Introduction à la programmation

Karl Tombre

Cours de tronc commun, 1re année

Version 1.3

Ce cours a pour objectif de vous initier à l'informatique, et en tout premier lieu à la programmation. Le langage « support » choisi est Java, et vous aurez donc l'occasion d'écrire des programmes dans ce langage. Mais avant tout, nous souhaitons vous familiariser avec les concepts et principes fondamentaux de la programmation ; de ce fait, vous devriez être capable, à l'issue de ce cours, de vous familiariser assez rapidement avec un autre langage de programmation que Java.

Le cours est basé sur plusieurs livres, dont vous trouverez les références en fin de polycopié. Le langage Java fait l'objet d'une abondante littérature, de qualité parfois inégale. De plus, il peut encore évoluer. En complément d'un livre, il peut donc être opportun de garder un signet sur le site de référence, `http://java.sun.com/`, où vous trouverez entre autres des tutoriaux en ligne, dans des versions parfois plus récentes que leurs versions imprimées. Notez bien toutefois que la plupart des livres sur le marché sont destinés à un public d'informaticiens, qui savaient déjà programmer avant d'apprendre Java...

Enfin, je vous invite à garder un signet sur ma propre page web à l'école[1], où je regroupe au fur et à mesure des informations plus ou moins directement liées à ce cours et des pointeurs sur des ressources Internet intéressantes. Vous y trouverez notamment un pointeur sur les *API* de Java, c'est-à-dire la documentation en ligne de l'ensemble des classes disponibles dans l'environnement Java.

Une page spécifique[2] regroupe les informations pratiques sur le déroulement du cours : horaires, énoncés et corrigés des TDs, groupes, programme des séances, etc.

© Karl Tombre, École des Mines de Nancy. Document édité avec XEmacs et formatté avec LaTeX.
Achevé d'imprimer le 2 septembre 2003. Un grand merci à tous ceux, collègues ou étudiants, qui, au fil des éditions successives de ce polycopié, ont contribué à l'améliorer par leurs relectures et leurs nombreuses suggestions d'amélioration, et en particulier à Philippe Dosch, Jacques Jaray, Luigi Liquori, Bart Lamiroy et (*last, but not least*) Guillaume Bonfante.

[1] `http://www.mines.inpl-nancy.fr/~tombre`
[2] `http://www.mines.inpl-nancy.fr/~tombre/java.html`

Table des matières

Chapitre 1

Introduction

L'informatique est avant tout une science de l'abstraction — il s'agit de créer le bon modèle pour un problème et d'imaginer les bonnes techniques automatisables et appropriées pour le résoudre. Toutes les autres sciences considèrent l'univers tel qu'il est. Par exemple, le travail d'un physicien est de comprendre le monde et non pas d'inventer un monde dans lequel les lois de la physique seraient plus simples et auxquelles il serait plus agréable de se conformer. À l'opposé, les informaticiens doivent créer des abstractions des problèmes du monde réel qui pourraient être représentées et manipulées dans un ordinateur.

Aho & Ullman [1]

1.1 Informatique = mécanisation de l'abstraction

Un *programme* est une suite d'instructions définissant des opérations à réaliser sur des données. Les instructions du programme sont exécutées les unes après les autres, le plus souvent dans l'ordre séquentiel dans lequel elles sont données dans le programme – on dit que le flot d'exécution, ou flot de contrôle, est séquentiel. Pour écrire des programmes, on se sert d'une notation, appelée *langage de programmation*. Les langages de programmation les plus rudimentaires sont ceux qui « collent » au jeu d'instructions de l'ordinateur ; on les appelle les langages d'assemblage (cf. annexe A). Un langage d'assemblage est par définition propre à un processeur donné (ou au mieux à une famille de processeurs) ; l'évolution de la programmation vers des projets à la fois complexes et portables a créé le besoin de langages de plus haut niveau. Ces langages permettent de gérer la complexité des problèmes traités grâce à la *structuration*, qu'on définira dans un premier temps, de manière très générale, comme le regroupement d'entités élémentaires en entités plus complexes (enregistrements regroupant plusieurs données, modules ou procédures regroupant plusieurs instructions élémentaires...).

Un bon programme doit être facile à écrire, à lire, à comprendre et à modifier. Il faut savoir que la mémoire humaine est limitée par le nombre plus que par la taille des structures à manipuler. Il devient donc vite difficile pour un programmeur de maîtriser de nombreuses lignes de code, si celles-ci ne sont pas regroupées en modules ou autres structures.

Les avantages de l'utilisation d'un langage de haut niveau sont multiples :

- Un programme n'est pas seulement destiné à être lu une fois ; plusieurs intervenants risquent de se pencher sur lui au cours de son existence, d'où la nécessité de la lisibilité.
- L'existence de langages évolués a permis la création de logiciels par des programmeurs qui n'auraient jamais appris un langage d'assemblage. Il faut noter à ce propos la puissance et la popularité croissante des langages dits de scriptage (Visual Basic, TCL/TK...) et des générateurs de programmes de gestion.
- Un programme en langage évolué peut être « porté » sur différentes architectures et réutilisé au fil de l'évolution du matériel.
- Un module clairement spécifié peut être réutilisé comme une brique de construction, dans divers contextes.

– Un langage peut choisir de laisser transparaître la machine sous-jacente, ou au contraire fournir un autre modèle de calcul (cf. parallélisme par exemple, même s'il y a un seul processeur physique, ainsi que la programmation fonctionnelle ou la programmation logique).
– Les langages de programmation offrent des outils pour l'abstraction, la modélisation et la structuration des données et des opérations. Ils permettent aussi la *vérification de type*, qui évite beaucoup d'erreurs (elle permet par exemple de refuser un programme qui voudrait additionner 1 et `"z"`).

1.2 Traduction

À partir du moment où l'on utilise un langage différent de celui de la machine, il devient nécessaire de lui traduire les programmes écrits dans ce langage. Deux approches sont possibles pour cela. La première consiste à lancer un programme de traduction simultanée, appelé *interprète*, qui traduit et exécute au fur et à mesure les instructions du programme à exécuter. La deuxième approche consiste à analyser l'ensemble du programme et à le traduire d'avance en un programme en langage machine, qui est ensuite directement exécutable. C'est la *compilation*.

Java fait partie des langages qui choisissent une approche hybride : un programme en Java est analysé et compilé, mais pas dans le langage de la machine physique. Pour des raisons de portabilité, le compilateur Java traduit dans un langage intermédiaire, universel et portable, le *byte-code*, qui est le langage d'une machine virtuelle, la JVM (*Java Virtual Machine*). Cette « machine » est exécutée sur une machine physique et un interprète le *byte-code*.

Pour traduire un langage, il faut tenir compte de ses trois niveaux :
– le niveau *lexical*, qui concerne le vocabulaire du langage, les règles d'écriture des mots du langage (identificateurs de variables ou fonctions), les mots clés, c'est-à-dire les éléments de structuration du discours, et les caractères spéciaux, utilisés par exemple pour les opérateurs ;
– le niveau *syntaxique*, qui spécifie la manière de construire des programmes dans ce langage, autrement dit les règles grammaticales propres au langage ;
– le niveau *sémantique*, qui spécifie la signification de la notation.

1.3 La programmation : du problème au programme

Nous attaquons ici toute la problématique de la programmation. On peut distinguer les étapes suivantes dans la vie d'un logiciel [1] :
– Définition du problème et spécification, incluant éventuellement plusieurs itérations de spécification avec les futurs utilisateurs du logiciel, un prototypage du produit final, et une modélisation des données.
– Conception, avec création de l'architecture de haut niveau du système, réutilisant si possible des composants déjà disponibles.
– Réalisation des composants et test de chacun d'eux pour vérifier qu'ils se comportent comme spécifié.
– Intégration et test du système : une fois chaque composant testé et validé, encore faut-il que le système intégré le soit.
– Installation et test « sur le terrain », c'est-à-dire en situation réelle.
– Maintenance : elle représente souvent plus de la moitié du coût de développement. Le système doit être purgé d'effets imprévisibles ou imprévus, ses performances doivent être corrigées ou améliorées, le monde réel dans lequel il est implanté n'étant pas immuable, il faut modifier le programme ou lui ajouter de nouvelles fonctionnalités, le système doit souvent être porté sur de nouvelles plateformes physiques, etc. Tous ces aspects de maintenance soulignent l'importance d'écrire des programmes lisibles, corrects, robustes, efficaces, modifiables et portables !

Nous voyons donc que l'activité d'analyse et de programmation, bien qu'elle soit au cœur de notre cours, ne représente qu'une des étapes dans le cycle logiciel. Ceci étant, nous allons essentiellement nous arrêter sur les phases qui permettent d'aller d'un problème bien spécifié à un programme, en passant par la réalisation d'un algorithme précis, c'est-à-dire de la conception d'une

démarche opératoire pour résoudre le problème donné, cette démarche étant ensuite transcrite dans un langage de programmation, en l'occurrence Java.

Dans ce cours, nous adoptons une démarche pragmatique, l'apprentissage des différentes notions se faisant par l'exemple, en l'occurrence le développement tout au long des chapitres d'un exemple de programme de gestion bancaire, que nous allons peu à peu enrichir de nouvelles fonctionnalités, qui illustrent les notions nouvelles introduites à chaque fois. Plutôt que de remplir de longues pages sur la syntaxe et les règles de programmation, nous avons préféré mettre beaucoup d'exemples complets de programmes, comptant sur votre capacité d'apprentissage par analogie.

Le chapitre 2 donne les premières bases nécessaires pour écrire un programme Java élémentaire. Le chapitre 3 introduit les outils fondamentaux de structuration des données et des instructions, à savoir les classes d'un côté et les fonctions et procédures de l'autre. Au chapitre 4, nous revenons sur la classe, mais pour introduire les bases de la programmation objet, méthodologie devenue quasiment incontournable à l'heure actuelle. Le chapitre 5, quant à lui, vous présente quelques exemples de modèles de données et la manière dont ils peuvent être représentés et manipulés. Il resterait tant à étudier, et le temps nous fait défaut ; j'ai donc réuni au chapitre 6 un ensemble probablement un peu disparate d'éléments supplémentaires, qui nous permettront cependant de compléter la partie programmation de ce cours par un programme qui dépasse légèrement la banalité habituelle des exercices d'école. Enfin, pour que vous n'ayez pas l'impression que l'informatique se résume à la programmation, nous apportons également au chapitre 7 un court éclairage sur le monde des bases de données.

J'ai choisi de reporter en annexes plusieurs ensembles d'informations utiles en soi, mais dont la présence directe dans le fil du polycopié aurait nui à la progression de la lecture. Je vous invite cependant à vous reporter aussi souvent que nécessaire à ces annexes. Vous y trouverez en particulier un glossaire (annexe C) que je vous conseille de consulter tout au long de votre lecture. Vous trouverez peut-être également intéressant de recourir parfois à l'aide-mémoire de programmation (annexe E) lors de vos séances de TD.

Chapitre 2

Les constructions de base en Java

... conduire par ordre mes pensées, en commençant par les objets les plus simples et les plus aisés à connaître, pour monter peu à peu, comme par degrés, jusques à la connaissance des plus composés ; et supposant même de l'ordre entre ceux qui ne se précèdent point naturellement les uns les autres.

Descartes

2.1 Constantes et variables

La manière la plus simple de manipuler des valeurs dans un programme est d'écrire directement ces valeurs. Ainsi, on pourra écrire `24` pour désigner un nombre entier, `176.54` pour désigner le nombre réel correspondant, `2.54E+12` pour désigner le nombre 2.54×10^{12}, `'a'` pour indiquer un caractère, `"Jean-Paul"` pour désigner une chaîne de caractères, et `true` pour indiquer la valeur booléenne « vrai ».

Toutes ces valeurs sont des *constantes*. Je peux par exemple demander à un programme d'afficher le résultat de l'opération `1+1`. Le seul problème est bien entendu que si je souhaite effectuer une autre opération, il faudra modifier le programme, par exemple en écrivant `1+2`. Or l'un des intérêts de la programmation est justement de pouvoir décrire une démarche opératoire qui reste la même alors que les données du problème peuvent changer. Par exemple, il est clair que l'opération « créditer un compte bancaire » correspond au même algorithme et au même programme, quels que soient le solde initial du compte et le montant dont on doit le créditer.

Il est donc indispensable de ne pas rester limité à l'emploi de constantes dans les programmes informatiques. D'un point de vue algorithmique, la *variable* est un objet qui a un nom (l'identificateur de la variable) et, au cours de sa vie, une valeur à chaque instant t donné. Concrètement, au sein de l'ordinateur, la variable va correspondre à un emplacement mémoire dans lequel on peut stocker une valeur. Quand on écrit un programme, le traducteur (compilateur ou interprète) associe à chaque variable l'adresse de l'emplacement mémoire correspondant et réalise l'adressage indirect nécessaire pour accéder en lecture ou en écriture à cet emplacement.

Il est bon de prendre l'habitude dès le début de donner à ses variables des noms mnémotechniques. Bien entendu, pour l'ordinateur, il est strictement équivalent que vous appeliez vos variables `a, b, c, d, i1, i2, i3` ou `soldeDuCompte, montantACrediter, ageDuCapitaine, nomDuClient` ; mais vous comprenez aisément que lorsqu'un programme dépasse quelques dizaines de lignes, il est beaucoup plus facile pour un être humain de comprendre, de modifier et de maintenir un programme utilisant le deuxième type de noms de variables.

Revenons aux constantes. Nous avons vu qu'une valeur constante peut être écrite telle quelle dans un programme, par exemple `19.6` ou `"Bonjour"`. Mais dans bien des cas, il est préférable de donner un nom symbolique aux constantes également, comme pour les variables. Pourquoi ? Prenons l'exemple d'un calcul de prix TTC. À l'heure où j'écris ces lignes, le taux de TVA standard est de 19,6%. Si donc je fais tous mes calculs de prix TTC dans mon programme en multipliant les variables désignant le prix hors taxe par `1.196`, je serai dans le pétrin le jour où le gouvernement relève ou baisse le taux de la TVA. Il me faudra chercher toutes les occurrences du nombre `1.196`

dans le programme, vérifier qu'elles correspondent bien à un calcul de TVA, et faire les modifications. Il est donc bien plus judicieux de désigner la constante par un nom, comme pour les variables. C'est ce que permet le mot clé `final`, grâce auquel on écrira :

```
final double tauxTVA = 1.196;
```

pour indiquer que `tauxTVA` est une constante de type réel (cf. § 2.3). Tous les calculs se feront avec cette constante nommée, et quand le taux de TVA changera, il y aura une seule ligne à modifier dans le programme... De même, en cas de programme multilingue, on pourra par exemple écrire :

```
final String salutation = "Bonjour";
```

ou au contraire

```
final String salutation = "Guten Tag";
```

ou

```
final String salutation = "Buenos dias";
```

2.2 Typage

Le type d'une expression ou d'une variable indique le domaine des valeurs qu'elle peut prendre et les opérations qu'on peut lui appliquer. En Java comme dans la plupart des langages évolués, toute variable doit être déclarée avec un *type* donné. Ce peut être un des types de base du langage, ou un type construit avec les outils de structuration fournis par le langage. Toute expression valide a aussi un type, et le résultat d'une opération doit *a priori* être affecté à une variable du type correspondant.

Il est relativement aisé de donner une interprétation ensembliste à la notion de type. Nous aurons l'occasion de le faire plus d'une fois, en particulier quand nous aborderons la notion de classe (cf. § 3.1 et 4.1).

- On dispose d'ensembles de base, auxquels correspondent en Java des types de bases[1] :
  - $\mathbb{R}$: `double`, `float`
  - $\mathbb{Z}$: `byte`, `short`, `int`, `long`
  - *Bool* : `boolean`
  - caractères : `char`
- On peut définir de nouveaux ensembles : si on a besoin d'un ensemble de couleurs, on pourra écrire quelque chose du genre `class Couleur`, comme nous le verrons ultérieurement. Cette capacité à définir de nouveaux ensembles ou types existe dans la grande majorité des langages de programmation.
- On peut définir des produits d'ensemble, comme par exemple $Couleur \times Bool$ ou $\mathbb{Z} \times \mathbb{Z} \times \mathbb{R}$. Nous verrons que les *classes* permettent en particulier ce genre de construction.
- On peut définir des fonctions, comme par exemple $f : \mathbb{Z}_! \times \mathbb{Z} \rightarrow Bool$, qui s'écrira en Java `boolean f(int a, int b)`.
- Grâce aux listes et autres collections, on peut définir des séquences d'objets.

La plupart des langages évolués offrent les constructions nécessaires pour mettre en œuvre les opérations ensemblistes de base que sont le produit de deux ensembles (et l'accès aux fonctions de projection associées), les fonctions et opérateurs, et la séquence ordonnée ou non d'éléments d'un ensemble.

2.3 Types élémentaires en Java

Le langage Java définit un certain nombre de types élémentaires ; les variables déclarées avec ces types correspondent des « cases mémoire » en adressage direct. La taille en mémoire de ces cases est définie par le type.

[1] *Stricto sensu*, les types Java entiers et réels représentent bien entendu des sous-ensembles de $\mathbb{Z}$ et de $\mathbb{R}$, du fait de la représentation discrète et bornée des nombres en informatique (cf. § F.2 et F.3).

- `boolean` : booléen, peut valoir `true` ou `false`
- `byte` : entier sur 8 bits [-128, 127]
- `char` : caractère Unicode (cf. § F.1) codé sur 16 bits
- `short` : entier codé sur 16 bits [-32 768, 32 767] (cf. § F.2)
- `int` : entier codé sur 32 bits
- `long` : entier codé sur 64 bits
- `float` : réel codé sur 32 bits, au format IEEE 754 (cf. § F.3)
- `double` : réel double précision codé sur 64 bits, au format IEEE 754

Voici quelques exemples de déclarations et d'initialisations valides en Java :

```
boolean b = true;
int i = 3;
b = (i != 0);
char c = 'A';
char newline = '\n';
char apostrophe = '\'';
char delete = '\377';
char aleph = '\u05D0';
long somme = 456L;
float val = -87.56;
float cumul = 76.3f;
double pi = 3.14159;
double large = 456738.65D;
double veryLarge = 657E+234;
```

Attention : comme l'indiquent les exemples ci-dessus, en Java, toute variable doit être déclarée (avec indication de son type) une et une seule fois, avant d'être utilisée.

Exercice 1 *Quelles sont les erreurs de syntaxe dans les lignes qui suivent ?*

```
boolean b;
double a2 = 5.2;
int k = 5;
int j = k;
int i = 7;
char a = i;
int k = 7;
int l = t;
int t = 7;
```

2.4 Expressions

Java offre un jeu riche d'opérateurs pour écrire toutes sortes d'expressions arithmétiques et logiques. Des règles de priorité entre opérateurs permettent d'écrire les expressions de la manière la plus « naturelle » possible ; par exemple, `a + b * c` équivaut à `a + (b * c)` et non à `(a + b) * c`. Ceci étant, en cas de doute, rien ne vous empêche de parenthéser vos expressions.

2.4.1 Opérateurs arithmétiques

Les opérateurs arithmétiques unaires sont `+` et `-` ; par exemple, `-a` équivaut à `-1 * a`. Les opérateurs arithmétiques binaires sont présentés dans le tableau ci-après.

Op.	Exemple	Description / remarques
+	`a + 5`	addition – ou concaténation de chaînes dans le cas particulier de String (cf. § 4.4)
-	`b - c`	soustraction
*	`13 * x`	multiplication
/	`a / b`	division – attention, si les opérandes sont entiers, il s'agit de la division euclidienne ! Ceci est une source connue d'erreurs chez les débutants.
%	`12 % 5`	modulo (reste de la division euclidienne)

Exercice 2 *Donner la valeur des variables après les instructions suivantes :*

```
int i = 13;
int j = 5;
i = i % 5;
j = j * i + 2;
int k = (i * 7 + 3) / 5;
```

2.4.2 Opérateurs logiques et relationnels

Tout d'abord, si vous n'avez jamais vu l'algèbre de Boole dans vos études, je vous conseille de lire au préalable l'annexe B, qui vous donne une courte introduction à la notion même d'opérations et de fonctions logiques.

Un opérateur relationnel compare deux valeurs. Les opérateurs logiques permettent d'opérer sur des valeurs logiques, habituellement obtenues grâce à l'application d'opérateurs relationnels.

Les opérateurs relationnels de Java sont les suivants :

Op.	Exemple	Description / remarques
>	`a > 0`	$a > 0$
>=	`b >= c`	$b \geq c$
<	`a < 0`	$a < 0$
<=	`a <= b`	$a \leq b$
==	`a == b`	test d'égalité $a = b$ – attention, ne pas confondre avec *l'affectation* d'une valeur à une variable (cf. § 2.5), c'est une source d'erreur fréquente, qui peut avoir des conséquences catastrophiques !
!=	`a != b`	test d'inégalité $a \neq b$
instanceof	`a instanceof C`	test d'appartenance à une classe (cf. p. 95).

Les opérateurs logiques disponibles sont les suivants :

Op.	Exemple	Description / remarques
&&	`(a>0) && (b<0)`	*et* logique – le deuxième opérande n'est évalué que si le premier opérande est vrai
||	`(a>0) || (b<0)`	*ou* logique – le deuxième opérande n'est évalué que si le premier opérande est faux
!	`!(b >= c)`	*non* logique
&	`(a>0) & (b<0)`	*et* logique – les deux opérandes sont toujours évalués – *Utilisation déconseillée*
|	`(a>0) | (b<0)`	*ou* logique – les deux opérandes sont toujours évalués – *Utilisation déconseillée*

Je vous conseille de prendre l'habitude d'utiliser par défaut les versions `&&` et `||` des opérateurs *et/ou*.

Java offre également un opérateur à trois opérandes, qui permet de construire des expressions conditionnelles du type `c? a : b`; celle-ci donne comme résultat le second opérande (a) si le premier opérande (c) est vrai, et le troisième opérande (b) dans le cas contraire. Ainsi, `(x > y)? x : y` est une expression qui a pour valeur $\max(x, y)$.

2.4.3 Opérateurs bit à bit

Ce sont des opérateurs qui permettent d'effectuer des opérations au niveau des bits. Bien que moins fréquemment utilisés, nous les mentionnons pour mémoire ; mais ne les utiliserons pas dans ce cours.

Op.	Exemple	Description / remarques
`>>`	`17 >> 3`	décalage de 3 bits sur la droite – ici le résultat est 2
`>>>`	`a >>> 5`	toujours un décalage de bits sur la droite (ici 5 bits), mais en version non signée
`<<`	`0x03 << 2`	décalage de 2 bits sur la gauche – je profite de cet exemple pour introduire la notation hexadécimale des nombres (10 s'écrit `A`, 11 s'écrit `B`, ... 15 s'écrit `F`) ; le résultat de l'exemple est `0x0C`
`&`	`0x75 & 0xF0`	*et* bit à bit – le résultat de l'exemple donné est `0x70`
`|`	`0x75 | 0xF0`	*ou* bit à bit – le résultat de l'exemple donné est `0xF5`
`^`	`0x75 ^ 0xF0`	*ou exclusif (xor)* bit à bit – le résultat de l'exemple donné est `0x85`
`~`	`~0xF0`	complément bit à bit – le résultat de l'exemple donné est `0x0F`

2.5 L'affectation

Nous avons vu (cf. § 2.1) qu'à une variable correspond un emplacement mémoire. Pour simplifier les explications qui vont suivre, considérons la mémoire comme une table M ; nous supposerons que la variable de nom `x` est stockée à la « case » $M[i]$. On appelle *l-valeur* (*l-value*, c'est-à-dire *left-value*) de la variable x son adresse en mémoire, dans notre cas i. On appelle *r-valeur* (*r-value*, c'est-à-dire *right-value*) de x la valeur stockée à cet emplacement, dans notre cas $M[i]$.

L'*affectation* est une opération fondamentale dans les langages de programmation impératifs, dont Java fait partie ; elle consiste à changer la *r-valeur* d'une variable, c'est-à-dire à stocker une nouvelle valeur à l'emplacement désigné par la *l-valeur* de la variable. De manière conceptuelle, on la note habituellement $\boxed{x \leftarrow y}$, qui signifie qu'on stocke la *r-valeur* de y à l'adresse désignée par la *l-valeur* de x. En Java, cette opération s'écrit `x = y`.

Attention – on ne pose pas ici une égalité, et on n'établit pas de lien pérenne entre les deux variables `x` et `y`. On se contente, à un instant précis, de recopier la valeur stockée dans `y` à l'adresse de `x`. Ce qui suit illustre ce point important :

```
int x = 3;
int y = 5;              // x == 3, y == 5
x = y;                  // x == 5, y == 5
y = 6;                  // x == 5, y == 6 : x et y ne sont pas des variables liées !
```

Il est très fréquent d'ajouter ou de soustraire une valeur à une variable, par exemple d'écrire `a = a + 3;`[2]. Java vous offre donc la possibilité d'écrire la même chose de manière raccourcie ; de même pour les autres opérations arithmétiques, ainsi que les opérations bit à bit :

```
a += 3;                 // équivaut à a = a + 3
a -= 3;                 // équivaut à a = a - 3
a *= 3;                 // équivaut à a = a * 3
a /= 3;                 // équivaut à a = a / 3
a %= 3;                 // équivaut à a = a % 3
a <<= 3;
// etc. pour >>=, >>>=, &=, |= et ^=
```

[2]Cet exemple illustre bien la différence fondamentale entre une égalité mathématique – qui n'aurait aucun sens ici – et l'affectation : ici, on prend la *r-valeur* de `a`, on lui ajoute `3`, et on stocke le résultat à l'adresse donnée par la *l-valeur* de cette même variable `a`.

Parmi ces opérations, le cas particulier de l'incrémentation et de la décrémentation est assez fréquent pour justifier d'une notation particulière :

```
a++;                // équivaut à a += 1 ou à a = a +1
++a;                // équivaut à a++, sauf pour la valeur rendue
a--;                // équivaut à a -= 1 ou à a = a -1
--a;                // équivaut à a--, sauf pour la valeur rendue
```

Cette question de valeur rendue vous semble probablement étrange ; aussi longtemps que vous utilisez ces opérations de la manière dont elles sont illustrées ci-dessus, vous n'avez pas besoin de vous tracasser à ce sujet. C'est lorsqu'on commence à utiliser la *r-valeur* de l'expression – ce qui est appelé par certains « programmer par effets de bord » et que je vous déconseille vivement, du moins dans un premier temps – qu'il faut y être attentif, dans la mesure où la forme préfixée rend la valeur de la variable *après* incrémentation/décrémentation, alors que la forme postfixée rend la valeur *avant* l'opération :

```
int x = 3;
int y = 5;          // x == 3, y == 5
x = y++;            // x == 5, y == 6
x = ++y;            // x == 7, y == 7
```

2.6 Mon premier programme Java

À ce stade, nous disposons de constantes, de variables, d'expressions et de l'instruction d'affectation d'une valeur à une variable. On peut déjà écrire des programmes simples avec ces éléments. Nous allons donc construire notre tout premier programme en Java. Nous nous rendrons vite compte qu'il faut recourir à plusieurs constructions que nous n'avons pas encore expliquées ; prenez-les dans un premier temps comme des « recettes », et nous indiquerons chaque fois la référence en avant de la partie du cours où la recette est expliquée.

Quand nous commençons ainsi à écrire des programmes, il est important de prendre tout de suite de bonnes habitudes d'écriture ; je vous invite donc dès maintenant à consulter l'annexe H.

Comme thème récurrent dans ce cours, nous allons choisir la gestion d'un compte bancaire. Commençons par écrire les lignes nécessaires pour créditer un compte bancaire d'un montant donné :

```
int solde = 0;     // variable désignant le solde et initialisée à 0
int montant = 16;  // variable désignant le montant à créditer

solde += montant;  // l'opération de crédit à proprement parler
```

Pour que ces lignes constituent un *programme* qui puisse s'exécuter, il faut les inclure dans une construction qui vous semblera probablement compliquée pour l'instant, mais que nous expliquerons par la suite. Tout d'abord, l'ensemble de lignes doit être inclus dans une *procédure* particulière appelée `main` (cf. § 3.2.4), dont la syntaxe est assez rébarbative au premier abord (mais nous aurons l'occasion d'expliquer les raisons de cette syntaxe – cf. § 4.3.1). Mais Java impose aussi que toute procédure doit appartenir à une *classe*, car on ne peut lancer une exécution Java qu'en précisant dans quelle classe se trouve le code à exécuter (cf. § 4.1). Nous définirons donc une classe `Banque` dans le fichier `Banque.java` (cf. § H.1).

Nous avons maintenant un programme, contenu dans la classe `Banque`, et qui est exécuté lorsque l'interprète Java est lancé sur cette classe. Le contenu du fichier `Banque.java` est donc :

```
public class Banque {
    public static void main(String[] args) {
        int solde = 0;
        int montant = 16;
```

```
        solde += montant;
    }
}
```

Nous avons maintenant un programme qui compile et qui s'exécute ; malheureusement, il ne communique pas avec le monde extérieur. On aimerait au minimum qu'il affiche le résultat de son calcul. Nous allons pour cela recourir à une fonction de la bibliothèque d'entrées/sorties fournie avec Java, appelée `java.io` (cf. § 6.3) ; cette fonction permet d'afficher à l'écran une chaîne de caractères, et convertit en chaîne de caractères tous les objets d'un autre type, tel que `int` dans notre exemple. Nous allons également utiliser une propriété de la classe `String`, représentant les chaînes de caractères (cf. § 4.4), à savoir que l'opérateur `+` permet de concaténer deux chaînes de caractères.

Le programme ainsi modifié est donné ci-après :

```
// Classe Banque - version 1.0
import java.io.*;       // Importer toutes les fonctionnalités de java.io

public class Banque {
    public static void main(String[] args) {
        int solde = 0;
        System.out.println("Le solde initial est de " + solde + " euros.");
        int montant = 16;

        solde += montant;
        System.out.println("Le nouveau solde est de " + solde + " euros.");
    }
}
```

Quand on l'exécute, un résultat s'affiche :

```
Le solde initial est de 0 euros.
Le nouveau solde est de 16 euros.
```

Allons maintenant un peu plus loin : une grosse faiblesse du programme reste que le montant à créditer est donné dans le programme lui-même, alors qu'on aimerait qu'il soit donné par l'utilisateur. Pour cela, nous allons recourir à une nouvelle fonction de la bibliothèque d'entrées/sorties ; malheureusement, pour l'appeler, nous avons d'abord besoin de recourir à une petite gymnastique mystérieuse de conversions. N'ayez crainte ; nous comprendrons plus tard les raisons de ces conversions (cf. § 6.3.1). Quand on lit quelque chose au clavier, il peut y avoir des problèmes (l'utilisateur peut par exemple donner une entrée invalide) et le système le sait ; il faut donc lui dire comment traiter ces éventuels problèmes, appelés *exceptions* (cf. § 6.4). Enfin, la fonction de lecture rend une chaîne de caractères, qu'il conviendra de convertir en un entier, par une fonction de conversion *ad hoc*. La nouvelle version du programme est donnée ci-après ; dans tout ce polycopié, nous indiquons sur fond grisé les lignes qui changent d'une version à l'autre, sauf quand les modifications sont majeures et nécessitent de reconsidérer l'ensemble des lignes :

```
// Classe Banque - version 1.1
import java.io.*;       // Importer toutes les fonctionnalités de java.io

public class Banque {
    public static void main(String[] args) {
        // Conversions mystérieuses...
        InputStreamReader ir = new InputStreamReader(System.in);
        BufferedReader br = new BufferedReader(ir);

        int solde = 0;
        System.out.println("Le solde initial est de " + solde + " euros.");
```

```
        System.out.print("Montant à créditer = ");
        System.out.flush();
        String lecture = "";      // la chaîne qu'on va lire, initialisée à rien
        try { // prise en compte des exceptions par try/catch
            lecture = br.readLine();
        } catch (IOException ioe) {}

        int montant = Integer.parseInt(lecture);  // conversion en int de la chaîne lue

        solde += montant;       // on retrouve l'incrémentation classique
        System.out.println("Le nouveau solde est de " + solde + " euros.");
    }
}
```

Le résultat de l'exécution donne (les données entrées par l'utilisateur sont toujours indiquées sur fond grisé) :

```
Le solde initial est de 0 euros.
Montant à créditer = 43
Le nouveau solde est de 43 euros.
```

Voilà ! nous avons un premier programme, certes rudimentaire, mais qui fait quelque chose d'à peu près sensé... Bien évidemment, nous avons dû passer sous silence les raisons de beaucoup de détails, qui vous paraissent peut-être pour l'instant relever plus de la formule magique que de la démarche raisonnée. Mais les choses vont se clarifier au fur et à mesure de la progression.

2.7 De la structuration du discours : les instructions de contrôle

Pour structurer et décomposer un programme, l'idée de base dans beaucoup de langages est que *stricto sensu*, on ne gère toujours qu'une seule *instruction* à chaque niveau, y compris le programme complet. Cette instruction peut être :

- L'appel à une fonction ou à une procédure ; nous verrons les fonctions et les procédures plus tard (§ 3.2), mais c'est en particulier l'instruction qui sert de point de départ à l'exécution d'un programme, par le lancement de la procédure `main()`, comme nous l'avons vu ci-dessus.
- Une instruction atomique de déclaration d'une variable[3] ou d'affectation du résultat d'un calcul à une variable, voire une instruction vide (`;`). Comme nous l'avons vu intuitivement dans l'exemple que nous avons déroulé au § 2.6, une expression ou une déclaration de variable (avec initialisation éventuelle) « devient » une instruction quand on lui ajoute le signe ' `;` ' qui joue le rôle de terminaison d'instruction[4].
- Une construction conditionnelle (cf. § 2.7.1) ou d'itération (cf. § 2.7.2).
- Un bloc composé ; l'idée est ici que si on a besoin de plusieurs instructions à un même niveau, elles sont regroupées dans un bloc, entre les délimitateurs '`{`' et '`}`'. Un bloc composé a sa propre portée lexicale (cf. § 6.1) ; on peut en particulier y définir et y utiliser des variables locales au bloc.

2.7.1 Instructions conditionnelles

Dans le modèle classique en vigueur depuis la machine de von Neumann (cf. chapitre A), le flot normal d'exécution est séquentiel, c'est-à-dire qu'après exécution d'une instruction élémentaire,

[3] En règle générale, la déclaration d'une variable doit être immédiatement suivie de l'initialisation de cette variable. Sauf cas très particulier, quand vous n'avez aucun moyen de donner une valeur d'initialisation réaliste à une variable, nous vous conseillons d'adopter ce principe.

[4] Pour ceux qui connaissent le langage Pascal, il faut noter qu'en Pascal, ' `;` ' est un *séparateur* d'instructions, ce qui implique entre autres que la dernière instruction d'un bloc n'est pas terminée par un ' `;` ' – en Java, en revanche, chaque instruction se termine par un ' `;` '

c'est l'instruction immédiatement consécutive dans le programme qui est activée. Les langages de programmation de haut niveau ont pour la plupart repris ce principe simple, et c'est le cas notamment de Java : des instructions atomiques vont être exécutées dans l'ordre dans lequel elles se succèdent dans le programme.

Cependant, pour élaborer des programmes plus complets, il est nécessaire de disposer d'instructions permettant de modifier dans certains cas ce déroulement linéaire. La première que nous allons voir est la conditionnelle. Pour oser une comparaison ferroviaire, on peut dire qu'il existe deux types de conditionnelles :

- l'aiguillage, qui permet de choisir l'une de deux « branches » d'exécution, suivant la valeur d'une condition ;
- la gare de triage, qui permet de multiples embranchements, suivant les conditions rencontrées.

L'aiguillage correspond à la structure algorithmique classique appelée *si–alors–sinon*, et qu'on peut écrire sous la forme suivante :

si *condition*
  alors $instruction_1$
  [**sinon** $instruction_2$]
finsi

La partie *sinon* est facultative. Pour l'indiquer, nous avons mis des crochets [ ] ci-dessus. La construction correspondante en Java utilise les mots clés `if` et `else`. Notons qu'il n'y a pas de mot clé particulier pour traduire *alors*.

Illustrons tout de suite cette construction en modifiant notre programme bancaire. Nous souhaitons maintenant pouvoir choisir d'effectuer une opération de crédit ou de débit. Il faut donc demander à l'utilisateur ce qu'il souhaite faire, et effectuer en conséquence les opérations appropriées. Dans le programme, nous utilisons aussi, sans l'expliquer pour l'instant[5], la construction `choix.equals("D")`, qui permet de comparer deux chaînes de caractères. L'interface reste ici rudimentaire, on pourrait entre autres imaginer que si l'utilisateur donne un choix invalide, on lui repose la question au lieu de décider par défaut que c'est une opération de crédit :

```
// Classe Banque - version 1.2
import java.io.*;       // Importer toutes les fonctionnalités de java.io

public class Banque {
    public static void main(String[] args) {
        InputStreamReader ir = new InputStreamReader(System.in);
        BufferedReader br = new BufferedReader(ir);

        int solde = 0;
        System.out.println("Le solde initial est de " + solde + " euros.");

        System.out.print("[D]ébit ou [C]rédit (choix par défaut) ? ");
        String choix = "";
        try {
            choix = br.readLine();
        } catch (IOException ioe) {}

        boolean credit = true;   // variable vraie si c'est un crédit (défaut)

        if (choix.equals("D") || choix.equals("d")) {
            credit = false;
        }
        // sinon on reste à true par défaut, donc crédit

        // Notez l'utilisation de l'opérateur à 3 opérandes ?:
```

[5] Pour ceux qui sont curieux de savoir comment ça marche, voir § 4.4.

```
        System.out.print("Montant à " + (credit ? "créditer" : "débiter") + " = ");

        System.out.flush();
        String lecture = "";      // la chaîne qu'on va lire, initialisée à rien
        try {
            lecture = br.readLine();
        } catch (IOException ioe) {}

        int montant = Integer.parseInt(lecture);  // conversion en int de la chaîne lue

        if (credit) {
            solde += montant;
        }
        else {
            solde -= montant;
        }
        System.out.println("Le nouveau solde est de " + solde + " euros.");
    }
}
```

Et voici le résultat de deux exécutions successives de ce programme :

```
Le solde initial est de 0 euros.
[D]ébit ou [C]rédit (choix par défaut) ? d
Montant à débiter = 124
Le nouveau solde est de -124 euros.

Le solde initial est de 0 euros.
[D]ébit ou [C]rédit (choix par défaut) ? c
Montant à créditer = 654
Le nouveau solde est de 654 euros.
```

Vous avez peut-être noté l'emploi de blocs composés, délimités par '`{`' et '`}`', bien que ceux-ci ne contiennent qu'une seule instruction. *Stricto sensu*, ils ne sont pas nécessaires ici, mais il est fortement conseillé d'en systématiser l'usage pour toutes les constructions conditionnelles ou itératives ; cela augmente la lisibilité et la clarté du programme, en mettant mieux en valeur la structure de votre « discours ».

La deuxième construction conditionnelle possible correspond à ce que nous avons appelé la gare de triage. Pour l'exprimer dans un algorithme, on écrira quelque chose du genre :

$$\begin{cases} cas_1 & \rightarrow instruction_1 \\ cas_2 & \rightarrow instruction_2 \\ \dots & \\ cas_n & \rightarrow instruction_n \end{cases}$$

En Java, il n'existe pas de vraie construction syntaxique permettant de traduire cette notion ; on utilise habituellement une série de `if` ... `else` imbriqués. Cependant, dans le cas particulier où les différents cas peuvent s'exprimer sous la forme de valeurs possibles pour une expression de type entier (`byte`, `short`, `int` ou `long`) ou caractère (`char`), on dispose de la construction `switch`. Cependant, comme nous l'illustrons dans l'exemple ci-après, cette construction correspond plutôt à un ensemble de branchements à des points précis du programme, ce qui explique la présence des instructions `break` ; sans celles-ci, le flot de contrôle se poursuivrait séquentiellement. Cela explique aussi l'absence de structuration par des blocs composés : on reste au même niveau de structuration. À noter aussi qu'on ne peut pas, dans une construction de type `switch`, employer des conditions booléennes, les étiquettes de branchement – introduites par le mot clé `case`, l'étiquette `default` indiquant une valeur par défaut – correspondant à des valeurs constantes et discrètes, de type

caractère ou entier. Il faut donc réserver cette construction à des cas très précis et spécifiques, et privilégier une suite de constructions `if ...else`, éventuellement imbriquées, dans la majorité des cas de conditionnelles.

Donnons toutefois une nouvelle variante du programme bancaire où nous utilisons une condition `switch` pour choisir le type d'opération à effectuer. Cette fois-ci, au lieu de comparer des chaînes de caractères, nous utilisons l'opération `choix.charAt(0)` pour récupérer le premier caractère (position 0) de la chaîne[6], le `switch` se faisant sur la valeur de ce caractère (majuscules et minuscules autorisées) :

```
// Classe Banque - version 1.3
import java.io.*;       // Importer toutes les fonctionnalités de java.io

public class Banque {
    public static void main(String[] args) {
        InputStreamReader ir = new InputStreamReader(System.in);
        BufferedReader br = new BufferedReader(ir);

        int solde = 0;
        System.out.println("Le solde initial est de " + solde + " euros.");

        System.out.print("Votre choix : [D]ébit, [C]rédit, [F]in ? ");
        String choix = "";
        try {
            choix = br.readLine();
        } catch (IOException ioe) {}
        boolean fin = false;     // variable vraie si on veut s'arrêter
        boolean credit = false;   // variable vraie si c'est un crédit

        // Récupérer la première lettre de la chaîne saisie
        char monChoix = choix.charAt(0);
        switch(monChoix) {
        case 'C':
        case 'c':       // Même chose pour majuscule et minuscule
            credit = true;
            fin = false;
            break;     // Pour ne pas continuer en séquence
        case 'd':
        case 'D':
            credit = false;
            fin = false;
            break;
        case 'f':
        case 'F':
            fin = true;
            break;
        default:      // Etiquette spéciale si aucun des cas prévus n'est vérifié
            fin = true;  // On va considérer que par défaut on s'arrête
        }

        // Poser la question uniquement si on n'a pas choisi de finir
        // Notez que toute cette partie reste identique au programme précédent
        // mais elle est maintenant incluse dans une construction if
        if (!fin) {
```

[6]Une fois de plus, les esprits curieux peuvent faire un saut en avant jusqu'au § 4.4 pour mieux comprendre cette opération.

```
            System.out.print("Montant à " + (credit ? "créditer" : "débiter") + " = ");

            System.out.flush();
            String lecture = "";      // la chaîne qu'on va lire, initialisée à rien
            try {
                lecture = br.readLine();
            } catch (IOException ioe) {}

            int montant = Integer.parseInt(lecture);  // conversion en int

            if (credit) {
                solde += montant;
            }
            else {
                solde -= montant;
            }
        }

        // Dans tous les cas, imprimer le nouveau solde
        System.out.println("Le nouveau solde est de " + solde + " euros.");
    }
}
```

Un exemple de deux exécutions :

```
Le solde initial est de 0 euros.
Votre choix : [D]ébit, [C]rédit, [F]in ? d
Montant à débiter = 123
Le nouveau solde est de -123 euros.

Le solde initial est de 0 euros.
Votre choix : [D]ébit, [C]rédit, [F]in ? f
Le nouveau solde est de 0 euros.
```

2.7.2 Instructions itératives

Pour garder l'analogie ferroviaire, la construction itérative permet de réaliser un circuit dans le flot de contrôle – ce qui plaira sûrement à tous les amateurs de modélisme. Plus sérieusement, une construction itérative permet de répéter les mêmes instructions

- tant qu'une condition reste vérifiée ; on écrit alors :
  tantque *condition* **faire**
  instructions
  fintantque
- pour tous les éléments d'un ensemble ; on écrira alors quelque chose du genre :
  pour tout $x \in E$ **faire**
  instructions
  finpour

Java offre deux constructions correspondant à ces deux cas de figure. Commençons par la première. En Java, elle s'écrit avec les mots clés `while` et éventuellement `do`. En reprenenant l'exemple bancaire, ajoutons maintenant la possibilité d'effectuer successivement plusieurs opérations de débit et de crédit, jusqu'à ce que l'utilisateur désire finir :

```
// Classe Banque - version 1.4
import java.io.*;       // Importer toutes les fonctionnalités de java.io

public class Banque {
    public static void main(String[] args) {
        InputStreamReader ir = new InputStreamReader(System.in);
```

```
        BufferedReader br = new BufferedReader(ir);

        int solde = 0;
        System.out.println("Le solde initial est de " + solde + " euros.");

        boolean fin = false;     // variable vraie si on veut s'arrêter
        while (!fin) {  // tant qu'on n'a pas demandé de finir
            System.out.print("Votre choix : [D]ébit, [C]rédit, [F]in ? ");
            String choix = "";
            try {
                choix = br.readLine();
            } catch (IOException ioe) {}

            boolean credit = false;   // variable vraie si c'est un crédit

            // Récupérer la première lettre de la chaîne saisie
            char monChoix = choix.charAt(0);
            switch(monChoix) {
            case 'C':
            case 'c':       // Même chose pour majuscule et minuscule
                credit = true;
                fin = false;
                break;     // Pour ne pas continuer en séquence
            case 'd':
            case 'D':
                credit = false;
                fin = false;
                break;
            case 'f':
            case 'F':
                fin = true;
                break;
            default:
                fin = true;  // On va considérer que par défaut on s'arrête
            }

            if (!fin) {
                System.out.print("Montant à " +
                                 (credit ? "créditer" : "débiter") +
                                 " = ");
                System.out.flush();
                String lecture = "";
                try {
                    lecture = br.readLine();
                } catch (IOException ioe) {}

                int montant = Integer.parseInt(lecture);  // conversion en int

                if (credit) {
                    solde += montant;
                }
                else {
                    solde -= montant;
                }
                // N'afficher le nouveau solde que s'il y a eu une opération
                System.out.println("Le nouveau solde est de " + solde + " euros.");
            }
        }
    }
```

```
}
```

Un exemple d'exécution est donné ci-après :

```
Le solde initial est de 0 euros.
Votre choix : [D]ébit, [C]rédit, [F]in ? C
Montant à créditer = 500
Le nouveau solde est de 500 euros.
Votre choix : [D]ébit, [C]rédit, [F]in ? D
Montant à débiter = 234
Le nouveau solde est de 266 euros.
Votre choix : [D]ébit, [C]rédit, [F]in ? D
Montant à débiter = 65
Le nouveau solde est de 201 euros.
Votre choix : [D]ébit, [C]rédit, [F]in ? C
Montant à créditer = 67
Le nouveau solde est de 268 euros.
Votre choix : [D]ébit, [C]rédit, [F]in ? F
```

La variante `do ...while` a comme seule différence que les instructions sont exécutées une première fois avant que soit testée la condition d'arrêt. Nous aurions par exemple pu écrire une version légèrement différente du programme précédent, avec :

```
      do {
          System.out.print("Votre choix : [D]ébit, [C]rédit, [F]in ? ");
          ... // etc.
      } while (!fin);
```

Dans le cas présent, la différence entre ces deux versions est minime ; il y a cependant des cas où le choix de l'une ou de l'autre des variantes rend l'écriture du programme plus simple ou plus claire.

