Objets
Algorithmes
Patterns

René Lalement

octobre 2000

Table des matières

Avant-propos

> *Dans la littérature de cet hémisphère (...) abondent les objets idéaux, convoqués et dissous en un moment, suivant les besoins poétiques. Ils sont quelquefois déterminés par la pure simultanéité. Il y a des objets composés de deux termes, l'un de caractère visuel et l'autre auditif : la couleur de l'aurore et le cri lontain d'un oiseau. Il y en a composé de nombreux termes : le soleil et l'eau contre la poitrine du nageur, le rose vague et frémissant que l'on voit les yeux fermés, la sensation de quelqu'un se laissant emporter par un fleuve et aussi par le rêve. Ces objets au second degré peuvent se combiner à d'autres ; le processus, au moyen de certaines abréviations, est pratiquement infini. Il y a des poèmes fameux composés d'un seul mot énorme. Ce mot intègre un* objet poétique *créé par l'auteur.*
>
> Jorge Luis BORGES

La définition rigoureuse de ce qu'est un *algorithme* est du ressort de l'informatique théorique, laquelle fait appel à des notions de logique mathématique ; ce n'est pas l'objet de ce cours. N'importe quel dictionnaire en donne une définition intuitive, et nous en verrons suffisamment d'exemples pour affiner cette intuition. Disons simplement que l'algorithme est la forme que revêtent les solutions que l'informatique sait donner aux problèmes qui lui sont posés. Certains algorithmes sont *implémentés* directement par des circuits électroniques numériques : c'est le cas de ceux qui réalisent les opérations arithmétiques, et d'algorithmes beaucoup plus élaborés comme la transformée de Fourier rapide. C'est aussi le rôle de certains circuits que l'on trouve sur des objets courants (carte à puce, appareil photo, téléphone portable) dont l'usage est bien délimité et non modifiable.

Un ordinateur ne diffère guère de ces objets courants, puisqu'il contient des circuits implémentant un (unique) algorithme dont l'usage est tout aussi délimité et non modifiable : « exécuter des programmes ». La description de cet algorithme étant le sujet d'un cours d'architecture des machines, et non de ce cours, nous ne pouvons ici qu'énoncer son caractère universel : l'algorithme implémenté par les circuits d'un ordinateur est capable de *simuler* n'importe quel autre algorithme. Les instructions nécessaires à cette simulation forment un *programme*, qu'il suffit d'exécuter.

Du modèle d'architecture conçu, dans les années 44-46, par von NEUMANN, WILKES, GOLDSTINE et BURKS, retenons qu'instructions et données sont représentées d'une même façon, par une suite d'éléments binaires. Les premiers ordinateurs construits suivant ce modèle furent programmés directement en binaire. On utilisa peu après des *langages d'assemblage*, qui forment une notation textuelle des instructions. Depuis la fin des années 50, on utilise des *langages de programmation*. Au moyen de ces langages, un programme est écrit sous la forme d'un texte, c'est-à-dire d'une suite de caractères, humainement lisible, appelé programme-*source*, qui n'est pas directement exécutable. Une des techniques employées pour faire exécuter ce programme est de le traduire en instructions de la machine, au moyen d'un *compilateur* ; on obtient ainsi le programme-*objet*. Celui-ci peut alors être soumis à la machine pour exécution.

De nombreux langages de programmation ont été utilisés ; parmi les plus connus figurent Fortran, Lisp, Prolog, Smalltalk, Cobol, Pascal, C, C++, Java, Caml. Le langage C a été conçu et implémenté par Dennis M. RITCHIE. Le Turing Award 1983 lui fut décerné, conjointement avec Ken L. THOMSON pour le développement et l'implémentation du système d'exploitation Unix. Sa définition, désormais qualifiée « Kernighan-Ritchie », a été clarifiée et modernisée au cours de sa normalisation ANSI, publiée en 1988, et qui a adopté certaines évolutions consacrées par l'usage. Certaines de ces évolutions sont apparues à l'occasion du développement du langage C++, dû à Bjarne STROUSTRUP, et conçu comme une extension de C, par l'addition des *classes*. Il s'agissait, dans les années 1983–85, de donner au programmeur la possibilité d'écrire en C des programmes dans le style « orienté objets », imaginé dès la fin des années 70, à la fois pour développer des applications de simulation (le langage Simula) et les premières interfaces graphiques (Smalltalk, puis Objective C). C++ s'est depuis considérablement enrichi et stabilisé sous la forme d'une norme ISO, adoptée en 1997 et ratifiée en août 1998, pour devenir l'un des langages majeurs de l'informatique contemporaine.

Mais les enjeux s'étaient déplacés entre-temps. Avec l'émergence des micro-ordinateurs, les années 80 ont consacré la fin des ordinateurs centraux, les « mainframes » ; avec l'émergence de l'Internet, les années 90 ont consacré la fin de l'ordinateur personnel ; avec l'émergence des systèmes embarqués, les années à venir consacreront la fin de l'ordinateur. Il faut comprendre que toujours plus d'ordinateurs centraux seront utilisés, plus de micro-ordinateurs, plus d'ordinateurs en général, mais que l'informatique aura cessé de s'identifier à l'ordinateur. Cette évolution se caractérise par la présence de techniques informatiques dans des champs d'activité de plus en plus larges, non comme outils, mais comme composants de systèmes et souvent comme composants critiques. Cette présence apparaît aussi bien dans les grands systèmes (industrie, transport, commerce, etc) que dans les objets de la vie courante (téléphone, montre, automobile, électroménager, etc). Les critères de sécurité passent au premier plan des préoccupations, le bug de l'an 2000 en étant une simple illustration.

L'informatique (ou science de l'ordinateur, *computer science*, ou science du calcul, *computation science*) s'est alors transformée en «sciences et techniques de l'information et de la communication», dont les mots-clés sont :

- *Convergence* : la représentation numérique devient l'unique forme de l'information vers laquelle convergent l'ordinateur, les télécommunications et l'audiovisuel ;
- *Multimédia* : la convergence autorise le traitement d'images, fixes ou animées, de sons, et de percepts mécaniques, non seulement comme information, mais comme moyen de communication, avec la réalité virtuelle ;
- *Communicabilité* : tout système informatique doit communiquer, non seulement avec l'utilisateur, via des interfaces homme-machine multimédia, mais aussi avec d'autres systèmes informatiques, via de multiples réseaux ;
- *Mobilité* : les organisations traditionnelles (ordinateur individuel, modèle client-serveur) sont remises en cause pour assurer la mobilité des données, des processus et des utilisateurs, dans le cadre de réseaux à configuration dynamique, à l'exemple de la téléphonie mobile ; c'est l'informatique nomade ;
- *Temps réel* : les composants informatiques doivent assurer des services en temps réel, notamment dans la commande des processus ;
- *Sécurité* : le caractère critique des composants informatiques et la généralisation du support et du traitement numérique de l'information imposent des critères de sécurité stricts ;
- *Mondialisation* : double processus d'interconnexion des réseaux (Internet), et de reconnaissance de standards de fait ; les services offerts doivent être indépendants de l'infrastructure (nature du réseau, de l'architecture matérielle et logicielle) ; le nombre d'utilisateurs et le volume de l'information croissent de façon spectaculaire.

Le développement du langage Java™, et de toute la technologie qui l'entoure, s'inscrit exactement dans cette évolution. Car Java ne peut pas simplement être vu comme un nouveau langage, mais plutôt comme le cœur d'une nouvelle technologie, qui répond à de nouveaux enjeux et qui se déploie dans de nombreuses applications. Java y intervient à la fois de façon matérielle (cartes à puce, systèmes embarqués) et de façon logicielle sous forme d'API spécialisées (*Application Programming Interface*), par exemple pour l'accès aux bases de données, la programmation réseau ou pour le graphique. L'utilisation de ces API fait que de nombreuses applications traditionnellement difficiles à programmer deviennent plus abordables.

La mise en œuvre des algorithmes et des structures de données, ainsi que la construction de composants logiciels supposent des techniques de modularité et de réutilisation qui conduisent à privilégier la programmation à objets, l'un des trois principaux styles de programmation, avec le style impératif et le style applicatif. Tout programme Java est une mixture de traits applicatifs, impératifs et objets. Les programmes sont organisés dans le style objet, lequel recourt à des expressions (style applicatif) et à des structures de contrôle (style impératif).

Le *style applicatif* est entièrement fondé sur l'évaluation d'expressions satisfaisant la propriété : *la valeur d'une expression ne dépend que des valeurs des sous-expressions et de l'opération qui les combine*. Ce style conduit souvent à des programmes concis et faciles à comprendre, et que certains langages, comme Caml, savent exécuter efficacement. Pour écrire des algorithmes quelconques dans le style applicatif, il faut recourir à la récursivité, c'est-à-dire à la capacité d'une fonction de s'appeler elle-même. Les définitions récursives sont souvent très naturelles et résultent de techniques générales de résolution de problèmes. Par contre, les programmes ainsi construits ne peuvent pas toujours être utilisés si l'on doit respecter des exigences d'efficacité.

Le *style impératif* représente un algorithme à l'aide de deux types de *structures* : les structures de contrôle (appel de fonction, branchements, itérations) et les structures de données, le contrôle permettant d'assembler des instructions qui opèrent sur des données, en transformant l'état de la mémoire. L'instruction typique de ce style est l'*affectation* , et les structures de contrôle typiques sont des itérations (les « boucles » `for` et `while`). Désormais, la valeur d'une expression dépend non seulement de l'opération et des valeurs des sous-expressions, mais aussi de l'état courant de la mémoire. La notion de valeur cède le pas à celle d'objet, la notion d'expression à celle d'instruction. Des objets sont construits, des objets sont modifiés, des objets sont détruits. C'est dans ce style qu'on écrira les programmes les plus efficaces, que les compilateurs sauront le mieux optimiser. Ses structures, qui sont souvent plus difficiles à maîtriser que celles du style applicatif, conduisent à des programmes plus difficiles à prouver, mais qui forment la majeure partie de ce que produisent les programmeurs. Fortran, Pascal, et C ont été conçus pour la programmation impérative.

L'approche orientée objet de la programmation tente de représenter un algorithme comme une communauté d'organismes vivants, chacun étant créé, disposant de ses propres ressources, ayant éventuellement son propre comportement et interagissant avec les autres membres de la communauté. Ceci reste de l'ordre de la métaphore, mais conduit en pratique à incorporer les instructions dans la définition même des objets, de sorte que contrôle et données cessent d'être découplés comme c'était le cas de l'approche classique. Comme en programmation impérative, des objets sont créés, modifiés, détruits ; mais ce sont maintenant les interactions entre objets qui constituent la trame des programmes écrits dans ce style. C++ et, plus récemment, Java et Objective CAML, sont des langages représentatifs de ce style de programmation.

