.1 – Système de coordonnées dans un composant graphique

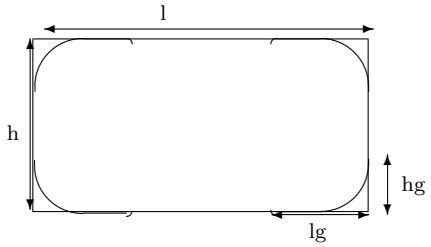

FIG. 13.2 – Paramètres d'un rectangle arrondi

Méthode	**Description**
`drawLine(x1,y1,x2,y2)`	ligne droite de (x1,y1)à (x2,y2)
`drawRect(x,y,l,h)`	rectangle vide, (x,y) coin sup.gauche, largeur l, hauteur h
`fillRect(x,y,l,h)`	rectangle plein, (x,y) coin sup.gauche, largeur l, hauteur h
`drawRoundRect(x,y,l,h,lg,hg)`	rectangle vide à coins arrondis,(x,y) coin sup.gauche, largeur l, hauteur h, angle largeur lg, hauteur hg
`fillRoundRect(x,y,l,h,lg,hg)`	rectangle plein à coins arrondis, ...
`draw3DRect(x,y,l,h,bool)`	rectangle vide effet 3D en relief (bool=true) ou enfoncé (bool=false)
`draw3DRect(x,y,l,h,bool)`	rectangle plein effet 3D...
`drawPolygon(tabX,tabY,n)`	polygone vide à n sommets, tabX tableau abscisses, tabY ordonnées
`fillPolygon(tabX,tabY,n)`	polygone plein à n sommets, tabX, tableau abscisses, tabY ordonnées
`drawPolygon(polyg)`	polygone vide défini par l'instance de `Polygon`
`fillPolygon(polyg)`	polygone plein défini par l'instance de `Polygon`
`drawOval(x,y,l,h)`	ovale vide délimité par le rectangle défini par x,y,l et h
`fillOval(x,y,l,h)`	ovale plein délimité par le rectangle défini par x,y,l et h

On dispose de deux méthodes pour l'affichage de texte :
- `drawString(chaine, x, y)`
- `drawChars(tabChar,dbt,fin,x,y)` (tableau de caractères, indice du permier caractère, indice du dernier, position d'affichage du premier).

Pour afficher du texte, il faut créer une instance de la classe `Font` définie par son nom, son style (`bold, italic`) et sa taille. Des noms de polices classques : ``TimesRoman'', ``Courrier'', Helvetica''. Les styles sont des constantes entières définies dans la classe `Font` : `Font.PLAIN`, `Font.BOLD`, `Font.ITALIC`. C'est la méthode `setFont(laPolice)` qui permet de spécifier la police à utiliser. Vous consulterez avec profit la documentation sur la classe `java.awt.Font`.

Pour dessiner un objet ou du texte, on utilise la couleur courante du contexte graphique. Cette couleur peut évidemment être changée avec la méthode `setColor(laCouleur)` qui prend en paramètre une instance de la classe `java.awt.Color`.

Bien sûr, on peut aussi spécifier la couleur de fond du composant graphique grâce à la méthode `setBackground(laCouleur)` de la classe `java.awt.Component` (dont hérite la classe `Applet`).

Le dernier exemple du chapitre 12 vous donne un exemple illustrant l'utilisation des couleurs.

13.2 Eléments d'interfaces

Il y a deux types d'éléments dans une interface, des conteneurs (*containers*) et des composants (*components*). Les premiers, comme leur nom l'indique sont susceptibles de contenir des éléments, ils constituent la base de l'interface (`Panel`, `Frame`, `Window`...). Les seconds sont ajoutés à un conteneur, il s'agit de boutons, cases à cocher, ascenseurs... (`Button`, `Checkbox`, `Scrollbar`, ...). En plus de ces éléments, le paquetage `java.awt` fournit des gestionnaires de mise en page (`FlowLayout`, `BorderLayout`, ...), la classe `Event` qui permet de détecter tous les événements externes (entrées clavier, clic de souris, ...), des exceptions.

Il n'est pas question ici de faire une présentation exhaustive de la programmation graphique en Java, mais plutôt de donner les éléments essentiels, à travers quelques exemples significatifs.

13.2.1 Conteneurs

Pour pouvoir placer les composants il faut un *conteneur*. Tous les conteneurs héritent de la classe `java.awt.Container`. Les deux principaux sont :

- la classe `Window` qui crée une nouvelle fenêtre, ses classes dérivées `Frame` (fenêtre avec un bord et une barre de menu) et `Dialog` (fenêtre pour le choix d'une fichier par exemple) sont également très utiles,
- la classe `Panel` est la classe mère de la classe `Applet`. Elle propose un espace dans lequel une application peut placer des composants mais aussi d'autres panneaux.

Par défaut les composants sont ajoutés de gauche à droite et de haut en bas. Il existe des gestionnaires pour gérer le placement des composants, ce sont les `LayoutManager`(cf. 13.3).

13.2.2 Composants

On les appelle aussi des *widgets*(pour WInDows gadGET), ce sont des composants graphiques que l'on peut ajouter dans des conteneurs.