Pour le deuxième type d'itération, qui permet de parcourir tous les éléments d'un ensemble ou d'une collection, il est souvent utile d'employer une autre construction Java, celle qui emploie le mot clé `for`. Si par exemple, au lieu de demander à l'utilisateur d'indiquer quand il a fini de saisir des écritures comptables, on sait qu'on va lui en demander dix, on pourra écrire :

```
for (int i = 0 ; i < 10 ;  i++) {
  System.out.print("Votre choix : [D]ébit ou [C]rédit ? ");
  ... // etc
}
```

Les trois expressions séparées par des ' `;` ' dans cette construction `for` correspondent respectivement à l'initialisation de la boucle, à la condition de continuation, et au passage à l'élément suivant de l'ensemble à traiter. Les plus perspicaces auront probablement compris qu'une construction équivalente[7], dans le cas présent, serait :

```
int i = 0;
while (i < 10) {
  System.out.print("Votre choix : [D]ébit ou [C]rédit ? ");
  ... // etc
  i++;
}
```

[7] Au petit détail près que dans un cas la variable `i` n'existe que lors de l'itération, alors que dans l'autre elle est déclarée avant le début de la boucle et continue donc sa vie à la sortie de l'itération.

NB : Une erreur assez commune faite par les débutants et de séparer les trois expressions constitutives de la construction `for` par des virgules et non par des points-virgules. Tenez-le vous pour dit !

Le principal avantage de `for` est de rendre explicite l'énumération des éléments d'un ensemble ou d'une collection. Nous aurons l'occasion d'illustrer plus complètement l'utilisation de cette construction dans d'autres paragraphes (cf. § 3.3).

Deux mots clés supplémentaires peuvent être utilisés dans les itérations[8] :

– `continue` permet de revenir de manière prématurée en début de boucle. Si par exemple on veut éviter d'effectuer une opération de crédit ou de débit si le montant de l'opération est nul, on peut modifier le programme vu précédemment en ajoutant les lignes sur fond grisé ci-dessous :

```
...
while (!fin) {  // tant qu'on n'a pas demandé de finir
    ...
    if (!fin) {
        System.out.print("Montant à " +
                         (credit ? "créditer" : "débiter") +
                         " = ");
        System.out.flush();
        String lecture = "";
        try {
            lecture = br.readLine();
        } catch (IOException ioe) {}

        int montant = Integer.parseInt(lecture);  // conversion en int

        if (montant == 0) {
            continue;    // repartir en début de boucle while(!fin)
        }

        if (credit) {
            solde += montant;
        }
        else {
            solde -= montant;
        }
        System.out.println("Le nouveau solde est de " + solde + " euros.");
    }
    ...
}
```

Il va sans dire qu'il faut éviter autant que faire se peut de recourir à ce genre de construction, qui rend le programme considérablement moins lisible.

– `break`, que nous avons déjà vu pour la construction `switch`, permet aussi de sortir d'une boucle d'itération sans repartir par le test de départ. Ainsi, nous aurions pu écrire notre programme différemment, en prévoyant une boucle infinie (construction `while (true)`) dont on sort explicitement dès que l'utilisateur donne le code de fin. Le programme peut sembler un peu plus élégant de cette manière ; cependant, il ne faut pas oublier qu'il est *a priori* contre-intuitif et peu explicite d'écrire `while (true)` quand on sait pertinemment qu'on va sortir de la boucle à un moment donné. Il faut donc utiliser des instructions comme `break` et `continue` avec beaucoup de modération...

Donnons néanmoins une version corrigée de notre programme, avec utilisation de `break` :

[8] On peut noter que ces deux instructions `break` et `continue` peuvent être suivies d'une étiquette, qui spécifie soit une boucle englobante (pour `continue`), soit une instruction quelconque englobante (pour `break`). Dans ce dernier cas, `break` indique que l'on veut quitter le niveau englobant ainsi spécifié. Nous ne vous conseillons pas d'utiliser ces fonctionnalités plus avancées...

```
// Classe Banque - version 1.5
import java.io.*;       // Importer toutes les fonctionnalités de java.io

public class Banque {
    public static void main(String[] args) {
        InputStreamReader ir = new InputStreamReader(System.in);
        BufferedReader br = new BufferedReader(ir);

        int solde = 0;
        System.out.println("Le solde initial est de " + solde + " euros.");

        boolean fin = false;     // variable vraie si on veut s'arrêter
        while(true) {  // boucle infinie dont on sort par un break
            System.out.print("Votre choix : [D]ébit, [C]rédit, [F]in ? ");
            String choix = "";
            try {
                choix = br.readLine();
            } catch (IOException ioe) {}

            boolean credit = false;   // variable vraie si c'est un crédit

            // Récupérer la première lettre de la chaîne saisie
            char monChoix = choix.charAt(0);
            switch(monChoix) {
            case 'C':
            case 'c':       // Même chose pour majuscule et minuscule
                credit = true;
                fin = false;
                break;     // Pour ne pas continuer en séquence
            case 'd':
            case 'D':
                credit = false;
                fin = false;
                break;
            case 'f':
            case 'F':
                fin = true;
                break;
            default:
                fin = true;  // On va considérer que par défaut on s'arrête
            }

            if (fin) {
                break;    // sortir de la boucle ici
            }
            else {
                System.out.print("Montant à " +
                                 (credit ? "créditer" : "débiter") +
                                 " = ");
                System.out.flush();
                String lecture = "";
                try {
                    lecture = br.readLine();
                } catch (IOException ioe) {}

                int montant = Integer.parseInt(lecture);  // conversion en int

                if (credit) {
                    solde += montant;
```

```
                }
                else {
                    solde -= montant;
                }
                System.out.println("Le nouveau solde est de " + solde + " euros.");
            }
        }
    }
}
```

Chapitre 3

Structuration

La programmation, c'est l'art d'organiser la complexité.

E. Dijkstra

Il est déjà possible d'écrire des programmes importants avec les constructions de base que nous avons vues. Cependant, il nous manque des outils permettant de *structurer* nos programmes, pour maîtriser une complexité qui deviendrait sinon trop grande pour permettre une gestion aisée par un être humain. Comme nous l'avons déjà vu (cf. § 1.1), la structuration peut être définie comme l'art de regrouper des entités élémentaires en entités de plus haut niveau, qui puissent être manipulées directement, permettant ainsi de s'affranchir de la maîtrise des constituants élémentaires quand on travaille avec la structure.

En programmation, cette structuration s'applique suivant deux axes :

1. On peut regrouper des opérations élémentaires, que l'on effectue régulièrement, en entités auxquelles on donnera un nom et une sémantique précise. C'est ce que nous verrons avec les fonctions et les procédures (§ 3.2).
2. On peut aussi regrouper des données, qui ensemble servent à représenter un concept unitaire. Nous allons commencer par ce premier type de structuration, en définissant la notion de classe (§ 3.1).

3.1 La classe, première approche : un regroupement de variables

Il est extrêmement fréquent que l'on manipule un concept complexe, qui nécessite pour être représenté l'utilisation de plusieurs variables. Restons dans notre domaine d'application bancaire ; il est clair que pour être utilisé, le programme que nous avons écrit doit s'appliquer à un *compte bancaire*, qui est caractérisé au minimum par :

- le nom du titulaire du compte,
- l'adresse de celui-ci,
- le numéro du compte,
- le solde du compte.

On pourrait bien sûr avoir quatre variables indépendantes, `nom`, `adresse`, `numéro` et `solde`, mais cela n'est guère satisfaisant, car rien n'indique explicitement que ces quatre variables sont « liées ». Que ferons-nous dans ces conditions pour nous y retrouver quand il s'agira d'écrire le programme de gestion d'une agence bancaire, dans laquelle il y a plusieurs centaines de comptes ?

La plupart des langages de programmation offrent donc la possibilité de regrouper des variables en une structure commune et nommée, que l'on peut ensuite manipuler globalement. En Java, cette structure s'appelle la *classe*[1].

[1] En fait, vous comprendrez au chapitre 4 que la classe est bien plus que cela, puisque c'est l'élément de base de la programmation objet. Mais nous nous contenterons dans un premier temps de cette vision simplifiée de la classe, qui la rend analogue à un `RECORD` en Pascal, par exemple.

Si je souhaite définir le concept de compte bancaire, je peux définir une classe, avec le mot clé `class`. Si j'appelle cette classe `CompteBancaire`, elle doit être définie dans un nouveau fichier, `CompteBancaire.java`, comme nous l'expliquons au paragraphe H.1. Voici le contenu de ce fichier :

```
// Classe CompteBancaire - version 1.0

/**
 * Classe permettant de regrouper les éléments qui décrivent un
 * compte bancaire. Pas de programmation objet pour l'instant
 * @author Karl Tombre
 */

public class CompteBancaire {
    String nom;       // le nom du client
    String adresse;   // son adresse
    int numéro;       // numéro du compte (int pour simplifier)
    int solde;        // solde du compte (opérations entières pour simplifier)
}
```

La définition d'une classe définit un nouveau type ; nous pouvons donc maintenant déclarer une variable de ce nouveau type : `CompteBancaire monCompte` et travailler directement avec cette variable. Cependant, il faut être capable d'accéder aux éléments individuels de cette variable, par exemple le solde du compte. La *notation pointée* permet de le faire : `monCompte.solde` désigne le solde du compte représenté par la variable `monCompte`.

NB : À ce stade, on notera qu'on peut considérer une classe comme un produit cartésien.

3.1.1 Allocation de mémoire

Une classe peut en fait être considérée comme un « moule »; la classe `CompteBancaire` décrit la configuration type de tous les comptes bancaires, et si on déclare deux variables :

```
CompteBancaire c1;
CompteBancaire c2;
```

chacune de ces variables possède son propre exemplaire de nom, d'adresse, de numéro et de solde. En effet, il ne serait pas normal que tous les comptes partagent le même numéro ou le même solde ! Donc `c1.numéro` et `c2.numéro`, par exemple, sont deux variables distinctes.

Une loi générale est que dans une « case » mémoire, on ne peut stocker que des nombres. Reste à savoir comment interpréter le nombre contenu dans une telle case ; cela va bien entendu dépendre du type de la variable :

En présence de types scalaires comme des entiers, l'interprétation est immédiate : le nombre contenu dans la case est la valeur de l'entier.

Pour les autres types simples (réels, caractères...) un codage standard de la valeur par un nombre permet également une interprétation immédiate (cf. annexe F).

Les choses se passent différemment pour toutes les variables de types définis à partir d'une classe : ces variables, telles que `c1` et `c2` dans notre exemple, sont techniquement des *références*. Cela signifie que les cases mémoires correspondantes ne « contiennent » pas directement les différentes données nécessaires pour représenter un compte, mais uniquement *l'adresse* d'un emplacement mémoire où un tel compte est représenté. Autrement dit, leur *r-valeur* contient une *l-valeur*.

Pour réserver un emplacement mémoire effectif, il faut alors passer par une opération d'allocation mémoire, grâce à l'instruction `new`. Aussi longtemps que celle-ci n'a pas été effectuée, la variable `monCompte`, par exemple, a une valeur spécifique appelée `null`, correspondant à une adresse indéfinie, et il se produit une erreur si on essaye d'accéder à `monCompte.numéro`. Il faut donc faire cette allocation mémoire, par exemple au moment de la déclaration de la variable :

```
CompteBancaire monCompte = new CompteBancaire();
```

avant de pouvoir continuer par exemple par :

```
monCompte.solde = 1000;
```

Aparté pour les petits curieux : que devient la mémoire qui n'est plus utilisée ?

Si vous avez l'expérience de certains langages de programmation comme Pascal, C ou C++, vous vous demandez peut-être comment procéder pour « rendre » la mémoire affectée par `new`, une fois que vous n'en avez plus besoin. En effet, en plus d'un mécanisme similaire à `new`, ces langages ont aussi une construction permettant de libérer de la mémoire.

Rien de tel en Java ; ses concepteurs ont choisi de le munir d'un mécanisme de *ramasse-miettes* (*garbage collector* en anglais). Un processus automatique, qui s'exécute en parallèle et en arrière-plan, sans que l'utilisateur ait à intervenir, entreprend de collecter, au fur et à mesure, les « cases mémoire » qui ne sont plus référencées par aucune variable.

3.1.2 Exemple : un embryon de programme de gestion de compte

Pour illustrer notre propos, reprenons notre exemple bancaire. Nous allons maintenant demander à l'utilisateur de donner les coordonnées du compte, et toutes les manipulations se feront sur une variable de type `CompteBancaire`. Notez bien que nous travaillons maintenant avec deux fichiers, `Banque.java` et `CompteBancaire.java`, selon un principe général en Java, qui impose qu'à chaque classe corresponde un fichier séparé :

```
// Classe Banque - version 2.0

/**
 * Classe contenant un programme de gestion bancaire, utilisant
 * cette fois-ci une variable de type CompteBancaire
 * @author Karl Tombre
 * @see    CompteBancaire
 */

import java.io.*;       // Importer toutes les fonctionnalités de java.io

public class Banque {
    public static void main(String[] args) {
        InputStreamReader ir = new InputStreamReader(System.in);
        BufferedReader br = new BufferedReader(ir);

        // On va travailler sur la variable monCompte
        CompteBancaire monCompte = new CompteBancaire();

        // Commencer par demander les valeurs des champs du compte
        System.out.print("Nom du titulaire = ");
        System.out.flush();
        try {
            monCompte.nom = br.readLine();
        } catch (IOException ioe) {}
        System.out.print("Son adresse = ");
        System.out.flush();
        try {
            monCompte.adresse = br.readLine();
        } catch (IOException ioe) {}
        System.out.print("Numéro du compte = ");
        System.out.flush();
```

```
String reponse = "";
try {
    reponse = br.readLine();
} catch (IOException ioe) {}
monCompte.numéro = Integer.parseInt(reponse);
// Solde initial à 0
monCompte.solde = 0;
System.out.println(monCompte.nom +
                   ", le solde initial de votre compte numéro " +
                   monCompte.numéro +
                   " est de " + monCompte.solde + " euros.");

boolean fin = false;     // variable vraie si on veut s'arrêter
while(true) {  // boucle infinie dont on sort par un break
    System.out.print("Votre choix : [D]ébit, [C]rédit, [F]in ? ");
    String choix = "";
    try {
        choix = br.readLine();
    } catch (IOException ioe) {}

    boolean credit = false;   // variable vraie si c'est un crédit

    // Récupérer la première lettre de la chaîne saisie
    char monChoix = choix.charAt(0);
    switch(monChoix) {
    case 'C':
    case 'c':       // Même chose pour majuscule et minuscule
        credit = true;
        fin = false;
        break;     // Pour ne pas continuer en séquence
    case 'd':
    case 'D':
        credit = false;
        fin = false;
        break;
    case 'f':
    case 'F':
        fin = true;
        break;
    default:
        fin = true;  // On va considérer que par défaut on s'arrête
    }

    if (fin) {
        break;    // sortir de la boucle ici
    }
    else {
        System.out.print("Montant à " +
                         (credit ? "créditer" : "débiter") +
                         " = ");
        System.out.flush();
        String lecture = "";
        try {
            lecture = br.readLine();
        } catch (IOException ioe) {}

        int montant = Integer.parseInt(lecture);  // conversion en int
```

```
                if (credit) {
                    monCompte.solde += montant;
                }
                else {
                    monCompte.solde -= montant;
                }
                System.out.println(monCompte.nom +
                                   ", le nouveau solde de votre compte numéro " +
                                   monCompte.numéro +
                                   " est de " + monCompte.solde + " euros.");
            }
        }
    }
}
```

Et voici un résultat d'exécution :

```
Nom du titulaire = Dominique Scapin
Son adresse = 1, rue des gros sous, 75002 Paris
Numéro du compte = 182456
Dominique Scapin, le solde initial de votre compte numéro 182456 est de 0 euros.
Votre choix : [D]ébit, [C]rédit, [F]in ? C
Montant à créditer = 876234
Dominique Scapin, le nouveau solde de votre compte numéro 182456 est de 876234 euros.
Votre choix : [D]ébit, [C]rédit, [F]in ? D
Montant à débiter = 542
Dominique Scapin, le nouveau solde de votre compte numéro 182456 est de 875692 euros.
Votre choix : [D]ébit, [C]rédit, [F]in ? F
```

Exercice 3 *Définir des classes qui permettent de représenter :*
- *des points sur un plan,*
- *des points dans l'espace,*
- *des villes de France,*
- *des villes du monde.*

3.2 Fonctions et procédures

3.2.1 Les fonctions – approche intuitive

Vous avez probablement remarqué que chaque fois que nous effectuons une lecture d'un entier, nous sommes obligés d'écrire plusieurs lignes de code, toujours à peu près les mêmes, ce qui finit par alourdir le programme. Nous sommes typiquement en présence du besoin de structuration des instructions : l'opération de lecture au clavier d'un entier, par exemple, serait bien plus à sa place dans une *fonction* de lecture qu'en ligne.

Java permet d'écrire de telles fonctions, qui prennent éventuellement des paramètres en entrée – ici nous choisirons la question à poser – et qui rendent un résultat – ici un entier.

Si vous avez bien suivi le dernier exemple que nous avons déroulé, vous avez probablement remarqué en plus que pour lire un entier, il faut d'abord lire une chaîne de caractères, puis convertir la chaîne résultat de cette opération de lecture en un entier. On peut donc écrire la fonction de lecture d'un entier à l'aide de la fonction de lecture d'une chaîne de caractères.

Où les mettre ?

Nous avons déjà vu qu'en Java, tout doit être défini au sein d'une classe. Se pose alors la question de la classe dans laquelle définir ces fonctions de lecture. On pourrait tout simplement

les ajouter à la classe `Banque` ; mais elles sont d'un caractère assez général pour qu'elles puissent aussi servir dans d'autres contextes que la gestion d'un compte bancaire. Nous choisissons donc de définir une nouvelle classe – notre troisième – que nous appelons `Utils` et dans laquelle nous regrouperons, au fur et à mesure que nous progressons, les fonctions utilitaires qui peuvent servir dans nos programmes. Voici donc une première version de cette classe, définie – faut-il le rappeler ? – dans le fichier `Utils.java` :

```
// Classe Utils - version 1.0

/**
 *  Classe regroupant les utilitaires disponibles pour les exercices
 * de 1ere année du tronc commun informatique.
 * @author Karl Tombre
 */

import java.io.*;

public class Utils {
    // Les fonctions lireChaine et lireEntier
    // Exemple d'utilisation :
    // String nom = Utils.lireChaine("Entrez votre nom : ");
    public static String lireChaine(String question) {
        InputStreamReader ir = new InputStreamReader(System.in);
        BufferedReader br = new BufferedReader(ir);

        System.out.print(question);
        System.out.flush();
        String reponse = "";
        try {
            reponse = br.readLine();
        } catch (IOException ioe) {}
        return reponse;
    }

    // La lecture d'un entier n'est qu'un parseInt de plus !!
    public static int lireEntier(String question) {
        return Integer.parseInt(lireChaine(question));
    }
}
```

Continuons pour l'instant à prendre comme un acquis la syntaxe `public static` ; nous y reviendrons pour en donner l'explication (cf. § 4.3).

Vous remarquerez que la fonction `lireChaine` prend en entrée une variable de type `String`, à savoir la question que l'on souhaite poser, et rend une variable de type `String` également, à savoir la réponse donnée par l'utilisateur à la question posée. Vous remarquerez aussi que la fonction `lireEntier` s'écrit par un simple appel à la fonction précédente, `lireChaine`, suivi d'un appel à une fonction de conversion d'une chaîne en un entier. Vous avez donc noté que les fonctions que nous avons définies incluent elles-mêmes des appels à d'autres fonctions définies dans les bibliothèque standard de Java, telles que `Integer.parseInt` justement. Enfin, vous noterez l'emploi du mot clé `return`, pour indiquer la valeur qui est renvoyée comme résultat de la fonction.

Reprenons maintenant notre programme bancaire et notons comment il se simplifie grâce à l'emploi des fonctions. Vous noterez entre autres que les variables rébarbatives de type `BufferedReader` et autres n'ont plus lieu d'être déclarées dans ce programme, puisqu'elles sont propres aux fonctions de lecture. Notez aussi l'emploi dans ce programme de la notation pointée `Utils.lireEntier`, pour indiquer dans quelle classe il faut aller chercher la fonction `lireEntier` :

```
// Classe Banque - version 2.1

/**
```

```
 * Classe contenant un programme de gestion bancaire, utilisant
 * cette fois-ci une variable de type CompteBancaire
 * @author Karl Tombre
 * @see    CompteBancaire
 */

public class Banque {
    public static void main(String[] args) {
        // On va travailler sur la variable monCompte
        CompteBancaire monCompte = new CompteBancaire();

        // Commencer par demander les valeurs des champs du compte
        monCompte.nom = Utils.lireChaine("Nom du titulaire = ");
        monCompte.adresse = Utils.lireChaine("Son adresse = ");
        monCompte.numéro = Utils.lireEntier("Numéro du compte = ");

        // Solde initial à 0
        monCompte.solde = 0;
        System.out.println(monCompte.nom +
                           ", le solde initial de votre compte numéro " +
                           monCompte.numéro +
                           " est de " + monCompte.solde + " euros.");

        boolean fin = false;     // variable vraie si on veut s'arrêter
        while(true) {  // boucle infinie dont on sort par un break
            String choix = Utils.lireChaine("Votre choix : [D]ébit, [C]rédit, [F]in ? ");
            boolean credit = false;   // variable vraie si c'est un crédit

            // Récupérer la première lettre de la chaîne saisie
            char monChoix = choix.charAt(0);
            switch(monChoix) {
            case 'C':
            case 'c':       // Même chose pour majuscule et minuscule
                credit = true;
                fin = false;
                break;     // Pour ne pas continuer en séquence
            case 'd':
            case 'D':
                credit = false;
                fin = false;
                break;
            case 'f':
            case 'F':
                fin = true;
                break;
            default:
                fin = true;  // On va considérer que par défaut on s'arrête
            }

            if (fin) {
                break;    // sortir de la boucle ici
            }
            else {
                int montant = Utils.lireEntier("Montant à " +
                                               (credit ? "créditer" : "débiter") +
                                               " = ");
                if (credit) {
                    monCompte.solde += montant;
                }
```

```
                else {
                    monCompte.solde -= montant;
                }
                System.out.println(monCompte.nom +
                                   ", le nouveau solde de votre compte numéro " +
                                   monCompte.numéro +
                                   " est de " + monCompte.solde + " euros.");
            }
        }
    }
}
```

3.2.2 Les fonctions – définition plus formelle

La notion de fonction rejoint en fait la fonction telle qu'elle est connue en mathématiques. Nous avons déjà vu les opérateurs du langage (§ 2.4), qui définissent en fait des fonctions intrinsèques ; par exemple + sur les entiers correspond à la fonction[2]

$$+ : \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}$$

Plus généralement, une fonction *func* qui serait définie mathématiquement par :

$$func : \mathbb{Z} \times Bool \times Char \to \mathbb{R}$$

aura comme définition en Java :

```
static double func(int x, boolean b, char c)
```

Cela s'étend à tous les types, aussi bien ceux qui sont prédéfinis que ceux qui sont définis dans une bibliothèque ou par le programmeur.

Le passage des paramètres

Prenons un autre exemple : la fonction puissance entière d'un réel pourra s'écrire par exemple :

```
static double puiss(double x, int n) {
    /* Il manque ici un test de n positif */
    double res = 1.0;
    int p = 0;
    while (p < n) {
        p++;
        res *= x;
    }
    return res;
}
```

Que se passe-t-il alors lors de l'appel de cette fonction ? Le cas le plus simple est celui de l'utilisation de constantes ; on peut par exemple écrire :

```
double x = puiss(5.4, 6);
```

ce qui va attribuer à `x` la valeur calculée de $5,4^6$. Mais on pourrait aussi écrire :

```
double r = Utils.lireReel("Donnez un nombre réel = ");
int p = Utils.lireEntier("Donnez un nombre entier = ");
double y = puiss(r, p);
```

[2] En toute rigueur, du fait du codage des entiers sur un ensemble fini de bits, on opère en Java sur une partie finie de $\mathbb{Z}$.

en supposant bien entendu que nous ayons écrit également la fonction `lireReel` pour lire au clavier une valeur réelle.

Dans ces deux cas, il est important de noter comment les valeurs sont passées à la fonction. Dans la définition de celle-ci, `x` et `n` sont appelés les *paramètres formels* de la fonction. L'appel `puiss(5.4, 6)` revient à affecter à `x` la valeur `5.4` et à `n` la valeur 6, avant d'exécuter la fonction. De même, l'appel `puiss(r, p)` revient à faire les affectations implicites

$$x \quad \text{(de la fonction)} \quad \leftarrow \quad r \quad \text{(du monde extérieur à la fonction)}$$

et

$$n \quad \text{(de la fonction)} \quad \leftarrow \quad p \quad \text{(du monde extérieur à la fonction)}$$

Il faut bien noter que cette affectation implicite revient à dire qu'on passe la r-valeur des paramètres avec lesquels on appelle la fonction – ici `r` et `p` – et que ceux-ci ne sont donc pas modifiés par la fonction, qui ne « récupère » qu'une copie de leur *r-valeur* (cf. § 2.5). On dit que le passage des paramètres se fait *par valeur*, et en Java c'est le seule mode de passage des paramètres[3].

Une fois que la fonction a rendu une valeur au « monde extérieur » (à celui qui l'a appelée), les paramètres `x` et `n` de la fonction n'ont plus aucune existence – ils n'en ont qu'au sein de la fonction.

3.2.3 Les procédures

La fonction correspond à une définition mathématique précise ; mais il arrive aussi souvent qu'on soit amené à effectuer à plusieurs endroits du programme une même opération, définie par un même ensemble d'instructions. On souhaite là aussi regrouper ces instructions et leur donner un nom. Une fois de plus, il est possible que ce regroupement d'instructions reçoive un ou plusieurs paramètres, mais à la différence d'une fonction, il ne « rend » aucune valeur particulière.

On appelle un tel regroupement une *procédure*. En Java, les procédures sont définies suivant la même syntaxe que les fonctions ; on se contente d'utiliser un mot clé particulier, `void`, à la place du type rendu par la fonction. On peut donc dire par souci de simplicité, même si c'est en fait un abus de langage, qu'en Java, une procédure est une fonction qui ne retourne rien...

Continuons de simplifier notre programme bancaire. Plusieurs informations sont maintenant associées au compte bancaire, et chaque fois que l'on souhaite afficher son solde, il faut énumérer ces informations. Nous allons regrouper toutes les instructions nécessaires pour afficher l'état d'un compte bancaire dans une procédure `étatCompte`, à laquelle nous passerons en paramètre un compte bancaire. Nous en profitons pour améliorer l'affichage de cet état :

```
// Classe Banque - version 2.2

/**
 * Classe contenant un programme de gestion bancaire, utilisant
 * une variable de type CompteBancaire
 * @author Karl Tombre
 * @see    CompteBancaire, Utils
 */

public class Banque {
    public static void étatCompte(CompteBancaire unCompte) {
        System.out.println("Compte numéro " + unCompte.numéro +
                           " ouvert au nom de " + unCompte.nom);
        System.out.println("Adresse du titulaire : " + unCompte.adresse);
        System.out.println("Le solde actuel du compte est de " +
                           unCompte.solde + " euros.");
        System.out.println("************************************************");
```

[3] Dans d'autres langages, on a parfois le choix entre passage par valeur – passage de la *r-valeur* – et passage par référence. Dans ce deuxième cas, on passe en fait la *l-valeur* du paramètre d'appel à la fonction, et celle-ci peut donc agir directement sur la variable passée en paramètre, et non seulement sur ses propres variables, qui désignent les paramètres formels.

```
    }

    public static void main(String[] args) {
        CompteBancaire monCompte = new CompteBancaire();

        // Commencer par demander les valeurs des champs du compte
        monCompte.nom = Utils.lireChaine("Nom du titulaire = ");
        monCompte.adresse = Utils.lireChaine("Son adresse = ");
        monCompte.numéro = Utils.lireEntier("Numéro du compte = ");

        // Solde initial à 0
        monCompte.solde = 0;

        // Afficher une première fois
        étatCompte(monCompte);

        boolean fin = false;     // variable vraie si on veut s'arrêter
        while(true) {  // boucle infinie dont on sort par un break
            String choix = Utils.lireChaine("Votre choix : [D]ébit, [C]rédit, [F]in ? ");
            boolean credit = false;   // variable vraie si c'est un crédit

            // Récupérer la première lettre de la chaîne saisie
            char monChoix = choix.charAt(0);
            switch(monChoix) {
            case 'C':
            case 'c':       // Même chose pour majuscule et minuscule
                credit = true;
                fin = false;
                break;     // Pour ne pas continuer en séquence
            case 'd':
            case 'D':
                credit = false;
                fin = false;
                break;
            case 'f':
            case 'F':
                fin = true;
                break;
            default:
                fin = true;  // On va considérer que par défaut on s'arrête
            }

            if (fin) {
                break;    // sortir de la boucle ici
            }
            else {
                int montant = Utils.lireEntier("Montant à " +
                                               (credit ? "créditer" : "débiter") +
                                               " = ");
                if (credit) {
                    monCompte.solde += montant;
                }
                else {
                    monCompte.solde -= montant;
                }
                // Afficher le nouveau solde
                étatCompte(monCompte);
            }
        }
    }
```

```
}
```

Une exécution de cette nouvelle version du programme done :

```
Nom du titulaire = Facture Portails
Son adresse = 165 rue M*crosoft, Seattle (WA), USA
Numéro du compte = 2000
Compte numéro 2000 ouvert au nom de Facture Portails
Adresse du titulaire : 165 rue M*crosoft, Seattle (WA), USA
Le solde actuel du compte est de 0 euros.
**********************************************
Votre choix : [D]ébit, [C]rédit, [F]in ? C
Montant à créditer = 189000
Compte numéro 2000 ouvert au nom de Facture Portails
Adresse du titulaire : 165 rue M*crosoft, Seattle (WA), USA
Le solde actuel du compte est de 189000 euros.
**********************************************
Votre choix : [D]ébit, [C]rédit, [F]in ? D
Montant à débiter = 654
Compte numéro 2000 ouvert au nom de Facture Portails
Adresse du titulaire : 165 rue M*crosoft, Seattle (WA), USA
Le solde actuel du compte est de 188346 euros.
**********************************************
Votre choix : [D]ébit, [C]rédit, [F]in ? F
```

3.2.4 Le cas particulier de `main`

Depuis le début, nous écrivons `public static void main(String[] args)` en début de programme. Il est temps de l'expliquer. En fait, un programme est une succession d'appels à des procédures et des fonctions. Il faut donc bien un « commencement », c'est-à-dire une procédure qui appelle les autres, qui est à l'origine de l'enchaînement des appels de procédures et de fonctions. Cette procédure particulière s'appelle toujours `main`. Elle prend en paramètre un tableau de chaînes de caractères, qui correspond à des arguments avec lesquels on peut éventuellement lancer le programme. Nous n'utiliserons jamais cette faculté dans ce cours – ce qui signifie que ce tableau d'arguments est vide dans notre cas – mais nous sommes bien obligés de donner la déclaration exacte de `main` pour que le système s'y retrouve...

3.2.5 Surcharge

En Java, rien n'interdit de définir plusieurs fonctions et procédures portant le même nom, à condition que leurs *signatures* – c'est-à-dire le nombre et le type de leurs paramètres formels – soient différentes. Ainsi, on pourra définir (par exemple dans la classe `Utils`) toutes les fonctions et procédures suivantes, de nom `max` :

```
public static int max(int a, int b) {...}
public static float max(char a, char b) {...}
public static double max(double a, double b) {...}
public static void max(char c, int y) {...}
public static void max(String s, int y) {...}
public static char max(String s) {...}
```

À noter toutefois que Java interdit de définir deux fonctions ayant la même signature mais des types de résultat différents, comme par exemple

```
public static int max(int a, int b) {...}
public static String max(int a, int b) {...}
```

Le type des paramètres à l'appel et la signature permettent au compilateur de déterminer laquelle de ces fonctions ou procédures doit être choisie lors de l'appel. Ainsi `max(3, 4)` va appeler la première fonction, alors que `max("Bonjour", 3)` va provoquer l'appel de la procédure de l'avant-dernière ligne.

Attention cependant à ne pas abuser de cette faculté ; pour des raisons de lisibilité par vous-même et par d'autres humains, et pour éviter les confusions, il vaut mieux n'utiliser la surcharge que dans les cas où les différentes fonctions ou procédures de même nom correspondent à la même opération sémantique, mais appliquée à des types différents.

Exercice 4 *Écrire en Java les 3 fonctions dont les définitions mathématiques sont les suivantes :*

$$\begin{aligned} \max &: \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z} \\ &(n, m) \mapsto n \quad si \quad n > m \\ &(n, m) \mapsto m \quad sinon \\ f &: \mathbb{Z} \to \mathbb{Z} \\ &n \mapsto 2^n + 3^n \\ se &: \mathbb{R}^+ \to \mathbb{N} \\ &x \mapsto \lfloor \sqrt{x} \rfloor \end{aligned}$$

3.3 Les tableaux

Le tableau est une structure de données de base que l'on retrouve dans beaucoup de langages. Il permet de stocker en mémoire un ensemble de valeurs de même type, et d'y accéder directement.

À tout type T de Java correspond un type $T[\,]$ qui indique un tableau d'éléments de type T. Pour créer un tableau, on fera une fois de plus appel à l'instruction **new**, en précisant cette fois-ci le nombre d'éléments que doit contenir le tableau. Ainsi, on écrira :

```
byte[] octetBuffer = new byte[1024];
```

pour définir un tableau de 1024 entiers codés sur un octet[4].

On accède à un élément du tableau en donnant son indice, sachant que *l'indice du premier élément d'un tableau est toujours 0 en Java* ! Ainsi `octetBuffer[0]` désigne le premier élément, `octetBuffer[1]` le deuxième, et `octetBuffer[1023]` le dernier. Par ailleurs, la taille d'un tableau est donnée par la construction *nomDuTableau*`.length` : ainsi, dans l'exemple ci-dessus, `octetBuffer.length` vaut 1024.

Attention ! Si vous déclarez un tableau d'objets, c'est-à-dire de variables déclarées d'un type défini par une classe, il ne faut pas oublier de créer les objets eux-mêmes ! Imaginons que nous voulions gérer un tableau de 10 comptes bancaires. En écrivant :

```
CompteBancaire[] tabComptes = new CompteBancaire[10];
```

on crée un tableau de 10 *références* à des comptes bancaires. La variable `tabComptes[0]`, par exemple, est de type `CompteBancaire`, mais si je veux qu'elle désigne un objet effectif et qu'elle ne vaille pas `null`, je dois également écrire :

```
tabComptes[0] = new CompteBancaire();
```

[4] En fait, on peut aussi écrire `byte octetBuffer[]` ; c'est-à-dire mettre les crochets après le nom de la variable et non après le type. Si vous êtes amené plus tard à « jongler » entre les langages C++ et Java, vous préférerez sans doute cette seconde solution, pour éviter la dyslexie, car c'est la seule notation autorisée en C++. Néanmoins, je conseille plutôt d'utiliser la notation donnée ici, qui a le mérite de désigner clairement la variable `octetBuffer` comme étant de type `byte[]`, c'est-à-dire tableau d'octets.

comme nous l'avons vu précédemment.

Illustrons tout cela en faisant une fois de plus évoluer notre célèbre programme bancaire. Nous allons maintenant justement gérer 10 comptes dans un tableau. Il faut donc commencer par initialiser ces 10 comptes. Ensuite, le programme est modifié en demandant l'indice dans le tableau avant de demander le montant à créditer ou débiter. Notez que je devrais en fait vérifier que l'indice saisi est bien dans les bornes valides, c'est-à-dire dans l'intervalle $[0, 10[$.

En prime, cet exemple me donne l'occasion d'illustrer l'emploi d'une itération avec la construction `for` :

```
// Classe Banque - version 2.3

/**
 * Classe contenant un programme de gestion bancaire, utilisant
 * un tableau de comptes bancaires
 * @author Karl Tombre
 * @see    CompteBancaire, Utils
 */

public class Banque {
    public static void étatCompte(CompteBancaire unCompte) {
        System.out.println("Compte numéro " + unCompte.numéro +
                           " ouvert au nom de " + unCompte.nom);
        System.out.println("Adresse du titulaire : " + unCompte.adresse);
        System.out.println("Le solde actuel du compte est de " +
                           unCompte.solde + " euros.");
        System.out.println("***********************************************");
    }

    public static void main(String[] args) {
        CompteBancaire[] tabComptes = new CompteBancaire[10];

        // Initialisation des comptes
        for (int i = 0 ; i < tabComptes.length ; i++) {
            // D'abord créer le compte !!
            tabComptes[i] = new CompteBancaire();
            // Et puis lire les données
            tabComptes[i].nom = Utils.lireChaine("Nom du titulaire = ");
            tabComptes[i].adresse = Utils.lireChaine("Son adresse = ");
            tabComptes[i].numéro = Utils.lireEntier("Numéro du compte = ");
            // Solde initial à 0
            tabComptes[i].solde = 0;
        }

        boolean fin = false;     // variable vraie si on veut s'arrêter
        while(true) {  // boucle infinie dont on sort par un break
            String choix = Utils.lireChaine("Votre choix : [D]ébit, [C]rédit, [F]in ? ");
            boolean credit = false;   // variable vraie si c'est un crédit

            // Récupérer la première lettre de la chaîne saisie
            char monChoix = choix.charAt(0);
            switch(monChoix) {
            case 'C':
            case 'c':       // Même chose pour majuscule et minuscule
                credit = true;
                fin = false;
                break;     // Pour ne pas continuer en séquence
            case 'd':
            case 'D':
```

```
                credit = false;
                fin = false;
                break;
            case 'f':
            case 'F':
                fin = true;
                break;
            default:
                fin = true;  // On va considérer que par défaut on s'arrête
            }

            if (fin) {
                break;    // sortir de la boucle ici
            }
            else {
                int indice = Utils.lireEntier("Indice dans le tableau des comptes = ");
                int montant = Utils.lireEntier("Montant à " +
                                               (credit ? "créditer" : "débiter") +
                                               " = ");
                if (credit) {
                    tabComptes[indice].solde += montant;
                }
                else {
                    tabComptes[indice].solde -= montant;
                }
                // Afficher le nouveau solde
                étatCompte( tabComptes[indice] );
            }
        }
    }
}
```

Une exécution de ce programme donne la trace suivante :

```
Nom du titulaire = Karl
Son adresse = Sornéville
Numéro du compte = 123

... Je vous passe un certain nombre de lignes - c'est fastidieux

Nom du titulaire = Bart
Son adresse = Bruxelles
Numéro du compte = 76243
Votre choix : [D]ébit, [C]rédit, [F]in ? D
Indice dans le tableau des comptes = 4
Montant à débiter = 8725
Compte numéro 524 ouvert au nom de François
Adresse du titulaire : Strasbourg
Le solde actuel du compte est de -8725 euros.
************************************************
Votre choix : [D]ébit, [C]rédit, [F]in ? C
Indice dans le tableau des comptes = 9
Montant à créditer = 8753
Compte numéro 76243 ouvert au nom de Bart
Adresse du titulaire : Bruxelles
Le solde actuel du compte est de 8753 euros.
************************************************
Votre choix : [D]ébit, [C]rédit, [F]in ? C
Indice dans le tableau des comptes = 0
```

```
Montant à créditer = 7652
Compte numéro 123 ouvert au nom de Karl
Adresse du titulaire : Sornéville
Le solde actuel du compte est de 7652 euros.
**********************************************
Votre choix : [D]ébit, [C]rédit, [F]in ? F
```

NB : Habituellement, on n'initialise pas un tableau au moment de sa création, mais on le « remplit » comme dans l'exemple que nous venons de donner. Cependant, il y a des cas où l'on souhaite initialiser un tableau ; on donnera alors entre accolades la liste des valeurs initiales, séparées par des virgules, comme l'illustre l'exemple suivant, où l'on définit un tableau de taille 12, initialisé directement par 12 valeurs (la taille du tableau est déduite par le compilateur du nombre de valeurs données en initialisation) :

```
static String[] moisEnFrançais = {"janvier", "février", "mars", "avril",
                                  "mai", "juin", "juillet", "août",
                                  "septembre", "octobre", "novembre",
                                  "décembre"};
```

Exercice 5 *Écrire les fonctions :*

- *`max(int[] tab)` qui retourne le plus grand élément d'un tableau d'entiers ;*
- *`somme(int[] tab)` qui calcule la somme des éléments d'un tableau d'entiers ;*
- *`estALIndice(int[] tab, int n)` qui retourne -1 si `n` n'est pas dans le tableau `tab`, et le premier indice dans `tab` auquel se trouve `n` sinon.*

Chapitre 4

Programmation objet

> *The "global" view of systems can be likened to that of the creator of a system who must know all interfaces and system types. In contrast, objects in a system are like Plato's cave-dwellers who can interact with the universe in which they live only in terms of observable communications, represented by the shadows on the walls of their cave. Creators of a system are concerned with classification while objects that inhabit a system are concerned primarily with communication.*
>
> Peter Wegner

L'école de programmation Algol (cf. chapitre A) a été la première à proposer une approche de la programmation qui est devenue classique : un programme est considéré comme un ensemble de procédures et un ensemble de données, *séparé*, sur lequel agissent ces procédures. Ce principe a parfois été résumé par une équation célèbre : $\boxed{\textit{Programmes} = \textit{Algorithmes} + \textit{Structures de données}}$. Les méthodes d'analyse qui vont de pair consistent à *diviser pour régner*, c'est-à-dire à découper la tâche à effectuer en un ensemble de modules indépendants, considérés comme des boîtes noires.

Cette technique a fait ses preuves depuis longtemps et continue à être à la base de la programmation structurée. Mais elle atteint ses limites lorsque l'univers sur lequel opèrent les programmes évolue, ce qui est en fait la régle générale plutôt que l'exception : de nouveaux types de données doivent être pris en compte, le contexte applicatif dans lequel s'inscrit le programme change, les utilisateurs du programme demandent de nouvelles fonctionnalités et l'interopérabilité du programme avec d'autres programmes, etc. Or dans un langage comme Pascal ou C, les applications sont découpées en procédures et en fonctions ; cela permet une bonne décomposition des traitements à effectuer, mais le moindre changement de la structuration des *données* est difficile à mettre en œuvre, et peut entraîner de profonds bouleversements dans l'organisation de ces procédures et fonctions.

C'est ici que la programmation objet apporte un « plus » fondamental, grâce à la notion d'*encapsulation* : les données et les procédures qui manipulent ces données sont regroupées dans une même entité, appelée l'*objet*. Les détails d'implantation de l'objet restent cachés : le monde extérieur n'a accès à ses données que par l'intermédiaire d'un ensemble d'opérations qui constituent l'*interface* de l'objet. Le programmeur qui utilise un objet dans son programme n'a donc pas à se soucier de sa représentation physique ; il peut raisonner en termes d'abstractions.

Java est l'un des principaux représentants actuels de la famille des langages à objets. Dans ce chapitre, nous allons nous familiariser progressivement avec les principales caractéristiques de cette famille, en nous appuyant comme dans les chapitres précédents sur le langage Java pour illustrer notre propos. Mais il faut savoir qu'à certains choix conceptuels près, on retrouvera le même style de programmation dans d'autres langages de la famille.

4.1 Retour sur la classe

Nous avons déjà introduit la classe comme permettant de regrouper des variables pour former une même entité. Mais en fait, la classe est plus que cela : elle est la description d'une famille d'objets

ayant même structure *et même comportement*. Elle permet donc non seulement de regrouper un ensemble de données, mais aussi les procédures et fonctions qui agissent sur ces données. Dans le vocabulaire de la programmation objet, ces procédures et fonctions sont appelées les *méthodes* ; elles représentent le comportement commun de tous les objets appartenant à la classe.