Le *style objet* s'est développé avec le triple objectif d'améliorer la capacité des programmes à modéliser les objets physiques, à organiser le logiciel et à gérer les ressources mises en œuvre. En premier lieu, le génie logiciel, pareil à toute autre activité d'ingénierie, a conduit à la notion de « composant » logiciel, forme de *module*. Une fois produit, un module peut être réutilisé : il offre un certain nombre de services, et en utilise d'autres, ce qui constitue l'*interface* du module. D'autre part, la programmation a pour objectif de construire des modèles de processus calculatoires, les algorithmes, laissant à d'autres disciplines le soin de construire des modèles des processus du monde physique. Le développement de la programmation objet est dû au souhait que la programmation elle-même puisse contribuer à cette modélisation. Enfin, la mise en œuvre d'un algorithme sur une machine suppose que certaines ressources matérielles sont disponibles : comment ces ressources sont acquises, désignées, utilisées et rendues. La plus importante de ces ressources est la mémoire. La gestion de ces ressources est partagée entre le programmeur et le système. Les langages à objets cherchent à rendre la gestion des ressources la plus uniforme possible, à travers la notion d'objet.

Certaines caractéristiques sémantiques des langages à objets (comme la modularité, l'encapsulation, l'héritage, le sous-typage, la liaison tardive) sont liées à ce style de programmation. L'utilisation judicieuse de ces techniques, requise pour le développement de systèmes logiciels complexes, reste cependant assez difficile – et une affaire de goût.

Ce document constitue le support des deux cours *Objets et patterns* (chapitres 1 et 3) et *Algorithmes* (chapitre 2). Le premier cours est consacré à la programmation à objets en Java. Les informaticiens, quand ils se font linguistes, distinguent la syntaxe, la sémantique, et la pragmatique d'un langage de programmation.

La *syntaxe* regroupe les règles lexicales et grammaticales de formation des programmes, qui permettent de décider si un texte est un « programme syntaxiquement correct », ou plus simplement, est un « programme », ou bien s'il comporte des « erreurs de syntaxe ». Tout débutant doit se sentir rebuté par cette forme d'écriture et sa rigidité qui ne pardonne pas la moindre faute de frappe ; la connaissance des règles syntaxiques facilitera surtout le travail du compilateur. L'idéal serait d'appliquer ces règles sans jamais devoir les apprendre, en utilisant un éditeur de texte bien configuré (Emacs, ou celui d'un environnement de programmation) en lisant beaucoup de programmes bien écrits et en procédant par imitation. Disons le clairement : la syntaxe de Java n'est pas un objectif de ce cours. On trouvera cependant en annexe la grammaire complète de Java, à titre de référence.

La *sémantique* explicite la signification des programmes : quelle est la valeur d'une expression, quel est l'effet d'une instruction. C'est ici que résident les concepts réellement importants de la programmation, qui s'appliquent également à d'autres langages : évaluation, portée des déclarations, appel de fonction, modes d'allocation, etc. Les objectifs de sûreté et d'efficacité des programmes nécessitent une connaissance précise de la sémantique : celle-ci sera donc traitée dans ce cours, de façon très informelle mais assez complète, et constitue l'essentiel de la partie *Objets*.

Formes (syntaxiques) et concepts (sémantiques) ne suffisent pas pour apprendre à programmer. La *pragmatique* désigne la mise en pratique de ces formes et concepts : quelle construction utiliser dans tel contexte, comment l'utiliser, etc. La pragmatique est aux langages de programmation ce que la rhétorique fut au langage parlé pendant des siècles : un ensemble de règles de l'art, de recettes, d'exemples et d'usages que reconnaissent les programmeurs. L'ambition du cours *Objets et patterns* est de faire partager des éléments, nécessairement disparates, de cette pragmatique de Java, en recourant à la notion de *pattern*.

Il faut encore avoir quelque chose à dire, pour écrire des programmes qui ont du sens. Les programmes de ce cours satisfont un unique objectif, qui est de résoudre des problèmes, au moyen d'*algorithmes*. Ceux-ci sont conçus, parfois par des informaticiens, et souvent par des spécialistes de la discipline dont émane le problème posé, qui maîtrisent des techniques de résolution spécifiques (par exemple en mathématiques appliquées). Le lecteur est renvoyé à d'autres cours (probabilités, calcul scientifique, recherche opérationnelle, etc) pour leur élaboration. À l'exception de quelques résultats élémentaires, les algorithmes ne feront l'objet d'aucune étude (preuve de propriétés, analyse de complexité) : le troisième chapitre, *Algorithmes*, n'aborde donc pas réellement l'*algorithmique*. Il présente plutôt quelques structures de données utiles, les principales formes d'algorithmes ainsi que quelques algorithmes résolvant des problèmes classiques. Les chapitres 2 et 3 sont très interdépendants, comme le lecteur le constatera en suivant les nombreux renvois mutuels.

René LALEMENT
Champs-sur-Marne, 31 août 2000

Mes remerciements à Gilbert CAPLAIN, *Jean-Philippe* CHANCELIER, *Olivier* CARTON, *Étienne* DURIS, *Hervé* GRALL, *Mathieu* JAUME, *Renaud* KERIVEN, *Bernard* LAPEYRE *et Thierry* SALSET *pour leur relecture de ce texte et leurs nombreuses corrections et suggestions. L'installation des logiciels et de l'environnement de travail a été effectuée par* TU *Dien Au, Jean-Louis* BOUDOULEC, *Jean* COUEDIC *et Thierry* SALSET. *La traduction de ce texte, de LaTeX en HTML, et son installation sur le Web ont été réalisées par Jean-Philippe* CHANCELIER.

1 Objets

1.1 Un exemple Voici un algorithme qui permet de calculer une approximation de π :

> « Aller dans un *pub*, s'assurer qu'il contient bien un jeu de fléchettes et que la cible est un disque inscrit dans un carré ; prier l'un des consommateurs de lancer des fléchettes n'importe où dans le carré ; compter le nombre de fléchettes qui sont plantées dans le disque ; faire le quotient de ce nombre par le nombre total de fléchettes plantées dans le carré ; multiplier par 4 ce quotient ; retourner ce produit. »

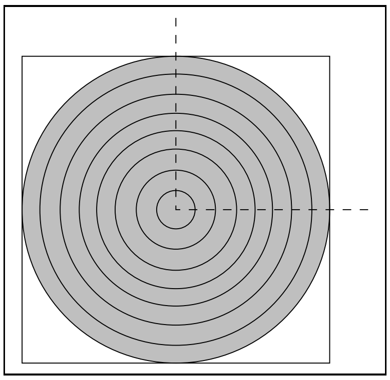

FIG. 1.1 – *Calculer π avec des fléchettes.*

Si vous ne disposez pas des ressources naturelles nécessaires à l'implémentation de cet algorithme, vous pouvez recourir à un ordinateur et le programmer comme suit ; vous devrez simuler le lancer de fléchettes par un tirage de nombres aléatoires, et représenter le quadrant supérieur droit du disque de la cible par une fonction $v = f(u) = \sqrt{1-u^2}$, avec $0 \leq u \leq 1$:

```
class Cible {
  private int dedans;

  private static double f(double x) {
    // x doit être compris entre -1 et 1
    return Math.sqrt(1 - x*x);
  }

  Cible() {
    dedans = 0;
  }

  int valDedans() {
    return dedans;
  }

  void lancer() {
    // tirage d'un point dans le carré [0,1[ X [0,1[
    double u = Math.random();
    double v = Math.random();
    if (v <= f(u)) dedans++;
  }
}

class Simulation {
  private static double calculAire(int n) {
    // évalue l'aire de la cible par une méthode de Monte-Carlo
    Cible cible = new Cible();
    for (int i=0; i<n; i++) cible.lancer();
    return (double)cible.valDedans()/n;
  }

  public static void main(String[] args) {
    int tirages = Integer.parseInt(args[0]);
    double pi = 4 * calculAire(tirages);
    System.out.println("Pi = " + pi);
  }
}
```

Il s'agit de la définition de deux classes, `Cible` et `Simulation`. La première définit un modèle d'objets, la seconde un modèle de calcul. La classe `Cible` est destinée à être *instanciée*, c'est-à-dire à produire des objets de type `Cible`. Une fois ces objets créés, les deux opérations que l'on peut faire sur ceux-ci sont : `lancer()` et `valDedans()`. Ces deux opérations utilisent `dedans`, un champ de la classe `Cible` qui indique le nombre de fléchettes dans la cible et une fonction `f()` qui détermine le contour d'une cible ; ce champ et cette fonction sont des membres de la classe `Cible`. La classe `Simulation` n'est pas destinée à être instanciée. Elle rassemble deux fonctions, `calculAire()` et `main()`. C'est cette dernière, dite fonction *principale*, par laquelle l'exécution du programme commence. Plaçons ces deux définitions de classe dans un fichier de nom `Simulation.java`. Puis compilons ce fichier par la commande suivante sous Linux :

```
linux% javac Simulation.java
```

L'exécution de cette commande produit deux autres fichiers, `Cible.class` et `Simulation.class`, qui contiennent des instructions exécutables par la *Machine Virtuelle Java*. La classe `Cible` n'est pas exécutable, parce qu'elle ne contient pas de fonction principale. La classe `Simulation` peut être exécutée, parce qu'elle en contient une :

```
linux% java Simulation 1000000
```

La commande `java` démarre une Machine Virtuelle Java, lui fournit la classe principale `Simulation` et l'argument `1000000`. La Machine Virtuelle Java *charge* la classe `Simulation` et les classes que celle-ci utilise, notamment la classe `Cible`; elle exécute ensuite les instructions définies par la fonction principale de `Simulation`. Tout programme doit contenir la définition d'une fonction principale qui doit s'appeler `main`; l'exécution d'un programme commence par l'exécution de la fonction `main()`, qui invoquera éventuellement d'autres fonctions.

```
  public static void main(String[] args) {
    int tirages = Integer.parseInt(args[0]);
    double pi = 4 * calculAire(tirages);
    System.out.println("Pi = " + pi);
  }
```

Cette définition de fonction a un *en-tête*,

```
  public static void main(String[] args)
```

et un *corps*, qui est placé entre l'accolade ouvrante `{` et l'accolade fermante `}`. L'en-tête exprime que la fonction `main()` a un paramètre de nom `args` et de type `String[]` (c'est le `(String[] args)` à la suite de `main`) et ne retourne rien (c'est le type `void`); on dit d'une telle fonction, qui ne retourne rien, qu'il s'agit d'une *procédure*. Son corps définit les deux *variables locales* `tirages` et `pi`, l'une de type `int` (un entier, l'un des types primitifs), l'autre de type `double` (un nombre flottant en double précision, un autre type primitif). À la première est affectée la valeur de l'expression `Integer.parseInt(args[0])`, à la seconde la valeur de l'expression `4 * calculAire(tirages)`. La valeur de la première expression est l'entier 1 000 000 fourni comme argument à la ligne de commande.