1. `Label` pour afficher du texte qui ne doit pas changer,
2. `TextField` pour une zone de saisie d'une ligne de texte,
3. `TextArea` pour une zone de saisie d'un paragraphe,
4. `Button` pour un bouton,
5. `Checkbox` pour une case à cocher,
6. `CheckboxGroup` pour un groupe de cases à cocher,
7. `Choice` pour une liste de choix, `List` pour une liste défilante,
8. `Scrollbar` pour un ascenseur,

Composant	Constructeurs	Méthodes
`Label`	`Label(), Label(String)`	`setText(String), getText()`
`TextField`	`TextField(),` `TextField(int), TextField(String)`	hérite de `TextComponent`, `getText()`
`TextArea`	`TextArea(),` `TextArea(int,int),TextArea(String)`	hérite de `TextComponent`, `getText()`
`Button`	`Button(), Button(String)`	`getLabel()`, `setLabel(String)`(cf.13.6)
`Checkbox`	`Checkbox(), Checkbox(String),` `Checkbox(String, boolean),` `Checkbox(String, boolean,` `CheckboxGroup)`	`getLabel()`, `getState()`, `setLabel(String)` `setState(boolean)`
`CheckboxGroup`	`CheckboxGroup()`	`getSelectedCheckbox()`, `setSelectedCheckbox(Checkbox)`
`Choice`	`Choice()`	`add(String), addItem(String),getItem(int)` `getSelectedIndex(), getSelectedItem(),` `remove(int), remove(String)`
`Scrollbar`	`Scrollbar()` (vertical),`Scrollbar(int)`	`setOrientation(int), setMinimum(int)` `setMaximum(),setValues(int,int,int,int)` `getValue(), getMinimum(), getMaximum()`

TAB. 13.1 – Composants et quelques unes de leurs méthodes

9. `Canvas` pour une zone graphique.

Chacun de ces composants hérite bien sûr des attributs et méthodes de la classe `component` mais a aussi ses attributs et méthodes spécifiques. Le tableau 13.1 donne les méthodes essentielles associées à chacun de ses composants, mais le mieux est, encore une fois, de consulter la documentation ! Nous ne donnons pas ici les méthodes de gestion d'événements, souvent communes à beaucoup de composants, qui seront décrites plus loin (cf. 13.4).

Exemple :

```
import java.applet.Applet;
import java.awt.*;
public class AjoutComposants extends Applet {
    private Label l=new Label("Titre");
    private TextField  t1=new TextField("Entrez ici votre nom");
    private Button b = new Button("appuyer ici");
    private Checkbox c1= new Checkbox("oui");
    private Checkbox c2= new Checkbox("non");
    private CheckboxGroup grp= new CheckboxGroup();
    private List liste = new List(3,false);
    private Scrollbar sb = new Scrollbar(Scrollbar.HORIZONTAL,30,500,0,1000);

    public void init(){
        add(l);
        add(t1);
        add(b);
        c1.setState(true);
        c1.setCheckboxGroup(grp);
```

```
        add(c1);
        c2.setState(false);
        c2.setCheckboxGroup(grp);
        add(c2);
        liste.addItem("d'accord");
        liste.addItem("pas d'accord");
        liste.addItem("ne sais pas");
        add(liste);
        add(sb);
    }
}
```

13.3 Organiser l'interface

Nous savons comment ajouter des composants à un conteneur, mais on aimerait bien pouvoir organiser nos composants de façon un peu claire ! Pour cela, on dispose des `LayoutManagers` ou gestionnaires de disposition. Il en existe cinq :

- le `FlowLayout` est choisi par défaut pour les `Applet` et `Panel`, et place les composants comme indiqué précédemment (de gauche à droite et de haut en bas),
  la méthode `setLayout(new FlowLayout())` indique que le conteneur courant utilise un `FlowLayout` gestionnaire, et la méthode `add(`*nom du composant*`)` indique que l'on rajoute le composant cité,
- le `BorderLayout` est choisi par défaut pour les `Window`, `Dialog` et `Frame` et dispose les composants selon cinq attributs (`North`, `South`, `West`, `East` et `Center`). La méthode `add()` a deux paramètres : la position et le composant à rajouter,
- le `GridLayout` met chaque composant sur une case d'une grille dont on donne la dimension à la création du gestionnaire. Cela se passe de la même manière que pour un `FlowLayout`, mais les composants disposent d'un espace de même taille,
- `CardLayout` permet de gérer des panneaux comme des cartes, une seule étant visible à la fois (cf. le fameux jeu du solitaire),
- `GridBagLayout` est le plus flexible et le plus compliqué des gestionnaires !...

Deux petits exemples :

```
import java.applet.Applet;
import java.awt.*;
public class Disposition extends Applet {
    private Button b1 = new Button("bouton1");
    private Button b2 = new Button("bouton2");
    private Button b3 = new Button("bouton3");
    private Button b4 = new Button("bouton4");
    private Panel p = new Panel();
    public void init(){
        setLayout(new BorderLayout(2,2));
        add("North",b1);        add("West",b2);
        add("East",b3);         add("South",b4);
        add("Center",p);
    }
    public void paint(Graphics g) {
        Graphics g2=p.getGraphics();
        g2.setColor(Color.blue);
```

```
        g2.drawOval(10,10,30,30);
    }
}
==========================================
import java.applet.Applet;
import java.awt.*;
public class Disposition2 extends Applet {
    private Button b1 = new Button("bouton1");
    private Button b2 = new Button("bouton2");
    private Button b3 = new Button("bouton3");
    private Button b4 = new Button("bouton4");
    public void init(){
        setLayout(new BorderLayout(2,2));
        add("North",b1);
        add("West",b2);
        add("East",b3);
        add("South",b4);
    }
}
```