Revenons à notre exemple bancaire pour illustrer notre propos. Nous avions défini au § 3.1 une classe `CompteBancaire` qui contient quatre variables. Par ailleurs, nous avions défini au § 3.2.3 une procédure `étatCompte` dans la classe `Banque` ; mais celle-ci accède directement aux variables de la classe `CompteBancaire`, ce qui est contraire au principe d'encapsulation précédemment énoncé. De même, si on veut vraiment considérer `CompteBancaire` comme une « boîte noire », dans laquelle les détails d'implantation sont cachés, il n'est pas judicieux de laisser le programme de la classe `Banque` accéder directement à la variable interne `solde`, pour l'incrémenter ou la décrémenter. Plus généralement, on souhaite avoir le contrôle sur les opérations autorisées sur les attributs internes d'une classe, ce qui est impossible à réaliser si les programmes du « monde extérieur à la classe » peuvent accéder directement à ces attributs.

Nous aboutissons donc à la spécification de l'interface souhaitable pour un objet de type compte bancaire :

– il faut pouvoir créditer et débiter le compte,
– il faut être en mesure d'afficher son état.

D'autres méthodes pourront bien entendu venir compléter cette première interface rudimentaire. À partir de là, écrivons une nouvelle version de la classe `CompteBancaire` ; notez bien la manière de « penser » quand on écrit une méthode, par exemple `créditer` : « je » reçois en argument un montant, et j'incrémente « mon » solde de ce montant. Notez aussi que pour interdire l'accès direct aux variables internes, on fait précéder leur déclaration du mot clé `private`, les méthodes, elles, étant caractérisées par le mot clé `public`, qui indique qu'elles sont accessibles de l'extérieur de l'objet.

```
// Classe CompteBancaire - version 2.0
/**
 * Classe représentant un compte bancaire et les méthodes qui lui
 * sont associées.
 * @author Karl Tombre
 */

public class CompteBancaire {
    private String nom;       // le nom du client
    private String adresse;   // son adresse
    private int numéro;       // numéro du compte
    private int solde;        // solde du compte

    // Les méthodes
    public void créditer(int montant) {
        solde += montant;
    }
    public void débiter(int montant) {
        solde -= montant;
    }
    public void afficherEtat() {
        System.out.println("Compte numéro " + numéro +
                           " ouvert au nom de " + nom);
        System.out.println("Adresse du titulaire : " + adresse);
        System.out.println("Le solde actuel du compte est de " +
                           solde + " euros.");
        System.out.println("***********************************************");
    }
```

```
}
```

Nous avons déjà vu qu'il y a un lien fort entre les notions de classe et de *type de données*. Le type en lui-même peut être vu comme *l'interface* de la classe, c'est-à-dire la spécification – ou la signature – des opérations valides sur les objets de la classe, alors que la classe elle-même est une mise en œuvre concrète de cette interface.

4.1.1 Fonctions d'accès

Une conséquence de l'encapsulation est que les variables d'instance `nom`, `adresse`, `numéro` et `solde`, sont maintenant cachées au monde extérieur. Malgré tous les avantages de cet état de fait, on peut avoir besoin de rendre certaines de ces informations accessibles, ne serait-ce qu'en lecture. Faudrait-il pour autant les rendre à nouveau publiques dans ce cas ? Non ! La solution est alors de munir l'interface de la classe de *fonctions d'accès* en lecture et/ou en écriture. Ainsi, si je veux permettre aux utilisateurs de la classe de consulter en lecture uniquement le nom et l'adresse du titulaire du compte, je pourrais définir dans la classe `CompteBancaire` les méthodes suivantes :

```
public String getNom() {
    return nom;
}
public String getAdresse() {
    return adresse;
}
```

L'avantage, par rapport au fait de rendre les variables publiques, est qu'on ne permet que l'accès en lecture, pas en écriture (c'est-à-dire en modification). Et même si on voulait également permettre l'accès en écriture à l'adresse, par exemple, il faut définir la fonction d'accès

```
public void setAdresse(String nouvelleAdresse) {
    adresse = nouvelleAdresse;
}
```

plutôt que de rendre la variable publique. En effet, dans une vraie application, on souhaitera probablement vérifier la cohérence de l'adresse, mettre à jour certaines tables statistiques de l'agence, voire activer l'impression d'un nouveau chéquier, par exemple, toutes choses qui sont possibles dans une méthode telle que `setAdresse`, mais qui sont impossibles à contrôler si le « monde extérieur » peut directement modifier la valeur de la variable `adresse`.

Le fait de munir ses classes de fonctions d'accès, même triviales, au lieu de rendre certaines variables publiques, est donc une « bonne pratique » en programmation qu'il vaut mieux acquérir tout de suite, même si cela peut parfois sembler fastidieux ou inutile sur les exemples simplistes que nous utilisons en cours. Vous pouvez voir illustré l'ajout de telles fonctions à la classe `CompteBancaire` au § 6.6, quand nous en aurons besoin pour définir une interface utilisateur. Jusqu'à ce moment-là, nous nous en passerons pour ne pas alourdir les exemples...

4.2 Les objets

Comme nous l'avons vu au § 3.1.1, une classe peut être considérée comme un « moule », et on crée des objets par « moulage » à partir du modèle donné par la classe. Cette opération, qui se fait grâce à l'instruction `new` déjà vue, s'appelle l'*instanciation*, car l'objet créé est dit être une *instance* de sa classe. Celle-ci permet donc de reproduire autant d'exemplaires – d'instances – que nécessaire.

Chaque instance ainsi créée possède son propre exemplaire de chacune des variables définies dans la classe ; on les appelle ses *variables d'instance*. Ainsi, si je crée deux objets de type `CompteBancaire`, ils auront chacun leur propre variable `solde`, leur propre variable `nom`, etc.

Se pose maintenant le problème de l'initialisation de ces variables, au moment de la création. Jusqu'à maintenant, dans notre programme bancaire, nous initialisions ces variables après la création par `new`, par simple affectation de leur valeur. Mais le principe d'encapsulation que nous avons

adopté, et la protection que nous avons donnée aux variables par le mot clé `private`, nous interdit maintenant d'accéder directement à ces variables pour leur affecter une valeur.

Il nous faut donc un mécanisme spécifique d'initialisation ; en Java, ce mécanisme s'appelle un *constructeur*. Un constructeur est une procédure d'initialisation ; il porte toujours le même nom que la classe pour laquelle il est défini – sans attributs de type –, et on peut lui donner les paramètres que l'on veut. Modifions donc notre classe `CompteBancaire` pour lui ajouter un constructeur :

```
// Classe CompteBancaire - version 2.1

/**
 * Classe représentant un compte bancaire et les méthodes qui lui
 * sont associées.
 * @author Karl Tombre
 */

public class CompteBancaire {
    private String nom;       // le nom du client
    private String adresse;   // son adresse
    private int numéro;       // numéro du compte
    private int solde;        // solde du compte

    // Constructeur : on reçoit en paramètres les valeurs du nom,
    // de l'adresse et du numéro, et on met le solde à 0 par défaut
    CompteBancaire(String unNom, String uneAdresse, int unNuméro) {
        nom = unNom;
        adresse = uneAdresse;
        numéro = unNuméro;
        solde = 0;
    }

    // Les méthodes
    public void créditer(int montant) {
        solde += montant;
    }
    public void débiter(int montant) {
        solde -= montant;
    }
    public void afficherEtat() {
        System.out.println("Compte numéro " + numéro +
                           " ouvert au nom de " + nom);
        System.out.println("Adresse du titulaire : " + adresse);
        System.out.println("Le solde actuel du compte est de " +
                           solde + " euros.");
        System.out.println("************************************************");
    }
}
```

Il est temps d'utiliser des objets de cette classe dans notre programme bancaire, que nous devons bien évidemment modifier pour tenir compte de l'approche objet que nous avons prise. Vous pouvez noter plusieurs choses dans l'exemple qui suit :

- le passage des arguments au constructeur au moment de l'allocation mémoire *via* le mot clé `new` ;
- l'appel des méthodes de l'interface par le moyen de la notation pointée ;
- le fait que nous ne voyons plus du tout apparaître dans ce programme les détails internes d'implantation d'un compte bancaire. Le concepteur de `CompteBancaire` est donc libre de modifier cette implantation interne, sans que cela influe sur le programme bancaire, à condition bien sûr que l'interface reste valide.

```
// Classe Banque - version 3.0

/**
 * Classe contenant un programme de gestion bancaire, utilisant
 * un tableau de comptes bancaires
 * @author Karl Tombre
 * @see    CompteBancaire, Utils
 */

public class Banque {
    public static void main(String[] args) {
        CompteBancaire[] tabComptes = new CompteBancaire[10];

        // Initialisation des comptes
        for (int i = 0 ; i < tabComptes.length ; i++) {
            // Lire les valeurs d'initialisation
            String monNom = Utils.lireChaine("Nom du titulaire = ");
            String monAdresse = Utils.lireChaine("Son adresse = ");
            int monNuméro = Utils.lireEntier("Numéro du compte = ");
            // Créer le compte -- notez la syntaxe avec new
            tabComptes[i] = new CompteBancaire(monNom, monAdresse, monNuméro);
        }

        boolean fin = false;     // variable vraie si on veut s'arrêter
        while(true) {  // boucle infinie dont on sort par un break
            String choix = Utils.lireChaine("Votre choix : [D]ébit, [C]rédit, [F]in ? ");
            boolean credit = false;   // variable vraie si c'est un crédit

            // Récupérer la première lettre de la chaîne saisie
            char monChoix = choix.charAt(0);
            switch(monChoix) {
            case 'C':
            case 'c':       // Même chose pour majuscule et minuscule
                credit = true;
                fin = false;
                break;     // Pour ne pas continuer en séquence
            case 'd':
            case 'D':
                credit = false;
                fin = false;
                break;
            case 'f':
            case 'F':
                fin = true;
                break;
            default:
                fin = true;  // On va considérer que par défaut on s'arrête
            }

            if (fin) {
                break;    // sortir de la boucle ici
            }
            else {
                int indice = Utils.lireEntier("Indice dans le tableau des comptes = ");
                int montant = Utils.lireEntier("Montant à " +
                                               (credit ? "créditer" : "débiter") +
                                               " = ");
                if (credit) {
                    tabComptes[indice].créditer(montant);
```

```
                }
                else {
                    tabComptes[indice].débiter(montant);
                }
                // Afficher le nouveau solde
                tabComptes[indice].afficherEtat();
            }
        }
    }
}
```

Pour conclure ce paragraphe, insistons à nouveau sur le point suivant :
- À l'intérieur de la classe, les variables d'instance et les méthodes de la classe, publiques ou privées, sont accessibles directement, comme nous l'avons vu avec la définition des méthodes `débiter`, `créditer` et `afficherEtat`, dans la classe `CompteBancaire`.
- En revanche, à l'extérieur de la classe, on n'accède aux variables et méthodes publiques de l'instance d'un objet que par la notation pointée. Ainsi, dans l'exemple que nous venons de voir, `tabComptes[indice].créditer(montant)` signifie que l'on appelle « la méthode `créditer` de l'objet `tabComptes[indice]` ». On ne peut pas accéder aux données et méthodes privées.

Exercice 6 *On souhaite écrire un programme Java qui gère un panier d'emplettes pour un site de commerce électronique. On représentera un item des emplettes par une classe* `Item` *dont un embryon est donné ci-dessous ; compléter cette classe en écrivant le corps du constructeur et de la méthode* `prix`*.*

```
public class Item {
    private String nom;
    private int quantité;
    private double prixUnitaire;

    Item(String n, int q, double pu) {
        // À faire
    }

    public double prix() {
        // À faire
    }
}
```

4.3 Méthodes et variables de classe

Nous avons vu que les méthodes et les variables d'instance sont propres à chaque instance ; ainsi, chaque objet de type `CompteBancaire` possède son propre exemplaire de la variable `solde` – ce qui permet fort heureusement à chaque titulaire de compte de disposer de son propre solde et non d'un solde commun !

Mais il peut être parfois nécessaire de disposer de méthodes et de variables qui soient propres à la classe, et non aux instances, et qui n'existent donc qu'en exemplaire unique, quel que soit le nombre d'instances. On les appelle des méthodes et variables de classe, et elles sont introduites par le mot clé `static`.

Illustrons tout de suite notre propos. Nous souhaitons modifier la classe `CompteBancaire` de manière à attribuer automatiquement le numéro de compte, par incrémentation d'une variable de classe, que nous appellerons `premierNuméroDisponible`. Celle-ci est initialisée à 1[1], et chaque fois qu'on crée un nouveau compte – concrètement, dans le constructeur de la classe – elle est incrémentée :

[1] Notez bien que l'initialisation de la variable de classe ne se fait pas dans un constructeur, mais directement dans la classe, au moment où elle est déclarée.

```
// Classe CompteBancaire - version 2.2

/**
 * Classe représentant un compte bancaire et les méthodes qui lui
 * sont associées.
 * @author Karl Tombre
 */

public class CompteBancaire {
    private String nom;       // le nom du client
    private String adresse;   // son adresse
    private int numéro;       // numéro du compte
    private int solde;        // solde du compte

    // Variable de classe
    public static int premierNuméroDisponible = 1;

    // Constructeur : on reçoit en paramètres les valeurs du nom et
    // de l'adresse, on met le solde à 0 par défaut, et on récupère
    // automatiquement un numéro de compte
    CompteBancaire(String unNom, String uneAdresse) {
        nom = unNom;
        adresse = uneAdresse;
        numéro = premierNuméroDisponible;
        solde = 0;
        // Incrémenter la variable de classe
        premierNuméroDisponible++;
    }

    // Les méthodes
    public void créditer(int montant) {
        solde += montant;
    }
    public void débiter(int montant) {
        solde -= montant;
    }
    public void afficherEtat() {
        System.out.println("Compte numéro " + numéro +
                           " ouvert au nom de " + nom);
        System.out.println("Adresse du titulaire : " + adresse);
        System.out.println("Le solde actuel du compte est de " +
                           solde + " euros.");
        System.out.println("************************************************");
    }
}
```

Comme nous avons modifié la signature du constructeur, nous devons aussi modifier légèrement notre programme bancaire ; je ne donne ici que les premières lignes de la nouvelle mouture, le reste ne changeant pas. Notez que l'on ne demande plus que le nom et l'adresse :

```
// Classe Banque - version 3.1

/**
 * Classe contenant un programme de gestion bancaire, utilisant
 * un tableau de comptes bancaires
 * @author Karl Tombre
 * @see    CompteBancaire, Utils
 */
```

```
public class Banque {
    public static void main(String[] args) {
        CompteBancaire[] tabComptes = new CompteBancaire[10];

        // Initialisation des comptes
        for (int i = 0 ; i < tabComptes.length ; i++) {
            // Lire les valeurs d'initialisation
            String monNom = Utils.lireChaine("Nom du titulaire = ");
            String monAdresse = Utils.lireChaine("Son adresse = ");
            // Créer le compte -- notez la syntaxe avec new
            tabComptes[i] = new CompteBancaire(monNom, monAdresse);
        }

        // etc.
```

À part la modification sur le nombre de questions posées à l'utilisateur en phase de création du tableau, le programme a le même comportement qu'avant :

```
Nom du titulaire = Karl
Son adresse = Sornéville
Nom du titulaire = Luigi
Son adresse = Rome

... Je vous passe un certain nombre de lignes - c'est fastidieux

Nom du titulaire = Robert
Son adresse = Vandoeuvre
Votre choix : [D]ébit, [C]rédit, [F]in ? D
Indice dans le tableau des comptes = 3
Montant à débiter = 123
Compte numéro 4 ouvert au nom de François
Adresse du titulaire : Strasbourg
Le solde actuel du compte est de -123 euros.
**********************************************
Votre choix : [D]ébit, [C]rédit, [F]in ? C
Indice dans le tableau des comptes = 0
Montant à créditer = 9800
Compte numéro 1 ouvert au nom de Karl
Adresse du titulaire : Sornéville
Le solde actuel du compte est de 9800 euros.
**********************************************
Votre choix : [D]ébit, [C]rédit, [F]in ? F
```

4.3.1 Retour sur la procédure `main`

Vous avez maintenant les éléments vous permettant de mieux comprendre la syntaxe *a priori* rébarbative que nous vous avons imposée dès le premier programme, à savoir `public static void main(String[] args)` :

- En Java, la classe est l'unité de compilation et rien ne peut être défini en dehors d'une classe ; la classe `Banque` joue donc le rôle de « classe d'hébergement » du programme.
- Cependant, nous ne manipulons jamais d'*instances* de cette classe `Banque` – la procédure `main` doit exister en exemplaire unique pour la classe, c'est donc une méthode de classe, d'où le mot clé `static`. D'ailleurs, dans la version 2.2 de la classe `Banque` (cf. § 3.2.3), nous avions une autre procédure, `étatCompte`, qui était également définie comme méthode de classe, avec le mot clé `static`.

– Nous avons déjà expliqué que `main` est un nom réservé au « commencement » du programme (cf. § 3.2.4) et qu'il prend en paramètres un tableau de chaînes de caractères, correspondant à des arguments passés au moment du lancement du programme, d'où le paramètre formel `String[] args`.

4.3.2 Comment accéder aux variables et méthodes de classe ?

Nous ne nous sommes pas posés pour l'instant la question de l'accès aux variables et méthodes de classe, car jusqu'à maintenant nous ne les avons utilisées qu'au sein de la classe dans laquelle elles sont définies, et la notation directe s'applique donc, comme pour les méthodes et variables d'instance. Mais la variable de classe `premierNuméroDisponible` ayant été déclarée publique, on pourrait imaginer que l'on souhaite y accéder directement depuis un programme ou une autre classe. Dans un programme extérieur, il n'est pas pour autant conseillé d'écrire[2] :

```
CompteBancaire c = new CompteBancaire("Capitaine Haddock", "Boucherie Sanzot");
System.out.println("Le premier numéro disponible est : " + c.premierNuméroDisponible);
```

car `premierNuméroDisponible` appartient à la classe, et non à l'instance `c`. Il vaut donc mieux utiliser la notation pointée en précisant que c'est à la classe qu'appartient la variable :

```
System.out.println("Le premier numéro disponible est : " +
                   CompteBancaire.premierNuméroDisponible);
```

C'est d'ailleurs ce que nous ferons lorsque nous aurons besoin de sauvegarder cette variable dans un fichier (§ 6.3.2).

Vous avez d'ailleurs déjà utilisé – sans le savoir – des variables de classe dans les programmes vus jusqu'à maintenant. Par exemple, `System.out` désigne la variable de classe `out` de la classe `System`, comme nous le verrons au § 6.3.1.

Les fonctions `lireChaine` et `lireEntier` sont également des méthodes de classe que nous avons définies dans la classe `Utils`, et que nous avons utilisées en employant la syntaxe que nous venons de voir, à savoir `Utils.lireChaine`.

4.4 Exemple d'une classe prédéfinie : `String`

La *chaîne de caractères* est une structure de données très fréquente dans les langages de programmation. Dans beaucoup de langages, elle est définie comme un tableau de caractères. Mais en Java, c'est l'approche objet qui est employée, et les bibliothèques standards de Java fournissent la classe `String`, que nous avons déjà eu l'occasion d'utiliser.

`String` fournit les opérations courantes sur les chaînes, comme nous allons le voir dans ce paragraphe. Ces opérations sont disponibles sous forme de méthodes définies dans la classe `String`.

Mais `String` a aussi une particularité. C'est la seule classe pour laquelle un opérateur est redéfini[3] : nous avons déjà eu l'occasion d'utiliser l'opérateur `+` sur des chaînes de caractères, et nous avons vu qu'il permet la concaténation de deux chaînes, comme dans :
`"Adresse du titulaire : " + adresse`.

Par ailleurs, Java prévoit une autre facilité syntaxique pour les chaînes de caractères, à savoir l'emploi des guillemets (`"`). Quand nous écrivons `String salutation = "bonjour";` nous créons en fait une constante de type `String`, qui vaut `"bonjour"`. L'affectation de celle-ci à la variable `salutation` revient tout simplement à faire « pointer » l'adresse contenue dans celle-ci sur la « case » où est stockée cette constante – souvenez-vous que les variables de types définis par des classes sont toujours des références (cf. § 3.1.1). On pourrait bien sûr recourir plus classiquement à un constructeur, comme pour toute autre classe, en écrivant `String salutation = new String("bonjour");` mais ce serait moins efficace, puisqu'on crée dans ce cas deux objets de type `String`, la constante `"bonjour"` puis la nouvelle chaîne *via* le constructeur.

[2] Bien que Java le permette...
[3] En C++, un langage proche de Java par la syntaxe, on peut redéfinir des opérateurs sur toutes les classes.

Sans chercher à être exhaustifs, nous donnons ici une liste très partielle des méthodes disponibles dans la classe `String` ; comme pour toutes les autres classes des bibliothèques de base de Java, la documentation complète est disponible à partir de ma page web à l'école (rubrique *APIs* de Java). J'indique à chaque fois la *signature* des méthodes ; pour `compareTo` par exemple, cette signature est `int compareTo(String)`, ce qui signifie qu'on l'utilisera de la manière suivante :

```
String s = ....;
String t = ...;
if (s.compareTo(t) > 0) {
  System.out.println("La chaîne " + s + " est supérieure à la chaîne " + t);
}
```

Méthode	Description
`char charAt(int)`	Rend le caractère à la position indiquée. Nous avons déjà utilisé cette méthode.
`int compareTo(String)`	Comparaison lexicographique avec la chaîne donnée en paramètre. Rend 0 si les deux chaînes sont égales, un nombre négatif si la chaîne est inférieure à celle donnée en paramètre, un nombre positif si elle est supérieure.
`boolean endsWith(String)`	Teste si la chaîne se termine par le suffixe donné en paramètre.
`boolean equals(Object)`	Teste l'égalité avec un autre objet. Nous avons déjà utilisé cette méthode.
`boolean equalsIgnoreCase(String)`	Teste l'égalité avec une autre chaîne sans tenir compte des majuscules/minuscules.
`int lastIndexOf(String)`	Rend l'index dans la chaîne de la dernière position de la sous-chaîne donnée en paramètre.
`int length()`	Rend la longueur de la chaîne.
`String toLowerCase()`	Conversion de la chaîne en minuscules.
`String toUpperCase()`	Conversion de la chaîne en majuscules.

4.4.1 Application : recherche plus conviviale du compte

Nous allons appliquer l'interface que nous venons de voir à notre programme bancaire. En effet, jusqu'à maintenant, nous demandions à l'utilisateur de donner l'indice dans le tableau des comptes, ce qui est peu naturel. Nous allons maintenant retrouver le compte dans le tableau par simple indication du nom du titulaire. Il faudra alors parcourir le tableau pour chercher un compte dont le titulaire correspond au nom cherché. Nous permettrons à l'utilisateur de ne pas tenir compte des majuscules ou des minuscules, en utilisant la méthode `equalsIgnoreCase`

En préalable, notons que nous n'avons pas pour l'instant de moyen d'accéder au nom du titulaire du compte, puisque la variable `nom` est privée. Nous pourrions la déclarer publique, mais ce serait contraire au principe d'encapsulation ; préférons donc définir des méthodes d'accès en lecture au nom et à l'adresse du titulaire du compte. Notez bien que si j'avais déclaré ces variables publiques, je pourrais les lire, mais aussi les *modifier* depuis l'extérieur de la classe, alors qu'avec la solution préconisée, je ne permets que l'accès en lecture :

```
// Classe CompteBancaire - version 2.3

/**
 * Classe représentant un compte bancaire et les méthodes qui lui
 * sont associées.
 * @author Karl Tombre
 */

public class CompteBancaire {
    private String nom;       // le nom du client
    private String adresse;   // son adresse
```

```
    private int numéro;       // numéro du compte
    private int solde;        // solde du compte

    // Variable de classe
    public static int premierNuméroDisponible = 1;

    // Constructeur : on reçoit en paramètres les valeurs du nom et
    // de l'adresse, on met le solde à 0 par défaut, et on récupère
    // automatiquement un numéro de compte
    CompteBancaire(String unNom, String uneAdresse) {
        nom = unNom;
        adresse = uneAdresse;
        numéro = premierNuméroDisponible;
        solde = 0;
        // Incrémenter la variable de classe
        premierNuméroDisponible++;
    }

    // Les méthodes
    public void créditer(int montant) {
        solde += montant;
    }
    public void débiter(int montant) {
        solde -= montant;
    }
    public void afficherEtat() {
        System.out.println("Compte numéro " + numéro +
                           " ouvert au nom de " + nom);
        System.out.println("Adresse du titulaire : " + adresse);
        System.out.println("Le solde actuel du compte est de " +
                           solde + " euros.");
        System.out.println("***********************************************");
    }
    // Accès en lecture
    public String nom() {
        return this.nom;
    }
    public String adresse() {
        return this.adresse;
    }
}
```

Vous remarquerez que j'ai donné à ces méthodes les mêmes noms qu'aux variables auxquelles elles accèdent. Dans le corps des méthodes, j'utilise le mot clé **`this`**, qui référencie toujours l'objet courant – « moi-même », qui possède les variables et les instances[4] – et la notation **`this.nom`** indique donc « ma variable privée **`nom`** ». Ceci n'était pas *stricto sensu* nécessaire dans le cas présent, car il n'y a pas ambiguité, mais cela augmente la lisibilité du programme quand il y a deux attributs de même nom.

Voici le programme bancaire modifié ; vous noterez l'emploi de la méthode **`equalsIgnoreCase`** de la classe **`String`** :

```
// Classe Banque - version 3.2

/**
 * Classe contenant un programme de gestion bancaire, utilisant
 * un tableau de comptes bancaires
```

[4]De manière caractéristique, c'est le mot clé **`self`** qui est utilisé dans le langage Smalltalk pour désigner l'objet courant.

```
 * @author Karl Tombre
 * @see    CompteBancaire, Utils
 */

public class Banque {
    public static void main(String[] args) {
        CompteBancaire[] tabComptes = new CompteBancaire[10];

        // Initialisation des comptes
        for (int i = 0 ; i < tabComptes.length ; i++) {
            // Lire les valeurs d'initialisation
            String monNom = Utils.lireChaine("Nom du titulaire = ");
            String monAdresse = Utils.lireChaine("Son adresse = ");
            // Créer le compte -- notez la syntaxe avec new
            tabComptes[i] = new CompteBancaire(monNom, monAdresse);
        }

        boolean fin = false;     // variable vraie si on veut s'arrêter
        while(true) {  // boucle infinie dont on sort par un break
            String choix = Utils.lireChaine("Votre choix : [D]ébit, [C]rédit, [F]in ? ");
            boolean credit = false;   // variable vraie si c'est un crédit

            // Récupérer la première lettre de la chaîne saisie
            char monChoix = choix.charAt(0);
            switch(monChoix) {
            case 'C':
            case 'c':       // Même chose pour majuscule et minuscule
                credit = true;
                fin = false;
                break;     // Pour ne pas continuer en séquence
            case 'd':
            case 'D':
                credit = false;
                fin = false;
                break;
            case 'f':
            case 'F':
                fin = true;
                break;
            default:
                fin = true;  // On va considérer que par défaut on s'arrête
            }

            if (fin) {
                break;    // sortir de la boucle ici
            }
            else {
                // Demander un nom
                String nomAChercher = Utils.lireChaine("Nom du client = ");
                boolean trouvé = false;  // rien trouvé pour l'instant
                int indice = 0;
                for (int i = 0 ; i < tabComptes.length ; i++) {
                    if (nomAChercher.equalsIgnoreCase(tabComptes[i].nom())) {
                        trouvé = true; // j'ai trouvé
                        indice = i;    // mémoriser l'indice
                        break;         // plus besoin de continuer la recherche
                    }
                }
```

```
                if (trouvé) { // Si j'ai trouvé quelque chose
                    int montant = Utils.lireEntier("Montant à " +
                                                   (credit ? "créditer" : "débiter") +
                                                   " = ");
                    if (credit) {
                        tabComptes[indice].créditer(montant);
                    }
                    else {
                        tabComptes[indice].débiter(montant);
                    }
                    // Afficher le nouveau solde
                    tabComptes[indice].afficherEtat();
                }
                else {
                    System.out.println("Désolé, " + nomAChercher +
                                       " ne fait pas (encore) partie de nos clients.");
                }
            }
        }
    }
}
```

Et voici une trace d'exécution de cette nouvelle version :

```
Nom du titulaire = Karl Tombre
Son adresse = Sornéville
Nom du titulaire = Luigi Liquori
Son adresse = Rome

... Je vous passe un certain nombre de lignes - c'est fastidieux

Nom du titulaire = Bill Clinton
Son adresse = Washington
Votre choix : [D]ébit, [C]rédit, [F]in ? c
Nom du client = Karl Tombre
Montant à créditer = 4500
Compte numéro 1 ouvert au nom de Karl Tombre
Adresse du titulaire : Sornéville
Le solde actuel du compte est de 4500 euros.
**********************************************
Votre choix : [D]ébit, [C]rédit, [F]in ? D
Nom du client = Jacques Chirac
Désolé, Jacques Chirac ne fait pas (encore) partie de nos clients.
Votre choix : [D]ébit, [C]rédit, [F]in ? F
```

Ce programme souffre encore de plusieurs faiblesses, ou disons de manques que je vous laisse compléter en exercice[5] :

- Nous partons ici du principe que le nom est unique, mais dans ce cas nous aurions dû vérifier au moment de la création des comptes que nous n'acceptons pas d'homonymes. Alternativement, on pourrait imaginer un système où l'on cherche sur l'adresse si on trouve des homonymes.
- Il y aurait des manières plus astucieuses d'organiser le tableau des comptes, par exemple en le triant par ordre lexicographique sur les comptes, ce qui permettrait une recherche plus efficace.

[5] Si je vous donnais les solutions de tous les problèmes dans ce polycopié, je me retrouverais en panne d'idées nouvelles pour les TDs !

4.5 La composition : des objets dans d'autres objets

Le programme bancaire commence à devenir complexe ; on peut relever qu'il mèle dans la même procédure `main` des éléments d'interface homme–machine (les dialogues avec l'utilisateur) et la gestion du tableau des comptes. De plus, il est assez rigide, car il fixe à exactement 10 le nombre de comptes à gérer. Il est donc temps de modéliser par une classe à part entière la notion d'agence bancaire, qui pour l'instant est juste représenté par le tableau. Cela nous donnera l'occasion de composer un objet (l'agence bancaire) à partir d'autres objets (les comptes individuels)[6].

Spécifions une version sommaire de l'interface d'une agence bancaire. Il faut être capable de :
- ajouter un nouveau compte ;
- afficher l'état de tous les comptes de l'agence ;
- retrouver un compte en donnant simplement le nom du titulaire.

Les choix de représentation interne sont les suivants :
- Un tableau de comptes, comme dans le cas précédent. Mais nous voulons maintenant être capables de faire varier de manière dynamique la capacité de ce tableau. Pour cela, nous choisissons une capacité de départ de 10 comptes, et quand cette capacité est atteinte, nous créons un nouveau tableau de capacité incrémentée de 10, dans lequel nous recopions les comptes existants. Nous avons donc besoin d'une variable d'instance pour mémoriser la capacité courante du tableau, et nous utilisons une méthode privée, `augmenterCapacité`, pour l'opération d'incrémentation de la capacité[7].
- Un compteur indiquant le nombre de comptes effectivement présents dans le tableau. Nous ne sommes donc plus condamnés à en donner tout de suite 10.

Ceci nous conduit à la classe `AgenceBancaire` suivante :

```
// Classe AgenceBancaire - version 1.0

/**
 * Classe représentant une agence bancaire et les méthodes qui lui
 * sont associées.
 * @author Karl Tombre
 */

public class AgenceBancaire {
    private CompteBancaire[] tabComptes;  // le tableau des comptes
    private int capacitéCourante;         // la capacité du tableau
    private int nbComptes;                // le nombre effectif de comptes

    // Constructeur -- au moment de créer une agence, on crée un tableau
    // de capacité initiale 10, mais qui ne contient encore aucun compte
    AgenceBancaire() {
        tabComptes = new CompteBancaire[10];
        capacitéCourante = 10;
        nbComptes = 0;
    }

    // Méthode privée utilisée pour incrémenter la capacité
    private void augmenterCapacité() {
        // Incrémenter la capacité de 10
        capacitéCourante += 10;
        // Créer un nouveau tableau plus grand que l'ancien
        CompteBancaire[] tab = new CompteBancaire[capacitéCourante];
        // Recopier les comptes existants dans ce nouveau tableau
        for (int i = 0 ; i < nbComptes ; i++) {
```

[6] Il arrive qu'il y ait confusion entre l'emploi de la composition et celui de l'héritage. Une bonne manière de choisir l'outil le plus approprié est de se demander si B est *une sorte de* A (dans ce cas B hérite de A) ou si B *contient* des objets de type A (dans ce cas, on définit une variable d'instance de type A dans B).

[7] Un vrai programmeur Java ne ferait pas ce genre de gymnastique ; il utiliserait directement la classe `Vector`, disponible dans la bibliothèque de base Java, qui met justement en œuvre un tableau de taille variable. Mais il nous semble utile d'avoir fait soi-même ce genre de manipulation au moins une fois, c'est pourquoi nous construisons ici notre propre tableau extensible.

```
            tab[i] = tabComptes[i];
        }
        // C'est le nouveau tableau qui devient le tableau des comptes
        // (l'ancien sera récupéré par le ramasse-miettes)
        tabComptes = tab;
    }

    // Les méthodes de l'interface
    // Ajout d'un nouveau compte
    public void ajout(CompteBancaire c) {
        if (nbComptes == capacitéCourante) { // on a atteint la capacité max
            augmenterCapacité();
        }
        // Maintenant je suis sûr que j'ai de la place
        // Ajouter le nouveau compte dans la première case vide
        // qui porte le numéro nbComptes !
        tabComptes[nbComptes] = c;
        // On prend note qu'il y a un compte de plus
        nbComptes++;
    }
    // Récupérer un compte à partir d'un nom donné
    public CompteBancaire trouverCompte(String nom) {
        boolean trouvé = false;  // rien trouvé pour l'instant
        int indice = 0;
        for (int i = 0 ; i < nbComptes ; i++) {
            if (nom.equalsIgnoreCase(tabComptes[i].nom())) {
                trouvé = true; // j'ai trouvé
                indice = i;    // mémoriser l'indice
                break;         // plus besoin de continuer la recherche
            }
        }
        if (trouvé) {
            return tabComptes[indice];
        }
        else {
            return null;  // si rien trouvé, je rend la référence nulle
        }
    }
    // Afficher l'état de tous les comptes
    public void afficherEtat() {
        // Il suffit d'afficher l'état de tous les comptes de l'agence
        for (int i = 0 ; i < nbComptes ; i++) {
            tabComptes[i].afficherEtat();
        }
    }
}
```

Nous allons maintenant construire une nouvelle version de notre programme bancaire. Profitons de cette « mise à plat » pour modifier un peu les dialogues et leur traitement. Vous remarquerez que maintenant, ce programme ne contient plus que des instructions de dialogue d'un côté, et des appels à l'interface des classes `CompteBancaire` et `GestionBancaire` de l'autre ; nous avons donc bien séparé ces deux aspects, ce qui rend le programme plus lisible :

```
// Classe Banque - version 4.0

/**
 * Classe contenant un programme de gestion bancaire, utilisant
 * un objet de type AgenceBancaire
 * @author Karl Tombre
 * @see    AgenceBancaire, CompteBancaire, Utils
 */
```

```
public class Banque {
    public static void main(String[] args) {
        AgenceBancaire monAgence = new AgenceBancaire();

        while (true) { // boucle infinie dont on sort par un break
            String monNom = Utils.lireChaine("Donnez le nom du client (rien=exit) : ");
            if (monNom.equals("")) {
                break;  // si on ne donne aucun nom on quitte la boucle
            }
            else {
                // Vérifier si le compte existe
                CompteBancaire monCompte = monAgence.trouverCompte(monNom);
                if (monCompte == null) {
                    // rien trouvé, on le crée
                    System.out.println("Ce compte n'existe pas, nous allons le créer");
                    String monAdresse = Utils.lireChaine("Adresse = ");
                    // Créer le compte
                    monCompte = new CompteBancaire(monNom, monAdresse);
                    // L'ajouter aux comptes de l'agence
                    monAgence.ajout(monCompte);
                }

                String choix = Utils.lireChaine("Votre choix : [D]ébit, [C]rédit ? ");
                boolean credit = false;   // variable vraie si c'est un crédit

                // Récupérer la première lettre de la chaîne saisie
                char monChoix = choix.charAt(0);
                if (monChoix == 'c' || monChoix == 'C') {
                    int montant = Utils.lireEntier("Montant à créditer = ");
                    monCompte.créditer(montant);
                }
                else if (monChoix == 'd' || monChoix == 'D') {
                    int montant = Utils.lireEntier("Montant à débiter = ");
                    monCompte.débiter(montant);
                }
                else {
                    System.out.println("Choix invalide");
                }
                // Dans tous les cas, afficher l'état du compte
                monCompte.afficherEtat();
            }
        }
        // Quand on sort de la boucle, afficher l'état global de l'agence
        System.out.println("Voici le nouvel état des comptes de l'agence");
        monAgence.afficherEtat();
    }
}
```

Voici une trace d'exécution de ce programme :

```
Donnez le nom du client (rien=exit) : Karl Tombre
Ce compte n'existe pas, nous allons le créer
Adresse = Sornéville
Votre choix : [D]ébit, [C]rédit ? D
Montant à débiter = 1000
Compte numéro 1 ouvert au nom de Karl Tombre
Adresse du titulaire : Sornéville
Le solde actuel du compte est de -1000 euros.
**********************************************
Donnez le nom du client (rien=exit) : Karl Tombre
```

```
Votre choix : [D]ébit, [C]rédit ? C
Montant à créditer = 2000
Compte numéro 1 ouvert au nom de Karl Tombre
Adresse du titulaire : Sornéville
Le solde actuel du compte est de 1000 euros.
**********************************************
Donnez le nom du client (rien=exit) : Luigi Liquori
Ce compte n'existe pas, nous allons le créer
Adresse = Rome
Votre choix : [D]ébit, [C]rédit ? c
Montant à créditer = 8900
Compte numéro 2 ouvert au nom de Luigi Liquori
Adresse du titulaire : Rome
Le solde actuel du compte est de 8900 euros.
**********************************************
Donnez le nom du client (rien=exit) : Jacques Jaray
Ce compte n'existe pas, nous allons le créer
Adresse = Laxou
Votre choix : [D]ébit, [C]rédit ? F
Choix invalide
Compte numéro 3 ouvert au nom de Jacques Jaray
Adresse du titulaire : Laxou
Le solde actuel du compte est de 0 euros.
**********************************************
Donnez le nom du client (rien=exit) : kArL toMbre
Votre choix : [D]ébit, [C]rédit ? c
Montant à créditer = 2400
Compte numéro 1 ouvert au nom de Karl Tombre
Adresse du titulaire : Sornéville
Le solde actuel du compte est de 3400 euros.
**********************************************
Donnez le nom du client (rien=exit) :
Voici le nouvel état des comptes de l'agence
Compte numéro 1 ouvert au nom de Karl Tombre
Adresse du titulaire : Sornéville
Le solde actuel du compte est de 3400 euros.
**********************************************
Compte numéro 2 ouvert au nom de Luigi Liquori
Adresse du titulaire : Rome
Le solde actuel du compte est de 8900 euros.
**********************************************
Compte numéro 3 ouvert au nom de Jacques Jaray
Adresse du titulaire : Laxou
Le solde actuel du compte est de 0 euros.
**********************************************
```

Exercice 7 *Voici une classe* `Point` *rudimentaire :*

```
public class Point {
    private double x;
    private double y;

    Point(double unX, double unY) {
        x = unX;
        y = unY;
    }

    public double getX() { return x; }
    public double getY() { return y; }
}
```

On souhaite définir une classe décrivant des rectangles à partir de leurs deux coins supérieur gauche et inférieur droit (dans le repère habituel en graphisme sur ordinateur, c'est à dire origine en haut à gauche, x croissant vers la droite, y croissant vers le bas). L'embryon de cette classe est donné ci-dessous :

```
public class Rectangle {
    private Point coinSupGauche;
    private Point coinInfDroit;

    Rectangle(Point p1, Point p2) {
        coinSupGauche = new Point(Math.min(p1.getX(), p2.getX()),
                                  Math.min(p1.getY(), p2.getY()));
        coinInfDroit = new Point(Math.max(p1.getX(), p2.getX()),
                                 Math.max(p1.getY(), p2.getY()));
    }
}
```

Ajouter à cette classe les deux méthodes suivantes :
- `aire()`*, qui rend l'aire du rectangle ;*
- `plusPetitQue(Rectangle r)`*, qui rend une valeur vraie si l'aire du rectangle est inférieure à celle de* `r`*.*

4.6 L'héritage

Quand le cahier des charges d'une application devient important, il vient un moment où il n'est plus ni pratique, ni économique de gérer tous les cas possibles dans une seule classe. Ainsi, supposons qu'il y a deux types de comptes bancaires : les comptes de dépôt et les comptes d'épargne. Pour les premiers, on permet au solde d'être négatif, mais des agios sont déduits chaque jour si le solde est négatif. Pour les seconds, le solde doit toujours rester positif, mais on ajoute des intérêts calculés chaque jour[8].

On pourrait bien entendu gérer ces différents cas de figure dans une seule et même classe `CompteBancaire`, mais celle-ci deviendrait complexe et peu évolutive, puisqu'un grand nombre de ses méthodes devraient prévoir les différences de traitement entre comptes de dépôt et comptes d'épargne. On pourrait aussi choisir de faire deux classes indépendantes, `CompteDepot` et `CompteEpargne`, mais on est alors condamné à dupliquer un grand nombre d'informations et de traitements communs aux deux types de comptes.

C'est dans de telles situations que le recours à l'*héritage* est particulièrement approprié. En effet, l'héritage est un mécanisme qui permet de partager et de réutiliser des données et des méthodes. L'héritage permet d'étendre une classe existante, au lieu d'en créer une nouvelle *ex nihilo*, l'idée étant de tendre vers une hiérarchie de classes réutilisables.

Il s'agit en fait d'un problème de partage efficace de ressources. La classe peut en effet être considérée comme un réservoir de ressources, à partir duquel il est possible de définir d'autres classes plus spécifiques, complétant les ressources de leur « classe mère », dont elles héritent. Les ressources les plus générales sont donc mises en commun dans des classes qui sont ensuite *spécialisées* par la définition de *sous-classes*.

Une sous-classe est en effet une spécialisation de la description d'une classe, appelée sa *super-classe*, dont elle partage – on dit aussi qu'elle hérite – les variables et les méthodes. La spécialisation d'une classe peut être réalisée selon deux techniques. La première est l'enrichissement : la sous-classe est dotée de nouvelles variables et/ou de nouvelles méthodes, représentant les caractéristiques propres au sous-ensemble d'objets décrit par la sous-classe. La seconde technique est la substitution, qui consiste à donner une nouvelle définition à une méthode héritée, lorsque celle-ci se révèle inadéquate ou trop générale pour l'ensemble des objets décrits par la sous-classe.

[8] Pour garder des programmes illustratifs simples et éviter une gestion lourde de dates et de périodes, nous avons bien entendu simplifié énormément la gestion de comptes bancaires, pour laquelle les intérêts et les agios sont calculés beaucoup moins souvent mais tiennent en revanche compte des dates et des durées. Depuis le début, nous avons aussi simplifié le problème des sommes en ne comptant qu'avec des valeurs entières ; cela nous évite des problèmes d'arrondi à deux chiffres après la virgule, mais va nous compliquer un peu la vie quand il s'agira de calculer des intérêts et des agios.

La notion d'héritage peut être vue sous deux angles complémentaires. Quand une classe B hérite de la classe A (on dit aussi que B est *dérivée* de A), l'ensemble des instances de A contient celui des instances de B. Du point de vue extensionnel, la sous-classe B peut donc être considérée comme un sous-ensemble de la classe A. Mais en même temps, du fait des possibilités d'enrichissement, l'ensemble des propriétés de A est un sous-ensemble des propriétés de B ! Le mot clé utilisé en Java pour marquer l'héritage, `extends`, reflète d'ailleurs bien cet état de fait.

Reprenons notre nouveau cahier des charges pour illustrer cette notion. Tout d'abord, nous allons décider de faire de `CompteBancaire` une classe *abstraite*, c'est-à-dire une classe qu'il est interdit d'instancier directement. En effet, nous ne voulons avoir que des comptes de dépôt et des comptes d'épargne, et aucun compte dont la catégorie n'est pas définie. Il faut donc dans notre cas interdire à qui que ce soit de créer un objet par instanciation de `CompteBancaire`. Les classes abstraites sont souvent utilisées dans les langages à objets pour factoriser toutes les caractéristiques communes à un ensemble de classes, qui deviennent ensuite des sous-classes de la classe abstraite.

Le fait d'avoir une classe abstraite nous permet aussi de définir des méthodes abstraites, c'est-à-dire des méthodes dont nous définissons la signature, mais dont nous ne donnons aucune implantation dans la classe abstraite. Pour ne pas être à son tour abstraite, une sous-classe doit alors obligatoirement donner une définition de cette méthode. Aussi bien la classe abstraite que la méthode abstraite sont introduites par le mot clé `abstract`. Dans l'exemple donné ci-après, nous avons ajouté à `CompteBancaire` la méthode abstraite `traitementQuotidien`, qui correspond au traitement qui est supposé être effectué tous les jours – ou plutôt toutes les nuits – sur tous les comptes par un hypothétique programme de gestion des comptes.

Une autre modification que nous sommes amenés à effectuer concerne la protection des variables d'instance. Jusqu'à maintenant, elles étaient toutes déclarées `private`. Mais nous allons avoir besoin d'accéder à la variable `solde` dans les classes héritées ; or une variable privée n'est même pas visible dans les sous-classes qui en héritent ! D'un autre côté, nous ne souhaitons pas que `solde` devienne une variable publique. Java prévoit un niveau de protection intermédiaire, qui correspond à ce que nous cherchons : une variable ou une méthode *protégée* (mot clé `protected`) est privée sauf pour les sous-classes de la classe où elle est définie, ainsi que pour les classes appartenant au même *package* (cf. § 6.2). Les autres variables de `CompteBancaire` n'ont quant à elles aucune raison de ne pas rester privées.