L'expression `calculAire(tirages)` étant une *invocation* de la fonction `calculAire()`, celle-ci est maintenant exécutée.

```
private static double calculAire(int n) {
  // évalue l'aire sous f par une méthode de Monte-Carlo
  Cible cible = new Cible();
  for (int i=0; i<n; i++) cible.lancer();
  return (double)cible.valDedans()/n;
}
```

La fonction `calculAire()` a un paramètre `n`, de type `int`, et retourne un `double`. Son corps définit une variable locale `cible`, de type `Cible`, crée un objet de ce type, au moyen de `new Cible()`, et affecte à la variable `cible` une référence à cet objet. Ensuite, au moyen de l'itération

```
for (int i=0; i<n; i++)
```

la méthode `lancer()` de l'objet `cible` est invoquée n fois. Enfin, la méthode `valDedans()` de l'objet `cible` est invoquée, retourne un entier qui est converti en `double`, divisé par n, puis retournée comme valeur de l'expression `calculAire(tirages)` qui figurait dans la fonction principale. Revenons à cette fonction. La valeur retournée est multipliée par 4 et le résultat est affecté à la variable `pi`. La dernière ligne,

```
System.out.println("Pi = " + pi);
```

a pour effet d'écrire à l'écran la suite de caractères `Pi = `, suivie de la valeur de la variable `pi`, en notation décimale :

```
Pi = 3.14138
```

L'exécution est alors terminée. On obtient `3.14138` comme estimation de π, soit seulement trois décimales exactes, après un million de lancers (c'est le `1000000` spécifié sur la ligne de commande). Il y a heureusement des méthodes de calcul de π qui sont bien meilleures[1]. D'autres exécutions de ce programme peuvent conduire à des résultats différents, à cause du `random`.

Il reste à revoir la classe `Cible` qui, rappelons-le, définit un modèle d'objets (qui sont ses instances). Même si, dans ce programme, la classe `Simulation` ne crée qu'un seul objet de type `Cible`, on peut imaginer d'autres situations où plusieurs cibles seraient nécessaires. Chaque cible est dotée d'un champ `dedans`, dont la valeur sera le nombre de fléchettes dans la cible (la cible ignore celles qui frappent le mur). Ce champ, de type `int`,

1. Cette méthode d'intégration, dite de Monte-Carlo, qui est un exemple d'algorithme probabiliste numérique, devient réellement intéressante pour calculer des intégrales multiples en grandes dimensions

est déclaré `private`, ce qui signifie qu'il n'est pas accessible de l'extérieur de la classe. On veut cependant pouvoir lire la valeur de ce champ, c'est ce que permet la méthode `valDedans()`. La différence entre une fonction et une méthode est visible dans leurs en-têtes : une fonction est déclarée `static`, une méthode ne l'est pas. Une méthode est destinée à être appliquée à une instance de la classe, tandis qu'une fonction, qui ne dépend d'aucune instance, doit être invoquée via la classe où elle est définie. C'est pourquoi les fonctions `sqrt()` (racine carrée), `random()` (génératrice de nombres pseudo-aléatoires) et `parseInt()` (conversion d'une chaîne de caractères en entier) sont invoquées sous la forme `Math.sqrt(x)`, `Math.random()`, et `Integer.parseInt(s)`, les deux premières étant définies dans la classe `Math`, la troisième dans `Integer`. Ainsi, la méthode `lancer()` n'a de sens que relativement à une cible particulière, et est invoquée sous la forme `cible.lancer()`, mais toutes les cibles partagent une même fonction `f()` qui définit un même contour. La fonction `f()` est également déclarée `private`, car on ne souhaite pas qu'une cible soit utilisée simplement pour obtenir les valeurs de cette fonction : on interdit ainsi à une autre classe de faire appel à elle. Inversement, la fonction `main()` de la classe `Simulation` est déclarée `public`, car elle doit être invoquée de l'extérieur, par la Machine Virtuelle Java. Le corps de `f()` contient une seule *instruction*,

```
return Math.sqrt(1 - x*x);
```

qui contient l'*expression* « `Math.sqrt(1 - x*x)` », exprimant $\sqrt{1-x^2}$, laquelle est une invocation de la fonction `Math.sqrt()` ; l'argument de cet appel est l'expression « `1 - x*x` ». Cette dernière expression est formée à partir des opérateurs « `-` » et « `*` », de la constante littérale `1` et du nom `x`.

Toutes ces définitions (de classe, de champ, de fonction, de méthode, de variable locale) ont plusieurs rôles :

- d'introduire un nom ;
- de déclarer la façon de l'utiliser ;
- de définir, éventuellement, sa signification.

On ne peut pas utiliser un nom sans que celui-ci soit déclaré quelque part, et il faut encore que ce quelque part soit *accessible*. La déclaration indique les conditions d'accessibilité, où (le nom est-il `private`, `public` ?) et comment (de quel type est-il ? si c'est une fonction ou une méthode, quels sont les types de ses arguments, de sa valeur de retour ?). Enfin, le corps d'une fonction ou d'une méthode, ou la valeur qui est affectée à une variable permettent de définir la signification attachée au nom. On notera que dans une déclaration de variable locale, comme dans une liste de paramètres, le type précède le nom, à la façon des adjectifs en anglais : `Cible cible`, `int i`, `double x`. Ces déclarations servent à contraindre l'usage qui peut être fait d'un nom : il est incorrect de former l'expression `2 * cible`, car aucune opération de multiplication n'est définie entre un entier et un objet de type `Cible`, ou de former l'expression `i.lancer()`, car la méthode `lancer()` n'est pas définie sur le type `int`. Tout ce travail de vérification est fait par le compilateur, qui détecte ainsi les erreurs les plus grossières (qui sont souvent dues à une faute de frappe).

Le texte compris entre « `//` » et la fin de la ligne est un commentaire : inutilisables par la Machine Virtuelle Java, les commentaires facilitent la compréhension des programmes.

1.2 Les types En Java, l'importance des types provient de ce que les programmes sont eux-mêmes des collections de types. Ce n'est pas le cas de tous les langages, qui utilisent pourtant tous, une ou plusieurs notions de type.

Typer les données De façon informelle, les types permettent de classifier les données manipulées par les programmes : caractères, entiers, flottants, tableaux, etc. Cette classification a au moins un intérêt, qui est de pouvoir interpréter et utiliser correctement ces données, alors qu'elles sont représentées en mémoire, de façon uniforme, par des suites de 0 et de 1. Par exemple, la même suite de 0 et de 1 doit être interprétée différemment selon qu'il s'agit d'un nombre entier ou d'un nombre flottant. De plus, ce ne sont pas les mêmes opérations qui seront exécutées sur des entiers ou sur des flottants : ce sont des « unités fonctionnelles » différentes du microprocesseur qui exécuteront une addition sur des entiers et une addition sur des flottants. Cette notion de type est commune à tous les langages : c'est le typage des données. Les données, c'est ce qui occupe la mémoire ; c'est ce qui est modifié, selon le point de vue impératif. En Java, leur typage les répartit en trois grandes catégories : les données primitives, les tableaux et les instances de classe.

Typer les expressions Des langages comme Pascal, C, Ada, C++ et Java sont dits *fortement typés*. Tout nom de variable doit être déclaré avec son type, tout nom de fonction ou de méthode doit être déclaré avec le type de ses arguments et le type de son résultat. Ceci permet d'une part de déterminer la quantité de mémoire nécessaire pour représenter un objet (par exemple, un entier est représenté sur 4 octets, un flottant en double précision sur 8 octets). D'autre part, ces déclarations permettent de vérifier que les expressions sont correctement typées, et que par exemple, on n'invoque pas une fonction avec un argument numérique alors qu'elle a été déclarée avec un paramètre de type tableau. Pour certains langages, comme CAML, le type des expressions peut même être déterminé sans qu'il soit nécessaire de déclarer le type des noms ; pour la plupart des autres, comme Java, la déclaration préalable des noms est essentielle. Ces vérifications sont faites par le compilateur, donc avant l'exécution (on dit qu'elles sont statiques) : c'est le typage des expressions. En Java, les types des expressions sont répartis en quatre grandes catégories : les types primitifs, les types de tableaux, les classes et les interfaces.

Sûreté et sous-typage Ces deux notions de typage — des données et des expressions — ne sont pas indépendantes. En refusant d'emblée des expressions mal typées, elles permettent d'éviter certaines erreurs qui se produiraient à l'exécution. En effet, si le compilateur détermine qu'une expression a un certain type, et si cette expression est ensuite évaluée, la *sûreté* du typage est une propriété du langage qui assure que :

– aucune erreur due à une opération incorrecte en raison du type des données ne peut se produire à l'exécution ;
– le résultat de l'évaluation de l'expression est une valeur d'un type compatible avec le type de l'expression.

Cette dernière notion, de compatibilité des types, mérite d'être approfondie. Considérons par exemple les nombres entiers et les nombres flottants. Un mathématicien, familier de la théorie des ensembles, sachant que l'ensemble des entiers est inclus dans l'ensemble des réels, à qui l'on demande de citer un nombre réel, n'hésitera pas à répondre 12 ; l'informaticien lui fera remarquer que 12 est un entier et que 12.0 est un flottant, que ces deux constantes appartiennent à des types distincts (en Java, `int` et `double`), mais acceptera cependant 12 comme un flottant. Inversement, ni le mathématicien ni l'informaticien n'accepterait 12.3 comme un entier. On caractérise la relation entre ces types entiers et flottants en disant que `int` est un sous-type de `double`, mais `double` n'est pas un sous-type de `int`. Cette relation, dite de *sous-typage*, déjà commode pour les différents types numériques, s'avère encore plus importante pour les types d'objet : pour dire, par exemple, que les rectangles sont des formes géométriques, on fera du type `Rectangle` un sous-type du type `Forme`.