13.4 Gestion d'événements

Il s'agit maintenant de pouvoir réagir aux événements extérieurs (entrée de texte, clic de souris, ...). Les réactions aux événements sont gérées par un ou plusieurs *adaptateurs* (instances de classes héritant de l'interface `eventListener` du paquetage `java.awt.event`. Il y a des adaptateurs différents selon les types d'événements pouvant survenir. Tout composant graphique peut, en s'inscrivant auprès de l'adaptateur adéquat, signifier qu'il réagit à un certain type d'événements. Il y a donc trois points dans la gestion d'événements :

- la déclaration d'une classe `Reaction` qui implémente une des onze interfaces du paquetage `java.awt.event`, par exemple : `public class Reaction implements ActionListener ...`
- dans la classe `Reaction`, la définition de la ou des méthode(s) de l'interface, par exemple `ActionListener` déclare une seule méthode) : public void actionPerformed(ActionEvent e)
- l'inscription d'un composant auprès de l'adaptateur `Reaction`, par exemple (`b` est un bouton, et `X` une instance de la classe `Reaction`) : `b.addActionListener(X) ;`

On appelle *classe adaptateur* d'une interface, une classe qui implémente ses méthodes, avec un corps vide (pour qu'elles n'accomplissent aucune action). Cela peut être très utile dans le cas où une interface a plusieurs méthodes (on doit toutes les définir) et qu'il n'y en a qu'une que l'on veut définir. Nous y reviendrons dans les exemples.

Le tableau 13.2 qui suit donne les principales interfaces, avec le nom de l'interface, la classe d'événement, le type de composants générant ce type d'événement, les méthodes et, éventuellement la classe adaptateur. Ce tableau n'est pas exhaustif, et ne recense que les éléments les plus courants. Reportez vous à la documentation pour une information complète.

Interface	Evénement	Composants	Méthode(s)	Adaptateur
ActionListener	ActionEvent	Button, Textfield List	actionPerformed	–
ItemListener	ItemEvent	List, Choice Checkbox	itemStateChanged	–
KeyListener	KeyEvent	Component	keyPressed keyReleased keyTyped	KeyAdapter
MouseListener	MouseEvent	Component	mouseClicked mouseEntered mouseExited mousePressed mouseReleased	MouseAdapter
MouseMotionListener	MouseEvent	Component	mouseDragged mouseMoved	MouseMotionAdapter
WindowListener	WindowEvent	Window	windowActivated windowClosed windowClosing windowDeactivated windowDeiconified windowIconified windowOpened	WindowAdapter

TAB. 13.2 – Interfaces de `java.awt.event`, événements, composants et classes adaptateurs associés,

13.5 Essentiel des méthodes de `java.awt.Component`

Il n'est pas question de donner ici toutes les méthodes de cette classe, elles sont trop nombreuses ! Mais vous en retrouverez certaines presque systématiquement, et tous les composants que vous utilisez en héritent. En voici donc un catalogue réduit, pour plus de détail, vous savez où aller prospecter ! !

- `addKeyListener(KeyListener)` : inscrit le composant auprès d'un écouteur de touches,
- `addMouseListener(MouseListener)` : inscrit le composant auprès d'un écouteur de souris,
- `addMouseMotionListener(MouseMotionListener)` : inscrit le composant auprès d'un écouteur de déplacement de souris,
- `contains(int, int)` : vérifie si le composant contient le point dont les coordonnées sont passées en paramètre,
- `getBackground()` : donne la couleur (classe `Color`) de fond du composant,
- `getFont()` : donne la police de caractères du composant,
- `getForeground()` donne la couleur de premier plan,
- `getGraphics()` : crée un objet de la classe `Graphics` pour ce composant
- `getMaximumSize()` : taille maximale
- `getMinimumSize()` : taille minimale
- `getName()` : nom
- `paint(Graphics)` : dessine le composant,
- `paintAll(Graphics)` :dessine le composant, et tous les sous-composants,
- `removeKeyListener(KeyListener)` : retire l'inscription du composant, qui ne reçoit plus les événements de touches,

- `removeMouseListener(MouseListener)` : idem pour les évéments souris,
- `removeMouseMotionListener(MouseMotionListener)` : idem pour les évéments mouvements de souris,
- `repaint()` : redessine le composant,
- `setBackground(Color)` : spécifie la couleur de fond,
- `setFont(Font)` : spécifie la police,
- `setForeground(Color)` : spécifie la couleur de premier plan,
- `setVisible(boolean)` : cache ou affiche le composant selon la valeur du booléen passé en paramètre,

13.6 Un composant détaillé : le bouton

Voici le détail de la classe `Button` et ce que vous pouvez en faire. D'abord, n'oubliez pas que cette classe hérite de `java.awt.Component`, et à ce titre dispose des méthodes définies pour tous les composants.

- `public Button()` : constructeur
- `public Button(String label)` : constructeur d'un bouton avec un label,
- `public String getLabel()` : le label du bouton (null si le bouton n'a pas de label),
- `public synchronized void setLabel(String label)` : spécifie le label du bouton,
- `public void setActionCommand(String command)` : donne le nom de la commande à exécuter pour l'événement de type `ActionEvent` déclenché par le bouton, par défaut cette commende est le label du bouton,
- `public String getActionCommand()` : retourne le nom de la commande associéee à l'événement de tupe `ActionEvent` déclenché par le bouton,
- `public synchronized void addActionListener(ActionListener l)` : inscrit le bouton auprès d'un écouteur de `ActionEvent`,
- `public synchronized void removeActionListener(ActionListener l)` : supprime l'inscription du bouton.