Voici donc la nouvelle version de la classe `CompteBancaire` :

```
// Classe CompteBancaire - version 3.0

/**
 * Classe abstraite représentant un compte bancaire et les méthodes qui lui
 * sont associées.
 * @author Karl Tombre
 */

public abstract class CompteBancaire {
    private String nom;       // le nom du client
    private String adresse;   // son adresse
    private int numéro;       // numéro du compte
    protected int solde;      // solde du compte -- variable protégée

    // Variable de classe
    public static int premierNuméroDisponible = 1;

    // Constructeur : on reçoit en paramètres les valeurs du nom et
    // de l'adresse, on met le solde à 0 par défaut, et on récupère
    // automatiquement un numéro de compte
    CompteBancaire(String unNom, String uneAdresse) {
        nom = unNom;
        adresse = uneAdresse;
        numéro = premierNuméroDisponible;
        solde = 0;
        // Incrémenter la variable de classe
        premierNuméroDisponible++;
```

```
    }

    // Les méthodes
    public void créditer(int montant) {
        solde += montant;
    }
    public void débiter(int montant) {
        solde -= montant;
    }
    public void afficherEtat() {
        System.out.println("Compte numéro " + numéro +
                           " ouvert au nom de " + nom);
        System.out.println("Adresse du titulaire : " + adresse);
        System.out.println("Le solde actuel du compte est de " +
                           solde + " euros.");
        System.out.println("************************************************");
    }
    // Accès en lecture
    public String nom() {
        return this.nom;
    }
    public String adresse() {
        return this.adresse;
    }

    // méthode abstraite, doit être implantée dans les sous-classes
    // traitement quotidien appliqué au compte par un gestionnaire de comptes
    public abstract void traitementQuotidien();
}
```

Nous allons maintenant créer les deux sous-classes `CompteDepot` et `CompteEpargne`. Examinons en détail la première :

```
// Classe CompteDepot - version 1.0

/**
 * Classe représentant un compte de dépôt, sous-classe de CompteBancaire.
 * @author Karl Tombre
 */

public class CompteDepot extends CompteBancaire {

    private double tauxAgios;   // taux quotidien des agios
```

Vous avez sûrement noté l'emploi du mot clé `extends`, qui permet d'indiquer la relation d'héritage entre `CompteDepot` et `CompteBancaire`.

Le premier problème qui se pose à nous est la manière d'initialiser une instance de cette nouvelle classe. Un compte de dépôt est un compte bancaire avec des propriétés en plus, mais pour l'initialiser, le constructeur de `CompteDepot` ne doit pas omettre d'initialiser la partie héritée, c'est-à-dire d'appeler le constructeur de `CompteBancaire`. Il doit en fait y avoir une chaîne d'appels aux constructeurs des superclasses. Ici, nous utilisons le mot clé `super`, qui permet dans un constructeur d'appeler le constructeur de la superclasse. Cet appel doit être la première instruction donnée dans le constructeur de la classe :

```
    // Constructeur
    CompteDepot(String unNom, String uneAdresse, double unTaux) {
        // on crée déjà la partie commune
        super(unNom, uneAdresse);
        // puis on initialise le taux des agios
```

```
        tauxAgios = unTaux;
    }
```

La seule méthode à redéfinir dans la classe `CompteDepot` est `afficherEtat`, pour laquelle on souhaite afficher une ligne de plus, indiquant qu'on a bien un compte de dépôts. Nous trouvons ici un deuxième emploi du mot clé `super` : en conjonction avec la notation pointée, il permet d'appeler une méthode masquée par l'héritage, en l'occurrence la méthode `afficherEtat` de `CompteBancaire`, que nous sommes justement en train de redéfinir :

```
    // Méthode redéfinie : l'affichage
    public void afficherEtat() {
        System.out.println("Compte de dépôts");
        super.afficherEtat(); // appeler la méthode de même nom dans la superclasse
    }
```

Il nous reste à définir la méthode `traitementQuotidien`, qui était définie comme abstraite dans la superclasse. Notez la double conversion de `solde` en `double` pour effectuer les opérations internes en réel double précision, puis du résultat à débiter en `int` pour rester dans des calculs entiers[9]. Cette conversion utilise l'opération de *cast*, notée par un type entre parenthèses. Notez aussi l'appel direct à la méthode `débiter`, héritée de `CompteBancaire` :

```
    // Définition de la méthode de traitementQuotidien
    public void traitementQuotidien() {
        if (solde < 0) {
            débiter((int) (-1.0 * (double) solde * tauxAgios));
        }
    }
}
```

Donnons plus rapidement la deuxième sous-classe, `CompteEpargne`. On notera juste la redéfinition en plus de la méthode `débiter`, pour vérifier que le solde ne devient pas négatif :

```
// Classe CompteEpargne - version 1.0

/**
 * Classe représentant un compte d'épargne, sous-classe de CompteBancaire.
 * @author Karl Tombre
 */

public class CompteEpargne extends CompteBancaire {
    private double tauxIntérêts;   // taux d'intérêts par jour

    // Constructeur
    CompteEpargne(String unNom, String uneAdresse, double unTaux) {
        // on crée déjà la partie commune
        super(unNom, uneAdresse);
        // puis on initialise le taux d'intérêt
        tauxIntérêts = unTaux;
    }

    // Méthode redéfinie : l'affichage
    public void afficherEtat() {
        System.out.println("Compte d'épargne");
        super.afficherEtat(); // appeler la méthode de même nom dans la superclasse
    }

    // Méthode redéfinie : débiter -- interdit de passer en-dessous de 0
```

[9] Une conséquence de la formule choisie est que notre banque ne débite rien si les agios quotidiens ne sont que des centimes...

```
    public void débiter(int montant) {
        if (montant <= solde) {
            solde -= montant;
        }
        else {
            System.out.println("Débit non autorisé");
        }
    }

    // Définition de la méthode de traitementQuotidien
    public void traitementQuotidien() {
        créditer((int) ((double) solde * tauxIntérêts));
    }
}
```

4.6.1 Héritage et typage

Nous avons vu qu'une classe peut être assimilée à un type, dans la mesure où elle sert à définir des objets auxquels s'applique un ensemble d'opérations. La règle usuelle en programmation est de vérifier la correction des programmes avant l'exécution, c'est-à-dire de garantir avant l'exécution du programme que les méthodes appelées existent bien, que les variables sont du bon type, etc. C'est déjà pour cette raison que Java, comme beaucoup d'autres langages, est fortement typé (cf. § 2.2) et qu'il faut connaître d'avance – c'est-à-dire au moment de la compilation (cf. § 1.2) – le type de toutes les variables et la signature de toutes les méthodes utilisées.

Comment cette notion de typage fort s'articule-t-elle avec l'héritage ? D'une certaine manière – bien que cela soit un peu réducteur du point de vue formel – une sous-classe peut être considérée comme définissant un sous-type. Il y a donc compatibilité de types ; en reprenant notre exemple, une variable déclarée de type `CompteBancaire` peut référencer un objet instance de `CompteDepot` ou de `CompteEpargne`. Attention, le contraire n'est pas vrai *a priori* ! Bien entendu, quand on manipule une variable de type `CompteBancaire`, c'est l'interface définie par cette classe qui est accessible, même si l'objet référencé possède d'autres attributs de par son « appartenance » à une sous-classe de `CompteBancaire`.

Nous allons mettre cette propriété à profit pour proposer une version légèrement modifiée de la classe `AgenceBancaire` :

```
// Classe AgenceBancaire - version 1.1

/**
 * Classe représentant une agence bancaire et les méthodes qui lui
 * sont associées.
 * @author Karl Tombre
 */

public class AgenceBancaire {
    private CompteBancaire[] tabComptes;  // le tableau des comptes
    private int capacitéCourante;         // la capacité du tableau
    private int nbComptes;                // le nombre effectif de comptes

    // Constructeur -- au moment de créer une agence, on crée un tableau
    // de capacité initiale 10, mais qui ne contient encore aucun compte
    AgenceBancaire() {
        tabComptes = new CompteBancaire[10];
        capacitéCourante = 10;
        nbComptes = 0;
    }

    // Méthode privée utilisée pour incrémenter la capacité
    private void augmenterCapacité() {
        // Incrémenter la capacité de 10
```

```
        capacitéCourante += 10;
        // Créer un nouveau tableau plus grand que l'ancien
        CompteBancaire[] tab = new CompteBancaire[capacitéCourante];
        // Recopier les comptes existants dans ce nouveau tableau
        for (int i = 0 ; i < nbComptes ; i++) {
            tab[i] = tabComptes[i];
        }
        // C'est le nouveau tableau qui devient le tableau des comptes
        // (l'ancien sera récupéré par le ramasse-miettes)
        tabComptes = tab;
    }

    // Les méthodes de l'interface
    // Ajout d'un nouveau compte
    public void ajout(CompteBancaire c) {
        if (nbComptes == capacitéCourante) { // on a atteint la capacité max
            augmenterCapacité();
        }
        // Maintenant je suis sûr que j'ai de la place
        // Ajouter le nouveau compte dans la première case vide
        // qui porte le numéro nbComptes !
        tabComptes[nbComptes] = c;
        // On prend note qu'il y a un compte de plus
        nbComptes++;
    }

    // Récupérer un compte à partir d'un nom donné
    public CompteBancaire trouverCompte(String nom) {
        boolean trouvé = false;  // rien trouvé pour l'instant
        int indice = 0;
        for (int i = 0 ; i < nbComptes ; i++) {
            if (nom.equalsIgnoreCase(tabComptes[i].nom())) {
                trouvé = true; // j'ai trouvé
                indice = i;    // mémoriser l'indice
                break;         // plus besoin de continuer la recherche
            }
        }
        if (trouvé) {
            return tabComptes[indice];
        }
        else {
            return null;  // si rien trouvé, je rend la référence nulle
        }
    }

    // Afficher l'état de tous les comptes
    public void afficherEtat() {
        // Il suffit d'afficher l'état de tous les comptes de l'agence
        for (int i = 0 ; i < nbComptes ; i++) {
            tabComptes[i].afficherEtat();
        }
    }

    // Traitement quotidien de tous les comptes
    public void traitementQuotidien() {
        for (int i = 0 ; i < nbComptes ;i++) {
            tabComptes[i].traitementQuotidien();
        }
    }
```

```
}
```

En fait, la seule différence est l'ajout de la méthode `traitementQuotidien`, qui permet d'appliquer la méthode `traitementQuotidien` à tous les comptes. Mais il y a une autre différence qui n'apparaît pas dans le code : maintenant, il est impossible de créer des instances de la classe `CompteBancaire`, puisque c'est une classe abstraite. Le tableau `tabComptes` est un tableau d'objets de type `CompteBancaire`, qui contiendra des références à des instances soit de `CompteDepot`, soit de `CompteEpargne`. Grâce à la propriété de sous-typage, ces deux types d'objets peuvent être regroupés dans un même tableau – à condition bien entendu d'y accéder *via* l'interface qu'ils ont en commun, à savoir l'interface de la classe abstraite `CompteBancaire`. C'est ce qui nous permet dans les méthodes `afficherEtat` et `traitementQuotidien` de parcourir le tableau en demandant à chaque objet d'effectuer l'opération `afficherEtat` qui lui est propre.

4.6.2 Liaison dynamique

L'approche objet induit en fait une programmation par requêtes adressées aux interfaces des classes : on n'appelle pas directement une fonction, mais on demande à un objet d'exécuter une méthode de son interface. Se pose alors la question de savoir quand il faut déterminer quelle fonction physique il faut concrètement appeler, c'est-à-dire quand il faut réaliser la *liaison* entre la requête et la méthode de la classe appropriée.

S'il fallait que le compilateur connaisse précisément d'avance quelle méthode de quelle classe doit être appelée, on perdrait le bénéfice de l'héritage. En effet, il serait alors impossible de garantir un comportement correct des méthodes `afficherEtat` et `traitementQuotidien` de la classe `AgenceBancaire`, puisque ce n'est qu'au moment de l'exécution que l'on saura lequel des comptes du tableau est un compte de dépôts et lequel est un compte d'épargne. C'est pourquoi, dans les langages à objets, la liaison entre la requête et la méthode effectivement activée est *dynamique*. Le compilateur est capable d'établir que la méthode existe bien – puisque `afficherEtat` et `traitementQuotidien` appartiennent bien toutes deux à l'interface de la classe `CompteBancaire`, et que les instructions `tabComptes[i].afficherEtat()` et `tabComptes[i].traitementQuotidien()` sont donc bien valides – mais le choix de la méthode qui s'exécutera est différé jusqu'au moment de l'exécution, quand on constate qu'à l'indice i, le tableau comporte soit une instance de `CompteDepot`, soit une instance de `CompteEpargne`.

Nous pouvons donc écrire une nouvelle version de notre programme de gestion bancaire ; à la création d'un nouveau compte, nous demandons maintenant à l'utilisateur quel type de compte doit être créé. Dans la version actuelle, nous avons fixé dans le programme le taux d'intérêt des comptes d'épargne et le taux des agios ; bien entendu, une version plus complète devrait prévoir de saisir ces taux, et éventuellement de pouvoir les modifier. Le reste du programme ne change pas, la liaison dynamique se chargeant d'activer les méthodes appropriées. Nous terminons le programme par l'appel de la méthode `traitementQuotidien`, suivi d'un nouvel affichage de l'état des comptes de l'agence, pour illustrer le traitement différencié qui est fait :

```
// Classe Banque - version 4.1

/**
 * Classe contenant un programme de gestion bancaire, utilisant
 * un objet de type AgenceBancaire
 * @author Karl Tombre
 * @see    AgenceBancaire, CompteBancaire, Utils
 */

public class Banque {
    public static void main(String[] args) {
        AgenceBancaire monAgence = new AgenceBancaire();

        while (true) { // boucle infinie dont on sort par un break
            String monNom = Utils.lireChaine("Donnez le nom du client (rien=exit) : ");
            if (monNom.equals("")) {
```

```
            break;  // si on ne donne aucun nom on quitte la boucle
        }
        else {
            // Vérifier si le compte existe
            CompteBancaire monCompte = monAgence.trouverCompte(monNom);
            if (monCompte == null) {
                // rien trouvé, on le crée
                System.out.println("Ce compte n'existe pas, nous allons le créer");
                String monAdresse = Utils.lireChaine("Adresse = ");
                // Choisir le type de compte
                String type =
                    Utils.lireChaine("Compte de [D]épôt (défaut) ou d'[E]pargne ? ");
                char choixType = type.charAt(0);
                // Créer le compte de type demandé
                // On autorise les lettres accentuées ou non
                if (choixType == 'e' || choixType == 'E' ||
                    choixType == 'é' || choixType == 'É') {
                    monCompte = new CompteEpargne(monNom, monAdresse, 0.00015);
                }
                else {
                    monCompte = new CompteDepot(monNom, monAdresse, 0.00041);
                }
                // L'ajouter aux comptes de l'agence
                monAgence.ajout(monCompte);
            }

            String choix = Utils.lireChaine("Votre choix : [D]ébit, [C]rédit ? ");
            boolean credit = false;   // variable vraie si c'est un crédit

            // Récupérer la première lettre de la chaîne saisie
            char monChoix = choix.charAt(0);
            if (monChoix == 'c' || monChoix == 'C') {
                int montant = Utils.lireEntier("Montant à créditer = ");
                monCompte.créditer(montant);
            }
            else if (monChoix == 'd' || monChoix == 'D') {
                int montant = Utils.lireEntier("Montant à débiter = ");
                monCompte.débiter(montant);
            }
            else {
                System.out.println("Choix invalide");
            }
            // Dans tous les cas, afficher l'état du compte
            monCompte.afficherEtat();
        }
    }
    // Quand on sort de la boucle, afficher l'état global de l'agence
    System.out.println("Voici le nouvel état des comptes de l'agence");
    monAgence.afficherEtat();
    System.out.println("-----------------------------------");
    System.out.println("On applique un traitement quotidien");
    System.out.println("-----------------------------------");
    monAgence.traitementQuotidien();
    monAgence.afficherEtat();
}
```

```
}
```

Voici une trace de l'exécution de ce programme ; vous noterez le débit non autorisé du compte d'épargne – puisqu'on passerait en-dessous de 0 –, le traitement différencié des deux comptes dans l'affichage des états, et le résultat différent du traitement quotidien :

```
Donnez le nom du client (rien=exit) : Karl
Ce compte n'existe pas, nous allons le créer
Adresse = Sornéville
Compte de [D]épôt (défaut) ou d'[E]pargne ? D
Votre choix : [D]ébit, [C]rédit ? D
Montant à débiter = 50000
Compte de dépôts
Compte numéro 1 ouvert au nom de Karl
Adresse du titulaire : Sornéville
Le solde actuel du compte est de -50000 euros.
************************************************
Donnez le nom du client (rien=exit) : Luigi
Ce compte n'existe pas, nous allons le créer
Adresse = Rome
Compte de [D]épôt (défaut) ou d'[E]pargne ? é
Votre choix : [D]ébit, [C]rédit ? C
Montant à créditer = 50000
Compte d'épargne
Compte numéro 2 ouvert au nom de Luigi
Adresse du titulaire : Rome
Le solde actuel du compte est de 50000 euros.
************************************************
Donnez le nom du client (rien=exit) : Luigi
Votre choix : [D]ébit, [C]rédit ? D
Montant à débiter = 60000
Débit non autorisé
Compte d'épargne
Compte numéro 2 ouvert au nom de Luigi
Adresse du titulaire : Rome
Le solde actuel du compte est de 50000 euros.
************************************************
Donnez le nom du client (rien=exit) :
Voici le nouvel état des comptes de l'agence
Compte de dépôts
Compte numéro 1 ouvert au nom de Karl
Adresse du titulaire : Sornéville
Le solde actuel du compte est de -50000 euros.
************************************************
Compte d'épargne
Compte numéro 2 ouvert au nom de Luigi
Adresse du titulaire : Rome
Le solde actuel du compte est de 50000 euros.
************************************************
----------------------------------
On applique un traitement quotidien
----------------------------------
Compte de dépôts
Compte numéro 1 ouvert au nom de Karl
Adresse du titulaire : Sornéville
Le solde actuel du compte est de -50020 euros.
************************************************
Compte d'épargne
Compte numéro 2 ouvert au nom de Luigi
Adresse du titulaire : Rome
```

```
Le solde actuel du compte est de 50007 euros.
**********************************************
```

Et si je veux interdire la définition de sous-classes ?

Il arrive que l'on souhaite qu'une classe donnée ne puisse pas servir de superclasse à des sous-classes. La raison principale est souvent une question de sécurité : la compatibilité de type entre une classe et ses sous-classes pourrait permettre à des programmeurs mal intentionnés de définir une sous-classe qui viendrait se substituer à la classe d'origine, en donnant l'apparence d'avoir le même comportement, alors qu'elle peut faire des choses radicalement différentes. Le mot clé `final` permet d'indiquer qu'une classe ne peut pas avoir de sous-classes. La classe `String`, par exemple, est tellement vitale pour les programmes que les concepteurs de Java l'ont déclarée finale :

```
public final class String {
   ...
}
```

Chapitre 5

Modèles et structures de données

> *Malgré leur appellation voisine, une « liste » et une « liste chaînée » sont des concepts très différents. Une liste est une abstraction mathématique ou un modèle de données. Une liste chaînée est une structure de données. En particulier, c'est la structure de données que nous utilisons habituellement en Pascal et dans d'autres langages similaires pour représenter les listes abstraites dans les programmes. Il existe d'autres langages pour lesquels il n'est pas nécessaire d'utiliser une structure de données pour représenter des listes abstraites. Par exemple, la liste* $(a_1, a_2, \ldots, a_n)$ *peut être représentée directement dans le langage Lisp, et d'une façon similaire dans le langage Prolog...*
>
> Aho & Ullman [1]

Un *modèle de données* est une abstraction utilisée pour décrire des problèmes. Cette abstraction spécifie les valeurs qui peuvent être attribuées aux objets, et les opérations valides sur ces objets.

La *structure de données*, quant à elle, est une construction du langage de programmation, permettant de représenter un modèle de données.

Dans ce chapitre, nous allons illustrer la différence entre ces deux notions en parlant assez longuement de la *liste* (modèle de données), et des structures de données qui permettent habituellement de la représenter. Cela nous donnera aussi l'occasion d'introduire la notion d'*interface*, telle qu'elle est définie en Java. Nous verrons aussi plus rapidement quelques autres modèles de données classiques.

5.1 Les listes

La liste est un modèle de données qui permet de décrire une séquence d'objets contenant chacun une valeur d'un type donné. À la base, on peut définir le type $L =$ *liste de* V à partir d'un type V donné, comme une suite finie d'éléments de V.

Les opérations basiques que l'on peut définir sur une telle liste sont :

$vide:$	L	$\rightarrow$	$Bool$	indique que la liste est vide
$tete:$	L	$\rightarrow$	V	donne la valeur de la tête de liste
$succ:$	L	$\rightarrow$	L	donne la suite de la liste
$adjTete:$	$L \times V$	$\rightarrow$	L	ajoute une valeur en tête de liste

En plus de ces opérations de base, on peut souhaiter définir des opérations plus complexes – qui s'expriment en termes de combinaison de ces opérations de base – comme l'adjonction ou la suppression à un endroit quelconque, la recherche de l'existence d'une valeur dans la liste, voire l'adjonction et la suppression en queue ou le tri...

5.1.1 Représentation par un tableau

Le premier type de représentation auquel on pense est bien entendu le tableau, que nous avons déjà vu. Si nous représentons une liste par un tableau $T[1 \ldots n]$, par exemple, nous pouvons réaliser les opérations de base comme suit :

$$
\begin{array}{rlcll}
vide: & T[1\ldots n] & \rightarrow & Bool & \begin{cases} vrai & \text{si} \quad n = 0 \\ faux & \text{sinon} \end{cases} \\
tete: & T[1\ldots n>0] & \rightarrow & x \in V & x = T[1] \\
succ: & T[1\ldots n>0] & \rightarrow & U[1\ldots n-1] & \forall i \in [1\ldots n-1] \quad U[i] = T[i+1] \\
adjTete: & T[1\ldots n] \times x \in V & \rightarrow & U[1\ldots n+1] & \begin{cases} & U[1] = x \\ \forall i \in [2\ldots n+1] & U[i] = T[i-1] \end{cases}
\end{array}
$$

5.1.2 Représentation par une liste chaînée

La *liste chaînée* est une structure de données que l'on retrouve fréquemment en informatique. Elle nécessite de représenter chaque élément de la liste par un couplet $(val, succ)$, désignant respectivement la valeur au point courant et le « pointeur » sur le chaînon suivant. En Java, cela s'écrit très aisément puisque, comme nous l'avons vu, toutes les variables déclarées d'un type défini par une classe sont des *références*. Prenons par exemple le cas d'une liste d'entiers. La classe qui va représenter un « maillon de la chaîne » pourra s'écrire :

```
public class ElementListInt {
    public int val;
    public ElementListInt succ;

    // Constructeur
    ElementListInt(int x) {
        val = x;
        succ = null;
    }
}
```

Nous pouvons maintenant définir la classe représentant la liste chaînée d'entiers. Vous remarquerez que je n'ai pas mis dans l'interface de cette classe les opérations `tete` et `succ`, que l'on peut effectuer directement par l'accès à `tete.val` et `tete.succ`, ces variables d'instance de `ElementListInt` ayant été déclarées publiques :

```
public class ListInt {
    public ElementListInt tete;

    // Constructeur -- liste vide
    ListInt() {
        tete = null;
    }

    // Méthodes
    public boolean vide() {
        return tete == null;
    }
    public void adjTete(int x) {
        // Créer le nouvel élément
        ElementListInt nouveau = new ElementListInt(x);
        // Le raccorder au reste
        nouveau.succ = tete;
        tete = nouveau;
    }
}
```

Notez la facilité de « chaînage » pour l'adjonction en tête : on crée le nouveau maillon, on lui attribue une valeur, et on l'accroche au reste de la chaîne.

5.1.3 Comparaison entre les deux représentations

On remarquera tout de suite que d'un point de vue pratique, l'adjonction en tête n'est pas une opération très efficace avec la représentation tableau, puisqu'elle nécessite un décalage vers la droite de toutes les valeurs du tableau. Il en serait de même de l'insertion ou de la suppression en une position quelconque.

Ces opérations sont bien plus immédiates avec la liste chaînée. En revanche, celle-ci nécessite plus de place en mémoire, puisque chaque élément de la liste requiert à la fois un emplacement mémoire pour la valeur et un autre emplacement pour le chaînage.

Il est très fréquent de devoir ainsi faire des choix entre temps de calcul et encombrement mémoire. D'autres facteurs peuvent intervenir, notamment le type et la fréquence des opérations que l'on veut effectuer. Si par exemple on a besoin de trier la liste régulièrement, la représentation par tableau présente l'avantage de permettre l'emploi d'algorithmes de tri plus efficaces. Si en revanche les opérations d'adjonction/suppression en un endroit quelconque sont fréquentes, la représentation par liste chaînée présente des avantages indéniables.

5.1.4 Application : l'interface `ListeDeComptes`

Il est temps de revenir à notre exemple de gestion bancaire. Nous avions décidé de représenter les comptes d'une agence bancaire par un tableau d'objets de type `CompteBancaire`. Mais à la réflexion, il est probablement dommage de se lier à ce type de représentation ; nous venons de voir que dans certains cas, d'autres types de représentation peuvent être plus appropriés.

Pour se détacher de la représentation, tout en garantissant un ensemble de services, nous allons recourir à un nouveau concept en Java : l'*interface*. De manière informelle, une interface peut être considérée comme une déclaration de classe – on parlera en fait plutôt de signature de type – sans comportement associé, c'est-à-dire sans code, ni variables d'instance. L'utilisation d'interfaces permet de spécifier un comportement – on parle aussi de *contrat* – que les objets d'un type donné doivent assurer, sans prendre aucune décision sur les structures de données à mettre en œuvre pour représenter concrètement cette interface. On favorise ainsi l'abstraction de données, puisqu'on se situe au niveau du modèle de données.

Définissons donc l'interface `ListeDeComptes`, dont nous attendons le comportement suivant :
- trouver un compte connaissant le nom du client,
- ajouter un compte,
- supprimer un compte,
- afficher l'état de tous les comptes,
- appliquer un traitement quotidien à tous les comptes.

Le mot clé `interface` introduit cette interface, dont la définition se met dans un fichier à part, comme pour les classes – ici le fichier `ListeDeComptes.java` :

```
// Interface ListeDeComptes - version 1.0

/**
 * Interface représentant une liste de comptes et les opérations
 * que l'on souhaite effectuer sur cette liste
 * @author Karl Tombre
 */

public interface ListeDeComptes {
    // Récupérer un compte à partir d'un nom donné
    public CompteBancaire trouverCompte(String nom);
    // Ajout d'un nouveau compte
    public void ajout(CompteBancaire c);
    // Suppression d'un compte connaissant le nom du client
    public void supprimer(CompteBancaire c);
    // Afficher l'état de tous les comptes
    public void afficherEtat();
    // Traitement quotidien de tous les comptes
```

```
    public void traitementQuotidien();
}
```

Vous noterez la syntaxe très proche de celle d'une classe, avec les deux différences fondamentales suivantes, dues au fait que nous sommes au niveau du modèle de données et non de sa représentation concrète :

- nous ne définissons aucune variable,
- nous ne donnons que la signature des méthodes, sans aucune information sur la manière de les exécuter.

L'intérêt d'une telle interface, c'est qu'elle définit un type valide ; nous pouvons donc maintenant écrire notre programme de gestion bancaire en termes plus abstraits, puisque la variable **monAgence** peut maintenant être déclarée de type **ListeDeComptes**.

Cependant, une interface ne comporte aucune variable d'instance et aucun constructeur, on ne peut donc pas l'instancier. Il faut donc bien au moins une classe concrète qui implante l'interface ainsi spécifiée, si on veut disposer d'objets de ce type. Reprenons notre classe **AgenceBancaire** ; elle met déjà en œuvre toutes les opérations de l'interface **ListeDeComptes**, sauf la suppression. Grâce au mot clé **implements**, nous allons déclarer que cette classe met en œuvre l'interface :

```
public class AgenceBancaire implements ListeDeComptes {
    ...
}
```

Mais si nous essayons de la compiler après avoir simplement ajouté cette déclaration, nous avons le message d'erreur suivant :

```
javac  AgenceBancaire.java
AgenceBancaire.java:11: class AgenceBancaire must be declared abstract.
It does not define void supprimer(CompteBancaire) from interface ListeDeComptes.
public class AgenceBancaire implements ListeDeComptes {
             ^
1 error
```

Que se passe-t-il ? En fait, en déclarant **AgenceBancaire implements ListeDeComptes**, nous nous sommes *engagés* à remplir le contrat exprimé par l'interface. Or nous n'avons pas pour l'instant dit comment réaliser la méthode **supprimer** ; le compilateur nous suggère donc de rendre la classe abstraite, supposant que nous voulons définir des sous-classes dans lesquelles cette méthode serait définie.

Il se trompe... Nous allons plutôt ajouter la méthode **supprimer** à la classe *AgenceBancaire*, dont la nouvelle version est donnée ci-après :

```
// Classe AgenceBancaire - version 1.2

/**
 * Classe représentant une agence bancaire et les méthodes qui lui
 * sont associées.
 * @author Karl Tombre
 */

public class AgenceBancaire implements ListeDeComptes {
    private CompteBancaire[] tabComptes;  // le tableau des comptes
    private int capacitéCourante;         // la capacité du tableau
    private int nbComptes;                // le nombre effectif de comptes

    // Constructeur -- au moment de créer une agence, on crée un tableau
    // de capacité initiale 10, mais qui ne contient encore aucun compte
    AgenceBancaire() {
        tabComptes = new CompteBancaire[10];
        capacitéCourante = 10;
```

```
        nbComptes = 0;
    }

    // Méthode privée utilisée pour incrémenter la capacité
    private void augmenterCapacité() {
        // Incrémenter la capacité de 10
        capacitéCourante += 10;
        // Créer un nouveau tableau plus grand que l'ancien
        CompteBancaire[] tab = new CompteBancaire[capacitéCourante];
        // Recopier les comptes existants dans ce nouveau tableau
        for (int i = 0 ; i < nbComptes ; i++) {
            tab[i] = tabComptes[i];
        }
        // C'est le nouveau tableau qui devient le tableau des comptes
        // (l'ancien sera récupéré par le ramasse-miettes)
        tabComptes = tab;
    }

    // Les méthodes de l'interface
    // Ajout d'un nouveau compte
    public void ajout(CompteBancaire c) {
        if (nbComptes == capacitéCourante) { // on a atteint la capacité max
            augmenterCapacité();
        }
        // Maintenant je suis sûr que j'ai de la place
        // Ajouter le nouveau compte dans la première case vide
        // qui porte le numéro nbComptes !
        tabComptes[nbComptes] = c;
        // On prend note qu'il y a un compte de plus
        nbComptes++;
    }
    // Récupérer un compte à partir d'un nom donné
    public CompteBancaire trouverCompte(String nom) {
        boolean trouvé = false;  // rien trouvé pour l'instant
        int indice = 0;
        for (int i = 0 ; i < nbComptes ; i++) {
            if (nom.equalsIgnoreCase(tabComptes[i].nom())) {
                trouvé = true; // j'ai trouvé
                indice = i;    // mémoriser l'indice
                break;         // plus besoin de continuer la recherche
            }
        }
        if (trouvé) {
            return tabComptes[indice];
        }
        else {
            return null;  // si rien trouvé, je rend la référence nulle
        }
    }
    // Afficher l'état de tous les comptes
    public void afficherEtat() {
        // Il suffit d'afficher l'état de tous les comptes de l'agence
        for (int i = 0 ; i < nbComptes ; i++) {
            tabComptes[i].afficherEtat();
        }
    }

    // Traitement quotidien de tous les comptes
    public void traitementQuotidien() {
        for (int i = 0 ; i < nbComptes ;i++) {
            tabComptes[i].traitementQuotidien();
```

```
        }
    }

    // Suppression d'un compte
    public void supprimer(CompteBancaire c) {
        boolean trouvé = false;  // rien trouvé pour l'instant
        int indice = 0;
        for (int i = 0 ; i < nbComptes ; i++) {
            if (tabComptes[i] == c) { // attention comparaison de références
                trouvé = true; // j'ai trouvé
                indice = i;    // mémoriser l'indice
                break;         // plus besoin de continuer la recherche
            }
        }
        if (trouvé) {
            // Décaler le reste du tableau vers la gauche
            // On "écrase" ainsi le compte à supprimer
            for (int i = indice+1 ; i < nbComptes ; i++) {
                tabComptes[i-1] = tabComptes[i];
            }
            // Mettre à jour le nombre de comptes
            nbComptes--;
        }
        else {
            // Message d'erreur si on n'a rien trouvé
            System.out.println("Je n'ai pas trouvé ce compte");
        }
    }
}
```

Notez bien la manière dont on « efface » le compte en décalant d'une position vers la gauche toutes les valeurs du tableau après la position du compte à supprimer. Notez aussi que nous fondons la recherche du compte à supprimer sur la comparaison des références (opérateur `==`) et non sur la comparaison des contenus, ce qui nécessiterait une opération plus complexe.

Modifions maintenant notre programme de gestion bancaire de deux manières :

- la variable `monAgence` est maintenant déclarée de type `ListeDeComptes`, donc en termes d'interface – bien entendu, l'instanciation se fait à partir d'une classe concrète, à savoir `AgenceBancaire` ;
- l'option de suppression d'un compte est proposée à l'utilisateur.

```
// Classe Banque - version 4.2

/**
 * Classe contenant un programme de gestion bancaire, utilisant
 * un objet de type AgenceBancaire
 * @author Karl Tombre
 * @see    ListeDeComptes, CompteBancaire, Utils
 */

public class Banque {
    public static void main(String[] args) {
        ListeDeComptes monAgence = new AgenceBancaire();

        while (true) { // boucle infinie dont on sort par un break
```

```
String monNom = Utils.lireChaine("Donnez le nom du client (rien=exit) : ");
if (monNom.equals("")) {
    break;  // si on ne donne aucun nom on quitte la boucle
}
else {
    // Vérifier si le compte existe
    CompteBancaire monCompte = monAgence.trouverCompte(monNom);
    if (monCompte == null) {
        // rien trouvé, on le crée
        System.out.println("Ce compte n'existe pas, nous allons le créer");
        String monAdresse = Utils.lireChaine("Adresse = ");
        String type =
            Utils.lireChaine("Compte de [D]épôt (défaut) ou d'[E]pargne ? ");
        char choixType = type.charAt(0);
        // Créer le compte de type demandé
        if (choixType == 'e' || choixType == 'E' ||
            choixType == 'é' || choixType == 'É') {
            monCompte = new CompteEpargne(monNom, monAdresse, 0.00015);
        }
        else {
            monCompte = new CompteDepot(monNom, monAdresse, 0.00041);
        }
        // L'ajouter aux comptes de l'agence
        monAgence.ajout(monCompte);
    }

    String choix =
        Utils.lireChaine("Votre choix : [D]ébit, [C]rédit, [S]upprimer ? ");
    boolean supprimer = false;  // pour savoir si on a supprimé

    // Récupérer la première lettre de la chaîne saisie
    char monChoix = choix.charAt(0);
    if (monChoix == 's' || monChoix == 'S') {
        monAgence.supprimer(monCompte);
        supprimer = true;
    }
    else if (monChoix == 'c' || monChoix == 'C') {
        int montant = Utils.lireEntier("Montant à créditer = ");
        monCompte.créditer(montant);
    }
    else if (monChoix == 'd' || monChoix == 'D') {
        int montant = Utils.lireEntier("Montant à débiter = ");
        monCompte.débiter(montant);
    }
    else {
        System.out.println("Choix invalide");
    }
    if (!supprimer) {
        // Si on n'a pas supprimé, afficher l'état du compte
        monCompte.afficherEtat();
    }
    else {
        // Sinon afficher l'état global de l'agence
        System.out.println("Nouvel état des comptes de l'agence");
        monAgence.afficherEtat();
    }
}
```

```
        }
        // Quand on sort de la boucle, afficher l'état global de l'agence
        System.out.println("Voici le nouvel état des comptes de l'agence");
        monAgence.afficherEtat();
        System.out.println("----------------------------------");
        System.out.println("On applique un traitement quotidien");
        System.out.println("----------------------------------");
        monAgence.traitementQuotidien();
        monAgence.afficherEtat();
    }
}
```

Une trace d'exécution de ce nouveau programme montre la possibilité de supprimer un compte :

```
Donnez le nom du client (rien=exit) : Karl
Ce compte n'existe pas, nous allons le créer
Adresse = Sornéville
Compte de [D]épôt (défaut) ou d'[E]pargne ? D
Votre choix : [D]ébit, [C]rédit, [S]upprimer ? D
Montant à débiter = 10000
Compte de dépôts
Compte numéro 1 ouvert au nom de Karl
Adresse du titulaire : Sornéville
Le solde actuel du compte est de -10000 euros.
**********************************************
Donnez le nom du client (rien=exit) : Luigi
Ce compte n'existe pas, nous allons le créer
Adresse = Rome
Compte de [D]épôt (défaut) ou d'[E]pargne ? E
Votre choix : [D]ébit, [C]rédit, [S]upprimer ? C
Montant à créditer = 100000
Compte d'épargne
Compte numéro 2 ouvert au nom de Luigi
Adresse du titulaire : Rome
Le solde actuel du compte est de 100000 euros.
**********************************************
Donnez le nom du client (rien=exit) : Jacques
Ce compte n'existe pas, nous allons le créer
Adresse = Laxou
Compte de [D]épôt (défaut) ou d'[E]pargne ? D
Votre choix : [D]ébit, [C]rédit, [S]upprimer ? C
Montant à créditer = 23000
Compte de dépôts
Compte numéro 3 ouvert au nom de Jacques
Adresse du titulaire : Laxou
Le solde actuel du compte est de 23000 euros.
**********************************************
Donnez le nom du client (rien=exit) : Luigi
Votre choix : [D]ébit, [C]rédit, [S]upprimer ? S
Nouvel état des comptes de l'agence
Compte de dépôts
Compte numéro 1 ouvert au nom de Karl
Adresse du titulaire : Sornéville
Le solde actuel du compte est de -10000 euros.
**********************************************
Compte de dépôts
Compte numéro 3 ouvert au nom de Jacques
Adresse du titulaire : Laxou
Le solde actuel du compte est de 23000 euros.
**********************************************
Donnez le nom du client (rien=exit) : Luigi
```

```
Ce compte n'existe pas, nous allons le créer
Adresse = Bari
Compte de [D]épôt (défaut) ou d'[E]pargne ? E
Votre choix : [D]ébit, [C]rédit, [S]upprimer ? C
Montant à créditer = 100000
Compte d'épargne
Compte numéro 4 ouvert au nom de Luigi
Adresse du titulaire : Bari
Le solde actuel du compte est de 100000 euros.
**********************************************
Donnez le nom du client (rien=exit) :
Voici le nouvel état des comptes de l'agence
Compte de dépôts
Compte numéro 1 ouvert au nom de Karl
Adresse du titulaire : Sornéville
Le solde actuel du compte est de -10000 euros.
**********************************************
Compte de dépôts
Compte numéro 3 ouvert au nom de Jacques
Adresse du titulaire : Laxou
Le solde actuel du compte est de 23000 euros.
**********************************************
Compte d'épargne
Compte numéro 4 ouvert au nom de Luigi
Adresse du titulaire : Bari
Le solde actuel du compte est de 100000 euros.
**********************************************
---------------------------------
On applique un traitement quotidien
---------------------------------
Compte de dépôts
Compte numéro 1 ouvert au nom de Karl
Adresse du titulaire : Sornéville
Le solde actuel du compte est de -10004 euros.
**********************************************
Compte de dépôts
Compte numéro 3 ouvert au nom de Jacques
Adresse du titulaire : Laxou
Le solde actuel du compte est de 23000 euros.
**********************************************
Compte d'épargne
Compte numéro 4 ouvert au nom de Luigi
Adresse du titulaire : Bari
Le solde actuel du compte est de 100014 euros.
**********************************************
```

Nous allons maintenant illustrer l'intérêt d'avoir utilisé une interface, en définissant une autre classe qui met également en œuvre l'interface `ListeDeComptes`, mais cette fois-ci au moyen d'une liste chaînée[1].

Définissons tout d'abord la classe `ChainonCompte`, qui définit un « maillon » de la chaîne, sur le modèle de ce que nous avons fait pour `ElementListInt` :

```
public class ChainonCompte {
    public CompteBancaire val;
    public ChainonCompte succ;

    // Constructeur à partir d'un compte bancaire
```

[1] De même que pour le tableau extensible, un vrai programmeur Java préférerait recourir directement à la classe `LinkedList`, disponible dans la bibliothèque de base Java, et qui met en œuvre une liste chaînée générique. Mais une fois de plus, il nous semble utile d'avoir fait soi-même ce genre de manipulation au moins une fois...

```
    ChainonCompte(CompteBancaire c) {
        val = c;
        succ = null;
    }
}
```

La classe qui met en œuvre l'interface `ListeDeComptes` s'écrit ensuite assez aisément :

```
// Classe AgenceBancaireBis - version 1.0

/**
 * Classe représentant une agence bancaire et les méthodes qui lui
 * sont associées, au moyen d'une liste chaînée
 * @author Karl Tombre
 */

public class AgenceBancaireBis implements ListeDeComptes {
    private ChainonCompte tete;           // la tête de liste

    // Constructeur -- au moment de créer une agence, on crée une liste vide
    AgenceBancaireBis() {
        tete = null;
    }

    // Les méthodes de l'interface
    // Ajout d'un nouveau compte
    public void ajout(CompteBancaire c) {
        // On va l'ajouter en tête, c'est le plus facile
        ChainonCompte n = new ChainonCompte(c);
        n.succ = tete;
        tete = n;
    }
    // Récupérer un compte à partir d'un nom donné
    public CompteBancaire trouverCompte(String nom) {
        ChainonCompte courant = tete;  // chaînon courant
        while ((courant != null) && (!nom.equalsIgnoreCase(courant.val.nom()))) {
            courant = courant.succ;  // aller au suivant
        }
        if (courant !=  null) { // on en a trouvé un
            return courant.val;
        }
        else {
            return null;  // si rien trouvé, je rend la référence nulle
        }
    }
    // Afficher l'état de tous les comptes
    public void afficherEtat() {
        // Il suffit d'afficher l'état de tous les comptes de l'agence
        ChainonCompte courant = tete;
        while (courant != null) {
            courant.val.afficherEtat();
            courant = courant.succ;
        }
    }

    // Traitement quotidien de tous les comptes
    public void traitementQuotidien() {
        ChainonCompte courant = tete;
        while (courant != null) {
            courant.val.traitementQuotidien();
            courant = courant.succ;
```

```
        }
    }

    // Suppression d'un compte
    public void supprimer(CompteBancaire c) {
        if ((tete != null) && (tete.val == c)) {
            // Cas particulier de la suppression en tête
            tete = tete.succ;
        }
        else {
            ChainonCompte courant = tete;  // chaînon courant
            ChainonCompte prev = null;     // celui qui précède le courant
            while ((courant != null) && (courant.val != c)) {
                prev = courant;  // mémoriser le chaînon d'où un vient
                courant = courant.succ;  // aller au suivant
            }
            if (courant != null) {
                // Si on a trouvé quelque chose
                prev.succ = courant.succ;   // on le court-circuite
            }
            else {
                // Message d'erreur si on n'a rien trouvé
                System.out.println("Je n'ai pas trouvé ce compte");
            }
        }
    }
}
```

NB : je vous conseille d'étudier attentivement la manière dont sont effectuées des opérations telles que la recherche ou la suppression. Elles sont caractéristiques des opérations sur les listes chaînées.

Le moment « magique » approche... Nous revenons maintenant au programme de gestion bancaire, et nous nous contentons de modifier la classe instanciée pour créer l'objet référencé par la variable `monAgence` :

```
ListeDeComptes monAgence = new AgenceBancaireBis();
```

Et le programme fonctionne de la même manière, bien que la représentation interne des données soit tout à fait différente ! Nous voyons bien que nous avons gagné un niveau d'abstraction en utilisant l'interface au lieu d'utiliser directement la classe...

Mais avons-nous vraiment le même fonctionnement ? En fait, l'opération d'affichage des états nous donne les comptes dans l'ordre inverse de celui donné avec une instance de `AgenceBancaire`. Cela est dû au fait que l'adjonction d'un nouveau compte se fait en tête de liste dans la classe `AgenceBancaireBis`, alors qu'elle se fait en queue du tableau dans `AgenceBancaire`. Voici les dernières lignes de la trace d'exécution de la nouvelle version du programme :

```
----------------------------------
On applique un traitement quotidien
----------------------------------
Compte de dépôts
Compte numéro 4 ouvert au nom de Luigi
Adresse du titulaire : Bari
Le solde actuel du compte est de 98765 euros.
**********************************************
Compte d'épargne
Compte numéro 3 ouvert au nom de Jacques
Adresse du titulaire : Laxou
Le solde actuel du compte est de 123 euros.
**********************************************
```

```
Compte de dépôts
Compte numéro 1 ouvert au nom de Karl
Adresse du titulaire : Sornéville
Le solde actuel du compte est de -12350 euros.
**********************************************
```

5.2 Les piles

Une pile est une liste particulière, sur laquelle on ne permet l'adjonction et la suppression qu'en tête. Ainsi, c'est toujours le dernier ajouté qui sera le premier sorti (on parle parfois de LIFO : *Last In First Out*).