Types primitifs Les *données primitives* sont celles de tout langage de programmation : entiers, nombres flottants, caractères. Ce sont les données dont le type est l'un des types primitifs de Java :

`void` : aucune donnée n'est de ce type
`boolean` : deux données, `true` et `false`

`char` : les caractères 16 bits Unicode
`byte` : les entiers 8 bits signés, en complément à 2
`short` : les entiers 16 bits signés, en complément à 2
`int` : les entiers 32 bits signés, en complément à 2
`long` : les entiers 64 bits signés, en complément à 2
`float` : les flottants 32 bits, selon la norme IEEE 754-1985
`double` : les flottants 64 bits, selon la norme IEEE 754-1985

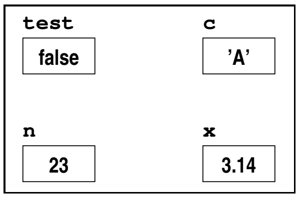

FIG. 1.2 – *Variables initialisées par des valeurs primitives.*

Ces données primitives sont des *valeurs*, qui peuvent être affectées à une variable, passées en argument à une méthode ou retournées comme résultat d'une méthode. Elles n'ont pas besoin d'être créées et peuvent être représentées par une valeur littérale, par exemple pour donner une valeur initiale à une variable (figure 1.2) :

```
boolean test = false;
char c = 'A';
int n = 23;
double x = 3.14;
```

Il existe des relations de sous-typage entre certains des types primitifs : chacun des types `byte`, `short`, `int`, `long`, `float`, `double` est un sous-type des suivants et `char` est un sous-type de `int`. Ces relations permettent de convertir une valeur d'un type en une valeur d'un sur-type : une telle conversion peut se faire sans perte d'information dans certains cas (de `byte` en `short`, de `short` ou de `char` en `int`, de `int` en `long`, de `float` en `double`) ou avec perte d'information dans les autres cas, l'ordre de grandeur étant cependant préservé (par exemple, de `int` en `float`). Notons que le type `boolean` n'est dans une relation de sous-typage avec aucun autre type. Quand les opérandes d'une opération ne sont pas de même type (par exemple un `int` et un `float`), il y a conversion implicite de l'un d'eux dans le type de l'autre ; par exemple, `int + float` est converti en `float + float`. Cette conversion implicite se fait toujours de sous-type vers sur-type : de `int` vers `float`, de `float` vers `double`.

Tableaux Un *tableau* est un objet rassemblant un nombre donné de variables déclarées d'un même type et installées consécutivement en mémoire ; ces variables sont appelées les éléments du tableau. À tout type correspond un type de tableaux obtenu en suffixant `[]` à son nom : par exemple, les types `int[]` des tableaux d'entiers, `String[]` des tableaux de chaînes de caractères, `int[][]` des tableaux de tableaux d'entiers. Les tableaux ne sont pas des valeurs, mais des objets qui n'existent dans la mémoire que s'ils sont créés. Un tableau est créé à l'aide de l'opérateur `new[]`, appliqué à un nom de type et à une expression entière, indiquant le nombre de ses éléments, celle-ci placée entre les crochets « `[` » et « `]` ». Par exemple, l'expression `new int[4]` crée en mémoire un tableau de 4 entiers, tous initialisés à zéro (figure 1.3).

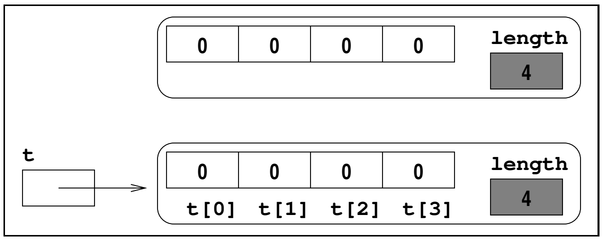

FIG. 1.3 – *Création d'un tableau de 4* `int`*, affectation à une variable* `t`*.*

La valeur de cette expression est une *référence* au tableau créé. Comme toute valeur, celle-ci peut être affectée à une variable, qui doit avoir été déclarée de type `int[]` :

```
int[] t = new int[4];
```

Les éléments d'un tableau sont indicés à partir de 0 : `t[0]`, `t[1]`, etc. La longueur d'un tableau, c'est-à-dire le nombre de ses éléments, est fixée lors de sa construction et ne peut être modifiée ultérieurement ; elle est obtenue comme la valeur du champ `length` du tableau ; par exemple, `t.length` vaut 4 si `t` est défini comme ci-dessus.

Instances Enfin, les *instances de classe* sont des données dont le type est une classe. Ces types résultent d'une définition de classe, de la forme

```
class ... { ... }
```

Comme les tableaux, les instances de classes sont des *objets*, et non des *valeurs*. Comme les tableaux, les instances n'existent dans la mémoire qu'après avoir été créées. C'est également l'opérateur `new`, appliqué cette fois-ci à un constructeur de la classe et à une liste d'arguments, placés entre les parenthèses « `(` » et « `)` », qui permet de construire des instances, par exemple `new Cible()`. L'opérateur `new` retourne une valeur, qui est une référence à l'objet créé.

Résumons. *Les données sont réparties en trois catégories : les données primitives, les tableaux et les instances. Les données primitives sont elles-mêmes des valeurs. Les tableaux et les instances sont des objets, et non des valeurs.*

Nous utiliserons par la suite une quatrième catégorie de types, les *interfaces*, mais aucune donnée ne peut être de ce type (voir § 3.1). Les bibliothèques de Java rassemblent des types, plus exactement des classes et des interfaces, et non des fonctions, comme c'est le cas en C ou en Fortran. Elles introduisent une profusion de types d'objets utiles au développement de programmes, par exemple le type `String` des chaînes de caractères, les types `InputStream` et `OutputStream` des flots d'entrée et de sortie. Ces types sont organisés en *paquets*, par exemple `java.io` ou `java.util`, qui rassemblent des types apparentés, eux-mêmes organisés de façon hiérarchique. On notera que l'usage est de donner aux classes et aux interfaces des noms commençant par une majuscule et aux paquets des noms commençant par une minuscule.

1.3 Valeurs et expressions Une valeur, c'est le résultat de l'évaluation d'une expression. Cette notion, essentielle au style applicatif, est également centrale en programmation objet, pour la simple raison que ce sont les valeurs qui circulent, et non les données : ce sont les valeurs qui sont affectées à une variable, passées en argument à une méthode, ou retournées comme résultat d'une méthode. Les valeurs se répartissent en deux catégories :

- les valeurs primitives,
- les références, qui sont
  - des références à des tableaux, ou
  - des références à des instances de classe.

Les valeurs primitives, c'est-à-dire dont le type est primitif, bénéficient d'une notation littérale (par exemple, `34`, `true`, `2.5`). Contrairement aux objets, les valeurs ne sont pas créées. Il n'y a pas de valeur de type `void`.

Une *variable* est un emplacement mémoire pouvant contenir une valeur. Il en existe plusieurs catégories :

- variable de classe ;
- champ, ou variable d'instance ;
- élément de tableau ;
- variable locale ;

– paramètre des fonctions, des méthodes, des constructeurs, ou des récupérateurs d'exception (qui seront expliqués ultérieurement, au § 3.21).

Les deux opérations que l'on peut faire sur une variable sont : la *lecture* de sa valeur, et l'*écriture* d'une valeur. L'écriture remplace une valeur précédemment écrite, ou bien *initialise* la variable, c'est-à-dire lui donne une valeur initiale. Évaluer une variable consiste à lire sa valeur. C'est une erreur de lire la valeur d'une variable sans qu'elle ait été préalablement écrite ; cette erreur est détectée par le compilateur qui émet un message « `Variable may not have been initialized` ». Les variables locales ne bénéficient pas d'une initialisation automatique, contrairement aux autres catégories de variables.

Java dispose d'un grand nombre de catégories d'expression, qui sont toutes construites à partir de noms (de type, de membre, de variable), de constantes littérales et d'opérateurs. Citons :

– les expressions arithmétiques (par exemple, `a - 2*b`)
– les expressions booléennes (comme `a<b/2 || a>=b`)
– les affectations (comme `a = b+1`)
– les expressions de création (comme `new int[4],new Cible()`)
– les sélection d'un champ (comme `cible.dedans`)
– les invocations de méthodes (comme `cible.lancer()`)
– les transtypages (comme `(double)m`)

Chaque catégorie d'expression a ses propres règles d'évaluation. Notons que l'évaluation d'une expression de type `void` ne produit pas de valeur ; l'évaluation d'une expression de type t différent de `void` produit une valeur de type t ou d'un sous-type de t. Cette propriété, dite de *sûreté de typage* est un facteur essentiel de sécurité.

Évaluer une affectation de la forme *var* = *exp* consiste à évaluer l'expression *exp*, ce qui produit une valeur, puis à écrire cette valeur dans la variable *var*. L'expression *var* = *exp* n'est correctement typée que si le type de `exp` est égal au type de `var` ou en est un sous-type ; la propriété de sûreté de typage garantit alors que la valeur affectée à *var* est du même type que *var* ou en est un sous-type.

Les noms de variable ne sont pas les seules expressions qui peuvent figurer à gauche d'une affectation : une expression d'accès à un élément d'un tableau (exemple : `t[i+1]`) ou à un champ d'une instance (exemple : `p.x`) sont également permises. Ces expressions sont qualifiées en anglais informatique de *lvalue*, ce qu'on pourrait traduire par *expression gauche*. Par contre, une expression comme `a+1` n'est pas une expression gauche et ne peut être affectée.

Les expressions arithmétiques et booléennes de Java, qui sont très voisines de celles que l'on trouve dans d'autres langages, sont reléguées en Annexe. Par contre, les expressions de création d'objet et celles d'invocation de méthode, propres à la programmation à objets, seront examinées en détail (notamment au § 1.9).

Si t est un type et E une expression, l'expression $(t)E$ est appelée expression de *transtypage* (en anglais, *cast*). Elle n'est généralement définie que si t est un sur-type (transtypage ascendant) ou un sous-type (transtypage descendant) du type de E. Quand elle est définie, l'expression $(t)E$ est de type t. Par exemple, pour effectuer une opération flottante quand les opérandes sont entiers, il faut d'abord appliquer un transtypage en `double` à l'un des opérandes : `(double)5/2` et `5/(double)2` ont ainsi pour valeur 2.5 et `(double)5` a pour valeur le flottant 5.0. Il existe des transtypages entre tous les types numériques. Pour ces types, l'évaluation d'une expression de transtypage réalise une *conversion* de la valeur. Les conversions descendantes, par exemple de `int` vers `byte`, ou de `double` vers *int*, peuvent conduire à une perte d'information par troncature. Les conversions entre `float` et `int` peuvent induire un perte d'information, bien que ces deux types aient la même taille.