Chapitre 14

Programmation concurrente : les *threads*

14.1 Introduction

Nous avons l'habitude d'écrire des programmes **séquentiels** : à partir d'un point de départ(la méthode `main` pour une application autonome), la machine exécute des instructions, les unes après les autres. Dans ce chapitre, les applications qui nous intéressent sont d'une autre nature, elles n'ont pas un unique *fil d'exécution* mais plusieurs, qui se déroulent en parallèle (ou du moins semblent le faire). On appelle *thread* un fil d'exécution, on parle aussi de processus léger. Les processus légers sont internes à une même application et partagent donc le même espace d'adressage, alors que les processus lourds (ou processus) sont gérés par le système d'exploitation qui leur alloue à chacun un espace de travail. La programmation concurrente suppose des mécanismes de **synchronisation** et d'**exclusion mutuelle**. Ces mécanismes sont présents dans le langage Java qui permet donc de programmer facilement l'exécution concurrente de plusieurs threads. Ce n'est pas le cas des langages classiques pour lesquels la programmation concurrente est une affaire de spécialistes !

Les threads se trouvent, à tout instant, dans un état particulier :
- instancié (`New Thread`),
- actif (`running`)
- endormi (`not running`),
- mort (`dead`).

14.2 Classe `Thread` et l'interface `Runnable`

14.2.1 Code

L'interface `java.lang.Runnable` doit être implémentée par toute classe dont les instances sont destinées à être exécutées par un thread. La classe doit alors définir la méthode `public abstract void run()`. Cette interface a été développée pour fournir un protocole commun pour les objets qui doivent exécuter du code pendant qu'ils sont actifs. C'est la méthode `run` qui contient les instructions à exécuter.

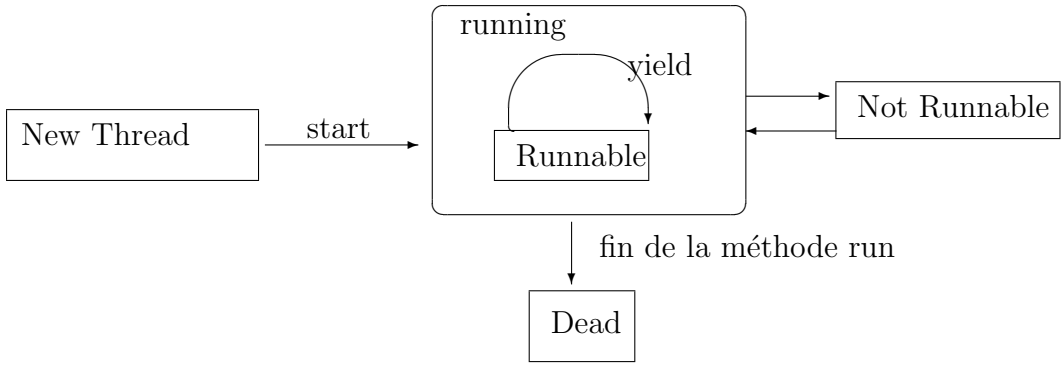

FIG. 14.1 – Cycle de vie d'un *Thread*

La création du filet d'exécution en parallèle consiste à exécuter la méthode `run` sur un objet particulier.

14.2.2 Création et exécution d'un thread sur un objet `Runnable`

Une fois définies les instructions à exécuter, il faut créer le filet d'exécution. Pour ce faire il faut créer un objet de la classe `java.lang.Thread`. Nous verrons la description de cette classe en détails un peu plus loin. Contentons nous pour l'instant de dire qu'elle dispose de plusieurs constructeurs, en particulier un constructeur prenant en argument un objet d'une classe implémentant l'interface `Runnable`. Attention, l'instanciation d'un thread n'implique pas le démarrage de son exécution (de l'exécution des instructions qui lui sont associées) : il existe mais n'est pas actif (cf figures 14.1 et 14.4).

```
class TesThread implements Runnable {
  ...
   public run(){
   ...
   }
}
...
TesThread test1 = new TesThread();
Thread leThread = new Thread(test1);
...
```

Pour rendre le thread **actif**, il faut le faire explicitement en invoquant la méthode `start()` de la classe `java.lang.Thread` :

```
leThread.start();
```

La méthode `start()` alloue les ressources nécessaires à l'exécution d'un thread et invoque la méthode `run()` de l'objet passé en paramètre lors de l'instanciation du thread. Rendre un thread actif ne signifie pas qu'il va s'exécuter en continu jusqu'à la fin. Il rejoint le groupe des threads actifs et le système se charge d'allouer régulièrement une tranche de temps pour qu'il puisse exécuter ses instructions.