Les opérations de la pile sont souvent appelées *empiler* et *dépiler*, ou en anglais *push* et *pop*.

Ce modèle de données trouve beaucoup d'applications pratiques quand il s'agit par exemple de mémoriser des contextes en programmation structurée (notion de pile des appels) ou des opérandes en attente d'opérateur, etc. En particulier, le calcul arithmétique avec notation suffixée (dite aussi polonaise inversée) se gère aisément au moyen d'une pile.

On peut bien entendu représenter une pile avec toutes les structures de données qui permettent de représenter des listes ou des tableaux. Nous donnons ci-après une esquisse d'une classe `PileEntiers` permettant d'empiler et de dépiler des entiers. Nous avons choisi une représentation par un tableau surdimensionné, mais vous savez maintenant comment le transformer en un tableau extensible... Nous utilisons aussi un traitement d'exception pour le cas où on essaie de dépiler d'une pile déjà vide. Nous reviendrons plus tard sur cette notion d'exceptions et nous reprendrons cet exemple en le détaillant plus (§ 6.4.2).

```
import java.util.*;

public class PileEntiers {
    private int[] tab;
    private int premierLibre;

    // Constructeur
    PileEntiers() {
        tab = new int[1000];   // surdimensionné
        premierLibre = 0;
    }

    public boolean pileVide() {
        return (premierLibre == 0);
    }
    public void empiler(int x) {
        tab[premierLibre] = x;
        ++premierLibre;
    }
    public int depiler() throws EmptyStackException {
        if (! pileVide()) {
            --premierLibre;
            return tab[premierLibre];
        }
        else {
            throw new EmptyStackException();
        }
    }
}
```

5.3 Les arbres

Si la liste permet de représenter une séquence d'éléments, l'arbre offre des moyens d'organisation plus puissants. Chaque nœud d'un arbre « contient » une valeur, et « pointe » sur les fils du nœud. L'*arbre binaire* est l'archétype des arbres ; chaque nœud a deux fils, et c'est un modèle de données qui permet en particulier de représenter les ensembles de manière efficace. Nous y reviendrons au § 6.5.1 ; contentons-nous pour l'instant de donner la classe qui représente les nœuds des arbres binaires d'entiers :

```
public class NoeudArbre {
    public int val;
    public NoeudArbre filsGauche;
    public NoeudArbre filsDroit;

    // Constructeur
    NoeudArbre(int x) {
        val = x;
        filsGauche = null;
        filsDroit = null;
    }
}
```

Chapitre 6

Programmation (un peu) plus avancée

Amis, ne creusez pas vos chères rêveries ;
Ne fouillez pas le sol de vos plaines fleuries ;
Et, quand s'offre à vos yeux un océan qui dort,
Nagez à la surface ou jouez sur le bord.
Car la pensée est sombre ! Une pente insensible
Va du monde réel à la sphère invisible ;
La spirale est profonde, et quand on y descend
Sans cesse se prolonge et va s'élargissant,
Et pour avoir touché quelque énigme fatale,
De ce voyage obscur souvent on revient pâle !

Victor Hugo

Comme je ne souhaite absolument pas que vous sortiez complètement pâles et désemparés de ce cours de tronc commun, je me contenterai dans ce chapitre d'aborder quelques-uns des nombreux points complémentaires dont j'aurais encore aimé vous parler, en ce qui concerne la programmation. Il va de soi que la science informatique – ou faut-il plutôt parler d'un art ? – recèle bien plus de joies, mais aussi de mystères, que ce que nous pouvons traiter en ces quelque séances. Espérons toutefois qu'à l'issue de ce cours, vous aurez vous-même envie d'en apprendre plus...

6.1 Portée

La portée d'une variable est le bloc de code au sein de laquelle cette variable est accessible ; la portée détermine également le moment où la variable est créée, et quand elle est détruite.

Les variables d'instance et variables de classe sont visibles et accessibles directement à l'intérieur de l'ensemble de la classe. Par exemple, la variable `tabComptes` est accessible dans l'ensemble de la classe `AgenceBancaire`, et nous avons bien entendu profité largement de cette visibilité pour l'utiliser dans les méthodes de cette classe. Si une variable de classe ou d'instance est déclarée publique, elle est également accessible de l'extérieur de la classe, mais en employant la notation pointée.

Les variables locales sont définies dans une méthode – ou dans un bloc interne à une méthode – et ne sont accessibles et visibles qu'au sein du bloc dans lequel elles ont été définies, ainsi que dans les blocs imbriqués dans ce bloc.

Reprenons par exemple la méthode `supprimer` de la classe `AgenceBancaire` :

```
public void supprimer(CompteBancaire c) {
    boolean trouvé = false;  // rien trouvé pour l'instant
    int indice = 0;
    for (int i = 0 ; i < nbComptes ; i++) {
        if (tabComptes[i] == c) { // attention comparaison de références
```

```
                trouvé = true; // j'ai trouvé
                indice = i;    // mémoriser l'indice
                break;         // plus besoin de continuer la recherche
            }
        }
        if (trouvé) {
            // Décaler le reste du tableau vers la gauche
            // On "écrase" ainsi le compte à supprimer
            for (int i = indice+1 ; i < nbComptes ; i++) {
                tabComptes[i-1] = tabComptes[i];
            }
            // Mettre à jour le nombre de comptes
            nbComptes--;
        }
        else {
            // Message d'erreur si on n'a rien trouvé
            System.out.println("Je n'ai pas trouvé ce compte");
        }
    }
```

– La variable booléenne `trouvé` existe tout au long de cette méthode ; elle est donc créée quand la méthode est lancée, et détruite à la sortie de la méthode.
– Il y a deux variables entières nommées `i`, qui sont créées chacune à un moment différent et qui n'existent qu'au sein du bloc dans lequel elles ont été déclarées. Les deux blocs correspondants sont :

```
        for (int i = 0 ; i < nbComptes ; i++) {
            if (tabComptes[i] == c) { // attention comparaison de références
                trouvé = true; // j'ai trouvé
                indice = i;    // mémoriser l'indice
                break;         // plus besoin de continuer la recherche
            }
        }
```

et

```
            for (int i = indice+1 ; i < nbComptes ; i++) {
                tabComptes[i-1] = tabComptes[i];
            }
```

Les paramètres d'appel d'une méthode (comme le paramètre `c` de la méthode `supprimer`) sont accessibles tout au long de la méthode.

Dans tous les cas, une variable peut être *masquée* si une autre variable de même nom est définie dans un bloc imbriqué. Le fait qu'une variable soit masquée signifie juste qu'elle n'est pas directement accessible ; elle n'en continue pas moins d'exister, et on peut même dans certains cas y accéder, en donnant les précisions nécessaires. Si par exemple je déclarais dans la méthode `supprimer` une variable locale de nom `nbComptes`, celle-ci masquerait la variable d'instance `nbComptes`. Pour accéder à cette dernière, je devrais dans ce cas écrire `this.nbComptes`. En règle générale, c'est une mauvaise idée de masquer une variable significative, car cela rend habituellement la lecture du programme difficile et confuse.

6.2 Espaces de nommage : les packages

On retrouve la notion de *bibliothèque* dans la grande majorité des langages de programmation. Pour beaucoup d'entre eux, la définition du langage est d'ailleurs accompagnée d'une définition normalisée de la bibliothèque de base, ensemble de fonctions fournissant les services fondamentaux tels que les entrées-sorties, les calculs mathématiques, etc. Viennent s'y ajouter des bibliothèques quasiment « standard » pour l'interface utilisateur, le graphisme, etc. D'autres bibliothèques accompagnent des produits logiciels. Enfin, certaines bibliothèques peuvent elles-mêmes être des produits logiciels, commercialisés en tant que tels.

Avec la notion de *package*, Java étend et modifie un peu, mais généralise aussi, le concept de bibliothèque.

La problématique du *nommage* est liée à celle de l'utilisation de bibliothèques. Si on peut théoriquement garantir l'absence de conflits de nommage quand un seul programmeur développe une application, c'est déjà plus difficile quand une équipe travaille ensemble sur un produit logiciel, et cela nécessite des conventions ou constructions explicites quand on utilise des bibliothèques en provenance de tiers.

De ce point de vue, Java systématise l'emploi des *packages*, qui sont entre autres des espaces de nommage permettant de réduire fortement les conflits de nommage et les ambiguités. Toute classe doit appartenir à un *package* ; si on n'en indique pas, comme cela a été le cas jusqu'à maintenant pour tous nos programmes, la classe est ajoutée à un *package* par défaut.

L'organisation hiérarchique en packages est traduite à la compilation en une organisation hiérarchique équivalente des répertoires, sous la racine indiquée comme le « réceptacle » de vos classes Java. La recommandation est faite d'utiliser pour ses noms de *packages* une hiérarchisation calquée sur celle des domaines Internet. Ainsi, les classes que j'écris pour les corrigés de mon cours devraient être déclarées dans le package `fr.inpl-nancy.mines.tombre.ens.cours1A` si je décide d'organiser mes programmes d'enseignement dans la catégorie *ens*.

Pour définir le *package* d'appartenance d'une classe, on utilise le mot clé `package` ; si je souhaite regrouper tous mes exemples bancaires dans le *package* `banque`, je peux donc écrire :

```
package fr.inpl-nancy.mines.tombre.ens.cours1A.banque;

public class Banque {
    ...
```

Les fichiers seront alors organisés de manière symétrique, dans un répertoire du genre :

```
fr/inpl-nancy/mines/tombre/ens/cours/cours1A/banque/
```

Pour faciliter l'utilisation des *packages*, Java fournit le mot clé `import`, que nous avons déjà utilisé plusieurs fois. Nous avons par exemple écrit :

```
import java.io.*;

public class Utils {
    public static String lireChaine(String question) {
        InputStreamReader ir = new InputStreamReader(System.in);
        BufferedReader br = new BufferedReader(ir);
        ...
```

Cela nous a permis d'utiliser directement la classe `InputStreamReader`, sans devoir donner son nom complet, qui est `java.io.InputStreamReader`. La directive `import` permet soit « l'importation » d'une classe (on écrirait alors par exemple `import java.io.InputStreamReader;`), soit celle de toutes les classes d'un *package* donné, ce que nous avons fait grâce à la forme `java.io.*`. Cette directive permet ultérieurement d'utiliser les classes ainsi importées sans donner leur nom complet (préfixé par leur package d'appartenance), autrement dit sous une forme abrégée, plus pratique. Rien ne vous empêche bien entendu d'utiliser une classe ou une interface publique sans l'importer, mais il faudra alors donner son nom complet, par exemple `java.awt.event.MouseListener` au lieu de `MouseListener`.

NB : Quand on ne spécifie pas la visibilité d'une variable ou d'une méthode (pas de mot clé `public`, `protected` ou `private`), celle-ci est visible dans toutes les classes de son *package*. Nous reproduisons ci-après un récapitulatif des règles de visibilité, suivant les cas.

Protection	**Classe**	**Sous-classe**	**Package**	**Monde entier**
`private`	oui	non	non	non
`protected`	oui	oui	oui	non
`public`	oui	oui	oui	oui
package (rien)	oui	non	oui	non

6.3 Entrées/sorties

Si les instructions d'entrées/sorties – c'est-à-dire de communication entre le programme et son environnement extérieur – sont très rarement définies dans le langage de programmation lui-même, les langages sont toujours accompagnés d'une bibliothèque de fonctions et procédures standards permettant d'effectuer ces opérations. Java n'est pas une exception.

Malgré sa simplicité apparente, la définition d'une bibliothèque générique d'entrées/sorties est une tâche très délicate. En effet, on a de multiples possibilités, qu'il faut de préférence prendre en compte de manière la plus homogène possible :

- On peut lire au clavier et écrire sur l'écran, mais aussi sur une imprimante ; on peut lire et écrire à partir d'un fichier, ou d'une adresse mémoire correspondant à un flux de données en provenance d'un réseau par exemple, etc.
- On peut avoir envie de lire ou écrire des caractères (en Java, souvenez-vous qu'ils sont représentés sur 2 octets), ou un flot d'octets, ou des objets plus structurés.
- On peut avoir des entrées et sorties « bufferisées », c'est-à-dire gérées ligne par ligne. C'est typiquement le cas dans des programmes interactifs simples, où vous saisissez vos réponses au programme au clavier, en souhaitant qu'elles ne soient prises en compte qu'une fois que vous tapez « Entrée ». Jusque là, vous êtes libres de revenir en arrière, d'effacer un caractère, etc. Les caractères que vous saisissez sont stockés dans une zone tampon (un *buffer* – d'où le nom). Dans d'autres cas, au contraire, on ne veut surtout pas que le caractère « Entrée » ait une signification particulière ; dans les flots à partir de fichiers, par exemple, on veut prendre en compte et traiter immédiatement chaque caractère entré (imaginez qu'on vous demande d'écrire un traitement de texte...)

À la base des entrées/sorties en Java se trouve la notion de flot de données (*stream* en anglais). Pour lire de l'information, un programme ouvre un flot de données connecté à une source d'information (un fichier, une connexion réseau, un clavier, etc.) et lit l'information de manière séquentielle sur ce flot. De même, pour envoyer de l'information vers une destination extérieure, un programme ouvre aussi un flot connecté à la destination et y écrit de manière séquentielle. Les opérations d'entrées/sorties se ressemblent donc toutes : on commence par ouvrir un flot, puis on lit (ou on écrit) dans ce flot tant qu'il y a de l'information à lire (respectivement à écrire), puis enfin on referme le flot.

L'ensemble des fonctionnalités standards d'entrées/sorties en Java se trouvent dans le *package* `java.io`. Celui-ci contient notamment 4 classes abstraites : `Reader` et `Writer` factorisent le comportement commun de toutes les classes qui lisent (respectivement écrivent) dans des flots de caractères (caractères au codage Unicode sur 16 bits – cf. § F.1). `InputStream` et `OutputStream`, quant à elles, sont les superclasses des classes qui permettent de lire et d'écrire dans des flots d'octets (sur 8 bits).

Le tableau qui suit donne un aperçu de quelques-unes des classes disponibles dans le *package* `java.io`. Il faut savoir qu'un certain nombre de fonctionnalités non mentionnées dans ce polycopié sont également disponibles dans ce *package*, par l'intermédiaire de classes spécialisées.

Superclasse →	`Reader`	`InputStream`	`Writer`	`OutputStream`
E/S bufferisées	`Buffered-Reader`	`Buffered-InputStream`	`Buffered-Writer`	`Buffered-OutputStream`
E/S avec la mémoire	`CharArray-Reader`	`ByteArray-InputStream`	`CharArray-Writer`	`ByteArray-OutputStream`
Conversion flots d'octets/flots de caractères	`InputStream-Reader`		`OutputStream-Writer`	
Pipelines (la sortie d'un programme est l'entrée d'un autre)	`PipedReader`	`PipedInput-Stream`	`PipedWriter`	`PipedOutput-Stream`
Fichiers	`FileReader`	`FileInput-Stream`	`FileWriter`	`FileOutput-Stream`

6.3.1 L'explication d'un mystère

Nous avons maintenant des éléments pour expliquer une bonne partie des notations ésotériques que nous vous avons imposées dès le début du cours, en particulier pour lire et écrire des informations.

La classe `System` permet à vos programmes Java d'accéder aux ressources du système hôte. Cette classe possède plusieurs variables de classe, dont deux vont nous intéresser ici. La variable de classe `System.out` est de type `PrintStream` ; elle permet d'écrire sur votre sortie standard (habituellement, la console Java sur votre écran). C'est un flot de sortie, appelé le flot de sortie standard, qui est ouvert d'office et prêt à recevoir des données. Les méthodes de `PrintStream` que nous avons utilisées tout au long de ce cours sont :

- `println` pour écrire une chaîne de caractères (`String`) en vidant le buffer, et passer à la ligne,
- `print` qui fait la même chose sans passer à la ligne, c'est-à-dire en écrivant dans le buffer,
- `flush` qui force l'affichage du buffer de sortie sans qu'il y ait eu de retour à la ligne.

Décortiquons maintenant la méthode de classe `Utils.lireChaine`, qui permet de lire une chaîne de caractères au clavier. La classe `System` fournit aussi une variable de classe `System.in`, qui est déclarée comme étant de type `InputStream`[1]. Comme son homologue `System.out`, cette variable désigne un flot ouvert d'office, le flot d'entrée standard. C'est un flot d'octets et non de caractères, ce qui est logique dans la mesure où les systèmes informatiques actuels fonctionnent avec des flots de caractères codés sur 8 bits en entrées/sorties. Mais comme nous voulons lire des chaînes de caractères, la première chose à faire est de créer une instance d'une sous-classe de `Reader`, pour nous retrouver dans la hiérarchie des flots de lecture de caractères. Il se trouve que la classe `InputStreamReader` permet justement de faire cette « conversion », comme nous l'avons vu dans le tableau. C'est donc la raison de l'instanciation d'un objet de type `InputStreamReader`, désigné par la variable `ir` :

```
public static String lireChaine(String question) {
    InputStreamReader ir = new InputStreamReader(System.in);
```

Mais en fait, nous voulons effectuer des entrées bufferisées, et il nous faut donc opérer une seconde « conversion », en créant une instance de la classe `BufferedReader`, désignée par la variable `br`. C'est cette variable que nous allons utiliser par la suite...

```
    BufferedReader br = new BufferedReader(ir);
```

On pose la question, sans aller à la ligne, et on force l'affichage du buffer, comme nous l'avons déjà expliqué :

```
    System.out.print(question);
    System.out.flush();
```

Il nous reste à appeler la méthode `readLine`, qui fait partie de l'interface de la classe `BufferedReader`, pour lire au clavier une ligne (jusqu'à ce que l'utilisateur tape « entrée ») :

```
    String reponse = "";
    try {
        reponse = br.readLine();
    } catch (IOException ioe) {}
    return reponse;
}
```

[1] `InputStream` étant une classe abstraite, cette variable désigne forcément une instance d'une de ses sous-classes, mais nous n'y accédons que par le biais de l'interface d'`InputStream`.

6.3.2 Les fichiers

La gestion de fichiers comprend deux aspects :

1. Les considérations générales de la gestion des entrées-sorties, dans la mesure où la lecture ou l'écriture dans un fichier n'est en fait qu'un flot parmi d'autres. Nous avons déjà vu que nous disposons dans la hiérarchie des classes du *package* `java.io` des classes `FileReader` et `FileWriter` pour lire et écrire des flots de caractères dans des fichiers, et de `FileInputStream` et `FileOutputStream` pour les flots d'octets.
2. L'établissement d'un lien avec le système de fichiers sous-jacent, tout en restant indépendant de la plateforme. Cet aspect est pris en compte par la classe `File`.

Prenons un exemple qui illustre ceci et montre comment lire et écrire dans un fichier. Une faiblesse importante de notre programme de gestion bancaire est bien entendu que nous perdons tout ce que nous avons saisi quand nous quittons le programme. Nous souhaitons donc maintenant être capables de sauvegarder les comptes de l'agence bancaire dans un fichier, au moment de quitter le programme, et de relire la sauvegarde quand on relance le programme.

La première chose à faire est d'ouvrir le fichier. Si le nom du fichier est `comptes.dat`, on écrira :

```
File fich = new File("comptes.dat");
```

Mais il se peut que ce fichier n'existe pas encore ; nous voulons alors le créer. Il se trouve que la classe `FileOutputStream`, quand elle est instanciée, crée le fichier s'il n'existe pas. On écrira donc :

```
if (!fich.exists()) {
    FileOutputStream fp = new FileOutputStream("comptes.dat");
    fich = new File("comptes.dat");
}
```

Vous noterez l'utilisation de la méthode `exists` de la classe `File` pour vérifier l'existence du fichier.

À condition que nous ayons le droit d'écrire dans ce fichier, nous passons ensuite à une phase de conversions très analogues à celle que nous avons effectuées à partir de `System.in`. En effet, nous voulons effectuer des entrées/sorties bufferisées, une fois de plus, et écrivons donc :

```
BufferedWriter bw = null;
try {
    if (fich.canWrite()) {
        FileWriter fw = new FileWriter(f);
        bw = new BufferedWriter(fw);
    }
}
catch (IOException ioe) {}
```

Nous avons maintenant une variable de type `BufferedWriter`, et pouvons écrire des chaînes de caractères. Comme nous souhaitons les séparer par le caractère « retour à la ligne », nous utilisons à la fois les fonctions `write` et `newLine` de la classe :

```
try {
    bw.write("Karl Tombre");
    bw.newLine();
    bw.write("Sornéville");
    bw.newLine();
}
catch (IOException ioe) {}
```

Pour écrire des entiers, nous choisirons de les convertir en chaînes de caractères, grâce à la méthode `toString` de la classe `Integer` :

```
try {
    Integer i = new Integer(numéro);
```

```
    bw.write(i.toString());
    bw.newLine();
    i = new Integer(solde);
    bw.write(i.toString());
    bw.newLine();
}
catch (IOException ioe) {}
```

Il reste pour finir à fermer le fichier, une fois que tout a été écrit :

```
bw.close();
```

Les opérations de lecture dans un fichier se font de manière similaire.

Avant de passer au programme de gestion bancaire, organisons-nous un peu. Nous allons ajouter dans la classe `Utils` des fonctions permettant de ne pas encombrer nos programmes, en y regroupant les lignes de code nécessaires aux opérations les plus courantes. La nouvelle version de la classe `Utils` se passe *a priori* de commentaires ; vous y retrouverez plusieurs des lignes de code que nous venons de voir :

```
// Classe Utils - version 2.0

/**
 *  Classe regroupant les utilitaires disponibles pour les exercices
 * de 1ere année du tronc commun informatique.
 * @author Karl Tombre
 */

import java.io.*;

public class Utils {
    // Les fonctions lireChaine et lireEntier
    // Exemple d'utilisation dans une autre classe :
    // String nom = Utils.lireChaine("Entrez votre nom : ");
    public static String lireChaine(String question) {
        InputStreamReader ir = new InputStreamReader(System.in);
        BufferedReader br = new BufferedReader(ir);

        System.out.print(question);
        System.out.flush();
        return loadString(br);
    }

    // La lecture d'un entier n'est qu'un parseInt de plus !!
    public static int lireEntier(String question) {
        return Integer.parseInt(lireChaine(question));
    }

    // écriture d'une chaîne dans un BufferedWriter
    public static void saveString(BufferedWriter bw, String s) {
        try {
            bw.write(s);
            bw.newLine();
        }
        catch (IOException ioe) {}
    }

    // écriture d'un entier : on convertit en chaîne et on se ramène
    // à la procédure précédente
```

```
    public static void saveInt(BufferedWriter bw, int i) {
        Integer q = new Integer(i);
        saveString(bw, q.toString());
    }

    // ouverture en écriture d'un fichier
    // rend un BufferedWriter -- null si pas possible de l'ouvrir
    public static BufferedWriter openWriteFile(File f) {
        BufferedWriter bw = null;
        try {
            if (f.canWrite()) {
                FileWriter fw = new FileWriter(f);
                bw = new BufferedWriter(fw);
            }
        }
        catch (IOException ioe) {}
        return bw;
    }

    // ouverture en lecture d'un fichier
    // rend un BufferedReader -- null si pas possible de l'ouvrir
    public static BufferedReader openReadFile(File f) {
        BufferedReader br = null;
        try {
            if (f.canRead()) {
                FileReader fr = new FileReader(f);
                br = new BufferedReader(fr);
            }
        }
        catch (IOException ioe) {}
        return br;
    }

    // lecture dans un BufferedReader d'une chaîne de caractères
    public static String loadString(BufferedReader br) {
        String s = "";
        try {
            s = br.readLine();
        }
        catch (IOException ioe) {}
        return s;
    }

    // lecture d'un entier dans un BufferedReader
    public static int loadInt(BufferedReader br) {
        return Integer.parseInt(loadString(br));
    }
}
```

Décidons ensuite comment un compte bancaire va être sauvegardé et rechargé dans un fichier. Nous optons pour la simplicité en écrivant et en lisant des chaînes de caractères. Dans l'ordre,

on écrira donc le nom, l'adresse, le numéro et le solde, suivi du taux d'agios pour un compte de dépôts, et du taux d'intérêts pour un compte d'épargne. Nous donnons ci-après la nouvelle version des trois classes **CompteBancaire**, **CompteEpargne** et **CompteDepot**. Notez qu'en plus des deux nouvelles méthodes qui y sont définies, nous avons ajouté un constructeur par défaut (sans paramètres) dont nous aurons besoin au moment du chargement à partir d'un fichier :

```
// Classe CompteBancaire - version 3.1

import java.io.*;

/**
 * Classe abstraite représentant un compte bancaire et les méthodes qui lui
 * sont associées.
 * @author Karl Tombre
 */

public abstract class CompteBancaire {
    private String nom;       // le nom du client
    private String adresse;   // son adresse
    private int numéro;       // numéro du compte
    protected int solde;      // solde du compte

    // Variable de classe
    public static int premierNuméroDisponible = 1;

    // Constructeur : on reçoit en paramètres les valeurs du nom et
    // de l'adresse, on met le solde à 0 par défaut, et on récupère
    // automatiquement un numéro de compte
    CompteBancaire(String unNom, String uneAdresse) {
        nom = unNom;
        adresse = uneAdresse;
        numéro = premierNuméroDisponible;
        solde = 0;
        // Incrémenter la variable de classe
        premierNuméroDisponible++;
    }

    // Constructeur par défaut, sans paramètres
    CompteBancaire() {
        nom = "";
        adresse = "";
        numéro = 0;
        solde = 0;
    }

    // Les méthodes
    public void créditer(int montant) {
        solde += montant;
    }
    public void débiter(int montant) {
        solde -= montant;
    }
    public void afficherEtat() {
        System.out.println("Compte numéro " + numéro +
                           " ouvert au nom de " + nom);
        System.out.println("Adresse du titulaire : " + adresse);
        System.out.println("Le solde actuel du compte est de " +
                           solde + " euros.");
        System.out.println("***********************************************");
```

```
    }
    // Accès en lecture
    public String nom() {
        return this.nom;
    }
    public String adresse() {
        return this.adresse;
    }

    // méthode abstraite, doit être implantée dans les sous-classes
    // traitement quotidien appliqué au compte par un gestionnaire de comptes
    public abstract void traitementQuotidien();

    // sauvegarde du compte dans un BufferedWriter
    public void save(BufferedWriter bw) {
        Utils.saveString(bw, nom);
        Utils.saveString(bw, adresse);
        Utils.saveInt(bw, numéro);
        Utils.saveInt(bw, solde);
    }

    // chargement du compte à partir d'un BufferedReader
    public void load(BufferedReader br) {
        nom = Utils.loadString(br);
        adresse = Utils.loadString(br);
        numéro = Utils.loadInt(br);
        solde = Utils.loadInt(br);
    }
}
```

```
// Classe CompteDepot - version 1.1

import java.io.*;

/**
 * Classe représentant un compte de dépôt, sous-classe de CompteBancaire.
 * @author Karl Tombre
 */

public class CompteDepot extends CompteBancaire {
    private double tauxAgios;   // taux quotidien des agios

    // Constructeur
    CompteDepot(String unNom, String uneAdresse, double unTaux) {
        // on crée déjà la partie commune
        super(unNom, uneAdresse);
        // puis on initialise le taux des agios
        tauxAgios = unTaux;
    }

    // Constructeur par défaut
    CompteDepot() {
        super();
        tauxAgios = 0.0;
    }
```

```
    // Méthode redéfinie : l'affichage
    public void afficherEtat() {
        System.out.println("Compte de dépôts");
        super.afficherEtat(); // appeler la méthode de même nom dans la superclasse
    }

    // Définition de la méthode de traitementQuotidien
    public void traitementQuotidien() {
        if (solde < 0) {
            débiter((int) (-1.0 * (double) solde * tauxAgios));
        }
    }
    // sauvegarde du compte dans un BufferedWriter
    public void save(BufferedWriter bw) {
        super.save(bw);
        Double d = new Double(tauxAgios);
        Utils.saveString(bw, d.toString());
    }

    // chargement du compte à partir d'un BufferedReader
    public void load(BufferedReader br) {
        super.load(br);
        tauxAgios = Double.valueOf(Utils.loadString(br)).doubleValue();
    }
}
```

```
// Classe CompteEpargne - version 1.1

import java.io.*;

/**
 * Classe représentant un compte d'épargne, sous-classe de CompteBancaire.
 * @author Karl Tombre
 */

public class CompteEpargne extends CompteBancaire {
    private double tauxIntérêts;   // taux d'intérêts par jour

    // Constructeur
    CompteEpargne(String unNom, String uneAdresse, double unTaux) {
        // on crée déjà la partie commune
        super(unNom, uneAdresse);
        // puis on initialise le taux d'intérêt
        tauxIntérêts = unTaux;
    }

    // Constructeur par défaut
    CompteEpargne() {
        super();
        tauxIntérêts = 0.0;
    }

    // Méthode redéfinie : l'affichage
    public void afficherEtat() {
        System.out.println("Compte d'épargne");
        super.afficherEtat(); // appeler la méthode de même nom dans la superclasse
    }
```

```
    // Méthode redéfinie : débiter -- interdit de passer en-dessous de 0
    public void débiter(int montant) {
        if (montant <= solde) {
            solde -= montant;
        }
        else {
            System.out.println("Débit non autorisé");
        }
    }

    // Définition de la méthode de traitementQuotidien
    public void traitementQuotidien() {
        créditer((int) ((double) solde * tauxIntérêts));
    }
    // sauvegarde du compte dans un BufferedWriter
    public void save(BufferedWriter bw) {
        super.save(bw);
        Double d = new Double(tauxIntérêts);
        Utils.saveString(bw, d.toString());
    }

    // chargement du compte à partir d'un BufferedReader
    public void load(BufferedReader br) {
        super.load(br);
        tauxIntérêts = Double.valueOf(Utils.loadString(br)).doubleValue();
    }
}
```

Il nous faut maintenant définir les opérations de sauvegarde et de chargement dans l'interface `ListeDeComptes` :

```
// Interface ListeDeComptes - version 1.1

import java.io.*;

/**
 * Interface représentant une liste de comptes et les opérations
 * que l'on souhaite effectuer sur cette liste
 * @author Karl Tombre
 */

public interface ListeDeComptes {
    // Récupérer un compte à partir d'un nom donné
    public CompteBancaire trouverCompte(String nom);
    // Ajout d'un nouveau compte
    public void ajout(CompteBancaire c);
    // Suppression d'un compte
    public void supprimer(CompteBancaire c);
    // Afficher l'état de tous les comptes
    public void afficherEtat();
    // Traitement quotidien de tous les comptes
    public void traitementQuotidien();
    // Sauvegarder dans un fichier
    public void sauvegarder(File f);
    // Charger à partir d'un fichier
    public void charger(File f);
```

```
}
```

Maintenant, nous devons l'implanter dans les classes qui mettent en œuvre cette interface. Nous ne donnons ici que la nouvelle version de la classe `AgenceBancaire`, c'est-à-dire la version avec tableaux extensibles :

```
// Classe AgenceBancaire - version 1.3

import java.io.*;

/**
 * Classe représentant une agence bancaire et les méthodes qui lui
 * sont associées.
 * @author Karl Tombre
 */

public class AgenceBancaire implements ListeDeComptes {
    private CompteBancaire[] tabComptes;  // le tableau des comptes
    private int capacitéCourante;         // la capacité du tableau
    private int nbComptes;                // le nombre effectif de comptes

    // Constructeur -- au moment de créer une agence, on crée un tableau
    // de capacité initiale 10, mais qui ne contient encore aucun compte
    AgenceBancaire() {
        tabComptes = new CompteBancaire[10];
        capacitéCourante = 10;
        nbComptes = 0;
    }

    // Méthode privée utilisée pour incrémenter la capacité
    private void augmenterCapacité() {
        // Incrémenter la capacité de 10
        capacitéCourante += 10;
        // Créer un nouveau tableau plus grand que l'ancien
        CompteBancaire[] tab = new CompteBancaire[capacitéCourante];
        // Recopier les comptes existants dans ce nouveau tableau
        for (int i = 0 ; i < nbComptes ; i++) {
            tab[i] = tabComptes[i];
        }
        // C'est le nouveau tableau qui devient le tableau des comptes
        // (l'ancien sera récupéré par le ramasse-miettes)
        tabComptes = tab;
    }

    // Les méthodes de l'interface
    // Ajout d'un nouveau compte
    public void ajout(CompteBancaire c) {
        if (nbComptes == capacitéCourante) { // on a atteint la capacité max
            augmenterCapacité();
        }
        // Maintenant je suis sûr que j'ai de la place
        // Ajouter le nouveau compte dans la première case vide
        // qui porte le numéro nbComptes !
        tabComptes[nbComptes] = c;
        // On prend note qu'il y a un compte de plus
        nbComptes++;
    }
    // Récupérer un compte à partir d'un nom donné
    public CompteBancaire trouverCompte(String nom) {
        boolean trouvé = false;  // rien trouvé pour l'instant
        int indice = 0;
```

```
        for (int i = 0 ; i < nbComptes ; i++) {
            if (nom.equalsIgnoreCase(tabComptes[i].nom())) {
                trouvé = true; // j'ai trouvé
                indice = i;    // mémoriser l'indice
                break;         // plus besoin de continuer la recherche
            }
        }
        if (trouvé) {
            return tabComptes[indice];
        }
        else {
            return null;  // si rien trouvé, je rend la référence nulle
        }
    }
    // Afficher l'état de tous les comptes
    public void afficherEtat() {
        // Il suffit d'afficher l'état de tous les comptes de l'agence
        for (int i = 0 ; i < nbComptes ; i++) {
            tabComptes[i].afficherEtat();
        }
    }

    // Traitement quotidien de tous les comptes
    public void traitementQuotidien() {
        for (int i = 0 ; i < nbComptes ;i++) {
            tabComptes[i].traitementQuotidien();
        }
    }

    // Suppression d'un compte
    public void supprimer(CompteBancaire c) {
        boolean trouvé = false;  // rien trouvé pour l'instant
        int indice = 0;
        for (int i = 0 ; i < nbComptes ; i++) {
            if (tabComptes[i] == c) { // attention comparaison de références
                trouvé = true; // j'ai trouvé
                indice = i;    // mémoriser l'indice
                break;         // plus besoin de continuer la recherche
            }
        }
        if (trouvé) {
            // Décaler le reste du tableau vers la gauche
            // On "écrase" ainsi le compte à supprimer
            for (int i = indice+1 ; i < nbComptes ; i++) {
                tabComptes[i-1] = tabComptes[i];
            }
            // Mettre à jour le nombre de comptes
            nbComptes--;
        }
        else {
            // Message d'erreur si on n'a rien trouvé
            System.out.println("Je n'ai pas trouvé ce compte");
        }
    }
```

Le début de la classe ne change pas, si ce n'est l'importation du *package* `java.io`. Passons maintenant aux deux nouvelles méthodes. La méthode `sauvegarder` est chargée d'écrire le contenu du tableau de comptes dans le fichier. Il faut bien entendu commencer par ouvrir le fichier en écriture ; pour cela, nous avons défini dans la classe `Utils` la méthode de classe `openWriteFile`, qui nous rend un objet de type `BufferedWriter`, que nous allons manipuler dans la suite des

opérations :

```
public void sauvegarder(File f) {
    BufferedWriter bw = Utils.openWriteFile(f);
```

À condition bien entendu que l'ouverture ait été possible (variable `bw` non égale à `null`), nous pouvons donc commencer à écrire dans le fichier. En y réfléchissant, nous nous rendons compte qu'il peut être judicieux de stocker le nombre de comptes dans le tableau ; cela simplifiera grandement la lecture. Donc la première ligne du fichier va contenir le nombre de comptes, ce que nous écrivons grâce à la méthode de classe `Utils.saveInt` :

```
    if (bw != null) {
        Utils.saveInt(bw, nbComptes);
```

Pour éviter de repartir à 1 dans la numérotation des comptes bancaires, il faut aussi sauvegarder la variable de classe `CompteBancaire.premierNuméroDisponible` :

```
        // Sauvegarder la variable de classe
        Utils.saveInt(bw, CompteBancaire.premierNuméroDisponible);
```

Il nous reste maintenant à parcourir le tableau en écrivant les comptes les uns après les autres. Si tous les comptes avaient été de la même nature, cela serait enfantin. Mais attention : nous avons à la fois des comptes d'épargne et des comptes de dépôt, et à la relecture du fichier il est nécessaire de savoir les distinguer ! Une première solution aurait été d'utiliser l'opérateur `instanceof` en écrivant quelque chose comme `if (tabComptes[i] instanceof CompteDepot) { ...} else { ... }`, mais cette solution reste très peu évolutive : qu'adviendrait-il du programme si nous ajoutions des comptes d'épargne en actions, des plans épargne logement, des livret d'épargne... ? Je propose donc une solution plus générique, qui fait appel aux propriétés de *réflexivité*[2] du langage Java. L'idée est d'écrire le nom de la classe d'appartenance de l'objet que l'on veut sauvegarder. En Java, il existe une classe `Class` dont toutes les classes sont instances. Une méthode `getClass` est définie sur tous les objets (techniquement, elle est définie dans la classe `Object`, racine de l'arbre d'héritage). Elle rend la classe d'appartenance de l'objet. Nous désignons cette classe par la variable `typeDeCompte`. La méthode `getName` de la classe `Class` rend une chaîne de caractères désignant le nom de la classe. Une fois cette chaîne écrite, nous pouvons déléguer à l'objet le soin de se sauvegarder (`tabComptes[i].save(bw)`) ; grâce à la liaison dynamique nous sommes sûrs d'appeler la bonne méthode, celle de la classe `CompteDepot` ou celle de la classe `CompteEpargne`, suivant les cas. L'avantage de cette solution est d'être extensible : si nous ajoutons ultérieurement d'autres sous-classes à `CompteBancaire`, la solution reste identique et le code ci-après n'a pas à être modifié :

```
        for (int i = 0 ; i < nbComptes ; i++) {
            Class typeDeCompte = tabComptes[i].getClass();
            Utils.saveString(bw, typeDeCompte.getName());
            tabComptes[i].save(bw);
        }
```

Il nous reste à fermer le flot d'écriture et à indiquer par un message comment l'opération s'est déroulée :

```
        try {
            bw.close();
        }
```

[2]La réflexivité peut se définir d'une manière générale comme la description du comportement d'un système qui entretient un rapport de cause à effet avec lui-même. Plus précisément, un système réflexif possède une structure, appelée auto-représentation, qui décrit la connaissance qu'il a de lui-même. Java possède certaines propriétés de réflexivité, notamment celles que nous utilisons ici pour demander à une classe son nom, ou pour récupérer un objet de type `Class` à partir de son nom.

```
            catch (IOException ioe) {}
            System.out.println("Sauvegarde effectuée");
        }
        else {
            System.out.println("Impossible de sauvegarder");
        }
    }
```

La méthode `charger` est symétrique. On commence par ouvrir le fichier en lecture, et on lit la première ligne qui, comme vous vous en souvenez, contient le nombre de comptes à charger. Cela nous donnera un compteur pour la boucle de lecture. On lit aussi sur la deuxième ligne la valeur à attribuer à la variable de classe `CompteBancaire.premierNuméroDisponible` :

```
    public void charger(File f) {
        BufferedReader br = Utils.openReadFile(f);
        if (br != null) {
            nbComptes = Utils.loadInt(br);
            CompteBancaire.premierNuméroDisponible = Utils.loadInt(br);
```

Il faut maintenant pour chaque compte sauvegardé commencer par lire le nom de sa classe d'appartenance (`Utils.loadString`), puis « convertir » ce nom en un objet de type `Class`, grâce à la méthode de classe `Class.forName`. Ensuite, nous créons une nouvelle instance de cette classe. Mais nous n'avons pas encore les paramètres d'initialisation, d'où l'intérêt de l'ajout de constructeurs par défaut – sans paramètres d'initialisation – dans les classes `CompteDepot` et `CompteEpargne`. Nous pouvons ainsi demander la création d'une nouvelle instance grâce à la méthode `newInstance` de la classe `Class`. Une fois cette instance créée, c'est à elle que nous laissons le soin de charger ses valeurs par la méthode `load`, qui active la bonne méthode, une fois de plus grâce à la liaison dynamique. Il ne restera plus qu'à refermer le fichier et à afficher un message sur le déroulement du chargement :

```
            for (int i = 0 ; i < nbComptes ; i++) {
                try {
                    Class typeDeCompte = Class.forName(Utils.loadString(br));
                    tabComptes[i] = (CompteBancaire) typeDeCompte.newInstance();
                    tabComptes[i].load(br);
                }
                catch(ClassNotFoundException cnfe) {}
                catch(IllegalAccessException iae) {}
                catch(InstantiationException ie) {}
            }
            try {
                br.close();
            }
            catch (IOException ioe) {}
            System.out.println("Comptes chargés");
        }
        else {
            System.out.println("Impossible de charger la sauvegarde");
        }
    }
}
```

Il nous reste simplement à intégrer ces deux opérations dans notre programme de gestion bancaire. Les seules modifications étant en début et en fin de ce programme, je vous passe les lignes intermédiaires, le programme interactif de débit/crédit n'étant affecté en rien par cette nouvelle fonctionnalité :

```
// Classe Banque - version 4.3
```

```
import java.io.*;
/**
 * Classe contenant un programme de gestion bancaire, utilisant
 * un objet de type AgenceBancaire
 * @author Karl Tombre
 * @see    ListeDeComptes, CompteBancaire, Utils
 */

public class Banque {
    public static void main(String[] args) {
        ListeDeComptes monAgence = new AgenceBancaire();
        File fich = new File("comptes.dat");
        // On commence par charger le fichier de sauvegarde éventuel
        monAgence.charger(fich);

        while (true) { // boucle infinie dont on sort par un break
          // ... la même chose qu'avant
        }
        // Quand on sort de la boucle, afficher l'état global de l'agence
        System.out.println("Voici le nouvel état des comptes de l'agence");
        monAgence.afficherEtat();
        System.out.println("-----------------------------------");
        System.out.println("On applique un traitement quotidien");
        System.out.println("-----------------------------------");
        monAgence.traitementQuotidien();
        monAgence.afficherEtat();

        // Et on termine par une sauvegarde
        try {
            fich = new File("comptes.dat");
            if (!fich.exists()) {
                // Si le fichier n'existe pas, le créer
                FileOutputStream fp = new FileOutputStream("comptes.dat");
                fich = new File("comptes.dat");
            }
            monAgence.sauvegarder(fich);
        }
        catch (IOException ioe) {}
    }
}
```

Une première exécution de ce programme ne permet pas de charger une sauvegarde qui n'existe pas encore ; mais sinon, les choses se déroulent normalement, et la sauvegarde s'effectue à la fin :

```
Impossible de charger la sauvegarde
Donnez le nom du client (rien=exit) : Karl Tombre
Ce compte n'existe pas, nous allons le créer
Adresse = Sornéville
Compte de [D]épôt (défaut) ou d'[E]pargne ? E
Votre choix : [D]ébit, [C]rédit, [S]upprimer ? C
Montant à créditer = 10000
Compte d'épargne
Compte numéro 1 ouvert au nom de Karl Tombre
Adresse du titulaire : Sornéville
Le solde actuel du compte est de 10000 euros.
**********************************************
Donnez le nom du client (rien=exit) : Luigi Liquori
```

```
Ce compte n'existe pas, nous allons le créer
Adresse = Bari
Compte de [D]épôt (défaut) ou d'[E]pargne ? D
Votre choix : [D]ébit, [C]rédit, [S]upprimer ? D
Montant à débiter = 20000
Compte de dépôts
Compte numéro 2 ouvert au nom de Luigi Liquori
Adresse du titulaire : Bari
Le solde actuel du compte est de -20000 euros.
**********************************************
Donnez le nom du client (rien=exit) :
Voici le nouvel état des comptes de l'agence
Compte d'épargne
Compte numéro 1 ouvert au nom de Karl Tombre
Adresse du titulaire : Sornéville
Le solde actuel du compte est de 10000 euros.
**********************************************
Compte de dépôts
Compte numéro 2 ouvert au nom de Luigi Liquori
Adresse du titulaire : Bari
Le solde actuel du compte est de -20000 euros.
**********************************************
----------------------------------
On applique un traitement quotidien
----------------------------------
Compte d'épargne
Compte numéro 1 ouvert au nom de Karl Tombre
Adresse du titulaire : Sornéville
Le solde actuel du compte est de 10001 euros.
**********************************************
Compte de dépôts
Compte numéro 2 ouvert au nom de Luigi Liquori
Adresse du titulaire : Bari
Le solde actuel du compte est de -20008 euros.
**********************************************
Sauvegarde effectuée
```

Examinons le contenu du fichier `comptes.dat` :

```
2
3
CompteEpargne
Karl Tombre
Sornéville
1
10001
1.5E-4
CompteDepot
Luigi Liquori
Bari
2
-20008
4.1E-4
```

Lançons maintenant le programme une seconde fois. Cette fois, le chargement se fait bien et nous retrouvons les comptes dans l'état où nous les avions laissés :

```
Comptes chargés
Donnez le nom du client (rien=exit) : Karl tOmbre
Votre choix : [D]ébit, [C]rédit, [S]upprimer ? d
```

```
Montant à débiter = 10000
Compte d'épargne
Compte numéro 1 ouvert au nom de Karl Tombre
Adresse du titulaire : Sornéville
Le solde actuel du compte est de 1 euros.
**********************************************
Donnez le nom du client (rien=exit) : Jacques Jaray
Ce compte n'existe pas, nous allons le créer
Adresse = Laxou
Compte de [D]épôt (défaut) ou d'[E]pargne ? E
Votre choix : [D]ébit, [C]rédit, [S]upprimer ? C
Montant à créditer = 100
Compte d'épargne
Compte numéro 1 ouvert au nom de Jacques Jaray
Adresse du titulaire : Laxou
Le solde actuel du compte est de 100 euros.
**********************************************
Donnez le nom du client (rien=exit) :
Voici le nouvel état des comptes de l'agence
Compte d'épargne
Compte numéro 1 ouvert au nom de Karl Tombre
Adresse du titulaire : Sornéville
Le solde actuel du compte est de 1 euros.
**********************************************
Compte de dépôts
Compte numéro 2 ouvert au nom de Luigi Liquori
Adresse du titulaire : Bari
Le solde actuel du compte est de -20008 euros.
**********************************************
Compte d'épargne
Compte numéro 1 ouvert au nom de Jacques Jaray
Adresse du titulaire : Laxou
Le solde actuel du compte est de 100 euros.
**********************************************
---------------------------------
On applique un traitement quotidien
---------------------------------
Compte d'épargne
Compte numéro 1 ouvert au nom de Karl Tombre
Adresse du titulaire : Sornéville
Le solde actuel du compte est de 1 euros.
**********************************************
Compte de dépôts
Compte numéro 2 ouvert au nom de Luigi Liquori
Adresse du titulaire : Bari
Le solde actuel du compte est de -20016 euros.
**********************************************
Compte d'épargne
Compte numéro 3 ouvert au nom de Jacques Jaray
Adresse du titulaire : Laxou
Le solde actuel du compte est de 100 euros.
**********************************************
Sauvegarde effectuée
```

Et bien entendu, le fichier `comptes.dat` est à jour :

```
3
4
CompteEpargne
Karl Tombre
Sornéville
```

```
1
1
1.5E-4
CompteDepot
Luigi Liquori
Bari
2
-20016
4.1E-4
CompteEpargne
Jacques Jaray
Laxou
3
100
1.5E-4
```

6.4 Exceptions

Dans tous les exemples que nous donnons depuis le début de ce cours, nous avons plus d'une fois employé la construction `try ... catch`. Celle-ci est liée au traitement des exceptions en Java, c'est-à-dire des conditions d'erreur ou « anormales », pour une raison ou une autre. Presque toutes les erreurs à l'exécution déclenchent une exception, qui peut dans beaucoup de cas être « attrapée ».