1.4 Tableaux Pour tout type t, le type des tableaux de t est noté t`[]` ; ce type est sous-type d'`Object`. La déclaration d'une variable de type tableau ne crée pas de tableau :

```
int[] a;
```

Un tableau est un objet rassemblant *un nombre donné de variables de même type*, installées consécutivement en mémoire ; ces variables sont appelées les éléments du tableau ; le nombre d'éléments, appelé la *longueur* du tableau, est représenté par le champ `length` du tableau ; ce champ est non modifiable ; un tableau de longueur nulle est dit vide.

Un tableau est créé grâce à l'opérateur `new[]`. L'expression spécifiant la longueur, entre les crochets « `[` » et « `]` », doit être de type `int` (ou d'un sous-type de `int`), et n'est pas nécessairement constante ; sa valeur doit être positive ou nulle. Les éléments d'un tableau sont toujours initialisés à la valeur nulle de leur type. Voici une définition d'un tableau de `int` de longueur 4 :

```
int[] a = new int[4];
```

Cette définition déclare `t` comme un nom dont le type est « *tableau de* `int` » ; l'évaluation de `int[4]` crée un objet qui comporte un bloc de mémoire pouvant contenir quatre `int` ; enfin, `t` est initialisé par une référence à cet objet, et chacun de ses quatre éléments est initialisé à 0 (figure 1.3).

Il est aussi possible de définir un tableau en initialisant explicitement ses éléments, comme dans l'exemple suivant qui initialise `a[0]` à `1`, `a[1]` à `2`, etc. (figure 1.4) :

```
int[] a = new int[] {1, 2, 3, 4};
int[] a = {1, 2, 3, 4};            // FORME SIMPLIFIÉE
```

La seconde forme ne peut être employée que lors de la définition d'une variable ; la première forme, `new` *t*`[] { ... }`, est une expression de type *t*`[]` qui définit et initialise un tableau *anonyme* et peut être employée dans d'autres contextes.

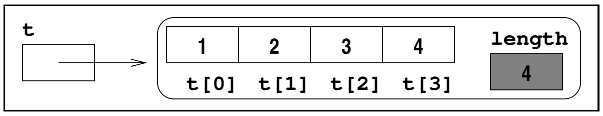

FIG. 1.4 – *Création d'un tableau avec initialisation explicite des éléments.*

Les éléments d'un tableau sont toujours accédés par indexation à partir de zéro : les quatre éléments du tableau `a` sont `a[0]`, `a[1]`, `a[2]` et `a[3]`. Ces `a[`*i*`]`, et plus généralement `a[`*exp*`]`, pour une expression arithmétique *exp* de type `int` (ou d'un sous-type de `int`), sont des variables désignant les éléments du tableau, et peuvent donc être le membre gauche d'une affectation :

```
int k = ...;
a[k+1] = 12;
```

C'est une erreur de tenter d'accéder à un élément hors des bornes du tableau (par exemple `a[-4]`, `a[4]`). Cette tentative déclenche à l'exécution l'exception `ArrayIndexOutOfBoundsException`.

Signalons qu'un tableau, une fois défini, ne peut pas être redimensionné ; c'est pourquoi le type `java.util.List`, qui n'a pas cette limitation, est fréquemment utilisé à la place d'un type tableau. Cependant, la longueur d'un tableau n'a pas à être connue à la compilation (ce qui est une contrainte des tableaux en Pascal, C et C++) ; on peut définir un tableau dont la longueur, puis les éléments, sont lus sur l'entrée standard en cours d'exécution, ou encore dont la longueur est calculée à l'exécution, par une fonction quelconque :

```
double[] t = new double[f(n)];
```

1.5 Classes et instances, membres et constructeurs Les classes sont tout d'abord des types. Ainsi, si une classe `Point` a été définie afin de modéliser les points du plan, il est possible de déclarer un nom dont le type est `Point` :

```
Point p;
```

Cette déclaration de `p`, qui ne crée aucun objet, dit simplement que le nom `p` pourra désormais être employé pour désigner des objets de classe `Point` (mais pas, par exemple, pour désigner des chaînes de caractères). Les objets sont créés par *instanciation*, en utilisant l'opérateur `new`, l'un des constructeurs de la classe, et une liste d'arguments. Par exemple, l'expression `new Point(2, 3)` permet de créer un objet de classe `Point`, de coordonnées $(2, 3)$; la valeur de cette expression est une *référence* à l'objet créé et cette valeur est *non nulle*, c'est-à-dire différente de la valeur `null`. Après l'affectation

```
p = new Point(2, 3);
```

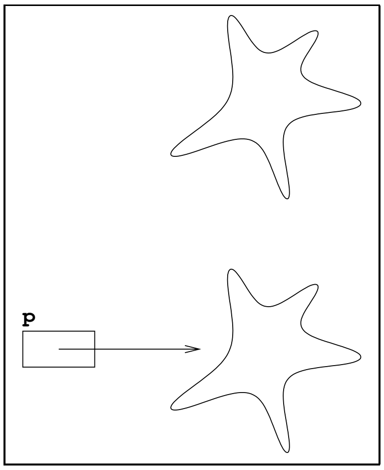

FIG. 1.5 – *Création d'un* `Point` *et affectation à la variable* `p`.

le nom `p` désignera l'objet nouvellement créé (figure 1.5). La déclaration d'un nom d'objet peut aussi comporter la création d'un objet ; les définitions suivantes de `p` et `o` introduisent ces noms et font en sorte qu'ils désignent des objets de la classe `Point` :

```
Point p = new Point(2, 3);
Point o = new Point();
```

Un constructeur d'une classe a le même nom que la classe, et une liste de paramètres. Il peut y avoir plusieurs constructeurs, qui sont distingués par le nombre et le type de leurs paramètres. L'opérateur `new`, appliqué à un constructeur d'une classe, a pour effet de créer un objet de cette classe et retourne une référence vers cet objet. C'est cette référence qui est affectée, comme valeur initiale, aux noms `p` et `o`. Une référence n'est pas un objet, c'est plutôt un numéro d'identification unique, qui est géré de façon interne par la Machine Virtuelle Java, alors que les noms sont choisis et gérés par le programmeur. L'unicité signifie que chaque invocation d'un constructeur retourne une référence à un objet différent. Malgré les apparences, les deux points suivants sont différents, et le test d'égalité `p1 == p2` aura pour valeur `false` :

```
Point p1 = new Point(2, 3);
Point p2 = new Point(2, 3);
```

Il est préférable de dire que `p` « désigne » un point, ou « réfère » à tel point, plutôt que de dire qu'il « est » un point. D'ailleurs, l'opération d'affectation permet de remplacer le point désigné par un autre. Après l'affectation suivante, les deux noms `p1` et `p2` désigneront le même point, celui référé par `p1` (figure 1.6):

```
p2 = p1;
```

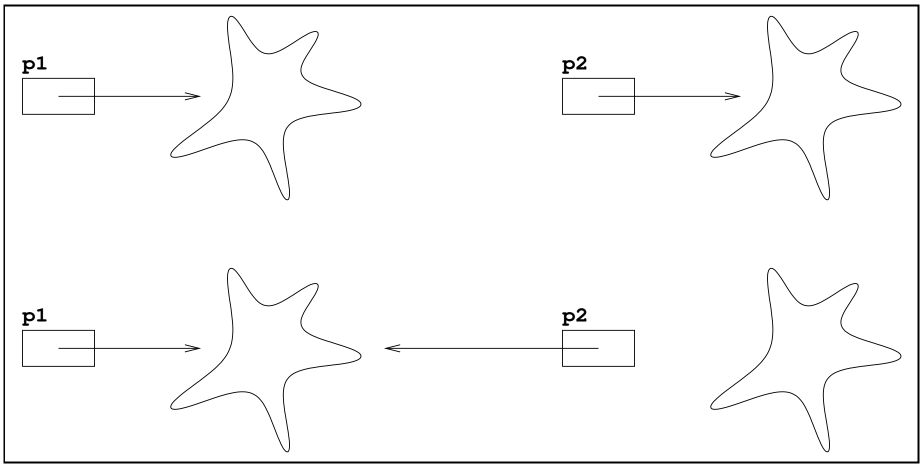

FIG. 1.6 – *Deux points `p1` et `p2` ; les deux variables désignent le même point après l'affectation `p1 = p2`.*

Champs Définir une classe consiste à définir ses constructeurs ainsi que ses membres. Les *membres* d'une classe sont des champs, des méthodes (et éventuellement des classes ou interfaces imbriquées). Nous allons définir une classe `Point` pour les points du plan. Il est naturel de doter cette classe de deux champs[2] de type `double`, de nom `x` et `y`, qui représentent les coordonnées d'un point. Les champs sont parfois appelés des variables d'instance ou encore des variables d'état, car ils décrivent l'état interne de l'objet (figure 1.7). La définition de la classe comprendra au moins ces deux champs :

```
class Point {
  double x, y;

  // ...
}
```

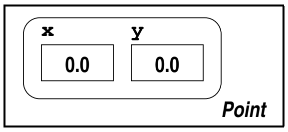

FIG. 1.7 – *Une instance de* `Point` *avec ses deux champs initialisés à zéro.*

Les champs sont d'abord initialisés à la *valeur nulle* de leur type. Ces valeurs nulles sont 0 pour les types numériques, `false` pour le type `boolean`, `'\u0000'` pour le type `char`, et `null` pour les types référencés (tableaux et classes).

On accède à un champ d'une instance en suffixant le nom d'un objet par le nom du champ ; par exemple, on accède aux champs `x` et `y` du point `p` au moyen des expressions `p.x` et `p.y`. C'est une erreur d'accéder à un champ d'un objet qui n'existe pas. Ainsi, l'évaluation de `p.x`, qui suit une déclaration de `p` et son initialisation par `null`, provoque le *déclenchement de l'exception* `NullPointerException` :

```
  Point p = null;
  p.x = 2;                // ERREUR -> NullPointerException
```

Pour éviter cette erreur, il est souvent nécessaire de commencer par faire le test `p != null`.