Premier exemple :

```
class ThreadTest implements Runnable{
   String s;
   public ThreadTest(String s){this.s=s;}
   public void run() {
      while (true) System.out.println(s);
   }
}
public class TicTac {
   public static void main(String arg[]){
      ThreadTest tic=new ThreadTest("TIC");
      ThreadTest tac=new ThreadTest("TAC");
      Thread t1=new Thread(tic);
      Thread t2=new Thread(tac);
      t1.start();
      t2.start();
   }
}
```

On peut remarquer que, chaque fois que l'on instancie un objet `ThreadTest`, il faut créer un thread et le démarrer. Ceci peut donc faire partie du constructeur de la classe `ThreadTest` :

```
class ThreadTest implements Runnable{
   String s;
   Thread t;
   public ThreadTest(String s){
      this.s=s;
      t=new Thread(this);
      t.start();
   }
   public void run() {
      while (true) System.out.println(s);
   }
}
public class TicTac {
   public static void main(String arg[]){
      ThreadTest tic=new ThreadTest("TIC");
      ThreadTest tac=new ThreadTest("TAC");
   }
}
```

Ce programme est supposé afficher alternativement les chaînes TIC et TAC. Si vous l'exécutez, vous constaterez, selon le système d'exploitation, que seule la chaîne TIC est affichée, ou bien que la chaîne TAC n'est affichée qu'au bout d'un certain temps.

Les différentes JVM ne partagent pas toujours correctement le temps CPU alloué à chaque thread (cela dépend de la plateforme). Il faut donc faire cette gestion "à la main" en cédant le contrôle régulièrement dans les threads avec les méthodes `sleep()` ou `yield()`, permettant ainsi aux autres threads de s'exécuter.

14.2.3 Suspendre et redémarrer un thread

Il existe diverses situations où il est nécessaire de suspendre l'exécution d'un thread. La remarque de la section précédente est un premier cas où l'entrelaçage des différents

processus doit être spécifié "à la main". Dans le cas d'applets qui définissent des animations graphiques, celles-ci n'ont pas lieu d'être exécutées si la page HTML qui contient l'applet n'est temporairement pas visible.

Les méthodes `suspend` et `resume` : La méthode `suspend()` doit être utilisée avec précaution, car il faut s'assurer que l'application invoque la méthode `resume()` ou la méthode `stop()` pour rendre le thread actif ou pour le tuer. Ces méthodes sont *deprecated*.

La méthode `sleep` : Une solution pour suspendre l'exécution d'un thread consiste à l'endormir pendant un certain temps. Dans le cas d'une animation graphique, pour éviter un affichage trop rapide, on utilise la méthode `sleep(tps)` pour suspendre l'exécution pendants un laps de temps passé en paramètre et exprimé en millisecondes. La méthode `sleep` lance l'exception `InterruptedException` si le thread est stoppé pendant son sommeil. L'invocation de cette méthode doit donc gérer cette exception.

L'exemple tic-tac est modifié ci-dessous pour forcer chacun des threads à s'endormir régulièrement de façon que l'autre puisse s'exécuter :

```
class TestThread2 implements Runnable{
   String s;
   Thread t;
   public TestThread2(String s){
      this.s=s;
      t=new Thread(this);
      t.start();
   }
   public void run() {
      while (true) {
         System.out.println(s);
         try { t.sleep(100);}
         catch(InterruptedException e){}
      }
   }
}
public class TicTac3 {
   public static void main(String arg[]){
      TestThread2 tic=new TestThread2("TIC");
      TestThread2 tac=new TestThread2("TAC");
   }
}
```

14.2.4 Autre méthode : hériter de la classe `Thread`

L'interface `Runnable` permet à tout objet d'une classe qui l'implémente d'être la cible d'un thread. Une autre façon de faire est de définir une classe qui hérite de la classe `Thread`. La classe `java.lang.Thread` implémente l'interface `Runnable` avec une méthode `run()` vide. En redéfinissant la méthode `run()` dans la sous-classe, il est possible de définir les actions que l'on veut faire exécuter au thread.

```
class TestThread extends Thread{
   String s;
   public TestThread(String s){
```

```
      this.s=s;
   }
   public void run() {
      while (true) {
         System.out.println(s);
         try { sleep(100);}
         catch(InterruptedException e){}
      }
   }
}
public class TicTac4 {
   public static void main(String arg[]){
      TestThread tic=new TestThread("TIC");
      TestThread tac=new TestThread("TAC");
      tic.start();
      tac.start();
   }
}
```

La méthode `start()` de la classe `Thread` invoque immédiatement, après initialisation du thread, la méthode `run()`.

14.3 Gestion des threads : synchronisation et communication

Nous l'avons dit, les threads partagent les données entre eux. Il faut donc être très vigilant dans l'utilisation de ces objets partagés. C'est le programmeur qui doit gérer la synchronisation des différents threads.

14.3.1 Exclusion mutuelle : `synchronized`

Java permet de verrouiller un objet (pas une variable de type primitif) pour empêcher les accès concurrents. Lorsqu'une méthode d'instance qualifiée de `synchronized` est invoquée sur un objet par un thread, elle pose un *verrou* sur l'objet. Ainsi, si un autre thread invoque une méthode `synchronized`, elle devra attendre que le verrou soit relâché. Le verrou est relâché si :

- le code synchronisé a fini de s'effectuer,
- la méthode `wait` est invoquée, que nous verrons plus loin.