Java fournit un mécanisme permettant de regrouper le traitement de ces exceptions. Une exception signalée dans un bloc est propagée (instruction `throw`), d'abord à travers les blocs englobants, puis à travers la pile des appels de méthodes et fonctions, jusqu'à ce qu'elle soit « attrapée » et traitée par une instruction `catch`. En fait, les exceptions sont des objets, instances de la classe `Throwable` ou de l'une de ses sous-classes.

Dans beaucoup de langages, le traitement des conditions d'erreur est disséminé à travers le programme, ce qui en rend la compréhension malaisée. Java fournit les mécanismes permettant au contraire de les regrouper, ce qui permet de se concentrer d'un côté sur les instructions à effectuer pour le déroulement normal des opérations, et de l'autre des interventions à faire si quelque chose se passe mal.

Les trois mots clés `try`, `catch` et `finally` permettent de structurer votre code de la manière schématique suivante :

```
try {
    // Code qui peut provoquer des exceptions
}
catch (UneException e1) {
    // Traitement de ce type d'exception
}
catch (UneAutreException e2) {
    // Traitement de l'autre type d'exception
}
finally {
    // Choses à faire dans tous les cas, même après une exception
}
```

`UneException` et `uneAutreException` doivent être des sous-classes de `Throwable`.

6.4.1 Les derniers éléments du mystère

Ces informations vont nous permettre d'élucider complètement le mystère de la notation ésotérique qui vous a été imposée pour les lectures de chaînes de caractères. Revenons à notre étude de la fonction `Utils.lireChaine` (cf. § 6.3.1). Nous avons vu que `br` est une variable de type `BufferedReader`. Si nous allons voir cette classe dans le *package* `java.io`, nous trouvons :

```
public class BufferedWriter extends Writer {
    // ... plein de choses que je passe
    public void readLine() throws IOException {
        // ... le corps de la méthode readLine
    }
    // ... etc.
}
```

Le mot clé `throws` indique que la méthode `readLine` est susceptible de provoquer une exception de type `IOException`. Je ne peux donc qu'essayer (`try`) de l'appeler, et dire ce que je fais (`catch`) si une exception `ioe` de type `IOException` se produit – en l'occurrence, rien du tout (`{}`) :

```
        String reponse = "";
        try {
            reponse = br.readLine();
        } catch (IOException ioe) {}
        return reponse;
```

Nous avons bien entendu vu de nombreux autres exemples de cette construction, tout au long du polycopié.

6.4.2 Exemple : une méthode qui provoque une exception

Nous avons écrit au § 5.2 une classe `PileEntiers` dont l'une des méthodes pouvait provoquer une exception :

```
    public int depiler() throws EmptyStackException {
        if (! pileVide()) {
            --premierLibre;
            return tab[premierLibre];
        }
        else {
            throw new EmptyStackException();
        }
    }
```

Si on essaie de dépiler une pile vide, on provoque une exception de type `EmptyStackException` – l'une des sous-classes de `Throwable` – grâce au mot clé `throw`. Vous noterez au passage que cette classe appartient au *package* `java.util`, qui n'est pas importé par défaut, ce qui nous oblige donc à une clause `import`.

Écrivons maintenant un petit programme utilisant une telle pile, dans lequel nous récupérerons cette erreur et en ferons quelque chose – en l'occurrence, nous afficherons un message :

```
import java.util.*;

public class TestPile {
    public static void main(String[] args) {
        PileEntiers maPile = new PileEntiers();
        maPile.empiler(3);
        maPile.empiler(4);
        try {
            System.out.println("Je dépile " + maPile.depiler());
        }
        catch(EmptyStackException ese) {
            System.out.println("Attention, la pile est vide");
        }
        try {
            System.out.println("Je dépile " + maPile.depiler());
```

```
        }
        catch(EmptyStackException ese) {
            System.out.println("Attention, la pile est vide");
        }
        try {
            System.out.println("Je dépile " + maPile.depiler());
        }
        catch(EmptyStackException ese) {
            System.out.println("Attention, la pile est vide");
        }
    }
}
```

Une exécution de ce programme va donner la trace suivante :

```
Je dépile 4
Je dépile 3
Attention, la pile est vide
```

6.5 La récursivité

Une procédure ou une fonction récursive s'appelle elle-même, directement ou indirectement. Elle découle souvent directement d'une formule de récurrence ; cela ne veut pas dire que toute formule de récurrence doit mener à une fonction récursive, car on peut éviter les récursivités inutiles, quand une construction itérative est plus claire.

Pour prendre un premier exemple simpliste, nous savons bien que la factorielle peut être définie par une formule de récurrence :

$$\forall n \in \mathbb{N}, \quad n! = \begin{cases} 1 & \text{si} \quad n \leq 1 \\ n \times (n-1)! & \text{sinon} \end{cases}$$

Cela peut se traduire directement en Java, sous forme d'une fonction récursive :

```
static int factorielle(int n) {
    /* n négatif non traité ici */
    if (n <= 1) {
        return 1;
    }
    else {
        return n * factorielle(n-1);
    }
}
```

En fait, le principe « diviser pour régner » est très souvent au cœur de l'analyse d'un problème complexe. La récursivité n'en est qu'un des avatars. Quand on use de cette stratégie, il est extrêmement important de s'assurer que :

1. Tous les cas ont été pris en compte (on n'a pas « perdu des bouts » dans le processus de division).
2. La division ne part jamais dans une boucle infinie (cas par exemple d'une récursivité mal analysée ou mal mise en œuvre).
3. Les sous-problèmes sont effectivement plus simples que le problème de départ !

6.5.1 Exemple : représentation d'un ensemble par un arbre

Revenons à la structure de données « arbre binaire » que nous avons commencé à esquisser au § 5.3. En effet, les manipulations d'arbres sont des exemples particulièrement frappants de la puissance d'une récursivité bien maîtrisée.

Pour tout ensemble E sur lequel on dispose d'une relation d'ordre total, il peut être judicieux d'utiliser un arbre binaire pour représenter la notion d'« ensemble d'éléments de E ». En effet, on peut construire un tel arbre binaire de telle manière que lorsqu'on accède à sa racine, tous les nœuds du sous-arbre gauche (celui dont la racine est le fils gauche de la racine) contiennent des valeurs inférieures à la valeur contenue dans la racine, et tous les nœuds du sous-arbre droit contiennent des valeurs supérieures à celle de la racine. De ce fait, pour peu que l'arbre soit équilibré[3], on peut tester l'appartenance d'un élément à l'ensemble représenté par l'arbre en un temps moyen de l'ordre de $O(\log n)$, où n est le nombre d'éléments de l'ensemble.

Illustrons notre propos en définissant une classe `ArbreBinaire`, s'appuyant sur la classe `NoeudArbre` que nous avons déjà vue (§ 5.3), qui permet de représenter un ensemble d'entiers. La seule variable d'instance dont on ait besoin ici est la racine, qui est un nœud, initialisé à `null` quand on crée un arbre nouveau :

```
public class ArbreBinaire  {
    private NoeudArbre racine;

    // Constructeur à partir de rien
    ArbreBinaire() {
        racine = null;
    }
```

Nous aurons également besoin de créer un arbre à partir d'un nœud donné, c'est-à-dire de considérer l'ensemble des nœuds rattachés directement ou indirectement à ce nœud comme un arbre. Nous définissons donc un constructeur à partir d'un nœud :

```
    // Constructeur à partir d'un noeud donné
    ArbreBinaire(NoeudArbre n) {
        racine = n;
    }
```

La première méthode, qui teste si l'ensemble est vide, consiste juste à tester si la racine est nulle :

```
    public boolean vide() {
        return racine == null;
    }
```

Dans les méthodes récursives de parcours de l'arbre que nous allons voir dans un instant, nous aurons besoin de récupérer les sous-arbres gauche et droit de l'arbre courant. Ici, nous utilisons le constructeur d'un arbre à partir d'un nœud, que nous venons de définir :

```
    public ArbreBinaire filsGauche() {
        if (racine == null) {
            return null;
        }
        else {
            return new ArbreBinaire(racine.filsGauche);
        }
    }

    public ArbreBinaire filsDroit() {
        if (racine == null) {
            return null;
        }
```

[3] L'arbre très simple que nous programmons dans l'exemple donné ici peut bien entendu être très déséquilibré, puisque sa configuration dépend fortement de l'ordre dans lequel on introduit les données. Mais il existe des versions plus élaborées de structures d'arbres où l'on rééquilibre l'arbre chaque fois qu'une adjonction ou une suppression menace de rompre l'équilibre.

```
        else {
            return new ArbreBinaire(racine.filsDroit);
        }
    }
```

La méthode `contient` permet de tester l'appartenance d'un entier à l'ensemble représenté par l'arbre. Nous voyons ici la puissance de la récursivité. Connaissant les propriétés de l'arbre, nous pouvons écrire l'algorithme comme suit :

$$x \in A = \begin{cases} faux & \text{si} \quad A = \emptyset \\ vrai & \text{si} \quad racine(A) = x \\ x \in filsDroit(A) & \text{si} \quad x > racine(A) \\ x \in filsGauche(A) & \text{si} \quad x < racine(A) \end{cases}$$

Cette formule de récurrence nous conduit directement à la méthode `contient` ; notez l'utilisation des méthodes `filsGauche` et `filsDroit` précédemment définies :

```
    public boolean contient(int x) {
        if (racine == null) {
            return false;
        }
        else if (racine.val == x) {
            return true;
        }
        else if (x > racine.val) {
            return filsDroit().contient(x);
        }
        else {
            return filsGauche().contient(x);
        }
    }
```

De même, la méthode d'ajout d'un élément à l'ensemble s'écrit de manière récursive, en s'inspirant de l'algorithme suivant :

$$A \cup \{x\} : \begin{cases} racine(A) = x & \text{si} \quad A = \emptyset \\ A & \text{si} \quad x = racine(A) \\ filsGauche(A) \cup \{x\} & \text{si} \quad x < racine(A) \\ filsDroit(A) \cup \{x\} & \text{si} \quad x > racine(A) \end{cases}$$

Notez que la méthode `ajouter` rend un arbre, pour permettre de « raccrocher » les branches quand on revient des appels récursifs :

```
    public ArbreBinaire ajouter(int x) {
        if (racine == null) {
            // créer un nouveau nœud
            NoeudArbre nouveau = new NoeudArbre(x);
            racine = nouveau;
        }
        else if (x < racine.val) {
            racine.filsGauche = filsGauche().ajouter(x).racine;
        }
        else if (x > racine.val) {
            racine.filsDroit = filsDroit().ajouter(x).racine;
        }
        else {
            // dernier cas : il y est déjà : rien à faire
        }
```

```
        // Dans tous les cas, rendre l'arbre créé
        return this;
    }
```

La dernière méthode que nous allons voir permet d'afficher le contenu de l'ensemble, dans l'ordre qui correspond à la relation d'ordre utilisée dans la construction de l'arbre. Cette méthode est très simple : imprimer l'arbre, c'est
- imprimer le sous-arbre gauche (appel récursif),
- imprimer la valeur de la racine,
- imprimer le sous-arbre droit (appel récursif).

```
    public void print() {
        if (racine != null) {
            filsGauche().print();
            System.out.print(racine.val + ", ");
            filsDroit().print();
        }
    }
}
```

Voici maintenant un petit programme de test, qui illustre le fonctionnement de cette classe :

```
import java.util.*;

public class TestArbre {
    public static void main(String[] args) {
        ArbreBinaire monArbre = new ArbreBinaire();
        monArbre.ajouter(3);
        monArbre.ajouter(4);
        monArbre.ajouter(18);
        monArbre.ajouter(9);
        monArbre.ajouter(1);
        monArbre.ajouter(20);
        monArbre.ajouter(87);
        monArbre.ajouter(9);
        monArbre.ajouter(11);
        monArbre.ajouter(65);
        monArbre.ajouter(41);
        monArbre.ajouter(19);
        if (monArbre.contient(19)) {
            System.out.println("19 y est");
        }
        else {
            System.out.println("19 n'y est pas");
        }
        monArbre.print();
        System.out.println("");
    }
}
```

La trace d'exécution de ce programme est la suivante :

```
19 y est
1, 3, 4, 9, 11, 18, 19, 20, 41, 65, 87,
```

Exercice 8 *Fibonacci, de son vrai nom Léonard de Pise, est né à la fin du 12^e siècle. Lors de nombreux voyages qui l'amenèrent en Égypte, en Syrie, en Grèce et en Sicile, il s'initia aux connaissances mathématiques du monde arabe. Quelque temps après son retour chez lui, à Pise, il publia*

en 1202 son célèbre Liber abbaci*, dans lequel il tente de transmettre à l'Occident la science mathématique des Arabes et des Grecs. C'est dans cet ouvrage qu'il pose le problème suivant :*

Si on part d'un couple de lapins, combien de couples de lapins obtiendra-t-on après un nombre donné de mois, sachant que chaque couple produit chaque mois un nouveau couple, lequel ne devient lui-même productif qu'à l'âge de deux mois. *Autrement dit :*

- *Au début : 1 couple*
- *Au bout de 1 mois : 1 couple*
- *Au bout de 2 mois : 2 couples*
- *Au bout de 3 mois : 3 couples*
- *Au bout de 4 mois : 5 couples*
- *Au bout de 5 mois : 8 couples*
- *Au bout de 6 mois : 13 couples...*

La suite des nombres de couples de lapins est appelée suite de Fibonacci, et possède beaucoup de propriétés intéressantes. Elle est décrite par la formule de récurrence suivante :

$$F_n = \begin{cases} 1 & si \quad n = 0 \quad ou \quad 1 \\ F_{n-1} + F_{n-2} & sinon \end{cases}$$

Écrire la fonction récursive correspondante.

6.6 Interface homme–machine et programmation événementielle

Au fur et à mesure que notre programme de gestion bancaire a grossi, le dialogue avec l'utilisateur est devenu de plus en plus complexe. Dans son état actuel, il souffre en fait d'un gros défaut : c'est le programme qui « prend l'utilisateur par la main » et qui lui pose les questions dans un certain ordre, alors qu'on souhaiterait bien entendu que l'utilisateur soit plus libre et que l'interface soit plus souple.

Cette interface textuelle est également bien « tristounette » à l'heure des environnements multi-fenêtrés, avec interactions à la souris et *via* des menus déroulants.

Bien entendu, Java offre toutes les fonctionnalités nécessaires pour créer de telles interfaces. Ce n'est que par souci d'une démarche pédagogique progressive que nous nous sommes cantonnés jusqu'ici à une interface aussi terne...

Dans ce dernier paragraphe du polycopié, nous allons vous donner un avant-goût de ce qu'est la programmation d'une vraie interface graphique. Il est évident que nous n'avons pas le temps de beaucoup étoffer notre programme, et l'exemple que nous allons développer reste très largement en-deçà des possibilités offertes par la bibliothèque de composantes graphiques *Swing* que nous allons utiliser.

Pour développer cette interface, nous allons devoir faire nos premiers pas en programmation événementielle. En programmation « classique », tout s'exécute sous le contrôle du programme, une fois qu'il est lancé ; en particulier, l'utilisateur doit répondre aux questions posées par le programme. En revanche, en programmation événementielle, le programme est en veille, attendant des stimuli externes, auxquels il réagit. C'est donc le programme qui doit réagir aux interactions de l'utilisateur, et non l'inverse.

Ces stimuli externes peuvent être de différentes natures : clic ou mouvement de souris, activation d'une touche, (dés)activation ou iconification d'une fenêtre, saisie d'un texte dans une zone d'interaction, signal arrivant sur un port de communication, etc.

Nous resterons fidèles à notre démarche tout au long du polycopié : au lieu de vous noyer sous les spécifications théoriques ou techniques, nous vous fournissons un exemple et vous invitons à « jouer » avec ce programme. Peut-être encore plus que beaucoup d'autres aspects, la programmation événementielle se prête bien à l'apprentissage par analogie ; avec un ou deux exemples sous les yeux, et un manuel des fonctionnalités offertes par la bibliothèque à portée de main (ou à portée de clic de souris, ces manuels étant habituellement en ligne), on est vite capable de mettre au point son propre programme interactif.

Tout d'abord, nous devons effectuer plusieurs modifications mineures dans les différentes classes que nous avons déjà conçues. En effet, les méthodes que nous avons écrites affichent parfois des messages ; or dans un système multi-fenêtré, nous n'avons plus de « console » sur laquelle écrire, et si on veut renvoyer un message il faut que la méthode en question renvoie une chaîne qui puisse être récupérée et affichée dans une fenêtre de dialogue.

À tout seigneur tout honneur – commençons par la classe **CompteBancaire**. Peu de modifications ici : nous avons ajouté une méthode d'accès au numéro du compte, pour pouvoir l'afficher en dehors de la classe. Nous n'en définissons pas pour **solde**, qui est **protected**, puisque le programme interactif que nous allons écrire se trouvera dans le même *package* que **CompteBancaire** ; la variable est donc visible (cf. § 6.2). Par ailleurs, la méthode **débiter** rend désormais un message (type **String**) indiquant si le débit a été autorisé ou non (*stricto sensu*, je devrais faire la même chose pour **créditer**, car on peut imaginer des types de compte pour lesquels on ne peut pas dépasser un certain plafond, par exemple).

```
// Classe CompteBancaire - version 3.2

import java.io.*;

/**
 * Classe abstraite représentant un compte bancaire et les méthodes qui lui
 * sont associées.
 * @author Karl Tombre
 */

public abstract class CompteBancaire {
    private String nom;       // le nom du client
    private String adresse;   // son adresse
    private int numéro;       // numéro du compte
    protected int solde;      // solde du compte

    // Variable de classe
    public static int premierNuméroDisponible = 1;

    // Constructeur : on reçoit en paramètres les valeurs du nom et
    // de l'adresse, on met le solde à 0 par défaut, et on récupère
    // automatiquement un numéro de compte
    CompteBancaire(String unNom, String uneAdresse) {
        nom = unNom;
        adresse = uneAdresse;
        numéro = premierNuméroDisponible;
        solde = 0;
        // Incrémenter la variable de classe
        premierNuméroDisponible++;
    }

    // Constructeur par défaut, sans paramètres
    CompteBancaire() {
        nom = "";
        adresse = "";
        numéro = 0;
        solde = 0;
    }

    // Les méthodes
    public void créditer(int montant) {
        solde += montant;
    }
    public String débiter(int montant) {
        solde -= montant;
        return "Débit accepté";
```

```
    }
    public void afficherEtat() {
        System.out.println("Compte numéro " + numéro +
                           " ouvert au nom de " + nom);
        System.out.println("Adresse du titulaire : " + adresse);
        System.out.println("Le solde actuel du compte est de " +
                           solde + " euros.");
        System.out.println("***********************************************");
    }
    // Accès en lecture
    public String nom() {
        return this.nom;
    }
    public String adresse() {
        return this.adresse;
    }

    public int numéro() {
        return this.numéro;
    }

    // méthode abstraite, doit être implantée dans les sous-classes
    // traitement quotidien appliqué au compte par un gestionnaire de comptes
    public abstract void traitementQuotidien();

    // sauvegarde du compte dans un BufferedWriter
    public void save(BufferedWriter bw) {
        Utils.saveString(bw, nom);
        Utils.saveString(bw, adresse);
        Utils.saveInt(bw, numéro);
        Utils.saveInt(bw, solde);
    }

    // chargement du compte à partir d'un BufferedReader
    public void load(BufferedReader br) {
        nom = Utils.loadString(br);
        adresse = Utils.loadString(br);
        numéro = Utils.loadInt(br);
        solde = Utils.loadInt(br);
    }
}
```

Les deux classes `CompteEpargne` et `CompteDepot` n'ont aussi que de légères modifications : les variables désignant les taux sont maintenant `protected` au lieu de `private`, et la modification de signature de la méthode `débiter` est répercutée :

```
// Classe CompteEpargne - version 1.2

import java.io.*;

/**
 * Classe représentant un compte d'épargne, sous-classe de CompteBancaire.
 * @author Karl Tombre
 */

public class CompteEpargne extends CompteBancaire {
    protected double tauxIntérêts;   // taux d'intérêts par jour

    // Constructeur
    CompteEpargne(String unNom, String uneAdresse, double unTaux) {
```

```
        // on crée déjà la partie commune
        super(unNom, uneAdresse);
        // puis on initialise le taux d'intérêt
        tauxIntérêts = unTaux;
    }

    // Constructeur par défaut
    CompteEpargne() {
        super();
        tauxIntérêts = 0.0;
    }

    // Méthode redéfinie : l'affichage
    public void afficherEtat() {
        System.out.println("Compte d'épargne");
        super.afficherEtat(); // appeler la méthode de même nom dans la superclasse
    }

    // Méthode redéfinie : débiter -- interdit de passer en-dessous de 0
    public String débiter(int montant) {
        if (montant <= solde) {
            solde -= montant;
            return "Débit accepté";
        }
        else {
            return "Débit non autorisé";
        }
    }

    // Définition de la méthode de traitementQuotidien
    public void traitementQuotidien() {
        créditer((int) ((double) solde * tauxIntérêts));
    }
    // sauvegarde du compte dans un BufferedWriter
    public void save(BufferedWriter bw) {
        super.save(bw);
        Double d = new Double(tauxIntérêts);
        Utils.saveString(bw, d.toString());
    }

    // chargement du compte à partir d'un BufferedReader
    public void load(BufferedReader br) {
        super.load(br);
        tauxIntérêts = Double.valueOf(Utils.loadString(br)).doubleValue();
    }
}
```

```
// Classe CompteDepot - version 1.2

import java.io.*;

/**
 * Classe représentant un compte de dépôt, sous-classe de CompteBancaire.
 * @author Karl Tombre
 */

public class CompteDepot extends CompteBancaire {
    protected double tauxAgios;   // taux quotidien des agios

    // Constructeur
```

```
    CompteDepot(String unNom, String uneAdresse, double unTaux) {
        // on crée déjà la partie commune
        super(unNom, uneAdresse);
        // puis on initialise le taux des agios
        tauxAgios = unTaux;
    }

    // Constructeur par défaut
    CompteDepot() {
        super();
        tauxAgios = 0.0;
    }

    // Méthode redéfinie : l'affichage
    public void afficherEtat() {
        System.out.println("Compte de dépôts");
        super.afficherEtat(); // appeler la méthode de même nom dans la superclasse
    }

    // Définition de la méthode de traitementQuotidien
    public void traitementQuotidien() {
        if (solde < 0) {
            débiter((int) (-1.0 * (double) solde * tauxAgios));
        }
    }
    // sauvegarde du compte dans un BufferedWriter
    public void save(BufferedWriter bw) {
        super.save(bw);
        Double d = new Double(tauxAgios);
        Utils.saveString(bw, d.toString());
    }

    // chargement du compte à partir d'un BufferedReader
    public void load(BufferedReader br) {
        super.load(br);
        tauxAgios = Double.valueOf(Utils.loadString(br)).doubleValue();
    }
}
```

Dans l'interface **ListeDeComptes**, plusieurs méthodes changent de signature pour la même raison, et rendent désormais une chaîne de caractères, à savoir le message qui doit être affiché dans une fenêtre. Il s'agit des méthodes **supprimer**, **sauvegarder** et **charger**. Par ailleurs, nous ajoutons deux nouvelles méthodes, **listeDesNoms**, qui rend un tableau des noms des titulaires des comptes, et **getData**, qui rend un tableau d'objets représentant les différents champs de tous les comptes. Ces deux méthodes sont utiles pour certaines fonctionnalités de l'interface, comme nous allons le voir.

```
// Interface ListeDeComptes - version 1.2

import java.io.*;

/**
 * Interface représentant une liste de comptes et les opérations
 * que l'on souhaite effectuer sur cette liste
 * @author Karl Tombre
 */

public interface ListeDeComptes {
    // Récupérer un compte à partir d'un nom donné
    public CompteBancaire trouverCompte(String nom);
```

```
    // Ajout d'un nouveau compte
    public void ajout(CompteBancaire c);
    // Suppression d'un compte
    public String supprimer(CompteBancaire c);
    // Afficher l'état de tous les comptes
    public void afficherEtat();
    // Traitement quotidien de tous les comptes
    public void traitementQuotidien();
    // Sauvegarder dans un fichier
    public String sauvegarder(File f);
    // Charger à partir d'un fichier
    public String charger(File f);
    // Liste des noms des titulaires de comptes
    public String[] listeDesNoms();
    // Récupérer toutes les données
    public Object[][] getData();
}
```

La classe `AgenceBancaire` doit bien entendu mettre en œuvre toutes ces modifications apportées à l'interface. Vous noterez dans la méthode `getData` l'emploi de l'opérateur `instanceof` pour donner à un objet booléen la valeur *ad hoc*. Vous noterez également que comme nous devons passer un tableau d'objets, tous les types élémentaires (entier et booléen ici) sont convertis en objets (instances de `Integer` et `Boolean` ici).

```
// Classe AgenceBancaire - version 1.4

import java.io.*;

/**
 * Classe représentant une agence bancaire et les méthodes qui lui
 * sont associées.
 * @author Karl Tombre
 */

public class AgenceBancaire implements ListeDeComptes {
    private CompteBancaire[] tabComptes;  // le tableau des comptes
    private int capacitéCourante;         // la capacité du tableau
    private int nbComptes;                // le nombre effectif de comptes

    // Constructeur -- au moment de créer une agence, on crée un tableau
    // de capacité initiale 10, mais qui ne contient encore aucun compte
    AgenceBancaire() {
        tabComptes = new CompteBancaire[10];
        capacitéCourante = 10;
        nbComptes = 0;
    }

    // Méthode privée utilisée pour incrémenter la capacité
    private void augmenterCapacité() {
        // Incrémenter la capacité de 10
        capacitéCourante += 10;
        // Créer un nouveau tableau plus grand que l'ancien
        CompteBancaire[] tab = new CompteBancaire[capacitéCourante];
        // Recopier les comptes existants dans ce nouveau tableau
        for (int i = 0 ; i < nbComptes ; i++) {
            tab[i] = tabComptes[i];
        }
        // C'est le nouveau tableau qui devient le tableau des comptes
```

```
        // (l'ancien sera récupéré par le ramasse-miettes)
        tabComptes = tab;
    }

    // Les méthodes de l'interface
    // Ajout d'un nouveau compte
    public void ajout(CompteBancaire c) {
        if (nbComptes == capacitéCourante) { // on a atteint la capacité max
            augmenterCapacité();
        }
        // Maintenant je suis sûr que j'ai de la place
        // Ajouter le nouveau compte dans la première case vide
        // qui porte le numéro nbComptes !
        tabComptes[nbComptes] = c;
        // On prend note qu'il y a un compte de plus
        nbComptes++;
    }
    // Récupérer un compte à partir d'un nom donné
    public CompteBancaire trouverCompte(String nom) {
        boolean trouvé = false;  // rien trouvé pour l'instant
        int indice = 0;
        for (int i = 0 ; i < nbComptes ; i++) {
            if (nom.equalsIgnoreCase(tabComptes[i].nom())) {
                trouvé = true; // j'ai trouvé
                indice = i;    // mémoriser l'indice
                break;         // plus besoin de continuer la recherche
            }
        }
        if (trouvé) {
            return tabComptes[indice];
        }
        else {
            return null;  // si rien trouvé, je rend la référence nulle
        }
    }
    // Afficher l'état de tous les comptes
    public void afficherEtat() {
        // Il suffit d'afficher l'état de tous les comptes de l'agence
        for (int i = 0 ; i < nbComptes ; i++) {
            tabComptes[i].afficherEtat();
        }
    }

    // Traitement quotidien de tous les comptes
    public void traitementQuotidien() {
        for (int i = 0 ; i < nbComptes ;i++) {
            tabComptes[i].traitementQuotidien();
        }
    }

    // Suppression d'un compte
    public String supprimer(CompteBancaire c) {
        boolean trouvé = false;  // rien trouvé pour l'instant
        int indice = 0;
        for (int i = 0 ; i < nbComptes ; i++) {
            if (tabComptes[i] == c) { // attention comparaison de références
                trouvé = true; // j'ai trouvé
                indice = i;    // mémoriser l'indice
                break;         // plus besoin de continuer la recherche
            }
        }
```

```
        if (trouvé) {
            // Décaler le reste du tableau vers la gauche
            // On "écrase" ainsi le compte à supprimer
            for (int i = indice+1 ; i < nbComptes ; i++) {
                tabComptes[i-1] = tabComptes[i];
            }
            // Mettre à jour le nombre de comptes
            nbComptes--;
            return "Compte supprimé";
        }
        else {
            // Message d'erreur si on n'a rien trouvé
            return "Je n'ai pas trouvé ce compte";
        }
    }

    public String sauvegarder(File f) {
        BufferedWriter bw = Utils.openWriteFile(f);
        if (bw != null) {
            Utils.saveInt(bw, nbComptes);
            // Sauvegarder la variable de classe
            Utils.saveInt(bw, CompteBancaire.premierNuméroDisponible);
            for (int i = 0 ; i < nbComptes ; i++) {
                Class typeDeCompte = tabComptes[i].getClass();
                Utils.saveString(bw, typeDeCompte.getName());
                tabComptes[i].save(bw);
            }
            try {
                bw.close();
            }
            catch (IOException ioe) {}
            return "Sauvegarde effectuée";
        }
        else {
            return "Impossible de sauvegarder";
        }
    }
    public String charger(File f) {
        BufferedReader br = Utils.openReadFile(f);
        if (br != null) {
            nbComptes = Utils.loadInt(br);
            CompteBancaire.premierNuméroDisponible = Utils.loadInt(br);
            for (int i = 0 ; i < nbComptes ; i++) {
                try {
                    Class typeDeCompte = Class.forName(Utils.loadString(br));
                    tabComptes[i] = (CompteBancaire) typeDeCompte.newInstance();
                    tabComptes[i].load(br);
                }
                catch(ClassNotFoundException cnfe) {}
                catch(IllegalAccessException iae) {}
                catch(InstantiationException ie) {}
            }
            try {
                br.close();
            }
            catch (IOException ioe) {}
            return "Comptes chargés";
        }
        else {
```

```
            return "Impossible de charger la sauvegarde";
        }
    }
    // Rendre la liste des noms
    public String[] listeDesNoms() {
        String[] l = new String[nbComptes]; // créer un tableau de chaînes
        for (int i = 0 ; i < nbComptes ; i++) {
            l[i] = tabComptes[i].nom();
        }
        return l;
    }

    public Object[][] getData() {
        Object[][] theData = new Object[nbComptes][5];
        for (int i = 0 ; i < nbComptes ; i++) {
            theData[i][0] = tabComptes[i].nom();
            theData[i][1] = tabComptes[i].adresse();
            theData[i][2] = new Integer(tabComptes[i].numéro());
            // solde est protected et on y accède directement
            // puisqu'on est dans le même package
            theData[i][3] = new Integer(tabComptes[i].solde);
            theData[i][4] = (tabComptes[i] instanceof CompteEpargne) ?
                new Boolean(true) : new Boolean(false);
        }
        return theData;
    }
}
```

Toutes ces modifications étant faites, passons au programme interactif proprement dit. Celui-ci va être défini dans une classe `GestionBanque` ; comme vous pouvez le constater, tout se passe dans le constructeur, la procédure `main` ne contient qu'une ligne. Si vous comparez ce programme avec celui de la classe `Banque` que nous avons fait évoluer jusque là, vous remarquerez premièrement qu'à part les opérations de chargement à partir d'un fichier et de sauvegarde à la sortie du programme, on ne reconnaît plus du tout le programme. Mais en étudiant le programme de plus près, vous constaterez que toutes les anciennes fonctionnalités sont présentes ; seulement, nous suivons maintenant un style de programmation événémentielle[4].

```
import javax.swing.*;
import javax.swing.event.*;
import java.awt.*;
import java.awt.event.*;
import java.io.*;

/**
 * Classe GestionBanque, permettant de gérer les comptes d'une agence bancaire
 * à partir d'une interface Swing
 *
 * @author "Karl Tombre" <Karl.Tombre@loria.fr>
 * @version 1.0
 * @see ActionListener
 */
```

[4] En fait, on ne retrouve pas vraiment toutes les fonctionnalités : je vous ai laissé la programmation d'un bouton permettant de supprimer un compte, à titre d'exercice d'application !

```
public class GestionBanque implements ActionListener {
    JFrame myFrame;           // la fenêtre principale
    ListeDeComptes monAgence; // l'agence bancaire

    /**
     * Constructeur - c'est ici que tout se passe en fait. On met en place
     * les éléments contenus dans la fenêtre et on active les événements.
     */

    GestionBanque() {
        // Créer une fenêtre
        myFrame = new JFrame("Gestion bancaire");  // titre de la fenêtre
        // arrêter tout si on ferme la fenêtre, pour éviter d'avoir un
        // processus qui continue à tourner en aveugle
        myFrame.addWindowListener(new WindowAdapter() {
                public void windowClosing(WindowEvent e) {
                    System.exit(0);
                }
            });

        // Création du panel principal
        JPanel mainPanel = new JPanel();
        // GridLayout(0, 2) --> 2 colonnes, autant de lignes que nécessaire
        mainPanel.setLayout(new GridLayout(0, 2));
        // une petite bordure de largeur 5
        mainPanel.setBorder(BorderFactory.createEmptyBorder(5,5,5,5));

        // On crée la liste des comptes et on charge la sauvegarde éventuelle
        monAgence = new AgenceBancaire();
        // On pourrait envisager de sélectionner le fichier via un FileChooser !
        File fich = new File("comptes.dat");
        monAgence.charger(fich);

        JOptionPane.showMessageDialog(mainPanel, monAgence.charger(fich));

        // Création des boutons et association avec les actions
        mainPanel.add(new JLabel("Gestion bancaire", JLabel.CENTER));
        JButton bQuitter = new JButton("Quitter");
        bQuitter.setVerticalTextPosition(AbstractButton.CENTER);
        bQuitter.setHorizontalTextPosition(AbstractButton.CENTER);
        bQuitter.setActionCommand("quitter");
        bQuitter.addActionListener(this);
        mainPanel.add(bQuitter);
        JButton bList = new JButton("Liste des comptes");
        bList.setVerticalTextPosition(AbstractButton.CENTER);
        bList.setHorizontalTextPosition(AbstractButton.CENTER);
        bList.setActionCommand("list");
        bList.addActionListener(this);
        mainPanel.add(bList);
        JButton bAjout = new JButton("Ajouter un compte");
        bAjout.setVerticalTextPosition(AbstractButton.CENTER);
        bAjout.setHorizontalTextPosition(AbstractButton.CENTER);
        bAjout.setActionCommand("ajout");
        bAjout.addActionListener(this);
        mainPanel.add(bAjout);
        JButton bCredit = new JButton("Créditer");
        bCredit.setVerticalTextPosition(AbstractButton.CENTER);
        bCredit.setHorizontalTextPosition(AbstractButton.CENTER);
        bCredit.setActionCommand("crédit");
        bCredit.addActionListener(this);
        mainPanel.add(bCredit);
```

```
        JButton bDebit = new JButton("Débiter");
        bDebit.setVerticalTextPosition(AbstractButton.CENTER);
        bDebit.setHorizontalTextPosition(AbstractButton.CENTER);
        bDebit.setActionCommand("débit");
        bDebit.addActionListener(this);
        mainPanel.add(bDebit);
        JButton bConsultation = new JButton("Consulter un compte");
        bConsultation.setVerticalTextPosition(AbstractButton.CENTER);
        bConsultation.setHorizontalTextPosition(AbstractButton.CENTER);
        bConsultation.setActionCommand("consulter");
        bConsultation.addActionListener(this);
        mainPanel.add(bConsultation);
        JButton bTraitement = new JButton("Traitement quotidien");
        bTraitement.setVerticalTextPosition(AbstractButton.CENTER);
        bTraitement.setHorizontalTextPosition(AbstractButton.CENTER);
        bTraitement.setActionCommand("traitement");
        bTraitement.addActionListener(this);
        mainPanel.add(bTraitement);


        //Ajouter le panel à la fenêtre et l'afficher
        myFrame.setContentPane(mainPanel);

        myFrame.pack();        // Taille naturelle
        myFrame.setVisible(true); // rendre visible
    }

    private CompteBancaire selectionCompte() {
        // Récupérer la liste des noms
        String[] l = monAgence.listeDesNoms();
        // Lancer un choix du compte sur le nom du titulaire
        ChoixCompte choix = new ChoixCompte(myFrame, l);
        // Quand on revient, on a choisi
        String nomChoisi = choix.value;
        // Rendre le compte correspondant
        return monAgence.trouverCompte(nomChoisi);
    }

    // Comme GestionBanque se déclare 'implements ActionListener'
    // et que c'est this qui a été donné comme ActionListener sur les
    // différents événements, c'est ici qu'il faut mettre la méthode
    // actionPerformed qui indique les actions à entreprendre suivant
    // les choix de l'utilisateur
    public void actionPerformed(ActionEvent e) {
        if (e.getActionCommand().equals("list")) {
            Object[][] t = monAgence.getData();
            // Afficher la table des comptes
            TableComptes tab = new TableComptes(myFrame, t);
        }
        else if (e.getActionCommand().equals("ajout")) {
            SaisieCompte s = new SaisieCompte(myFrame);
            // récupérer le compte saisi dans la variable c
            CompteBancaire c = s.leCompte;
            if (monAgence.trouverCompte(c.nom()) == null) {
                monAgence.ajout(c);
            }
            else {
                JOptionPane.showMessageDialog(myFrame,
                                              "Impossible, ce nom existe déjà");
            }
        }
```

```
        else if (e.getActionCommand().equals("débit")) {
            CompteBancaire c = selectionCompte();
            ChoixMontant m = new ChoixMontant(myFrame, "débiter");
            int montant = m.montant;
            // afficher le message rendu par le débit
            JOptionPane.showMessageDialog(myFrame, c.débiter(montant));
        }
        else if (e.getActionCommand().equals("crédit")) {
            CompteBancaire c = selectionCompte();
            ChoixMontant m = new ChoixMontant(myFrame, "créditer");
            int montant = m.montant;
            c.créditer(montant);
        }
        else if (e.getActionCommand().equals("quitter")) {
            // Sauvegarder la base et afficher le message rendu
            try {
                File fich = new File("comptes.dat");
                if (!fich.exists()) {
                    FileOutputStream fp =
                        new FileOutputStream("comptes.dat");
                    fich = new File("comptes.dat");
                }
                JOptionPane.showMessageDialog(myFrame,
                                              monAgence.sauvegarder(fich));
            }
            catch (IOException ioe) {}
            myFrame.setVisible(false);
            System.exit(0);
        }
        else if (e.getActionCommand().equals("consulter")) {
            CompteBancaire c = selectionCompte();
            FicheCompte f = new FicheCompte(myFrame, c);
        }
        else if (e.getActionCommand().equals("traitement")) {
            monAgence.traitementQuotidien();
        }
    }

    // La méthode main est réduite à la création de la fenêtre
    public static void main(String[] args) {
        GestionBanque g = new GestionBanque();
    }
}
```

La figure 6.1 illustre l'interface ainsi créée. Cette classe fait appel à un certain nombre d'autres

FIG. 6.1 – Interface créée par la classe `GestionBanque`.

classes, chacune gérant un type de fenêtre particulier. Les seules fenêtres non gérées par une classe explicitement créée sont les messages courts, que nous affichons dans une fenêtre de dialogue grâce à la méthode de classe `JOptionPane.showMessageDialog` (cf. Fig. 6.2).

FIG. 6.2 – Exemple de fenêtre de dialogue bref.

La classe `ChoixCompte` est utilisée chaque fois que l'on souhaite choisir l'un des comptes existants, à partir de la liste des noms des titulaires :

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class ChoixCompte extends JDialog {
    // Le nom sélectionné
    public String value;
    private JList list;

    ChoixCompte(Frame f, String[] tabNoms) {
        super(f, "Choix d'un compte", true);
        final JButton okButton = new JButton("OK");
        okButton.addActionListener(new ActionListener() {
                public void actionPerformed(ActionEvent e)  {
                    value = (String) list.getSelectedValue();
                    setVisible(false);
                }
            });
        list = new JList(tabNoms);
        // Valeur par défaut
        if (tabNoms.length > 0) {
            value = tabNoms[0];
            list.setSelectedIndex(0); // Par défaut
        }
        else {
            value = "";
        }
        // On n'a le droit que de choisir un nom à la fois
        list.setSelectionMode(ListSelectionModel.SINGLE_SELECTION);
        list.addMouseListener(new MouseAdapter() {
                public void mouseClicked(MouseEvent e) {
                    if (e.getClickCount() == 2) {
                        okButton.doClick();
                    }
                }
            });
        // La liste peut être longue, donc mettre un scroll
        JScrollPane listScroller = new JScrollPane(list);
        // Créer un container avec un titre et la liste de choix
        JPanel listPane = new JPanel();
        listPane.setLayout(new BoxLayout(listPane, BoxLayout.Y_AXIS));
        JLabel label = new JLabel("Choisissez un compte");
        listPane.add(label);
        listPane.add(listScroller);
```

```
        listPane.setBorder(BorderFactory.createEmptyBorder(10,10,10,10));

        // Mettre aussi le bouton
        JPanel buttonPane = new JPanel();
        buttonPane.add(okButton);

        // Mettre tout ça dans la fenêtre
        Container contentPane = getContentPane();
        contentPane.add(listPane, BorderLayout.CENTER);
        contentPane.add(buttonPane, BorderLayout.SOUTH);

        pack();
        setVisible(true);
    }
}
```

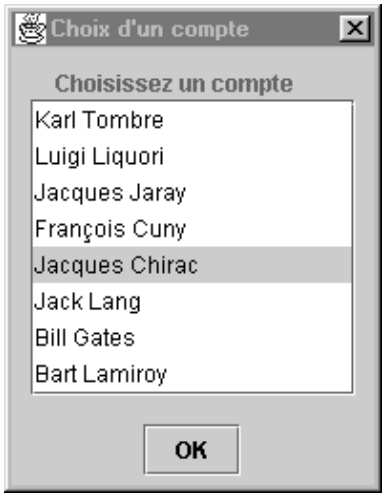


FIG. 6.3 – Fenêtre ouverte par la classe `ChoixCompte`.

La classe `TableComptes`, quant à elle, affiche l'état de tous les comptes :

```
import javax.swing.*;
import javax.swing.table.AbstractTableModel;
import java.awt.*;
import java.awt.event.*;

public class TableComptes extends JDialog {

    class MyTableModel extends AbstractTableModel {
        final String[] nomsCol = {"Nom", "Adresse", "Numéro", "Solde", "Compte d'épargne"};
        Object[][] data;

        MyTableModel(Object[][] myData) {
            super();
            data = myData;
        }

        public int getColumnCount() {
            return nomsCol.length;
        }
        public int getRowCount() {
            return data.length;
```

```
        }
        public String getColumnName(int col) {
            return nomsCol[col];
        }
        public Object getValueAt(int row, int col) {
            return data[row][col];
        }
        public Class getColumnClass(int c) {
            return getValueAt(0, c).getClass();
        }
        // Interdire l'édition
        public boolean isCellEditable(int row, int col) {
            return false;
        }
    }

    TableComptes(Frame f, Object[][] myData) {
        super(f, "Liste des comptes", true);
        MyTableModel myModel = new MyTableModel(myData);
        JTable table = new JTable(myModel);
        // Mettre dans un ascenseur
        JScrollPane scrollPane = new JScrollPane(table);

        // Créer un panel pour mettre tout ça
        JPanel listPane = new JPanel();
        listPane.setLayout(new BoxLayout(listPane, BoxLayout.Y_AXIS));
        JLabel label = new JLabel("Liste des comptes");
        listPane.add(label);
        listPane.add(scrollPane);
        listPane.setBorder(BorderFactory.createEmptyBorder(10,10,10,10));

        // Mettre un bouton OK
        final JButton okButton = new JButton("OK");
        okButton.addActionListener(new ActionListener() {
                public void actionPerformed(ActionEvent e) {
                    setVisible(false);
                }
            });
        // Le mettre dans un panel
        JPanel buttonPane = new JPanel();
        buttonPane.add(okButton);

        // mettre le tout dans la fenêtre de dialogue
        Container contentPane = getContentPane();
        contentPane.add(listPane, BorderLayout.CENTER);
        contentPane.add(buttonPane, BorderLayout.SOUTH);

        pack();
        setVisible(true);
    }
}
```

La classe `SaisieCompte` crée une fenêtre de saisie, dans laquelle l'utilisateur peut entrer les données d'un nouveau compte qu'il veut créer. L'ajout lui-même reste dans `GestionBanque`, et on commence par vérifier que le nom du titulaire ne figure pas déjà dans la liste.