Constructeurs Outre ses champs, la définition d'une classe comporte la définition de ses *constructeurs*, qui portent le même nom que la classe, ont comme les méthodes, une liste de paramètres, mais n'ont pas de type de retour. Quand ceux-ci seront invoqués, au moyen de l'opérateur `new`, une instance sera créée, ce qui signifie qu'une zone de la mémoire lui sera attribuée, et ses champs pourront alors être initialisés. La définition des constructeurs consiste essentiellement à spécifier les différents cas d'initialisation

2. Les champs d'une classe sont analogues à ceux des types d'enregistrement d'autres langages (`struct` en C, `record` de Pascal ou de CAML).

utiles. Par exemple, on pourra initialiser les champs d'un point à l'aide d'un couple de coordonnées, ou bien on pourra les initialiser à zéro :

```
class Point {
  // ...

  // les deux constructeurs :

  Point() {}

  Point(double x, double y) {
    this.x = x;
    this.y = y;
 }
  // ...
}
```

Employé dans un constructeur, le nom `this` réfère à l'instance créée par ce constructeur. Ceci permet en particulier d'employer le nom du champ `x` comme paramètre du constructeur, dans `this.x = x`.

Il n'est pas obligatoire de définir des constructeurs. Si l'on n'en définit pas explicitement, le constructeur sans paramètre et de corps vide est toujours défini implicitement : c'est le *constructeur par défaut*. Par contre, dès que l'on définit un (autre) constructeur, ce constructeur par défaut n'est plus défini implicitement et ne peut donc plus être utilisé. C'est pourquoi, il faut ici définir explicitement le constructeur par défaut, du moins si l'on envisage de l'utiliser.

Méthodes Analogues à des fonctions, les *méthodes* de la classe rassemblent des instructions qui opèrent sur les instances de la classe. On pourra doter la classe `Point` d'une méthode `translater()`, qui a deux paramètres `dx` et `dy`, de type `double` et qui applique à un point une translation selon le vecteur (`dx, dy`). On invoque une méthode d'une instance en suffixant le nom d'un objet par le nom de la méthode, suivi par ses arguments entre parenthèses : par exemple, on invoque la méthode `translater()` sur le point `p` au moyen de l'expression `p.translater(1, 1)`. Dans un langage procédural, on devrait faire figurer le point `p` comme argument d'une procédure et écrire `translater(p, 1, 1)`. En programmation à objets, on dit parfois que l'on envoie à l'objet `p` le message `translater`, avec les arguments `1`, `1`, selon l'usage consacré par le langage Footnotesizetalk. C'est une erreur d'invoquer une méthode d'un objet qui n'existe pas. Ainsi, l'évaluation de `p.translater(1, 2)`, quand `p` a été initialisé par `null` ou n'a pas été initialisé, provoque le déclenchement de l'exception `NullPointerException`, de la même façon que l'accès à un champ de `p` :

```
  Point p = null;
  p.translater(1, 2);   // ERREUR -> NullPointerException
```

Voici enfin la définition de la classe `Point`, qui comprend deux champs, deux constructeurs et une méthode :

```
class Point {
  double x, y;

  Point() {}

  Point(double x, double y) {
    this.x = x;
    this.y = y;
 }

  void translater(double dx, double dy) {
    this.x = this.x + dx;        // x = x + dx
    this.y = this.y + dy;        // y = y + dy
 }
}
```

`this` Employé dans une méthode, le nom `this` réfère à l'instance de la classe à laquelle la méthode sera appliquée : c'est un paramètre implicite de toute méthode d'instance, qui permet d'accéder aux membres de l'instance (par exemple, `this.x`, `this.y` et `this.translater()`). D'autre part, le nom `this()` peut aussi être employé avec une liste d'arguments, mais seulement en première instruction du corps d'un constructeur, pour invoquer explicitement un autre constructeur de la classe. Par exemple, le constructeur sans argument pourrait être défini à l'aide du constructeur à deux arguments en employant `this()` :

```
 Point() {
   this(0, 0);
}
```

1.6 Méthodes et fonctions Les mathématiciens définissent maintenant les fonctions comme des relations, c'est-à-dire des sous-ensembles du produit cartésien de deux ensembles qui vérifient simplement des propriétés d'existence et d'unicité (tout élément a une image et une seule) : c'est ainsi que les fonctions sont comprises en théorie des ensembles.

Les informaticiens n'adoptent pas cette définition pour au moins deux raisons : d'une part, une fonction n'est pas seulement un ensemble de couples, mais est une méthode de calcul (ou algorithme), et d'autre part une fonction, pour un argument donné, peut ne pas toujours rendre le même résultat (c'est heureusement le comportement de la fonction `Math.random()` de génération de nombres pseudo-aléatoires). C'est notamment le cas des méthodes de Java, dont la valeur ne dépend pas seulement de la valeur de ses arguments, mais aussi de l'état interne de l'instance à laquelle elle est appliquée, c'est-à-dire de la valeur de ses champs.

On définit une méthode par son en-tête et son corps, c'est-à-dire :

- l'en-tête comprend
  - des modificateurs (`public`, `private`, `static`,...) éventuels,
  - le type de retour ,
  - le nom de la méthode,
  - une liste de paramètres, avec leur type ;
- le corps est un bloc, composé de déclarations, dites *locales*, et d'instructions.

Sauf quand le type de retour est `void`, le corps d'une méthode contient au minimum une instruction « `return` *expression*; » qui permet de communiquer son résultat. Quand le type du résultat est `void`, son corps peut contenir l'instruction « `return;` », mais ce n'est pas obligatoire.

Invocation La seule opération que l'on peut faire sur une méthode est de l'*invoquer*, ou de l'*appeler* avec des arguments. Nous considérons d'abord le cas des méthodes d'instance, et nous verrons plus loin le cas des méthodes de classe. Une *invocation*, ou *appel* de méthode, est une expression formée à partir d'une expression cible, d'un « `.` », du nom de la méthode invoquée et d'une liste d'arguments placés entre « `(` » et « `)` », par exemple `p.translater(1,1)`. L'expression cible peut être un nom (par exemple, `p`), et en particulier `this` ; l'invocation `this.m(...)` peut être simplifiée en `m(...)` à l'intérieur de la classe qui définit la méthode `m`. Signalons bien qu'une méthode `f` sans argument doit être invoquée par `p.f()` et non par `p.f`, notation qui confondrait invocation de méthode sans argument et accès à un champ. Le nombre des arguments (dans l'expression d'invocation) et le nombre des paramètres (dans la définition de la méthode) doivent être identiques : une invocation de `p.translater(1, 2, 3)` est illégale.

Le type d'un argument doit être égal au type du paramètre correspondant ou doit en être un sous-type ; par exemple, une méthode dont la déclaration comporte un paramètre de type `int` pourra être invoquée avec un argument de type `byte`, car `byte` est un sous-type de `int`, mais ne pourra pas être invoqué par un argument de type `long`, ni de type `String`, car `long` et `String` ne sont pas des sous-types de `int`.

L'évaluation d'une invocation de méthode, de la forme *cible.m(*arg_1*, ...)*, se fait en plusieurs étapes que nous préciserons au § 1.9. Certaines étapes, fondées uniquement sur le type de l'expression cible et sur les types des arguments sont effectuées à la compilation. D'autres étapes nécessitent la connaissance de l'objet cible, et ne peuvent se faire qu'à l'exécution. Plusieurs traits importants sont mis en œuvre dans ces diverses étapes : la résolution de la surcharge, la liaison tardive, le passage des arguments par valeur. On représentera une invocation par le symbole $\rightarrow$: par exemple, `main()` $\rightarrow$ `calculAire(1000000)` exprime que `main()` invoque `calculAire()`, avec l'argument `1000000`. Il y a alors création en mémoire d'un cadre d'invocation de la méthode invoquée, qui a pour but l'exécution du corps de cette méthode, et dont le contenu sera détaillé un peu plus loin.

Quand l'exécution du corps de la méthode invoquée est terminée, la méthode invoquée *retourne*. Si le type du résultat de la méthode invoquée est différent de `void`, la méthode *retourne une valeur*, qui est son résultat. La valeur retournée est celle de l'expression figurant dans l'instruction `return` exécutée (le corps de la méthode peut contenir plusieurs `return`, mais une seule sera exécutée, puisque la méthode retourne aussitôt après). Le type de cette expression doit être égal au type de retour de la méthode, ou en être un sous-type ; notamment, si ces types sont numériques, il y a conversion implicite de la valeur de l'expression de retour vers le type de retour quand cela est possible : par exemple un « `return 3;` » dans une méthode déclarée retourner un `double` ne retournera pas l'entier 3, mais le double 3.0. On notera le retour d'une valeur v par le symbole $\stackrel{v}{\rightarrow}$ ou par $\stackrel{v}{\leftarrow}$: `calculAire(1000000)` $\stackrel{3.14}{\rightarrow}$ `main()`, ou `main()` $\stackrel{3.14}{\leftarrow}$ `calculAire(1000000)`. Une méthode dont le type du résultat est `void`, retourne, mais ne retourne pas de valeur. On notera ces deux usages du verbe « retourner », intransitif en général, et transitif dans le cas d'une valeur.

L'invocation d'une méthode implique la création d'un *cadre d'invocation*, lequel a une existence dans la mémoire de la Machine Virtuelle Java et une durée de vie. La zone mémoire qui lui est attribuée comporte, entre autres, les informations suivants :

- les *paramètres*, initialisés par la *valeur* des arguments ;
- les *variables locales* ;
- la *variable de retour*, qui est initialisée par la *valeur* de l'expression E quand une instruction « `return` E; » est exécutée par la méthode invoquée ;
- le *point de retour*, c'est-à-dire la désignation, dans le code du programme, de l'instruction qui sera exécutée juste après le retour de la méthode invoquée.

Ce cadre d'invocation est placé dans une partie de la mémoire appelée la *pile*. La durée de vie de ce cadre est déterminée par l'exécution du corps de la méthode : il est *empilé* à l'invocation, c'est-à-dire placé au sommet de la pile, et il est *dépilé* au retour, c'est-à-dire retiré du sommet de la pile. Quand une méthode retourne, le cadre d'invocation qui avait été créé lors de son invocation est détruit, et les données qu'il contenait cessent d'être utilisables. Ce mode d'allocation de la mémoire est appelé *allocation automatique*, ou allocation *sur la pile* ; la plupart des langages disposent de ce mode d'allocation, à l'exception notable de Fortran 77.