L'exemple ci-dessous décrit l'utilisation d'un même mégaphone par trois orateurs. Chaque orateur attendra la fin de l'utilisation du mégaphone par le précédent. L'objet de la classe `Megaphone` est vérouillé dès qu'il est pris par un thread `Orateur`, même si celui-ci n'utilise pas le mégaphone à plein temps (voir l'instruction `t.sleep(100)`).

```
class Megaphone { // classe qui décrit un mégaphone
    synchronized void parler(String qui, String quoi, Thread t) {
        // méthode synchronized car si un orateur utilise le mégaphone
        // ce dernier n'est pas disponible pour un autre orateur
        for (int i=0; i<10; i++){
```

```
            System.out.println(qui+" affirme : "+quoi +i+"ème fois ");
            try{t.sleep(100);}
            catch(InterruptedException e){}
        }
    }
}

class Orateur extends Thread { // classe qui décrit un orateur
    String nom, discours;
    Megaphone m;
    public Orateur(String s, String d, Megaphone m){
       nom=s;    discours=d;
       this.m=m;
    }
    public void run(){
        m.parler(nom,discours,this);
    }
}

class Reunion {
    public static void main(String[] arg){
    Megaphone m=new Megaphone();
    Orateur o1=new Orateur("Orateur 1","je suis le premier !",m);
    Orateur o2=new Orateur("Orateur 2","je suis le deuxième !",m);
    Orateur o3=new Orateur("Orateur 3","je suis le troisième !",m);
    o1.start();o2.start();o3.start();
    }
}
```

Une méthode d'instance `synchronized` pose un verrou sur l'objet par lequel elle a été invoquée. Une méthode **statique** (de classe) peut être qualifiée de `synchronized`. Elle pose alors un verrou sur la classe et ainsi, deux méthodes statiques `synchronized` ne peuvent être exécutées en même temps. Mais **attention**, il n'y a aucun lien entre les *verrous de classes* et les *verrous d'instances*. Une classe verrouillée (par une méthode statique `synchronized`) n'empêche pas l'exécution d'une méthode d'instance `synchronized` et inversement.

Avec le mot clé `synchronized` on a vu comment verrouiller une instance sur toute une méthode. En fait, il est possible de ne verrouiller qu'une partie du code d'une méthode. L'instruction comporte alors deux parties : l'objet à verrouiller et la ou les instructions à exécuter :

```
synchronized(objet)
   instruction-simple-ou-bloc-d'instructions
```

L'exemple qui suit reprend le cas du mégaphone, qui signale quand un orateur a demandé à l'utiliser :

```
class Megaphone { // classe qui décrit un mégaphone
     void parler(String qui, String quoi, Thread t) {
         System.out.println("mégaphone demandé par orateur "+qui);
         synchronized(this){
```

```
         // c'est seulement pour parler que le mégaphone est verrouillé
            for (int i=1; i<=10; i++){
                System.out.println(qui+" affirme : "+quoi);
                try{ t.sleep(100);}
                catch(InterruptedException e){ }
            }
         }
     }
}
class Orateur extends Thread { // classe qui décrit un orateur
    String nom, discours;
    Megaphone m;
    int sommeil;
    public Orateur(String s, String d, Megaphone m){
       nom=s;    discours=d;
       this.m=m;
    }
    public void run(){
        m.parler(nom,discours,this);
    }
}
class Reunion {
    public static void main(String[] arg){
    Megaphone m=new Megaphone();
    Orateur o1=new Orateur("Orateur 1","je suis le premier ! ",m);
    Orateur o2=new Orateur("Orateur 2","je suis le deuxième ! ",m);
    Orateur o3=new Orateur("Orateur 3","je suis le troisième ! ",m);
    o1.start();o2.start();o3.start();
    }
}
```

14.3.2 Synchronisation entre threads : `wait`, `notify`, `notifyAll`

Il s'agit de faire coopérer des threads. La méthode `wait` suspend l'exécution d'un thread, en attendant qu'une certaine condition soit réalisée. La réalisation de cette condition est signalée par un autre thread par les méthodes `notify` ou `notifyAll`. Ces trois méthodes, dont les prototypes sont donnés ci-après, sont définies dans la classe `java.lang.Object` et sont donc héritées par toute classe.

```
public final void wait() throws InterruptedException
public final native void notify()
public final native void notifyAll()
```

Lorsque la méthode `wait` est invoquée à partir d'une méthode `synchronized`, en même temps que l'exécution est suspendue, le verrou posé sur l'objet par lequel la méthode a été invoquée est relâché. Dès que la condition de réveil survient, le thread attend de pouvoir reprendre le verrou et continuer l'exécution. Notez qu'une autre version de `wait` prend en argument un entier de type `long` qui définit la durée d'attente maximale (en millisecondes). Si ce temps est dépassé, le thread est *réveillé*.

La méthode `notify` réveille un seul thread. Si plusieurs threads sont en attente, c'est celui qui a été suspendu le plus longtemps qui est réveillé. Lorque plusieurs threads sont

en attente et qu'on veut tous les réveiller, il faut utiliser la méthode `notifyAll`. L'exemple qui suit est une adaptation du précédent avec des orateurs qui interrompent leur discours de temps en temps et libèrent le mégaphone pour les orateurs en attente. Voilà une réunion plus conviviale!!

```
class Megaphone {
     synchronized void parler(String qui, String quoi, Thread t) {
 System.out.println("mégaphone demandé par orateur "+qui);
         for (int i=1; i<=10; i++){
     System.out.println(qui+" affirme : "+quoi);
             notifyAll();     // libère le mégaphone
     try{wait();}  // se met en attente
     catch(InterruptedException e){ }
 }
     }
}
```