```
import javax.swing.*;
import javax.swing.event.*;
import java.awt.*;
import java.awt.event.*;
```

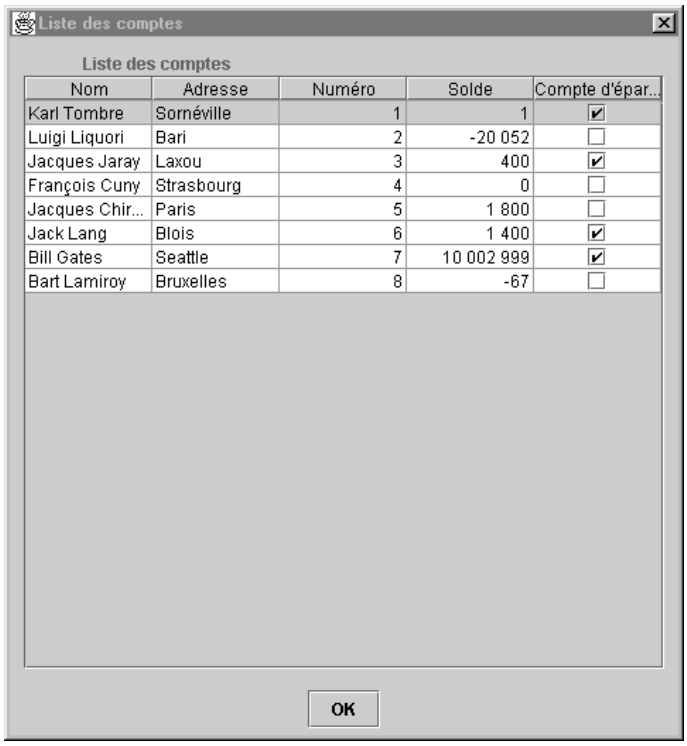


FIG. 6.4 – Fenêtre ouverte par la classe `TableComptes`.

```
public class SaisieCompte extends JDialog {
    public CompteBancaire leCompte;
    JTextField nom;
    JTextField adresse;
    JCheckBox épar;
    boolean épargne;

    SaisieCompte(Frame f) {
        super(f, "Saisie d'un nouveau compte", true);

        // Pour l'instant, pas de compte saisi
        leCompte = null;
        épargne = false;

        final JButton okButton = new JButton("OK");
        okButton.addActionListener(new ActionListener() {
                public void actionPerformed(ActionEvent e) {
                    if (épargne) {
                        leCompte = new CompteEpargne(nom.getText(),
                                                     adresse.getText(),
                                                     0.00015);
                    }
                    else {
                        leCompte = new CompteDepot(nom.getText(),
                                                   adresse.getText(),
                                                   0.00041);
                    }
                    setVisible(false);
                }
            });
```

```
        // On crée un panel pour la saisie
        JPanel pane = new JPanel();
        pane.setLayout(new GridLayout(0, 2));

        pane.add(new JLabel("Nom"));
        nom = new JTextField(25);
        pane.add(nom);
        pane.add(new JLabel("Adresse"));
        adresse = new JTextField(25);
        pane.add(adresse);

        épar = new JCheckBox("Compte d'épargne");
        épar.setSelected(false);
        épar.addItemListener(new ItemListener() {
                public void itemStateChanged(ItemEvent e) {
                    if (e.getStateChange() == ItemEvent.SELECTED) {
                        épargne = true;
                    }
                    else {
                        épargne = false;
                    }
                }
            });
        pane.add(new JLabel("Type du compte"));
        pane.add(épar);

        pane.setBorder(BorderFactory.createEmptyBorder(10,10,10,10));

        // Mettre aussi le bouton
        JPanel buttonPane = new JPanel();
        buttonPane.add(okButton);

        // Mettre tout ça dans la fenêtre
        Container contentPane = getContentPane();
        contentPane.add(pane, BorderLayout.CENTER);
        contentPane.add(buttonPane, BorderLayout.SOUTH);

        pack();
        setVisible(true);

    }
}
```

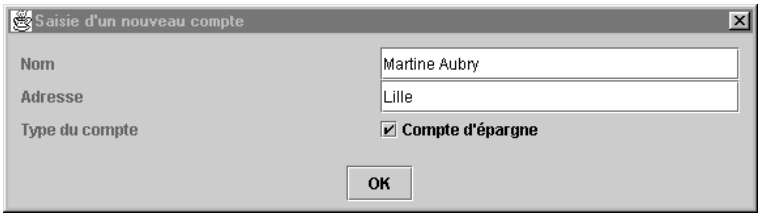


FIG. 6.5 – Fenêtre ouverte par la classe `SaisieCompte`.

La classe `ChoixMontant` permet de saisir un montant à créditer ou à débiter :

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
```

```
public class ChoixMontant extends JDialog {
    // Le montant sélectionné
    public int montant;
    JTextField myTextField;

    ChoixMontant(Frame f, String objet) {
        super(f, "Choix du montant", true);
        final JButton okButton = new JButton("OK");
        okButton.addActionListener(new ActionListener() {
                public void actionPerformed(ActionEvent e) {
                    montant = Integer.parseInt(myTextField.getText());
                    setVisible(false);
                }
            });
        montant = 0; // par défaut
        myTextField = new JTextField(10);
        myTextField.setText("0");
        // Créer un container avec un titre et le textfield
        JPanel tPane = new JPanel();
        tPane.setLayout(new GridLayout(0, 2));
        JLabel label = new JLabel("Montant à " + objet);
        tPane.add(label);
        tPane.add(myTextField);
        // Puis le bouton
        // Mettre aussi le bouton
        JPanel buttonPane = new JPanel();
        buttonPane.add(okButton);

        // Mettre tout ça dans la fenêtre
        Container contentPane = getContentPane();
        contentPane.add(tPane, BorderLayout.CENTER);
        contentPane.add(buttonPane, BorderLayout.SOUTH);

        pack();
        setVisible(true);
    }
}
```

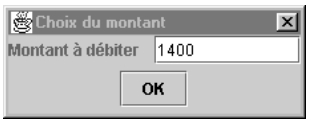


FIG. 6.6 – Fenêtre ouverte par la classe `ChoixMontant`.

Enfin, la classe `FicheCompte` crée une fenêtre où les détails d'un compte sont affichés :

```
import javax.swing.*;
import javax.swing.event.*;
import java.awt.*;
import java.awt.event.*;

// Solution simple : FicheCompte est fortement inspiré de SaisieCompte
// mais avec des labels seulement
public class FicheCompte extends JDialog {
    FicheCompte(Frame f, CompteBancaire c) {
        super(f, "Fiche du compte sélectionné", true);
```

```
        // Le bouton OK de la fin
        final JButton okButton = new JButton("OK");
        okButton.addActionListener(new ActionListener() {
                public void actionPerformed(ActionEvent e) {
                    setVisible(false);
                }
            });

        // On crée un panel
        JPanel pane = new JPanel();
        // 2 colonnes, autant de lignes qu'on veut
        pane.setLayout(new GridLayout(0, 2));

        pane.add(new JLabel("Nom "));
        pane.add(new JLabel(c.nom()));
        pane.add(new JLabel("Adresse "));
        pane.add(new JLabel(c.adresse()));
        pane.add(new JLabel("Numéro "));
        pane.add(new JLabel(Integer.toString(c.numéro())));
        pane.add(new JLabel("Solde "));
        pane.add(new JLabel(Integer.toString(c.solde)));

        if (c instanceof CompteDepot) {
            pane.add(new JLabel("Statut "));
            pane.add(new JLabel("Compte de dépôt"));

            pane.add(new JLabel("Taux quotidien des agios "));
            pane.add(new JLabel(Double.toString(((CompteDepot) c).tauxAgios)));
            }
        else {
            pane.add(new JLabel("Statut "));
            pane.add(new JLabel("Compte d'épargne"));

            pane.add(new JLabel("Taux quotidien des intérêts "));
            pane.add(new JLabel(Double.toString(((CompteEpargne) c).tauxIntérêts)));
        }

        pane.setBorder(BorderFactory.createEmptyBorder(10,10,10,10));

        // Mettre aussi le bouton
        JPanel buttonPane = new JPanel();
        buttonPane.add(okButton);

        // Mettre tout ça dans la fenêtre
        Container contentPane = getContentPane();
        contentPane.add(pane, BorderLayout.CENTER);
        contentPane.add(buttonPane, BorderLayout.SOUTH);

        pack();
        setVisible(true);

    }
}
```

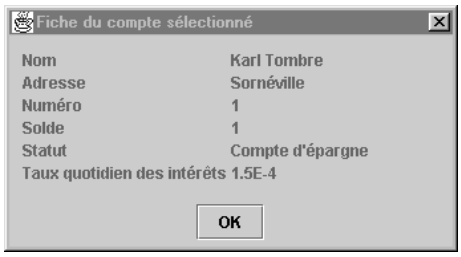


FIG. 6.7 – Fenêtre ouverte par la classe `FicheCompte`.

6.7 Conclusion

Ils n'ont rien rapporté que des fronts sans couleur
Où rien n'avait grandi, si ce n'est la pâleur.
Tous sont hagards après cette aventure étrange ;
Songeur ! tous ont, empreints au front, des ongles d'ange,
Tous ont dans le regard comme un songe qui fuit,
Tous ont l'air monstrueux en sortant de la nuit !
On en voit quelques-uns dont l'âme saigne et souffre,
Portant de toutes parts les morsures du gouffre !

Victor Hugo

Chapitre 7

Introduction sommaire au monde des bases de données

On doit être préoccupé uniquement de l'impression ou de l'idée à traduire. Les yeux de l'esprit sont tournés au dedans, il faut s'efforcer de rendre avec la plus grande fidélité possible le modèle intérieur. Un seul trait ajouté (pour briller, ou pour ne pas trop briller, pour obéir à un vain désir d'étonner, ou à l'enfantine volonté de rester classique) suffit à compromettre le succès de l'expérience et la découverte d'une loi. On n'a pas trop de toutes ses forces de soumission au réel, pour arriver à faire passer l'impression la plus simple en apparence, du monde de l'invisible dans celui si différent du concret où l'ineffable se résout en claires formules.

Marcel Proust

Nous avons abordé au chapitre 5 la notion de modèle de données, que nous avons définie comme une *abstraction* d'un type donné d'information, spécifiant les valeurs que peut prendre cette information, et les opérations qui permettent de la manipuler. Si les langages de programmation tels que Java offrent des mécanismes de représentation permettant de construire des structures de données et des classes représentant certains modèles de données, ils manquent cependant de souplesse quand les informations à manipuler comportent de nombreuses relations entre elles, par exemple. Nous avons vu au chapitre 6 comment la représentation d'un ensemble de comptes bancaires – problème que nous avons pourtant beaucoup simplifié par rapport à la réalité – nécessite la mise en place de fichiers, de mécanismes de sauvegarde et de chargement, etc. On imagine aisément que dans la réalité, on ne s'amuse pas, pour chaque application nécessitant la gestion d'informations un peu complexes, à remettre en place de tels mécanismes au moyen de lignes de code telles que nous les avons vues en Java. Au contraire, on recourt alors aux systèmes de gestion de bases de données[1].

On peut définir sommairement une *base de données* comme un ensemble de données représentant l'information sur un domaine d'application particulier, pour lequel on aura pris soin de spécifier les propriétés de ces données, ainsi que les relations qu'elles ont les unes avec les autres.

Un *système de gestion de bases de données* (SGBD) est un ensemble logiciel qui permet de créer, de gérer, d'enrichir, de maintenir de manière rémanente, et d'interroger efficacement des bases de données. Les SGBD sont des outils génériques, indépendants des domaines d'application des bases de données. Ils fournissent un certain nombre de services :

- capacité de spécifier les informations propres à une base de données, en s'appuyant sur un modèle de base – dans notre cas, nous nous intéresserons au modèle relationnel ;
- gestion de la rémanence des données – les opérations explicites de sauvegarde et récupération que nous avons programmées en Java au chapitre 6 sont fournies en standard par un SGBD, ce qui libère le concepteur d'une application informatique de tous les soucis de ce type ;

[1] La présentation qui suit reste très sommaire, et correspond juste à une première sensibilisation. Le lecteur qui voudrait en savoir plus peut se reporter à des ouvrages de référence comme [8], ou à des cours en ligne tel que celui d'Hervé Martin, à l'adresse
`http://www-lsr.imag.fr/Les.Personnes/Herve.Martin/HTML/FenetrePrincipale.htm`.

– possibilité de partage des données entre différents utilisateurs, voire différents logiciels ;
– confidentialité, intégrité et cohérence des données – dans un contexte réel, la gestion des droits d'accès, en consultation ou en modification, aux données d'une base est bien évidemment une contrainte absolue, tout comme l'assurance de la préservation de l'intégrité des données entre deux accès[2] ou de la cohérence entre les différentes informations que comporte la base ;
– possibilité de consultation des données (interrogation de la base), soit par des langages d'accès (nous allons voir dans ce chapitre le langage SQL), soit par le biais d'une page Internet, soit directement à partir d'un programme ;
– capacité de stockage élevée, afin de permettre la gestion de données volumineuses.

De nombreux SGBD existent, de complexité et de capacités variables. On peut citer des produits tels que Oracle dans le haut de gamme, le logiciel Access de Microsoft, plus rudimentaire, mais omniprésent en micro-informatique, ou des logiciels libres tels que MySQL[3].

7.1 Le modèle relationnel

Le modèle relationnel a été proposé en 1970 par Ted Codd, un chercheur chez IBM, sur la base de la théorie des ensembles et de l'algèbre relationnelle. Il correspond à une vision tabulaire des données : les attributs des données, et les relations entre elles, sont représentés dans des tables. Présentons rapidement ses principaux éléments :

– Un *domaine* est un ensemble de valeurs possibles. Chaque valeur du domaine est atomique. Le domaine correspond à la notion de type de données, tel que nous l'avons vue dans les chapitres précédents – mais restreint à des types élémentaires (atomiques).
– Un *schéma de relation* $R(A_1 : D_1, A_2 : D_2, \ldots A_n : D_n)$ permet de regrouper un ensemble d'attributs A_i, qui sont « typés » par leur domaine D_i. Ainsi, on pourra définir le schéma de relation `Client(idClient : entier, nom : chaîne(30), prénom : chaîne(30))`. Pour chaque schéma, on peut avoir un ensemble de *relations*, c'est-à-dire d'« instances » du schéma, par exemple :

idClient	nom	prénom
27384	Tombre	Karl
39827	Bonfante	Guillaume
73625	Lamiroy	Bart
98362	Chirac	Jacques
99375	Gates	Bill

Certaines valeurs peuvent prendre une valeur spéciale nulle qui indique qu'il n'y a pas d'information pour cet attribut.
– Chaque relation est caractérisée de manière unique par une ou plusieurs *clés*, qui sont des sous-ensembles d'attributs tels qu'il ne peut pas y avoir deux relations identiques avec les mêmes valeurs pour ce sous-ensemble. Parmi les clés possibles, on choisit une *clé primaire* – dans notre exemple, il s'agira assez naturellement de l'attribut `idClient`.
– Un *schéma de base de données* est un ensemble de schémas de relations assorti d'un ensemble de contraintes, dites contraintes d'intégrité. Une contrainte d'intégrité est une propriété, par exemple l'interdiction pour un attribut donné de prendre une valeur nulle, ou – et c'est là que les choses deviennent intéressantes – l'obligation pour une valeur d'attribut dans une première relation R_1 d'apparaître comme valeur de clé dans une autre relation R_2.
Ainsi, on si on définit un schéma de relation `Compte(numCompte : entier, idClient : entier, solde : réel)`, on pourra donner comme contrainte que toute relation instanciée doit correspondre à un client existant parmi les relations du schéma `Client`.
On dispose ainsi d'un ensemble d'*instances de base de données*, qui sont des instances des relations définies dans le schéma de la base de données, chaque relation de ces instances respectant les contraintes d'intégrité.

[2] Imaginez vos sentiments si un programme pirate pouvait accéder à vos informations médicales ou bancaires, par exemple, ou si une panne de courant entraînait la remise à zéro de votre compte bancaire...

[3] Logiciel libre, très facile à installer sous Linux et sous Windows. Voir `http://www.mysql.com/`. Le serveur MySQL de l'École se trouve à l'adresse `https://cesar.mines.inpl-nancy.fr/phpmyadmin`.

Complétons maintenant notre schéma de base de données exemple, en nous inspirant de l'exemple bancaire qui nous accompagne tout au long de ce polycopié. On peut imaginer les schémas de relation suivants (la clé primaire est mise en exergue pour chaque schéma). Il va sans dire que l'exemple reste très sommaire par rapport à la réalité bancaire...

```
Client(idClient : entier, nom : chaîne(30), prénom : chaîne(30))
Compte(numCompte : entier, idClient : entier, solde : réel, codeType : caractère)
Adresse(idClient : entier, rue : chaîne(50), codePostal : entier, commune : chaîne(40))
TypeCompte(code : caractère, nom : chaîne(40), tauxAgios : réel, tauxIntérêts : réel)
```

On pourrait alors avoir les instances suivantes pour les autres schémas de relation (les noms des attributs ont été abrégés pour des questions de place sur la page) :

Client

id	nom	prénom
27384	Tombre	Karl
39827	Bonfante	Guillaume
73625	Lamiroy	Bart
98362	Chirac	Jacques
99375	Gates	Bill

Compte

num	id	solde	code
12536	39827	345,40	D
16783	99375	63703,40	E
18374	39827	2500,45	E
26472	98362	3746,23	D
28374	73625	837,23	D
37468	27384	-981,34	D
37470	27384	1345,34	E
48573	73625	1673,87	E
57367	27384	938,34	E

Adresse

id	rue	cp	com
27384	53 Grande rue	54280	Sornéville
39827	3 avenue des tandems	54000	Nancy
73625	Bureau 498 à l'ENSMN	54000	Nancy
98362	Palais de l'Élysée	75000	Paris
99375	1 rue des fenêtres	99999	Seattle

TypeCompte

code	nom	txAg	txInt
D	Dépôts	14,3	0,0
E	Épargne	0,0	4,4

7.2 SQL

7.2.1 Introduction

SQL (*Structured Query Language*) est un langage conçu à partir des études sur le modèle relationnel, pour définir et accéder de manière normalisée aux bases de données relationnelles. Il est actuellement supporté par la plupart des SGBD du commerce et fait l'objet d'une norme. On peut donc le considérer comme une référence essentielle dans le domaine des bases de données.

SQL est un langage structuré, permettant d'exprimer de manière simple et lisible (proche de la langue anglaise) des requêtes qui traduisent les opérations de manipulation de données dans le modèle relationnel. Il permet d'une part de définir des schémas de bases de données relationnelles, ainsi que les données qui composent une base de données, et d'autre part de définir des requêtes, c'est-à-dire des manipulations dans une base de données pour en extraire les informations recherchées.

7.2.2 Création de schémas et insertion de données

Plutôt que de grandes explications théoriques, illustrons l'emploi de ce langage en créant tout d'abord les tables données précédemment, grâce à l'instruction `CREATE TABLE`. À noter que pour

détruire une table, on utilise l'instruction **DROP**, et que pour modifier un schéma de relation, on utilise l'instruction **ALTER TABLE**.

```
CREATE TABLE CLIENT(IDCLIENT INTEGER NOT NULL, NOM CHAR(30), PRENOM CHAR(30),
                    PRIMARY KEY (IDCLIENT))
CREATE TABLE COMPTE(NUMCOMPTE INTEGER NOT NULL, IDCLIENT INTEGER NOT NULL,
                    SOLDE DECIMAL(15,2), CODETYPE CHAR,
                    PRIMARY KEY (NUMCOMPTE))
CREATE TABLE ADRESSE(IDCLIENT INTEGER NOT NULL, RUE CHAR(50), CODEPOSTAL INTEGER,
                     COMMUNE CHAR(40),
                     PRIMARY KEY (IDCLIENT))
CREATE TABLE TYPECOMPTE(CODE CHAR NOT NULL, NOM CHAR(40), TAUXAGIOS DECIMAL(4,2),
                        TAUXINTERETS DECIMAL(4,2),
                        PRIMARY KEY (CODE))
```

Peuplons maintenant ces tables avec l'instruction **INSERT** (l'instruction **UPDATE** permet ultérieurement de modifier ces valeurs, et l'instruction **DELETE** de supprimer certaines relations d'une table – dans les deux cas, ces instruction se servent de la clause **WHERE**, que nous illustrons par la suite) :

```
INSERT INTO CLIENT VALUES(27384, "Tombre", "Karl")
INSERT INTO CLIENT VALUES(39827, "Bonfante", "Guillaume")
INSERT INTO CLIENT VALUES(73625, "Lamiroy", "Bart")
INSERT INTO CLIENT VALUES(98362, "Chirac", "Jacques")
INSERT INTO CLIENT VALUES(99375, "Gates", "Bill")

INSERT INTO COMPTE VALUES(12536, 39827, 345.40, 'D')
INSERT INTO COMPTE VALUES(16783, 99375, 63703.40, 'E')
INSERT INTO COMPTE VALUES(18374, 39827, 2500.45, 'E')
INSERT INTO COMPTE VALUES(26472, 98362, 3746.23, 'D')
INSERT INTO COMPTE VALUES(28374, 73625, 837.23, 'D')
INSERT INTO COMPTE VALUES(37468, 27384, -981.34, 'D')
INSERT INTO COMPTE VALUES(37470, 27384, 1345.34, 'E')
INSERT INTO COMPTE VALUES(48573, 73625, 1673.87, 'E')
INSERT INTO COMPTE VALUES(57367, 27384, 938.34, 'E')

INSERT INTO ADRESSE VALUES(27384, "53 Grande rue", 54280, "Sornéville")
INSERT INTO ADRESSE VALUES(39827, "3 avenue des tandems", 54000, "Nancy")
INSERT INTO ADRESSE VALUES(73625, "Bureau 498 à l'ENSMN", 54000, "Nancy")
INSERT INTO ADRESSE VALUES(98362, "Palais de l'Élysée", 75000, "Paris")
INSERT INTO ADRESSE VALUES(99375, "1 rue des fenêtres", 99999, "Seattle")

INSERT INTO TYPECOMPTE VALUES('D', "Dépôts", 14.3, 0.0)
INSERT INTO TYPECOMPTE VALUES('E', "Épargne", 0.0, 4.4)
```

7.2.3 Projections et sélections

Passons maintenant à la manipulation de données. On exprime habituellement une requête de la manière suivante : **SELECT ... FROM ... WHERE ....** La clause **SELECT** indique que l'on effectue une opération sur la base de données, en lui appliquant certains opérateurs ou fonctions, afin d'obtenir un résultat. La clause **FROM** donne la liste des tables utilisées dans l'opération en cours. La clause optionnelle **WHERE**, quant à elle, donne une liste de conditions que doivent respecter les données sélectionnées.

L'opération la plus simple consiste à faire une *projection*, c'est-à-dire d'extraire un ou plusieurs attributs d'un schéma. Ainsi, voici une requête simple de type projection, et le résultat obtenu sur l'exemple développé :

```
SELECT NOM FROM CLIENT
```

```
Tombre
Bonfante
Lamiroy
Chirac
Gates
```

Voici un deuxième exemple dans lequel on extrait deux attributs :

```
SELECT CODEPOSTAL, COMMUNE FROM ADRESSE

54280, Sornéville
54000, Nancy
54000, Nancy
75000, Paris
99999, Seattle
```

Un cas particulier est le caractère `*`, qui correspond à l'extraction de tous les attributs :

```
SELECT * FROM TYPECOMPTE

D, Dépôts, 14.30, 0.00
E, Épargne, 0.00, 4.40
```

La deuxième opération est la *sélection*, dans laquelle on limite la projection aux données qui vérifient une certaine condition (spécifiée grâce à la clause `WHERE`). Ainsi, pour extraire les identificateurs de tous les clients qui habitent à Nancy, on pourra écrire :

```
SELECT IDCLIENT FROM ADRESSE WHERE COMMUNE="NANCY"

39827
73625
```

7.2.4 Jointure

On peut réaliser le produit cartésien de deux (ou plus) tables. Par exemple, pour sélectionner l'ensemble des données obtenues par le produit des tables `COMPTE` et `TYPECOMPTE`, il suffit d'écrire :

```
SELECT * FROM COMPTE, TYPECOMPTE

12536, 39827, 345.40, D, D, Dépôts, 14.3, 0.0
16783, 99375, 63703.40, E, D, Dépôts, 14.3, 0.0
18374, 39827, 2500.45, E, D, Dépôts, 14.3, 0.0
26472, 98362, 3746.23, D, D, Dépôts, 14.3, 0.0
28374, 73625, 837.23, D, D, Dépôts, 14.3, 0.0
37468, 27384, -981.34, D, D, Dépôts, 14.3, 0.0
37470, 27384, 1345.34, E, D, Dépôts, 14.3, 0.0
48573, 73625, 1673.87, E, D, Dépôts, 14.3, 0.0
57367, 27384, 938.34, E, D, Dépôts, 14.3, 0.0
12536, 39827, 345.40, D, E, Épargne, 0.0, 4.4
16783, 99375, 63703.40, E, E, Épargne, 0.0, 4.4
18374, 39827, 2500.45, E, E, Épargne, 0.0, 4.4
26472, 98362, 3746.23, D, E, Épargne, 0.0, 4.4
28374, 73625, 837.23, D, E, Épargne, 0.0, 4.4
37468, 27384, -981.34, D, E, Épargne, 0.0, 4.4
37470, 27384, 1345.34, E, E, Épargne, 0.0, 4.4
48573, 73625, 1673.87, E, E, Épargne, 0.0, 4.4
57367, 27384, 938.34, E, E, Épargne, 0.0, 4.4
```

On comprend aisément que ce produit cartésien brut a peu d'intérêt ; il en a néanmoins beaucoup plus si on ne retient que les enregistrements cohérents, typiquement, pour notre exemple, ceux dont les codes de type correspondent entre les deux tables. Cette opération s'appelle la *jointure* et s'écrit de la manière suivante :

```
SELECT * FROM COMPTE, TYPECOMPTE WHERE COMPTE.CODETYPE = TYPECOMPTE.CODE

12536, 39827, 345.40, D, D, Dépôts, 14.3, 0.0
26472, 98362, 3746.23, D, D, Dépôts, 14.3, 0.0
28374, 73625, 837.23, D, D, Dépôts, 14.3, 0.0
37468, 27384, -981.34, D, D, Dépôts, 14.3, 0.0
16783, 99375, 63703.40, E, E, Épargne, 0.0, 4.4
18374, 39827, 2500.45, E, E, Épargne, 0.0, 4.4
37470, 27384, 1345.34, E, E, Épargne, 0.0, 4.4
48573, 73625, 1673.87, E, E, Épargne, 0.0, 4.4
57367, 27384, 938.34, E, E, Épargne, 0.0, 4.4
```

Si on associe maintenant une projection et une jointure, on voit qu'on peut extraire des informations intéressantes, comme par exemple le nom, le code postal et la ville des clients qui ont un compte d'épargne :

```
SELECT NOM, CODEPOSTAL, COMMUNE FROM CLIENT, COMPTE, ADRESSE
  WHERE CLIENT.IDCLIENT = ADRESSE.IDCLIENT AND CLIENT.IDCLIENT = COMPTE.IDCLIENT
  AND CODETYPE='E'

Gates, 99999, Seattle
Bonfante, 54000, Nancy
Tombre, 54280, Sornéville
Lamiroy, 54000, Nancy
Tombre, 54280, Sornéville
```

La présence de plusieurs lignes identiques dans le résultat ci-dessus n'est pas une erreur, mais résulte du produit cartésien, qui a trouvé plusieurs comptes d'épargne pour le même client. Si on souhaite éliminer de tels doublons, on peut utiliser la clause `DISTINCT` :

```
SELECT DISTINCT NOM, CODEPOSTAL, COMMUNE FROM CLIENT, COMPTE, ADRESSE
  WHERE CLIENT.IDCLIENT = ADRESSE.IDCLIENT AND CLIENT.IDCLIENT = COMPTE.IDCLIENT
  AND CODETYPE='E'

Gates, 99999, Seattle
Bonfante, 54000, Nancy
Tombre, 54280, Sornéville
Lamiroy, 54000, Nancy
```

7.2.5 Quelques autres clauses et fonctions

Une fois de plus, nous n'avons pas la prétention de donner un cours complet sur les bases de données relationnelles, ni même sur SQL, mais simplement de vous faire goûter à la puissance de ce type de modèle et de langage. Nous illustrons donc sans beaucoup de commentaires quelques autres constructions possibles en SQL...

Pour connaître le nombre de comptes de chaque type, on peut utiliser la fonction `COUNT`, associée à la clause `GROUP BY` pour regrouper les comptes de même type dans le produit cartésien :

```
SELECT NOM, COUNT(*) FROM COMPTE, TYPECOMPTE
  WHERE COMPTE.CODETYPE = TYPECOMPTE.CODE
  GROUP BY CODE

Dépôts, 4
Épargne, 5
```

Pour connaître le nombre de clients dans les différentes communes de Meurthe-et-Moselle, on pourra écrire

```
SELECT COMMUNE, COUNT(*) FROM CLIENT, ADRESSE
  WHERE CLIENT.IDCLIENT = ADRESSE.IDCLIENT
  AND CODEPOSTAL >= 54000 AND CODEPOSTAL < 55000
  GROUP BY COMMUNE

Nancy, 2
Sornéville, 1
```

Si maintenant on veut limiter la recherche précédente aux communes ayant plus d'un client, on ne peut plus recourir simplement à la clause `WHERE`, car ce genre de condition ne porte pas sur les lignes individuelles sélectionnées, mais sur les groupes constitués par la clause `GROUP BY`. C'est à cela que sert la clause `HAVING` :

```
SELECT COMMUNE, COUNT(*) FROM CLIENT, ADRESSE
  WHERE CLIENT.IDCLIENT = ADRESSE.IDCLIENT
  AND CODEPOSTAL >= 54000 AND CODEPOSTAL < 55000
  GROUP BY COMMUNE
  HAVING COUNT(*) > 1

Nancy, 2
```

Pour calculer le solde cumulé de tous les comptes de chaque client, on utilisera la fonction `SUM` – à noter que d'autres fonctions existent, notamment `MAX`, `MIN` et `AVG` :

```
SELECT PRENOM, NOM, SUM(SOLDE) FROM CLIENT, COMPTE
  WHERE CLIENT.IDCLIENT = COMPTE.IDCLIENT
  GROUP BY CLIENT.IDCLIENT

Karl, Tombre, 1302.34
Guillaume, Bonfante, 2845.85
Bart, Lamiroy, 2511.10
Jacques, Chirac, 3746.23
Bill, Gates, 63703.40
```

Enfin, si je veux présenter la liste ci-dessus dans l'ordre alphabétique des noms de famille, on peut ajouter la clause `ORDER BY` :

```
SELECT PRENOM, NOM, SUM(SOLDE) FROM CLIENT, COMPTE
  WHERE CLIENT.IDCLIENT = COMPTE.IDCLIENT
  GROUP BY CLIENT.IDCLIENT
  ORDER BY NOM

Guillaume, Bonfante, 2845.85
Jacques, Chirac, 3746.23
Bill, Gates, 63703.40
Bart, Lamiroy, 2511.10
Karl, Tombre, 1302.34
```

7.3 JDBC

Jusqu'ici, nous avons vu les bases de données de manière complètement déconnectée de la programmation. On peut bien évidemment accéder directement à une base de données *via* un interprète du langage SQL ou par une interface Web, par exemple. Cependant, l'intérêt principal reste bien entendu d'intégrer directement l'accès à ces bases aux programmes écrits par ailleurs,

pour remplacer en particulier les manipulations lourdes de fichiers illustrées par exemple au § 6.3.2, par une gestion directe des données au moyen d'une base de données.

En Java, l'interface (API) JDBC a été justement conçue pour permettre aux applications écrites en Java de communiquer avec les SGBD, en exploitant notamment SQL. Nous nous contentons ici d'illustrer très sommairement l'emploi de cette API dans un programme Java ; le lecteur qui voudrait en savoir plus se reportera à l'un des nombreux ouvrages techniques disponibles, tel que [9]. À noter que les classes JDBC se trouvent dans le *package* `java.sql`. On n'oubliera donc pas de commencer par dire :

```
import java.sql.*;
```

Il faut noter que le SGBD peut très bien s'exécuter sur une autre machine que celle sur laquelle s'exécute l'application qui l'utilise. Il faut donc examiner ce qui se passe aussi bien du côté du *serveur*, en l'occurrence le SGBD, que du côté du *client*, à savoir l'application. En supposant qu'un SGBD s'exécute bien du côté serveur, il faut un *pilote*[4] pour y accéder. Les pilotes sont bien entendu propres à chaque SGBD ; si on imagine par exemple que l'on accède au SGBD MySql, on invoquera l'un des pilotes disponibles pour ce SGBD. Chaque pilote a ses règles propres pour le lancement ; celui que nous utilisons en TD à l'École, par exemple, est lancé par la commande suivante :

```
Class.forName("org.gjt.mm.mysql.Driver").newInstance();
```

La première chose à effectuer, une fois le pilote lancé, est de se connecter à la base de données. On utilise pour cela une méthode de la classe `DriverManager` du *package* `java.sql`, dont le rôle est de tenir à jour une liste des implantations de pilotes et de transmettre à l'application toute information utile pour se connecter aux bases. Ainsi, on pourra écrire :

```
try {
    Connection c = DriverManager.getConnection(
        "jdbc:mysql://cesar.mines.inpl-nancy.fr/cours_1a?
        user=cours_1a_user&
        password=mines");
} catch (SQLException err) {
    System.out.println("SQLException: " + err.getMessage());
    System.out.println("SQLState:     " + err.getSQLState());
    System.out.println("VendorError:  " + err.getErrorCode());
}
```

à condition bien entendu que le SGBD tourne bien sur la machine `cesar`, avec les informations d'authentification données ici. Vous noterez au passage que comme nombre de méthodes des classes du *package* `java.sql`, `getConnection` est susceptible de provoquer une exception, et que je prévois d'afficher un maximum d'informations dans ce cas.

La méthode `getConnection` rend une référence à un objet de type `Connection` ; cette dernière classe symbolise une transaction avec une base de données. C'est en utilisant une méthode de cette classe que l'on crée un objet de type requête SQL, représenté par la classe `Statement` :

```
try {
    Statement stmt = c.createStatement();
} catch (SQLException err) { ... }
```

Nous sommes maintenant prêts pour récupérer le résultat d'une requête, grâce à une méthode de cette dernière classe :

```
try {
    ResultSet rs = stmt.executeQuery(
    "SELECT * FROM COMPTE, TYPECOMPTE WHERE COMPTE.CODETYPE = TYPECOMPTE.CODE");
} catch (SQLException err) { ... }
```

[4] *Driver* en anglais.

Vous comprendrez aisément qu'à la place de la chaîne de caractères constante ci-dessus, la requête peut être constituée d'une chaîne construite par le programme d'application, à partir par exemple des données fournies par un utilisateur dans une partie interactive, ou de la logique propre à l'application. Nous vous laissons d'ailleurs le soin de le vérifier dans l'exercice qui vous est proposé dans le TD accompagnant ce cours.

Le résultat de la requête est une instance de la classe `ResultSet` ; les différents éléments peuvent être récupérés pour affichage, ou pour exploitation dans les calculs propres à l'application, par l'intermédiaire de l'interface de cette classe. Le résultat de la requête est une table, et les méthodes `getXXX` de la classe `ResultSet` permettent de lire le contenu des différentes colonnes de cette table, colonnes qui sont numérotées à partir de 1. Ainsi, dans l'exemple donné ci-dessus, si on veut afficher dans l'ordre le type de compte (dépôts ou épargne), le numéro de compte, et le solde, on pourra par exemple écrire :

```
try {
    while (rs.next()) {
        System.out.print("Compte de type " + rs.getString(6) + " numéro ");
        System.out.print(rs.getInt(1) + " -- solde : ");
        System.out.println(rs.getDouble(3));
    }
    // On ferme proprement
    rs.close();
} catch (SQLException err) {
    ...
}
```

ce qui donnera une sortie du genre :

```
Compte de type Dépôts numéro 12536 -- solde : 345.40
Compte de type Dépôts numéro 26472 -- solde : 3746.23
Compte de type Dépôts numéro 28374 -- solde : 837.23
Compte de type Dépôts numéro 37468 -- solde : -981.34
Compte de type Épargne numéro 16783 -- solde : 63703.40
Compte de type Épargne numéro 18374 -- solde : 2500.45
Compte de type Épargne numéro 37470 -- solde : 1345.34
Compte de type Épargne numéro 48573 -- solde : 1673.87
Compte de type Épargne numéro 57367 -- solde : 938.34
```

Annexe A

Quelques éléments d'histoire

Dès 1864, Babbage avait proposé un « moteur analytique » contenant un « magasin » où étaient stockées toutes les variables, et un « moulin » dans lequel étaient chargées les quantités sur lesquelles s'effectuaient les opérations. Les bases des architectures informatiques actuelles ont cependant été posées à Princeton vers 1940, par von Neumann. Un ordinateur est composé de :

- une unité centrale de traitement (CPU), composée elle-même
  - d'une unité arithmétique et logique (UAL), chargée d'effectuer les calculs proprement dits,
  - d'une unité de contrôle chargée du pilotage des calculs,
  - d'une unité d'entrées-sorties, car un ordinateur qui ne communiquerait pas d'une manière ou d'une autre avec le monde extérieur serait un système fermé et n'aurait bien entendu pas beaucoup de sens.
- une mémoire, constituée de cases repérables par une adresse, dans lesquelles sont stockées les données sur lesquelles on travaille *et les programmes*.

La première machine de von Neumann ne traitait que des données entières. Elle savait effectuer des opérations arithmétiques et stocker les résultats dans un accumulateur. La valeur de cet accumulateur pouvait être rangée dans une case quelconque de la mémoire. Le contrôle était constitué par un flot séquentiel d'instructions, interrompu par des branchements conditionnels ou inconditionnels.

Il est important de noter que dans la machine de von Neumann, la mémoire ne contient pas seulement les données, mais aussi le programme lui-même ; cela permet notamment les branchements conditionnels ou inconditionnels, c'est-à-dire la continuation de l'exécution à partir d'une adresse en mémoire non consécutive à l'adresse courante. Cette adresse de branchement peut être calculée ou stockée en mémoire.

Les instructions qu'on pouvait effectuer sur la machine de von Neumann étaient très simples :

- des instructions arithmétiques :

```
A := A + M[i] ; A := A - M[i] ; A := A + |M[i]| ;
A :=  A - |M[i]| ; A := M[i] ; A := -M[i] ;
A := |M[i]| ; A := - |M[i]| ;
A := A*2 ; A := A div 2 ;
A, R := (M[i]*R) div 2^39, (M[i]*R) mod 2^39  ;
A, R := A mod M[i], A div M[i] ;
```

- des déplacements entre l'accumulateur, le registre et la mémoire :

```
A := M[i] ;
M[i] := A ; R := M[i] ; A := R ;
```

- des instructions de transfert du contrôle :

```
goto M[i].{left,right} ;
if A >= 0 goto M[i].{left,right} ;
```

- la possibilité de modifier la partie gauche ou la partie droite de `M[i]`, correspondant à une adresse stockée.

Le *langage machine* est la notation directement compréhensible par la machine. Le langage machine « colle » au jeu d'instructions de l'ordinateur. Les tout premiers programmes étaient d'ailleurs entrés par le positionnement de clés correspondant aux 0 et 1 ! Mais rapidement, la notion de programme sur support extérieur s'est imposée ; d'ailleurs, cette idée était antérieure à l'informatique, puisque le métier à tisser de Jacquard au début du 19^{e} siècle était déjà contrôlé par un programme sur cartes perforées.

À la base, les programmes restent encore de nos jours des séquences de 0 et de 1, ou d'octets, ou de nombres, correspondant au jeu d'instructions de la machine sur laquelle ils sont exécutés. Toutefois, il est rapidement devenu nécessaire d'augmenter la lisibilité des programmes grâce aux langages d'assemblage (souvent appelés incorrectement *assembleur*), qui indiquent par des mots clés mnémotechniques les instructions du processeur et les divers registres disponibles.

Le premier langage de haut niveau a été FORTRAN, issu des besoins de création de gros programmes de calcul numérique. FORTRAN a continué à évoluer et reste le langage de choix pour ce type d'applications. En parallèle naissait le besoin d'un langage permettant d'exprimer et de structurer aisément les constructions algorithmiques. Algol 60 est l'ancêtre d'une longue lignée. Une première branche s'est poursuivie par Algol 68, qui a lui-même inspiré des langages tels que Pascal (1971), Modula-2 (1983), BCPL (1969) ou C (1972). Une deuxième branche, suscitée au départ par des besoins de simulation du monde réel, a donné la notion de langage à objets, avec l'ancêtre Simula-67, le fondateur Smalltalk (1972, puis 1980), ou Eiffel. C++ (1986) fait une synthèse de ces deux branches en gardant les constructions de C tout en ajoutant la *programmation objet*. Java, le langage que nous utilisons comme support de ce cours, est conceptuellement plus proche de Smalltalk, mais reprend beaucoup d'éléments de syntaxe de C++.

Ce très bref historique serait incomplet si on n'évoquait pas quelques-unes des autres familles, nées de besoins divers et ayant chacune ses fondements en termes de modèles de calcul et de modèles de données : les langages orientés gestion dans la lignée de COBOL, les langages fonctionnels (LISP, ML), la programmation logique (Prolog), la programmation parallèle/concurrente, etc.

Annexe B

La logique et l'algèbre de Boole

La principale contribution de George Boole, mathématicien et logicien anglais (1815–1864), a consisté en une mathématisation de la logique. Pour cela, il a introduit le traitement algébrique de variables sans signification numérique. L'algèbre de Boole, qu'il définit en 1854 dans son traité *An investigation of the laws of thought*, est une base mathématique de la logique des circuits électroniques, notamment.

Une algèbre de Boole est un domaine muni des constantes 0, 1 et des opérations inverse, somme et produit. Nous noterons l'inverse par une barre au dessus de l'opérande inversé, la somme par $\vee$ et le produit par $\wedge$. Ces opérations ont les propriétés suivantes :

1. la somme et le produit sont associatifs, commutatifs et idempotents ;
2. le produit est distributif sur la somme et la somme est distributive sur le produit ;
3. 0 est élément neutre de la somme, 1 est élément neutre du produit ;
4. 0 est élément absorbant du produit, 1 est élément absorbant de la somme ;
5. $x \vee \bar{x} = 1$ (loi du tiers exclu), $x \wedge \bar{x} = 0$ (principe de contradiction), $\bar{\bar{x}} = x$;
6. $\overline{x \wedge y} = \bar{x} \vee \bar{y}$, $\overline{x \vee y} = \bar{x} \wedge \bar{y}$ (lois de Morgan).

Dans la logique des circuits électroniques, on appelle habituellement le produit « ET » (AND), la somme « OU » (OR) et l'inverse « NON » (NOT), et le domaine est réduit à $\{0, 1\}$, qu'on appelle aussi $\{\text{faux}, \text{vrai}\}$ ou $\{\text{false}, \text{true}\}$[1].

Les fonctions logiques suivantes sont souvent utilisées :
- ou exclusif (XOR) : $x \,\text{XOR}\, y = (x \wedge \bar{y}) \vee (y \wedge \bar{x})$
- implication : $x \Rightarrow y = \bar{x} \vee y$
- équivalence : $x \Leftrightarrow y = (x \Rightarrow y) \wedge (y \Rightarrow x)$
- « NON OU » (NOR) : $x \,\text{NOR}\, y = \overline{x \vee y} = \bar{x} \wedge \bar{y}$
- « NON ET » (NAND) : $x \,\text{NAND}\, y = \overline{x \wedge y} = \bar{x} \vee \bar{y}$

Pour tous ces opérateurs et fonctions, on recourt habituellement à une *table de vérité*, qui permet d'énumérer toutes les solutions selon les états des entrées. J'en donne quelques-unes ci-dessous, à titre d'exemple :

x	y	$x \vee y$
0	0	0
0	1	1
1	0	1
1	1	1

x	y	$x \wedge y$
0	0	0
0	1	0
1	0	0
1	1	1

x	y	$x \Rightarrow y$
0	0	1
0	1	1
1	0	0
1	1	1

x	y	$x \Leftrightarrow y$
0	0	1
0	1	0
1	0	0
1	1	1

x	y	$x \,\text{XOR}\, y$
0	0	0
0	1	1
1	0	1
1	1	0

[1] Mais on notera que $(E, \cup, \cap)$, par exemple, définit également une algèbre de Boole (l'algèbre des parties d'un ensemble E), le produit étant dans ce cas l'intersection, la somme l'union, l'inverse la partie complémentaire, 1 est E et 0 est $\emptyset$.