Membres de classe Les membres déclarés `static`, appelés *membres de classe*, par opposition aux membres d'instance, sont communs à toutes les instances d'une classe. Un champ statique, ou *variable de classe* est donc une variable unique, partagée par toutes les instances de la classe ; cette variable est toujours créée, même si aucune instance de la classe n'est construite.

Une méthode qui ne dépend pas d'une instance est appelée une *méthode de classe* , ou plus couramment, une *fonction*. Sa définition doit spécifier le mot-clé `static`, et elle ne doit pas alors être invoquée sur une instance, mais en utilisant le nom de la classe (si elle est invoquée dans la classe où elle est définie, elle peut être invoquée simplement par son nom, sans la préfixer du nom de sa classe). Voici la définition d'une fonction, elle même invoquant la fonction `sqrt()`, méthode statique de la classe `Math` :

```
private static double f(double x) {
  return Math.sqrt(1 - x*x);
}
```

Une méthode statique ne peut accéder qu'aux membres statiques de la classe ; elle ne doit pas utiliser `this`. C'est le cas de la méthode `main()` qui ne peut invoquer directement d'autres méthodes ou accéder directement à d'autres champs de la classe principale que si ces méthodes et champs sont également statiques. Par contre, une méthode statique peut construire un objet et accéder à tous ses membres.

Passage par valeur La description des opérations réalisées lors de l'invocation d'une méthode spécifie la relation entre l'argument, qui figure dans l'expression d'invocation, et le paramètre correspondant, qui figure dans la définition de la méthode : pour chaque paramètre d'une méthode, et pour chaque invocation, il y a création d'une variable dans son cadre d'invocation et initialisation de cette variable par la *valeur de l'argument* correspondant. Les affectations à un paramètre ne peuvent donc pas modifier la valeur de l'argument correspondant. Par exemple, la procédure `f()` suivante, après avoir initialisé le paramètre `x` à la valeur de l'argument `n`, lui affecte 0 (et n'en fait rien d'utilisable), mais ne modifie évidemment pas la variable `n` :

```
class Test {
```

```
  static void f(int x) {
    x = 0;
  }
  public static void main(String[] args) {
    int n = 4;
    f(n);
    System.out.println("n = " + n);        // --> n = 4
  }
}
```

Dans le cas d'un type de références (tableau, classe ou interface), la valeur n'est bien sûr pas l'objet lui-même, mais une référence à cet objet. Par exemple, la procédure `f()` suivante, après avoir initialisé le paramètre `x` à la valeur de `p` (qui est une référence au point $(2,3)$), lui affecte une référence au point $(0,0)$, ce qui ne modifie pas la variable `p`, utilisée comme argument de `f()`.

```
class Test {
  static void f(Point x) {
    x = new Point();
  }

  public static void main(String[] args) {
    Point p = new Point(2, 3);
    f(p);
    System.out.println("x=" + p.x + ", y=" + p.y);  // x=2, y=3

  }
}
```

Cependant, à la différence d'une valeur d'un type primitif, une méthode ou l'accès à un champ peut être appliqué à une référence, et l'objet référencé peut être modifié de cette façon :

```
class Test {
  static void f(Point x) {
    x.translater(1, 1);
  }

  public static void main(String[] args) {
    Point p = new Point(2, 3);
    f(p);
    System.out.println("x=" + p.x + ", y=" + p.y);  // x=3, y=4

  }
}
```

Dans ce cas, la méthode `f()` ne modifie pas la valeur de son argument, qui reste une référence au même objet, mais elle modifie l'objet référencé. Signalons que certains langages, comme Pascal ou C++, disposent d'un autre mode de passage des arguments, le passage *par référence*, qui permet de modifier la valeur de l'argument.

Comme les données des types primitifs (`int`, `double`, etc) ne sont par référencées, il n'est pas possible d'écrire de méthodes qui aient le même effet pour les types primitifs. Par exemple, les expressions `x++` et `++x` qui incrémentent de 1 la valeur de la variable `x` ne peuvent être définies par des méthodes ; signalons au passage que la première a comme valeur celle de `x` *avant* l'incrémentation, et la seconde, *après* l'incrémentation[3].

Tableaux et méthodes Un tableau peut être argument d'une méthode ou retourné par une méthode. Le tableau argument est passé par valeur ; or, la valeur de cet argument est une référence à un tableau ; c'est cette référence qui sert à initialiser le paramètre correspondant, lors de l'invocation de la méthode. Ceci permet à la méthode d'opérer sur le tableau. On peut donc en profiter pour faire modifier la valeur d'un élément du tableau par la fonction. Par exemple, on échange la valeur de deux éléments d'un tableau par :

```
static void échangerÉléments(double[] a, int m, int n) {
  double temp = a[m] ;
  a[m] = a[n] ;
  a[n] = temp;
}

public static void main (String[] args) {
  double[] a = {1, 2, 3, 4, 5, 6};
  échangerÉléments(a, 2, 4);
  // ...
}
```

Un tableau peut être retourné par une méthode. Par exemple, la méthode suivante calcule le tableau somme de deux tableaux :

```
static double[] vecteurAdd(double[] x, double[] y) {
  int n = Math.min(x.length, y.length);
  double[] z = new double[n];

  for (int i=0; i<n; i++) {
    z[i] = x[i] + y[i];
  }
  return z;
}
```

On l'invoque ainsi :

```
double[] x = {1.0, 2.0, 3.0};
```

3. Les classes enveloppantes `Integer`, `Double`, etc. qui encapsulent un `int`, un `double`, etc. permettent de réaliser cet effet.

```
  double[] y = {4.0, 5.0, 6.0};
  double[] z = vecteurAdd(x, y);
```

Il est souvent commode d'utiliser un tableau pour rassembler plusieurs valeurs de retour, quand ces valeurs sont de même nature et que leur nombre n'est pas connu. Par exemple, si l'on demande de calculer les solutions réelles d'une équation du deuxième degré (représentées grâce à une classe `Trinome`), on pourra retourner ces solutions dans un tableau :

```
  double[] solutions() { ... }
```

Une invocation de cette méthode aura la forme suivante :

```
  Trinome e = new Trinome(3.0, -2.6, 0.5);
  double[] racines = e.solutions();
```

On pourra alors accéder au nombre de solutions par `racines.length` et, selon cette valeur, aux racines éventuelles par `racines[0]` et `racines[1]`.

Surcharge De façon analogue aux constructeurs, il est fréquent que plusieurs méthodes d'une classe portent le même nom, chacune ayant soit un nombre d'arguments distinct, soit le même nombre d'arguments, mais des types distincts : on dit que ces méthodes n'ont pas le même *profil*. On parle dans ce cas de *surcharge* du nom. Par exemple, le nom `translater` est surchargé dans l'exemple suivant, où deux méthodes portent le même nom :

```
class Point {
  // ...
  void translater(double dx, double dy) {
    x = x + dx;          // this.x = this.x + dx
    y = y + dy;
  }
  void translater(double d) {
    translater(d, d);
  }
}
```

Les profils de ces deux méthodes sont respectivement :

- `(double, double)`
- `(double)`

Le compilateur est capable de déterminer laquelle de ces méthodes homonymes doit être invoquée, en se basant uniquement sur le nombre et le type des expressions arguments : c'est ce que l'on appelle la *résolution de la surcharge*. Quand il y a plusieurs méthodes candidates, c'est la plus *spécifique* qui est sélectionnée. Dans l'exemple suivant,

```
class Test {
```

```
  static void p(int x) {
    System.out.println("int");
  }
  static void p(long x) {
    System.out.println("long");
  }
  public static void main(String[] args) {
    p(1);  // --> "int"
    p(1L); // --> "long"
  }
}
```

les deux méthodes `p(int)` et `p(long)` sont candidates pour l'expression `p(1)`, parce que `1` est de type `int` et que `int` est un sous-type de `long`. Cependant, la méthode `p(int)` est plus spécifique que `p(long)` puisque tout argument accepté par la première est accepté par la seconde, mais pas réciproquement (ce qui provient du fait que `int` est un sous-type de `long` mais que `long` n'est pas un sous-type de `int`). C'est donc `p(int)` qui est sélectionnée. Dans le cas de l'expression `p(2L)`, l'argument est un `long`, donc seule la méthode `p(long)` est candidate.

Un programme pour lequel cette résolution ne serait pas possible est incorrect : cette erreur est détectée par le compilateur. Dans l'exemple suivant, l'expression `p(1, 2)` a deux arguments de type `int` et il y a deux méthodes candidates : `p(int, long)` et `p(long, int)`.

```
class Test {

  static void p(int x, long y) {
    System.out.println("IL");
  }
  static void p(long x, int y) {
    System.out.println("LI");
  }
  public static void main(String[] args) {
    p(1, 2);                          // ERREUR : AMBIGU
  }
}
```

Chacune pourrait être invoquée puisque `int` est un sous-type de `long`, mais aucune n'est plus *spécifique* que l'autre.

Notons que le type de retour de la méthode n'est pas pris en considération dans la résolution de la surcharge. Enfin, deux méthodes de même nom ne peuvent pas être définies avec le même profil et des types de retour différents. Ce n'est pas une surcharge, c'est une erreur :

```
  void f(int x) { ... }    // ERREUR
  int f(int x) { ... }     // ERREUR
```

C'est également une erreur de redéfinir une méthode avec le même profil en changeant simplement le nom des paramètres, car le nom des paramètres n'est pertinent que dans le corps de la méthode :

```
void f(int x) { ... }    // ERREUR
void f(int y) { ... }    // ERREUR
```

Toutes ces erreurs sont détectées par le compilateur.