L'exemple qui suit est celui classique du producteur et du consommateur qui produisent et consomment dans un même buffer :

```
import java.util.*;
class Buffer extends Stack {
    public synchronized void poser(char donnee) {
       // attendre tant que le buffer est plein
       while (full()) {
          try {
               wait();  // mise en attente
          }
          catch(Exception e) {}
       }
       // au moins une place libre
       push(new Character(donnee));
       notify(); // fin de mise en attente
    }
    public synchronized char prendre(){
       // attendre tant que le buffer est vide
       while (empty()){
          try{
             wait(); // mise en attente
          }
          catch(Exception e){}
       }
       notify();
       return ((Character)pop()).charValue();
    }
    public boolean full() {return (size()==2);}
    public boolean empty() {return (size()==0);}
}
class Producteur extends Thread {
    private Buffer buffer;
    private String donnee;
    public Producteur(Buffer buffer, String donnee) {
```

```
       this.buffer=buffer;
       this.donnee=donnee;
    }
    public void run(){
       for (int i=0;i<donnee.length();i++) {
          // produire les donnees
          buffer.poser(donnee.charAt(i));
          try {
               // rythme de production aleatoire
               sleep((int) (Math.random()*25));
          }
          catch(Exception e){}
       }
       System.out.println("\nProduction terminee");
   }
}
class Consommateur extends Thread {
    private Buffer buffer;
    private int nombre;
    public Consommateur(Buffer buffer, int nombre) {
       this.buffer=buffer;
       this.nombre=nombre;
    }
    public void run(){
       for (int i=0;i<nombre;i++) {
          // consommer les donnees
          char car= buffer.prendre();
          System.out.print(car);
          try {
              // rythme de consommation aleatoire
              sleep((int) (Math.random()*100));
          }
          catch(Exception e){}
       }
       System.out.println("\nConsommation terminee");
    }
}
public class ProdCons {
    public static void main(String arg[]) {
       String donnee="Java est un langage merveilleux !";
       Buffer buffer=new Buffer();
       Producteur producteur = new Producteur(buffer,donnee);
       Consommateur consommateur=new Consommateur(buffer,donnee.length());
       producteur.start();
       consommateur.start();
    }
}

/***** exemples d'execution *****
gbm-server:~/coursJava/Thread> java ProdCons
Jaa est un langage merveilleux
Production terminee
!v
Consommation terminee
```

```
gbm-server:~/coursJava/Thread> java ProdCons
Jva est un langage merveilleux
Production terminee
!a
Consommation terminee
```

14.3.3 Stopper un thread

Lorsqu'une application `Java` démarre, un premier thread s'exécute ; c'est le thread principal, qui démarre l'exécution de la méthode `main` dans le cas d'applications autonomes. Lorsque la méthode `main` est terminée, et si aucun autre thread n'a été créé, l'application s'arrête. Mais si d'autres threads ont été créés, l'application attend la fin de leur exécution pour s'arrêter(sauf pour les threads démons, dont nous ne parlerons pas ici).

Un thread termine son exécution lorsque toutes les instructions de sa méthode `run` ont été exécutées. Mais, souvent, cette méthode est conçue pour tourner indéfiniment (c'est le cas par exemple dans les applications graphiques). Pour arrêter des threads (actifs ou endormis) :

- utiliser une variable booléenne (c'est la méthode conseillée) :
  ```
  void run(){
     while (! stopperThreads) {...}
  }
  ```
- invoquer la méthode `stop` de la classe `Thread`, cette méthode `public final void stop()` force le thread à arrêter son exécution. Attention, cette méthode est peu sûre, elle a donc été supprimée des nouvelles versions de Java (à partir de 1.2). Il est permis de stopper un thread qui n'a pas encore démarré. La méthode lance une erreur `ThreadDeath` (dérivée de `java.lang.Error`), mais en général on ne la rattrape pas.

Notez qu'il est important d'arrêter les threads en cours d'exécution, car ils consomment des ressources système.

14.3.4 Un exemple de gestion de threads

L'exemple ci-dessous est tiré du support de cours de I.Charon (http ://www.infres.enst.fr/charon/coursJava/). Notez que les méthodes `suspend` (suspension d'un thread permettant une reprise au point d'arrêt), `resume` (reprise d'un thread arrêté par `suspend`) et `stop` sont *deprecated* (c'est-à-dire qu'elles ne sont conservées que par soucis de compatibilité, mais elles ne font plus partie des nouvelles versions) depuis la version 1.2 du langage.