Annexe C

Glossaire

Je donne ici la définition d'un certain nombre de termes employés tout au long de ce polycopié, ou en tout cas très usuels en informatique. Ces définitions sont tirées pour bonne partie du glossaire informatique des termes publiés au Journal officiel par la Commission générale de terminologie et de néologie le 22 septembre 2000[1], et pour certaines d'entre elles de la référence [7] ou d'autres sources.

Tous les termes répertoriés dans le glossaire apparaissent en italiques dans les définitions. La traduction anglaise des termes est donnée entre parenthèses.

abstraction de données (data abstraction) : principe selon lequel un *objet* est complètement défini par son *interface*, c'est-à-dire l'ensemble des opérations qui lui sont applicables. La réalisation de ces opérations et la représentation physique de l'état de l'objet restent cachées et inaccessibles au monde extérieur.

algorithme (algorithm) : jeu de règles ou de procédures bien défini qu'il faut suivre pour obtenir la solution d'un problème dans un nombre fini d'étapes.

algorithmique (algorithmics) : étude de la résolution de problèmes par la mise en œuvre de suites d'opérations élémentaires selon un processus défini aboutissant à une solution.

base de données (database) : ensemble de *données* organisé en vue de son utilisation par des *programmes* correspondant à des applications distinctes et de manière à faciliter l'évolution indépendante des données et des programmes.

bit ou élément binaire (bit) : information représentée par un symbole à deux valeurs, généralement notées 0 et 1, associées aux deux états d'un dispositif. NB : le terme « bit » résulte de la contraction de l'anglais « binary digit ».

bogue (bug) : défaut de conception ou de réalisation se manifestant par des anomalies de fonctionnement.

café (coffee) : matière première noire transformée en *programmes* par les informaticiens. || Également prétexte à l'allongement inconsidéré des pauses prises par les élèves-ingénieurs.

classe (class) : description d'une famille d'*objets* similaires possédant un état, décrit par des *variables*, et un comportement, décrit par des *méthodes*. Elle sert de modèle pour créer ses représentants, les *instances* (cf. § 4.1).

constructeur (constructor) : procédure d'initialisation, activée au moment de la création d'une nouvelle *instance*. En Java, le constructeur porte toujours le même nom que la *classe* pour laquelle il est défini (cf. § 4.2).

donnée (data) : représentation d'une information sous une forme conventionnelle destinée à faciliter son traitement.

éditeur (editor) : *programme* qui permet, à partir d'un écran, d'introduire des données textuelles ou graphiques ou d'en modifier la disposition. XEmacs est un exemple d'éditeur.

[1] Disponible en ligne sur le site RETIF à l'adresse `http://www-Rocq.INRIA.Fr/qui/Philippe.Deschamp/RETIF/`

encapsulation (encapsulation) : mécanisme permettant de regrouper dans une même entité des *données* et les opérations qui s'appliquent à ces données. Il permet de réaliser l'*abstraction de données*.

génie logiciel (software engineering) : application systématique des connaissances, des méthodes et des acquis scientifiques et techniques pour la conception, le développement, le test et la documentation de *logiciels*, afin d'en rationaliser la production, le suivi et la qualité.

héritage (inheritance) : mécanisme permettant le partage et la réutilisation de propriétés entre les *objets*. La relation d'héritage est une relation de généralisation/spécialisation, qui organise les objets en une structure hiérarchique (cf. § 4.6).

instance (instance) : représentant physique d'une *classe* obtenu par « moulage » du dictionnaire des *variables* d'instance et détenant les valeurs de ces variables. Son comportement est défini par les *méthodes* de sa classe (cf. § 4.2).

interactif (interactive) : qualifie les matériels, les *programmes* ou les conditions d'exploitation qui permettent des actions réciproques avec des utilisateurs ou avec des appareils.

interface (interface) : ensemble des opérations applicables à un *objet* et connues du monde extérieur (cf. § 4.1). ‖ En Java, l'interface est aussi une déclaration de *classe* sans comportement associé, c'est-à-dire sans code, ni *variables* d'instance. L'utilisation d'interfaces permet de spécifier un comportement que les *objets* d'un *type* donné doivent assurer, sans prendre aucune décision sur les structures de *données* à mettre en œuvre pour représenter concrètement cette interface. On favorise ainsi l'*abstraction de données* (cf. § 5.1.4).

liaison (binding) : mécanisme permettant d'associer un sélecteur à la *méthode* à appliquer. En Java, la liaison est toujours dynamique (cf. § 4.6.2).

logiciel (software) : ensemble des *programmes*, procédés et règles, et éventuellement de la documentation, relatifs au fonctionnement d'un ensemble de traitement de *données*.

méthode (method) : procédure ou fonction appartenant à l'*interface* d'une *classe* et désignée par un sélecteur (cf. § 4.1).

objet (object) : entité regroupant des *données* et des procédures et fonctions opérant sur ces données (cf. § 4.2). ‖ Par abus de langage, terme générique pour désigner une *instance*.

octet (byte) : unité de codage des informations. Un octet peut prendre 256 valeurs différentes.

programme (program) : suite d'instructions définissant des opérations à réaliser sur des *données*.

type (type) : le type d'une expression ou d'une *variable* indique le domaine des valeurs qu'elle peut prendre et les opérations qu'on peut lui appliquer (cf. § 2.2).

variable (variable) : grandeur qui, dans un *programme*, reçoit un nom et un *type* par une déclaration au cours de la programmation. Une variable est attribuée à un emplacement de mémoire qui, au cours de l'exécution du programme, peut recevoir différentes valeurs (cf. § 2.1). ‖ Une variable d'instance est déclarée dans une *classe*, mais chaque *instance* de la classe en possède son propre exemplaire (cf. § 4.2). Une variable de classe, au contraire, n'existe qu'en un seul exemplaire (cf. § 4.3).

Annexe D

Les mots clés de Java

Nous donnons ici la liste des mots clés de Java, avec une brève description de leur signification, et la référence à la page du polycopié où la construction ou le concept en question est expliqué.

Mot clé	**Description**	**Page**
`abstract`	Permet de rendre une classe, une méthode ou une interface abstraite	57
`boolean`	Type booléen	7
`break`	Sortie prématurée d'une itération ; également utilisé en liaison avec les constructions `switch`	14 & 19
`byte`	Type entier sur 8 bits	7
`case`	Permet d'étiqueter les différents cas dans une construction `switch`	14
`catch`	Récupération d'une exception	100
`char`	Type caractère	7
`class`	Définition d'une classe	24 & 40
`continue`	Rebouclage prématuré sur le début d'une boucle d'itération	19
`default`	Étiquette par défaut dans les constructions `switch`	14
`do`	Variante de la construction `while`	18
`double`	Type réel double précision	7
`else`	Partie *sinon* d'une conditionnelle	13
`extends`	Déclaration d'une sous-classe	58
`final`	Déclaration d'une constante. Accolé à une classe, indique qu'on interdit à cette classe d'avoir des sous-classes.	6 & 65
`finally`	Dans le mécanisme de traitement des exceptions, définition d'un bloc d'instructions qui doivent toujours être exécutées, qu'il y ait eu exception ou non	100
`float`	Type réel simple précision	7
`for`	Construction d'une itération	18
`if`	Partie *si ... alors* d'une conditionnelle	13
`implements`	Indique qu'une classe met en œuvre une interface	70
`import`	« Importation » d'une ou plusieurs classes d'un *package*	83
`instanceof`	Opérateur permettant de tester l'appartenance d'un objet à une classe	95
`int`	Type entier sur 32 bits	7
`interface`	Définition d'une interface	69
`long`	Type entier sur 64 bits	7
`native`	Mot clé permettant de qualifier des fonctions écrites dans un autre langage que Java, mais qu'on souhaite appeler à partir d'un programme en Java	Non traité dans ce cours

Mot clé	Description	Page
`new`	Création d'un nouvel objet par allocation de la mémoire nécessaire et appel éventuel du constructeur approprié	24 & 42
`null`	Valeur d'une référence nulle (à un objet inexistant ou non valide)	24
`package`	Déclaration du *package* d'appartenance de la classe	83
`private`	Définition d'une variable ou méthode privée	40
`protected`	Définition d'une variable ou méthode protégée	57
`public`	Définition d'une variable ou méthode publique	40
`return`	Valeur rendue par une fonction	28
`short`	Type entier sur 16 bits	7
`static`	Définition d'une variable ou méthode de classe	44
`super`	Référence à la méthode masquée par héritage, ou au constructeur de la super-classe	58
`switch`	Construction conditionnelle permettant de traiter différents cas	14
`synchronized`	Déclaration d'une méthode qui peut être appelée par plusieurs *threads*	Non traité dans ce cours
`this`	Référence de l'objet courant	49
`throw`	Déclenchement d'une exception	101
`throws`	Déclaration du fait qu'une exception peut être déclenchée par une méthode donnée	101
`transient`	Utilisé pour marquer des variables qui ne doivent pas être sérialisées	Non traité dans ce cours
`try`	Bloc d'instructions dans lequel une exception est suceptible d'être déclenchée	100
`void`	Indique une procédure (« fonction » qui ne rend rien)	31
`volatile`	Indique au compilateur qu'il ne faut pas effectuer certains types d'optimisation sur la variable ainsi marquée	Non traité dans ce cours
`while`	Construction d'une itération	16

Annexe E

Aide-mémoire de programmation

En complément de la progression du cours, où ces différentes constructions sont introduites, et de la table des mots clés donnée à l'annexe D, nous vous proposons une illustration par l'exemple des principales constructions de Java. Ces exemples n'ont aucune prétention d'exhaustivité, ni même d'utilité pratique, mais pourraient être utiles lorsque vous avez besoin de vous rafraîchir la mémoire sur la manière de réaliser telle ou telle construction.

Dans la même logique d'exemples, je ne donne que très peu de commentaires et d'explications complémentaires, et vous invite à vous reporter aux pages correspondantes (cf. table des mots clés par exemple), le cas échéant.

E.1 Constructions conditionnelles

La construction habituelle (cf. § 2.7.1) s'écrit :

```
if (<condition>) {
    <bloc-1>
}
else {
    <bloc-2>
}
```

la partie `else` étant facultative. Voici un exemple :

```
if (monnaie < prix) {
    System.out.println("Paiement insuffisant - veuillez ajouter des pièces");
}
else {
    System.out.println("Voici votre café");
    livraisonCafé();
    double rendu = prix - monnaie;
    if (rendu > 0) {
        System.out.println("Rendu = " + rendu);
    }
}
```

Une autre construction, moins souvent utilisée, permet de représenter un aiguillage par cas (cf. p. 14) :

```
switch(<expression>) {
case <label-1>:
    <instructions>
    break;
```

```
case <label-2>:
    <instructions>
    break;
...
default:
    <instructions>
}
```

Voici un exemple :

```
switch(sélection) {
case 'c':
    System.out.println("Café");
    break;
case 't':
    System.out.println("Thé");
    break;
case 'b':
    System.out.println("Bière");
    break;
case 'w':
    System.out.println("Whisky");
    break;
default:
    System.out.println("Choix invalide");
}
```

E.2 Constructions itératives

La construction itérative la plus simple est la construction « tant-que », qui s'écrit (voir § 2.7.2) :

```
while (<condition>) {
    <bloc>
}
```

Une variante – la seule différence est que le bloc s'exécute au moins une fois avant le test – s'écrit :

```
do {
    <bloc>
} while (<condition>);
```

Voici des exemples de ces deux constructions :

```
boolean faim = true;
while (faim) {
    manger();
    faim = getEtatEstomac();
}

int x = Utils.lireEntier("Donnez un nombre");
do {
    x += 3;
}
while (x < 100);
```

L'autre construction itérative, bien adaptée aux énumérations ou aux parcours de tableaux, est la suivante :

```
for (<initialisation> ; <condition de poursuite> ; <passage au suivant>) {
    <bloc>
}
```

Et voici un exemple :

```
int[] tab = new int[30];

// ...

int somme = 0;
for (int i = 0 ; i < 30 ; i++) {
    somme += tab[i];
}
```

E.3 Définition et appel d'une fonction

Quand on définit une fonction, on donne son type, son nom, et le type de ses paramètres, puis son corps, et on n'oublie pas de retourner un résultat :

```
[public] [static] <type> <fonction> (<type-1> <param-1>, ..., <type-n> <param-n>) {
    <corps>
    return <résultat à retourner>;
}
```

Voici un exemple :

```
public static double consommation(double nbKilomètres, double nbLitres) {
    return nbLitres * 100.0 / nbKilomètres;
}
```

La fonction s'utilise ensuite de la manière suivante :

```
double km = 458;
double l = Utils.lireDouble("Litres consommés ?");
double c = consommation(km, l);
System.out.println("Consommation = " + c + "litres aux 100 km");
```

E.4 Définition et appel d'une procédure

Une procédure est définie quasiment de la même manière ; la différence est dans le fait qu'elle ne « retourne » rien, ce qui est indiqué par le mot clé `void` :

```
public static void imprimerLaCommande(String[] articlesCommandés) {
    for (int i = 0 ; i < articlesCommandés.length ; i ++) {
        System.out.println(articlesCommandés[i]);
    }
}
```

Voici un exemple d'appel de procédure :

```
String[] commande = new String[3];
commande[0] = "Ordinateur";
commande[1] = "Imprimante";
commande[2] = "Scanner";
imprimerLaCommande(commande);
```

E.5 Définition d'une classe

```
public class Point {
    // variables d'instance
    protected double x;
    protected double y;

    // constructeur
    Point(double unX, double unY) {
        x = unX;
        y = unY;
    }

    // méthodes d'accès en lecture
    public double getX() {
        return x;
    }
    public double getY() {
        return y;
    }

    // méthodes d'accès en écriture
    public void setX(double unX) {
        x = unX;
    }
    public void setY(double unY) {
        y = unY;
    }

    // autres méthodes
    public double getRho() {
        return Math.sqrt(x * x + y * y);
    }
    // etc.
}
```

E.6 Instanciation d'une classe et accès aux méthodes

```
    double x1 = 3.4;
    double y1 = 2.6;
    double x2 = Utils.lireRéel("Abscisse = ");
    double y2 = Utils.lireRéel("Ordonnée = ");
    Point p1 = new Point(x1, y1);
    Point p2 = new Point(x2, y2);

    System.out.println("P1 : (" + p1.getX() + "," + p1.getY() + ")");
    p1.setX(5.6);
    p1.setY(p2.getX());
    double rho = p1.getRho();

    // etc.
```

E.7 Récupération d'une exception

```
    PileEntiers maPile = new PileEntiers();

    // ...
```

```
try {
    System.out.println("Je dépile " + maPile.depiler());
}
catch(EmptyStackException ese) {
    System.out.println("Attention, la pile est vide");
}
```

Annexe F

Quelques précisions sur le codage

Pour ceux qui souhaitent approfondir leur compréhension du codage interne des données, nous donnons dans ce chapitre quelques informations complémentaires[1].

Les ordinateurs, comme les humains, ont besoin de « parler le même langage » pour pouvoir échanger des données. Dans les temps héroïques des premiers ordinateurs, chaque constructeur de machine choisissait ses propres codages pour les données à traiter et à stocker. Mais dès qu'on a commencé à échanger des fichiers de données entre différentes machines, il s'est avéré nécessaire de fixer des normes de représentation. Quand la norme concerne même la représentation interne des données (voir par exemple la norme IEEE 754 décrite ci-après), elle permet de plus la conception de circuits de calcul indépendants qui viennent s'ajouter de manière modulaire à l'ordinateur (co-processeurs de calcul intensif, cartes graphiques...)

F.1 Le codage des caractères

Les premiers ordinateurs étaient américains et les données textuelles échangées étaient en anglais. On a très vite vu émerger la norme ASCII[2], qui fixe le codage des caractères sur 7 bits, ce qui permet donc 128 positions, dans l'intervalle $[0, 127]$. L'*octet* (*byte* en anglais) – ensemble de 8 bits – étant une entité de base en informatique, cela laissait l'intervalle $[128, 255]$ à la disposition des concepteurs ; certains ont utilisé ces positions pour des codes graphiques, d'autres – et notamment les européens – pour coder leurs caractères nationaux (é, è, ç, å, ø, ß, ñ, etc.)

L'internationalisation a ensuite imposé une normalisation des positions entre 128 et 255 également ; mais on se heurte déjà là à la diversité des situations : on ne peut pas « caser » sur 128 positions l'alphabet cyrillique, l'alphabet hébreu, l'alphabet grec, les caractères particuliers des différentes langues, les signes monétaires courants, etc. C'est ainsi qu'est apparue la norme ISO 8859, déclinée en plusieurs variantes ; pour l'Europe occidentale, c'est-à-dire ce qui nous concerne, la norme s'appelle ISO 8859-1, également appelée ISO-Latin-1. Elle reprend le codage ASCII de 0 à 127, et fixe les caractères correspondant aux positions 128 à 255[3].

Mais 256 positions ne suffiront jamais si on veut être capable de coder de manière uniforme tous les alphabets, y compris les idéogrammes orientaux. À l'ère de la mondialisation, il apparaît donc nécessaire d'étendre l'espace de codage, c'est pourquoi la norme Unicode a vu le jour. Celle-ci code chaque caractère sur 2 octets, soit 16 bits, ce qui donne 65 536 positions possibles ! Pour l'instant, seules 34 168 positions ont été attribuées, il reste donc de la place pour les extra-terrestres...

Java est le premier langage de programmation – à ma connaissance – qui a choisi Unicode comme codage interne de ses caractères. Un programme en Java peut donc avoir des variables qui portent des noms japonais et qui sortent des messages d'erreur en russe ; ce programme compilera et s'exécutera sur votre ordinateur français (mais évidemment, pour que vous puissiez visualiser le

[1] Merci à Bertrand Gaiffe, qui a rédigé une première version de cette « digression » relative au codage, en s'inspirant en partie de notes de cours trouvées sur le Web.

[2] *American Standard Code for Information Interchange*.

[3] Malheureusement pour nous, les grandes entreprises américaines, notamment Microsoft, ne respectent pas toujours cette norme, d'où les soucis qu'on a parfois sur le codage de certains caractères, quand on échange des fichiers entre le monde Microsoft et le monde Unix, par exemple.

code source du programme et les messages d'erreur sur votre écran, il faudra que vous installiez des polices de caractères japonaises et cyrilliques sur votre ordinateur).

Rassurez-vous quand même :

- Dans ce cours, nous nous limitons à des noms et à des messages en français... *and maybe in english.*
- Coder du français en Unicode, c'est coder en ISO-Latin-1 en intercalant à chaque fois un octet de valeur 0 – en d'autres termes, les 256 premières positions du codage Unicode correspondent au codage ISO-Latin-1.
- De toute façon, Java a prévu les filtres permettant de lire des fichiers où les caractères sont codés sur 8 bits, de convertir les données en interne en caractères sur 16 bits, et de sauvegarder à nouveau en 8 bits (cf. § 6.3.2).

F.2 Le codage des entiers

Les entiers regroupent les types `byte`, `short`, `int` et `long`. Le type de codage employé est le même, au nombre d'octets utilisés près. Plus précisement :

- 1 octet pour les `byte` $\rightarrow [-128, +127]$
- 2 octets pour les `short` $\rightarrow [-32768, +32767]$
- 4 octets pour les `int` $\rightarrow [-2^{31}, +2^{31} - 1]$
- 8 octets pour les `long` $\rightarrow [-2^{63}, +2^{63} - 1]$

Quand on code des entiers signés, il faut choisir la manière de représenter les nombres négatifs. Java code les entiers en complément à 2. Il est plus simple cependant de voir d'abord les autres formes de codage, pour comprendre l'intérêt du complément à 2.

F.2.1 Codage avec bit de signe

Dans la mesure où on manipule des entiers signés, la technique la plus simple serait d'avoir un bit de signe. Si le premier bit est 0, on a un entier positif, si c'est 1, on a un entier négatif.

Le problème est qu'on a alors une arithmétique compliquée : $a + (-b)$ ne se calcule pas comme $a - b$.

F.2.2 Le complément à un

Pour simplifier l'arithmétique, on peut employer le complément à un. Pour un entier positif, on ne change rien ; pour un entier négatif, on inverse tous les bits. Exemple :

$$\begin{array}{rcl} 8 & \rightarrow & 0000\ 1000 \\ -8 & \rightarrow & 1111\ 0111 \end{array}$$

On a donc le premier bit à 0 pour les positifs et le premier bit à 1 pour les négatifs. Pour additionner, on fait l'addition binaire et si on a une retenue, on l'ajoute au résultat.

Exemple :

$$\begin{array}{rrl} & 0001\ 0001 & (17) \\ + & 1111\ 0111 & (-8) \\ \hline & 1\ 0000\ 1000 & \\ & +1 & (\textit{retenue}) \\ \hline & 0000\ 1001 & (9) \end{array}$$

Problème : on a toujours deux représentations de zéro !

F.2.3 Le complément à deux

On représente toujours de la même manière les nombres positifs. En revanche, les nombres négatifs sont représentés par le complément à un auquel on ajoute 1.

Exemple :

$$\begin{array}{rcl} 8 & \rightarrow & 0000\ 1000 \\ -8 & \rightarrow & 1111\ 0111 + 1 = 1111\ 1000 \end{array}$$

Remarque :

$$\begin{array}{rcl} +0 & \rightarrow & 0000\ 0000 \\ -0 & \rightarrow & 1111\ 1111 + 1 = 0000\ 0000 \end{array}$$

On n'a donc plus cette fois-ci qu'un seul zéro.
Exemple d'addition (on ne tient pas compte du bit qui sort en débordement sur la gauche) :

$$\begin{array}{rll} & 0001\ 0001 & (17) \\ + & 1111\ 1000 & (-8) \\ \hline & 0000\ 1001 & (9) \end{array}$$

Pour ceux que cela amuse, formellement, étant donné un entier en binaire :
$b_n\ b_{n-1}\ \cdots\ b_0$, sa valeur est :

$$(\sum_{i=0}^{n-1} b_i.2^i) - b_n.2^n$$

d'où le nom de complément à 2 ; en toute rigueur, c'est un complément à 2^n.
Comme déjà précisé, tous les types entiers sont représentés en Java en complément à 2.

F.3 Le codage des réels

Les nombres flottants, utilisés pour représenter les réels, sont en fait des décimaux ; dans un univers numérique (tout du moins dans un univers numérique qui veut rester simple), on n'a pas de nombres en précision infinie.

Pour représenter les flottants, on emploie la norme IEEE 754. Dans une première étape, on considère les nombres décimaux comme :

$$N = (-1)^s.m.2^e$$

Avec s le signe, m la mantisse, e l'exposant. C'est la même chose qu'en base 10, avec bien sûr des puissances de 2 plutôt que des puissances de 10.

On normalise ensuite cette représentation en imposant à la mantisse d'être comprise entre 1 et 2 (c'est ce qu'on fait en base 10 en imposant à la mantisse d'être entre 1 et 10).

Puisque la mantisse commence alors forcément par 1, on n'a pas besoin de représenter le dit 1.

Remarque : on n'a pas de représentation de 0.

F.3.1 IEEE 754 pour les `float`

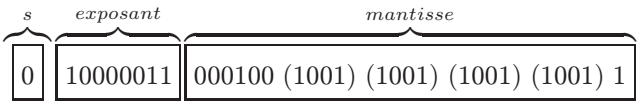
La norme IEEE 754, employée par Java, normalise simplement :

1. la taille prise pour représenter la mantisse et l'exposant,
2. une représentation de 0,
3. des représentations pour $+\infty$ et $-\infty$ et les débordements (comme après une division par 0 par exemple).

Dans le cas des `float` (32 bits), on a :
– le signe dans b_{31} (le bit le plus à gauche),
– l'exposant dans b_{30}–b_{23} ce qui donne 8 bits,
– la mantisse dans le reste : b_{22}–b_0.
L'exposant est représenté en ajoutant 127 (ce qui permet de balayer de -127 à +127).
De plus, conventionnellement, on a :
– 0 est représenté par exposant = mantisse = 0.

– ∞ est représenté par exposant = 255 (des 1 partout) et mantisse = 0. Le bit de signe distingue entre $+\infty$ et $-\infty$.
– exposant = 255 et mantisse $\neq 0$ désigne un débordement dans une opération précédente, et est généralement appelé NaN (*not a number*).

Pour ceux qui nous suivent encore, un dernier exemple :
on veut coder 17.15.
17 en base 2 $\rightarrow$ 10001
0.15 en base 2 $\rightarrow$ 0.00 (1001) (1001) $\cdots$
17.15 $\rightarrow$ 10001.00 (1001) (1001) $\cdots$
on normalise : $17.15 = 1.000100(1001)(1001)\cdots \times 2^4$
On a donc une mantisse $M = 000100(1001)(1001)\cdots$ (on ne représente pas le 1 « implicite »).
L'exposant est représenté par $E = e + 127 = 131 = 1000.0011$. Le nombre est positif donc $s = 0$.
Tout cela mis ensemble nous donne :



F.3.2 IEEE 754 pour les `double`

Pour les `double`, c'est pareil, sauf que l'exposant est codé sur 11 bits et la mantisse sur 52 bits, ce qui nous donne bien 64 bits avec le bit de signe.

Annexe G

Quelques conseils pour utiliser l'environnement JDE sous XEmacs

L'environnement JDE (*Java Development Environment*) vous permet d'écrire vos programmes Java, de les compiler et de les exécuter en restant sous XEmacs.

Sans vouloir donner un mode d'emploi complet de cet environnement très riche, voici quelques conseils et réponses aux questions les plus fréquentes :

Comment passer en mode JDE ? Normalement il suffit de charger un fichier ayant l'extension `.java`. Une erreur fréquente est de commencer à entrer son programme dans le buffer `*scratch*`, puis de le sauvegarder comme `MonProgramme.java`. Dans ces conditions, XEmacs ne passe pas en mode JDE. Il vaut mieux ouvrir le fichier `MonProgramme.java` directement – s'il n'existe pas encore, XEmacs se charge de le créer pour vous – comme cela vous êtes dès le départ en mode JDE. Si vous avez fait ce genre d'erreur, la manière la plus simple de s'en sortir est de fermer le fichier, puis de l'ouvrir à nouveau ; cette fois-ci, XEmacs passera en mode JDE, puisque votre fichier a l'extension `.java`.

Si tout cela ne suffit pas, vérifiez que vous avez bien la ligne

```
(require 'jde)
```

dans votre fichier `.emacs`, qui se trouve normalement dans votre répertoire de base.

Comment avoir les couleurs ? Normalement, vous devriez avoir un programme où les mots clés sont dans une certaine couleur, les noms des variables dans une autre, etc. Si ce n'est pas le cas chez vous, commencez par vérifier que vous êtes bien en mode JDE.

Si vous êtes bien en mode JDE, mais sans couleurs, vérifiez le menu *Options/Syntax Highlighting* de XEmacs. Vous devez normalement avoir sélectionné *Automatic*, *Fonts*, *Colors* et *Most*.

Comment avoir une indentation correcte ? Le mode JDE est construit au-dessus du mode d'édition Java, qui vous assure *a priori* une indentation correcte, à condition de prendre dès le départ de bonnes habitudes. Quand vous passez à la ligne suivante, commencez à taper ; le mode Java doit normalement vous insérer l'indentation correcte. Respectez-la ! À tout moment, vous pouvez taper la touche « tabulation » n'importe où dans la ligne ; cela indente la ligne courante de manière correcte.

Quand vous tapez une accolade fermante (`'}'`), l'indentation courante recule pour marquer la fermeture d'un bloc.

Je vous conseille par ailleurs de vous reporter aux règles d'indentation données au § H.2.

Comment compiler ? Une erreur classique est de cliquer sur l'icône « *Compile* » de XEmacs. En effet, celle-ci permet de lancer une compilation pour n'importe quel langage compilé *via* un utilitaire appelé `make`, qui n'est pas installé sur nos machines. De toute façon, JDE fournit un moyen beaucoup plus facile de compiler : sélectionnez l'item *Compile* dans le menu *JDE* (raccourci clavier `C-c C-v C-c`). Cela lance le compilateur Java – `javac` – sur le fichier courant, et donne une trace de la compilation dans un buffer séparé.

Un truc : si vous avez plusieurs fichiers (plusieurs classes) et que votre programme est tout prêt de marcher (vous n'êtes plus dans les toutes premières phases de débogage), compilez la classe dans laquelle se trouve la méthode de classe `main`. En effet, le compilateur va alors automatiquement compiler les autres fichiers dont il a besoin pour que le programme soit prêt à être exécuté – à condition bien sûr qu'il n'y ait plus d'erreur.

Comment exécuter le programme ? Toujours dans le menu *JDE*, cliquez sur l'item *Run App* (raccourci clavier `C-c C-v C-r`). Attention : il faut compiler avant de lancer l'exécution, sinon au pire vous avez une erreur, au mieux vous exécutez la dernière version compilée de votre programme...

Aussi longtemps que vous exécutez des applications avec entrées clavier et sorties écran (les programmes que nous avons écrits jusqu'au paragraphe 6.5.1 inclus), l'interprète Java qui doit être lancé est `java`. Quand on lance un programme avec interface graphique (§ 6.6), c'est la variante `javaw` qui doit être lancée.

Si JDE n'est pas configuré correctement pour le cas dans lequel vous êtes, vous pouvez mettre à jour l'option en sélectionnant la suite de menus suivante :

Options/Customize/Emacs/Programming/Tools/Jde/Project/Run Java Vm W.

Remplacez `java` par `javaw` ou réciproquement, puis faites *set* et *save*.

Annexe H

Conventions d'écriture des programmes Java

Ce chapitre rassemble quelques règles usuelles de présentation de programmes en Java. Il est fortement inspiré du document Sun qu'on peut par exemple trouver au centre de référence Java[1].

Il est important de noter que ces règles sont bien des *conventions* et ne font pas partie de la syntaxe de Java. Vous pouvez parfaitement écrire des programmes Java corrects mais illisibles ou très confus. Mais si une syntaxe correcte est nécessaire pour que la machine vous comprenne, une écriture claire et cohérente est nécessaire pour que d'autres humains vous comprennent !

H.1 Fichiers

Les noms de fichiers de source Java ont le suffixe `.java`. Les noms de fichiers compilés (bytecode) ont le suffixe `.class`.

Chaque fichier source Java contient une seule classe ou interface publique. Le nom du fichier (sans le suffixe `.java`) est *obligatoirement* le nom de cette classe ou interface publique. Si on y met des classes ou interfaces privées, la classe ou interface publique doit être la première à être définie dans le fichier.

On organisera le contenu de chaque fichier dans l'ordre suivant :

- Commentaires de début, comportant entre autres le nom de la classe, le nom de l'auteur, la date de création et de dernière mise à jour.
- Déclaration du *package* éventuel, suivie des importations éventuelles, par exemple :

```
package dupont.exercices;
import java.awt.*;
```

- Définition de la classe ou de l'interface (s'il y en a plusieurs, la classe ou interface publique en premier). Cette définition se fait dans l'ordre suivant :

[1] `http://java.sun.com/docs/codeconv/html/CodeConventionsTOC.doc.html`

Documentation de la classe ou de l'interface (commentaires `/** ... */`)	cf. § H.3
Déclaration de la classe ou de l'interface	Exemple : `public class Banque`
Commentaires de programmation (`/* ... */`), si nécessaire, c'est-à-dire commentaires généraux sur la mise en œuvre de la classe, qui n'ont pas leur place dans la documentation	cf. § H.3
Déclaration des variables de classe (`static`)	D'abord les `public`, puis les `protected`, puis les `private`
Déclaration des variables d'instance	D'abord les `public`, puis les `protected`, puis les `private`
Constructeurs	
Méthodes	Les regrouper de préférence par grandes fonctionnalités, pour rendre le tout le plus lisible possible

H.2 Indentation

Utiliser de préférence 4 espaces comme unité d'indentation. Règle de base : le niveau d'indentation correspond au degré d'imbrication.

Soyez cohérent dans vos alignements d'accolades ouvrantes et fermantes (utilisez partout la même convention). Le document SUN recommande de mettre l'accolade ouvrante à la fin de la ligne où commence le bloc concerné, et l'accolade fermante sur une ligne à part, indentée au niveau de l'instruction qui a entraîné l'ouverture du bloc, sauf pour les méthodes à corps vide, pour lesquelles l'accolade fermante suit immédiatement l'accolade ouvrante. Exemples :

```
class Test extends Object {
    int var1;
    int var2;

    Test(int i, int j) {
        var1 = i;
        var2 = j;
    }

    int methodeVide() {}

    void calculs() {
        ...
        if (var1 > 0) {
            var2--;
            ...
        }
        ...
        while (var1 != var2) {
            ...
        }
    }
    ...
}
```

Éviter les lignes trop longues (plus de 80 caractères), et souvenez-vous que vous pouvez toujours écrire une expression très longue sur plusieurs lignes. Dans ce cas, on cherchera à les couper aux endroits qui laissent la plus grande lisibilité :

- après une virgule,
- avant un opérateur,

– au niveau de parenthésage le plus englobant possible,
– aligner la nouvelle ligne avec le début de l'expression qui est au même niveau à la ligne précédente,
– si tout ce qui précède marche mal, indenter tout simplement de 4 ou 8 espaces.

Quelques exemples :

```
fonc(expressionTresLongue1, expressionTresLongue2, expressionTresLongue3,
     expressionTresLongue4, expressionTresLongue5);

var = fonc1(expressionTresLongue1,
            fonc2(expressionTresLongue2,
                  expressionTresLongue3));

impotAPayer = ageDuCapitaine * (longueur + largeur - hauteur)
              + 4 * tailleChemise;
```

Pour les instructions `if`, s'il faut présenter la condition sur plusieurs lignes, indenter de 8 espaces au lieu de 4 pour que l'instruction qui suit reste lisible :

```
if ((condition1 && condition2)
        || (condition3 && condition4)
        || (condition5 && condition6)) {
    faireQuelqueChose()   // reste lisible
    ...
}
```

H.3 Commentaires

H.3.1 Commentaires de programmation

On les utilise soit pour « masquer » une partie du code, en cours de développement, soit pour expliquer certains détails de mise en œuvre. Ils peuvent être mis sous la forme de l'un des styles suivants :

1. Blocs de commentaires : servent à expliciter des fichiers, des méthodes, des structures de données et des algorithmes. Les précéder d'une ligne blanche, et les présenter comme suit :

   ```
   /*
    * Ceci est un bloc de commentaire.
    * Il peut comporter plusieurs lignes, chacune commençant par une
    * étoile entourée de deux espaces, pour la lisibilité de l'ensemble
    */
   ```

2. Une ligne de commentaire : si votre commentaire ne tient pas sur une ligne, ne faites pas suivre deux lignes de commentaires ; utilisez de préférence un bloc de commentaires. Exemple de ligne de commentaire :

   ```
   if (condition) {
       /* Traiter le cas où tout va bien */
       ....
   }
   ```

3. Commentaires de fin de ligne : ils peuvent prendre les deux formes `/* ... */` ou `// ...`, la deuxième servant en outre à « masquer » une partie du code en cours de développement ou de déboguage. Exemples :

   ```
   if (a == 2) {
       return TRUE;      /* Cas particulier */
   ```

```
}
else if (toto > 1) {
    // Permuter les deux valeurs
    ...
}
else
    return FALSE;     // Explications

// if (tata > 1) {
//
//    // Trois valeurs à permuter
//    ...
//}
//else
//    return FALSE;
```

H.3.2 Commentaires de documentation

Ces commentaires sont traités comme les autres par le compilateur, mais sont interprétés par l'outil `javadoc`, qui permet de créer des pages WWW de documentation de vos programmes. Pour une introduction à javadoc, voir son site de référence[2], où vous trouverez aussi un manuel avec des exemples pour Windows. Un commentaire de documentation commence par `/**` et se termine par `*/`. Exemples de commentaires de documentation :

– Pour une classe :

```
/**
 * Cette classe est une spécialisation de la classe générale des clients.
 * @author   Jean Lingénieur
 * @see      Client
 */
class ClientPrivilegie extends Client
{
...
}
```

– Pour une méthode :

```
    /**
     * Rend la somme TTC à payer après application de la TVA et d'une
     * réduction éventuelle.
     * @param    sommeHT le montant hors taxes et avant réduction de la commande
     * @return   montant TTC net à payer
     */
    double sommeTTC(double sommeHT) {
        ...
    }
```

H.4 Déclarations et instructions

Il est recommandé de ne mettre qu'une déclaration par ligne, et de regrouper les déclarations en début de bloc. Essayez d'initialiser les variables locales dès leur déclaration (il y a bien sûr des cas où cela est difficile ou impossible, comme quand l'initalisation dépend d'un calcul préalable).

Par souci de clarté, évitez de masquer des variables à un certain niveau par des variables de même nom à un niveau d'imbrication inférieur, comme dans :

```
int total;
...
```

[2] `http://java.sun.com/j2se/javadoc/`

```
fonction() {
    if (condition) {
        int total;       // à éviter !!
        ...
    }
    ...
}
```

Mettez une seule instruction par ligne ; évitez par exemple :

```
compte++; valeur -= compte;    // à éviter !!
```

N'utilisez des parenthèses avec l'instruction `return` que lorsqu'elles clarifient la lecture du code.

H.5 Noms

Choisissez toujours des noms significatifs et non ambigus quant au concept qu'ils désignent. Tenez-vous en à une des deux grandes règles de composition : majuscules à chaque nouveau mot de la composition (`ageDuCapitaine`) ou soulignés (`age_du_capitaine`). Dans le tableau suivant, nous choisissons la première.

Type	**Règles de nommage**	**Exemples**
Classes	La première lettre doit être une majuscule. Éviter les acronymes, choisir des noms simples et descriptifs.	`class Image ;` `class ClientPrivilégié ;`
Interfaces	Mêmes règles que pour les classes	`interface ImageAccesDirect ;` `interface Stockage ;`

Méthodes	Les noms de méthodes (fonctions) doivent refléter une action. Choisir de préférence des verbes. Première lettre en minuscules.	`afficher() ;`
		`getValue() ;`
		`setValue() ;`

Variables	Première lettre en minuscules. Noms relativement courts mais descriptifs (mnémoniques). Éviter les noms à une lettre, sauf pour compteurs internes et variables temporaires.	`int i ;`
		`float largeur ;`
		`String nomDuCapitaine ;`

Constantes	En majuscules. Le séparateur devient donc forcément un souligné.	`final static int VAL_MAX = 999 ;`

Annexe I

Corrigé des exercices

Exercice 1

Les erreurs sont indiquées par des commentaires ci-dessous :

```
boolean b;
double a2 = 5.2;
int k = 5;
int j = k;
int i = 7;
char a = i;     // On ne peut pas affecter un entier à un caractère
int k = 7;      // La variable k a déjà été déclarée
int l = t;      // On utilise une variable t non (encore) déclarée
int t = 7;
```

Exercice 2

```
int i = 13;                // i = 13
int j = 5;                 // j = 5
i = i % 5;                 // i = 3
j = j * i + 2;             // j = 17
int k = (i * 7 + 3) / 5;   // k = 4
```

Exercice 3

```
public class Point2D {
  double x;
  double y;
}
```

```
public class Point3D {
  double x;
  double y;
  double z;
}
```

```
public class VilleFrance {
  String nom;
  int codePostal;
}
```

```
public class VilleMonde {
  String nom;
```

```
  int codePostal;
  String pays;
}
```

NB : dans le dernier cas, on pourrait aussi définir une classe `Pays` et écrire :

```
public class VilleMonde {
  String nom;
  int codePostal;
  Pays pays;
}
```

Exercice 4

La première fonction peut s'écrire de différentes manières ; la plus condensée est probablement la suivante :

```
  public static int max(int n, int m) {
      return (n > m) ? n : m;
  }
```

Pour écrire la deuxième fonction, on a d'abord besoin de définir une fonction auxiliaire qui calcule x^y, x et y étant deux entiers :

```
  public static int puissance(int x, int y) {
      int res = 1;
      int p = 0;
      while (p < y) {
          p++;
          res *= x;
      }
      return res;
  }
```

Ensuite la fonction f s'écrit tout simplement :

```
  public static int f(int n) {
      return puissance(2, n) + puissance(3, n);
  }
```

Pour la troisième fonction, on a recours aux fonctions définies dans la classe `Math` de la bibliothèque standard de Java, et à une conversion d'un réel vers un entier :

```
  public static int se(double x) {
      // Il faudrait provoquer une exception si x est négatif
      return (int) Math.floor(Math.sqrt(x));
  }
```

Bien entendu, toutes ces fonctions doivent être définies dans une classe pour qu'elles puissent être compilées et utilisées...

Exercice 5

```
  public static int max(int[] tab) {
      int res = 0;           // résultat provisoire
      if (tab.length > 0) {  // Au moins un élément
          res = tab[0];      // On initialise par la valeur du premier élément
```

```
        }
        // A partir du deuxième et jusqu'à la fin, chercher un plus grand que
        // le résultat provisoire et remplacer celui-ci, le cas échéant
        for (int i = 1 ; i < tab.length ; i++) {
            if (tab[i] > res) {
                res = tab[i];
            }
        }
        return res;
    }
```

```
    public static int somme(int[] tab) {
        int s = 0;
        for (int i = 0 ; i < tab.length ; i++) {
            s += tab[i];
        }
        return s;
    }
```

```
    public static int estALIndice(int[] tab, int n) {
        int indice = -1;
        for (int i = 0 ; i < tab.length ; i++) {
            if (tab[i] == n) {
                indice = i;
                break; // pas besoin de poursuivre le parcours du tableau
            }
        }
        return indice;
    }
```

Exercice 6

```
public class Item {
    private String nom;
    private int quantité;
    private double prixUnitaire;

    Item(String n, int q, double pu) {
        nom = n;
        quantité = q;
        prixUnitaire = pu;
    }

    public double prix() {
        return quantité * prixUnitaire;
    }
}
```

Exercice 7

```
public class Rectangle {
    private Point coinSupGauche;
    private Point coinInfDroit;

    Rectangle(Point p1, Point p2) {
        coinSupGauche = new Point(Math.min(p1.getX(), p2.getX()),
                                  Math.min(p1.getY(), p2.getY()));
        coinInfDroit = new Point(Math.max(p1.getX(), p2.getX()),
                                 Math.max(p1.getY(), p2.getY()));
```

```
    }

    public double aire() {
        return (coinInfDroit.getX() - coinSupGauche.getX()) *
            (coinInfDroit.getY() - coinSupGauche.getY());
    }

    public boolean plusPetitQue(Rectangle r) {
        return aire() < r.aire();
    }
}
```

Exercice 8

```
public static int fib(int n) {
    // Stricto sensu il faudrait provoquer une erreur pour n négatif
    if (n <= 1) {
        return 1;
    }
    else {
        return fib(n-1) + fib(n-2);
    }
}
```

Bibliographie

[1] Alfred Aho and Jeffrey Ullman. *Concepts fondamentaux de l'informatique*. Dunod, 1993. 856 pages. Traduit de l'anglais, ce livre a pour origine un cours d'introduction à l'informatique de Stanford. Bien que ne faisant aucune référence au langage Java, c'est probablement un des meilleurs ouvrages pour maîtriser les bases de l'informatique, et nous vous le recommandons fortement.

[2] Alfred Aho, John Hopcroft, and Jeffrey Ullman. *Data Structures and Algorithms*. Addison-Wesley, Bien que déjà assez ancien, dans un domaine où les publications sont nombreuses, ce livre reste un très grand classique. Une traduction française a été publiée en 1987 par InterÉditions. 1983.

[3] Ravi Sehti. *Programming Languages : Concepts and Structures*. Addison-Wesley, 1989. 478 pages. Certaines parties introductives de notre cours sont inspirées de cette introduction très claire et pédagogique aux modèles sous-jacents des langages de programmation. La 2^{e} édition de ce livre est disponible à la bibliothèque de l'École.

[4] Mary Campione and Kathy Walrath. *The Java Tutorial, Second Edition*. The Java Series. Addison-Wesley, 1998. 964 pages. Excellent tutoriel pour apprendre Java, également disponible sur Internet.

[5] Mary Campione, Alison Huml, and the Tutorial Team. *The Java Tutorial Continued : The Rest of the JDK*. The Java Series. Addison-Wesley, 1998. 896 pages. Complément de [4] pour les notions plus avancées.

[6] Kathy Walrath and Mary Campione. *The JFC Swing Tutorial*. The Java Series. Addison-Wesley, 1999. 953 pages. Autre complément de [4], dédié à la programmation événementielle avec Swing.

[7] Gérald Masini, Amedeo Napoli, Dominique Colnet, Daniel Léonard, and Karl Tombre. *Les langages à objets*. InterÉditions, 1989. 584 pages. Si je cite ce livre, ce n'est pas pour vous le recommander, car il n'est plus assez d'actualité. À l'époque de sa rédaction, les langages à objets n'étaient qu'émergents, Java était encore inconnu et on commençait juste à parler de C++. Mais étant l'un de ses auteurs, je me suis inspiré de ses deux premiers chapitres dans la rédaction des parties du polycopié qui concernent la programmation objet, car je crois que nous avions fait un bon travail de synthèse et d'explication dans ces chapitres. Plusieurs définitions du glossaire sont également tirées de ce livre.

[8] Chris J. Date. *Introduction aux bases de données*. Vuibert, 2001. 926 pages. Un classique, traduit de l'anglais.

[9] George Reese. *JDBC et Java – guide du programmeur*. Éditions O'Reilly, 1998. 203 pages. Traduit de l'anglais.