La méthode `isAlive` retourne `true` si le thread a été démarré et n'est pas arrêté. Si elle retourne `false`, le thread est soit un *New Thread* soit il est *Dead*.

```
import java.awt.*;
import java.awt.event.*;
import java.util.*;

class Ecouteur extends WindowAdapter { // pour les evenements window
    private Component fenetre;
    public Ecouteur(Component f) {
        fenetre = f;
    }
    public void windowClosing(WindowEvent e) {
```

```
        if (e.getSource()==fenetre){
            System.exit(0);
        }
    }
}

class RondsConcentriques extends Thread {
  int r=10;
  int debut;
  Component fenetre;
  boolean continuer;
  boolean finir;
  RondsConcentriques(int debut, Component fenetre) {
    this.debut = debut;
    this.fenetre = fenetre;
  }
  void suspendre() {
    continuer = false;
  }
  synchronized void reprendre() {
    continuer = true;
    notify();
  }
  synchronized void stopper() {
    finir = true;
    notify();
  }
  public void run() {
    Graphics g = fenetre.getGraphics();
    continuer = true;
    finir = false;
    for (int i = 0; i < 50; i++) {
        try {
            sleep(200);
            synchronized(this) {
                while (!continuer && !finir) wait();
            }
        }
        catch (InterruptedException exc) {}
        if (finir) break;
        g.setColor(new Color((debut+528424*i)%Integer.MAX_VALUE));
        g.drawOval(250-r, 250-r,2*r,2*r);
        r += 2;
    }
  }
}

class EssaiGestionThread extends Panel implements ActionListener {
  RondsConcentriques thread = null;
  Random alea;
  Button tracer = new Button("tracer");
  Button pauser = new Button("pauser");
  Button stopper = new Button("stopper");
  Button effacer = new Button("effacer");
```

```
  EssaiGestionThread() {
      tracer.addActionListener(this);
      pauser.addActionListener(this);
      stopper.addActionListener(this);
      effacer.addActionListener(this);
      add(tracer);
      add(pauser);
      add(stopper);
      add(effacer);
      alea = new Random((new Date()).getTime());
      setVisible(true);
  }
  public void actionPerformed(ActionEvent evt) {
      Object source = evt.getSource();
      if (source == tracer) {
          int debut= (int)Math.abs(alea.nextLong());
          if ((thread == null)||(!thread.isAlive())) {
              thread = new RondsConcentriques(debut, this);
              thread.start();
          }
          else thread.reprendre();
      }
      else if ((source == pauser) && (thread != null))
          thread.suspendre();
      else if (source == stopper)  {
          if (thread != null) thread.stopper();
          thread = null;
      }
      else if (source == effacer) repaint();
  }
  public static void main(String[] argv) {
    Frame f = new Frame();
    f.addWindowListener(new Ecouteur(f));
    f.setSize(500,500);
    f.add(new EssaiGestionThread());
    f.setVisible(true);
  }
}
```

14.4 Grouper des threads et priorités

Chaque thread appartient à un groupe. Par défaut, les threads font partie du même groupe que le thread qui les ont créés. La classe `ThreadGroup` permet d'organiser les threads en groupes. Lors de la création d'un thread, on peut spécifier à quel groupe il appartient (cf. constructeur de la classe `Thread`).

A chaque thread est associée une priorité (entier compris entre les valeurs `MIN_PRIORITY` et `MAX_PRIORITY` de la classe `Thread`. Lorsque plusieurs threads sont démarrés, celui de plus forte priorité s'exécute d'abord. S'il existe deux threads de même priorité en attente d'exécution, Java choisit l'un d'entre eux et l'exécute jusqu'à ce que :

- un thread de plus grande priorité est en attente,
- le thread en cours d'exécution se termine,

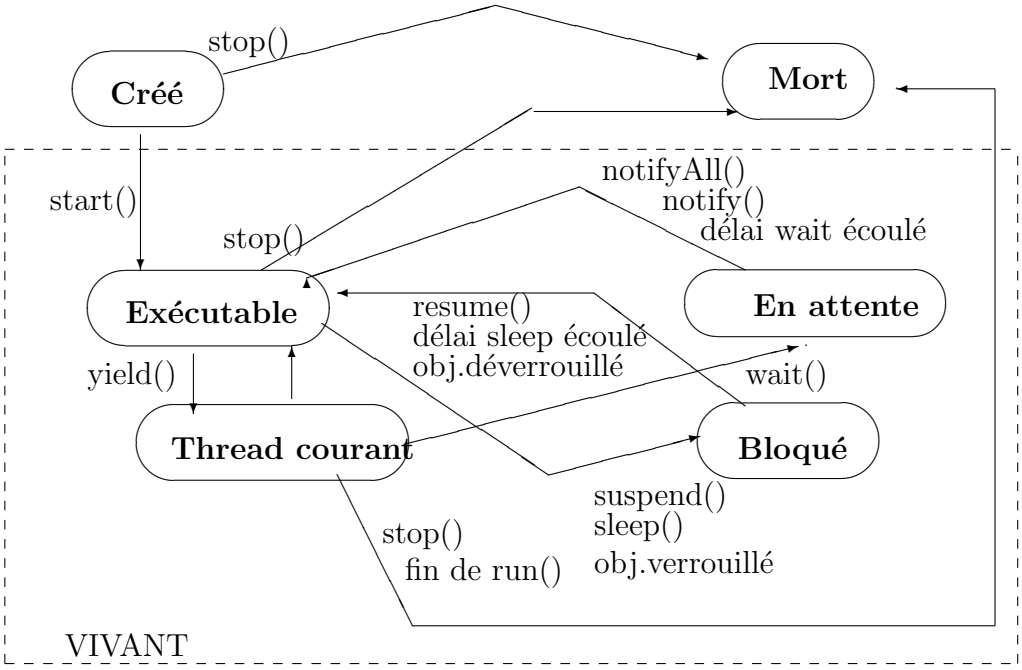

FIG. 14.2 – Etats d'un *Thread*

– le thread en cours d'exécution demande un partage du temps d'exécution (`sleep` ou encore `yield()`).

Lorsque l'on crée un nouveau thread, il hérite par défaut de la priorité du thread qui le crée. Cette priorité peut être modifiée en utilisant la méthode `setPriority`.