1.7 Types abstraits et sous-typage On souhaite à présent modéliser plusieurs classes d'objets géométriques : les points, les cercles et les rectangles. Tous ces objets ont une surface, mais la méthode de calcul de cette surface dépend de la classe de l'objet considéré. Un point du plan est défini par ses deux coordonnées, un cercle est défini par son centre et son rayon, un rectangle est défini par ses points supérieur gauche et inférieur droit. Voici par exemple la construction de deux points, d'un cercle et d'un rectangle et l'invocation d'une méthode `surface()` sur ces objets, en supposant que les classes correspondantes ont été définies par ailleurs, et qu'elles contiennent une méthode `surface()` :

```
class Test {
  public static void main(String[] args) {
    Point o = new Point(), p = new Point(1,1);
    Cercle c = new Cercle(o, 2);
    Rectangle r = new Rectangle(o, p);
    double sp = p.surface();
    double sc = c.surface();
    double sr = r.surface();
  }
}
```

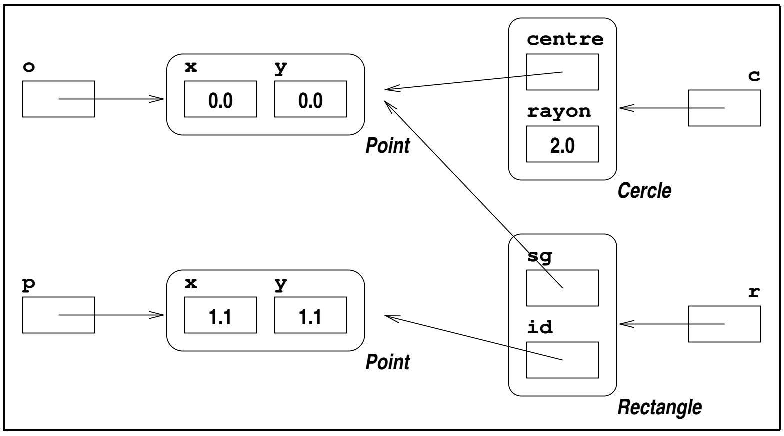

FIG. 1.8 – *Objets définis par composition.*

Il n'y a pas de difficulté pour définir trois classes `Cercle`, `Rectangle` et `Point`, chacune disposant d'une méthode `surface()` appropriée. Voici par exemple la classe `Cercle`, qui comporte deux champs, un constructeur et une méthode :

```
class Cercle {
  Point centre;
  double rayon;
  Cercle(Point centre, double rayon) {
    this.centre = centre;
    this.rayon = rayon;
  }
  double surface() {
    return Math.PI * rayon * rayon;
  }
}
```

On notera au passage que la classe `Cercle` a un champ `centre` dont le type, `Point` est une classe. Ceci ne signifie pas qu'un objet de type `Cercle` contienne un objet de type `Point`. La valeur d'un champ est toujours une *valeur*, ce ne peut pas être un objet ; la valeur de `centre` est une référence à un objet (figure 1.8). De la même façon, la classe `Rectangle` a deux champs `sg` et `id` (qui désignent les sommets « supérieur gauche » et « inférieur droit » du rectangle) de type `Point`. Les instances des classes `Cercle` et `Rectangle` sont des exemples d'objets composés à partir d'autres objets (ici, des points). Il est fréquent qu'une classe soit définie par *composition* à partir d'autres classes.

Supposons maintenant que l'on veuille tirer une forme au hasard, à pile ou face, puis calculer sa surface. Il semble naturel de former l'expression suivante :

```
  (Math.random()>0.5 ? c : r).surface();
```

Cependant cette expression n'est pas typée correctement parce que `c` et `r` ne sont pas du même type et aucun n'est un sous-type de l'autre. Il en est de même de l'expression `Math.random()>0.5 ? true : 3`, car `boolean` n'est pas un sous-type de `int` et vice-versa. La solution est d'introduire un nouveau type, `Forme`, dont `Cercle` et `Rectangle` seront des sous-types, de sorte que `Math.random()>0.5 ? c : r` soit une expression de type `Forme`. La classe `Forme` devra donc avoir une méthode `surface()`, puisque toute forme a une surface. Cependant, comme le calcul de la surface ne peut être fait que pour une forme particulière, la méthode `surface` de la classe `Forme` ne peut pas avoir de corps ; on dit que c'est une méthode *abstraite* et on la déclare en remplaçant le corps par un '`;`' et en faisant précéder son type de retour par le mot-clé `abstract`.

Une classe qui contient au moins une méthode abstraite, comme la classe `Forme`, est appelée une *classe abstraite*. Sa définition doit aussi être précédée du mot-clé `abstract` :

```
abstract class Forme {
  abstract double surface();
}
```

Notons qu'on ne peut pas créer un objet d'un type abstrait : une classe abstraite n'a pas d'instance, l'expression `new Forme()` n'est pas correcte. Pour faire de `Point`, `Cercle` et `Rectangle` des sous-types de `Forme`, il suffit de les déclarer comme des *sous-classes* de `Forme`, au moyen du mot-clé `extends` :

```
class Cercle extends Forme {
  Point centre;
  double rayon;
  Cercle(Point centre, double rayon) {
    this.centre = centre;
    this.rayon = rayon;
  }
  double surface() {
    return Math.PI * rayon * rayon;
  }
}

class Point extends Forme { ... }
class Rectangle extends Forme { ... }
```

Ces définitions font des trois types `Point`, `Cercle`, `Rectangle` des *sous-types* du type `Forme`. L'expression `Math.random()>0.5 ? c : r` est alors de type `Forme` et on peut lui appliquer la méthode `surface()`, qui est abstraite dans `Forme`, mais *implémentée* dans ses sous-types `Point`, `Cercle` et `Rectangle`. On dit parfois que ces trois types sont des types concrets. Ceci permet aussi de placer dans un tableau de `Forme` des objets de types différents :

```
Forme[] t = {o, c};
```

Plus généralement, la relation de sous-typage permet d'utiliser une expression d'un sous-type de t dans certains contextes où une expression de type t serait attendue :

- dans le membre droit d'une affectation à une variable de type t ;
- comme argument d'une méthode correspondant à un paramètre de type t.

Ceci permet d'affecter à une variable de type `Forme` un objet de type `Cercle` :

```
Forme c = new Cercle(new Point(), 2);
```

Par contre, il n'est pas possible d'affecter à une variable de type t une expression d'un sur-type de t, ou d'invoquer une méthode ou un constructeur avec un argument dont le type est un sur-type du type du paramètre correspondant :

```
Point q = p;          // ERREUR (à la compilation)
```

Une opération de transtypage est indispensable dans ce cas et conduira à une vérification, à l'exécution du fait que `p` désigne effectivement un objet de classe `Point` :

```
Point q = (Point)p;
```

Une expression comportant un transtypage vers un sur-type est toujours correctement typée ; son exécution pourra déclencher une exception à la compilation.

```
Point q = (Point)r;   // correct à la compilation
                      // ERREUR à l'exécution
```

1.8 *Héritage*

L'héritage est un mécanisme des langages à objets particulièrement utile en génie logiciel, car il offre des possibilités de réutilisation par enrichissement de classes déjà écrites. Il permet de définir une classe à partir d'une autre, en héritant des membres de cette dernière.

En voici un exemple. Nous devons définir une classe `PointColore`, dont les instances sont des points colorés [4]. Une première solution consisterait à définir deux champs, un point et une couleur, en utilisant la classe `Point` déjà définie et la classe `java.awt.Color` (du paquet `java.awt`) :

```
class PointColore {
 Point point;
 java.awt.Color couleur;

 PointColore(double x, double y, java.awt.Color couleur) {
   point = new Point(x, y);
   this.couleur = couleur;
 }
 void translater(double dx, double dy) {
   point.translater(dx, dy);
 }
}
```

4. Bien que Java autorise les lettres accentuées dans les noms, il est préférable de ne pas les utiliser dans les noms de type car ceux-ci servent à former des noms de fichier et les systèmes d'exploitation ne sont pas toujours capables de les traiter correctement ; par contre, les lettres accentuées peuvent être utilisées sans problème dans les noms de variable ou de méthode.

Cette classe utilise la classe `Point`, un de ses constructeurs et une de ses méthodes. On peut l'utiliser ainsi :

```
class Test {
 public static void main(String[] args) {
   PointColore pointRouge =
     new PointColore(1, 2, java.awt.Color.red);
   pointRouge.translater(1, 1);
 }
}
```

Ce mode de définition d'une classe est dit par *composition* : un `PointColore` se compose d'un `Point` et d'une `java.awt.Color`. En outre, un `PointColore` *délègue* la translation à son composant `point`. Ces deux techniques, de composition et de délégation, sont très utilisées. Par exemple, on définira une interface graphique à partir de plusieurs composants graphiques (des boutons, barres de menus, menus, etc.) et on déléguera à des observateurs le soin de traiter certains événements (presser un bouton, choisir un item dans un menu, etc.).

Java propose une autre technique, dite d'*extension*, qui permet de réutiliser une (seule) classe et ses méthodes : il suffit de déclarer la classe `PointColore` comme une extension de `Point`, à l'aide de la clause `extends`, et de lui ajouter un champ de type `java.awt.Color`. La classe `Point` est dite *parente* ou *sur-classe directe* de `PointColore`, celle-ci étant *dérivée*, ou *sous-classe directe* de `Point`.

```
class PointColore extends Point {
 java.awt.Color couleur;

 PointColore(double x, double y, java.awt.Color couleur) {
   super(x, y);
   this.couleur = couleur;
 }
 PointColore() {
   super();
   this.couleur = java.awt.Color.black;
 }
}
```

Ses constructeurs commencent par invoquer le constructeur de la classe parente, par l'opérateur `super`. En effet, le nom `super` peut être employé avec une liste d'arguments pour invoquer explicitement un constructeur de la classe parente, celui qui accepte les mêmes types d'arguments que cette invocation de `super`. Une invocation de `super( ... )` ne peut figurer qu'en première instruction du corps du constructeur (retenir qu'avant de créer un objet, il faut d'abord créer son « parent »). Une invocation de `super()` n'est pas obligatoire, mais si elle n'est pas explicite, une invocation implicite de `super()` a toujours lieu, sans argument, ce qui suppose que la classe parente a un constructeur sans paramètre.

Tous les membres de la classe `Point`, c'est-à-dire ses deux champs `x` et `y` et sa méthode `translater`, sont alors hérités par `PointColore` (figure 1.9) :

```
PointColore pc = new PointColore(1, 2, java.awt.Color.red);
pc.translater(2, 2);
```

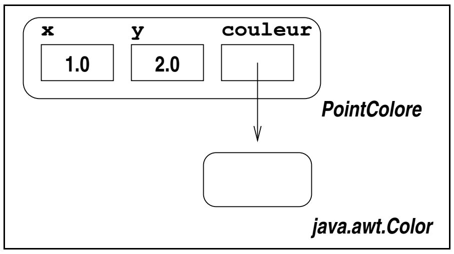

FIG. 1.9 – *Une instance de* `PointColore` *a trois champs :* `x` *et* `y` *sont hérités de* `Point`*, tandis que* `couleur` *est défini par* `PointColore`*.*

De façon générale, ce mécanisme d'extension spécifié par la clause `extends` a deux effets :

– la classe dérivée hérite de *certains* membres de sa classe parente ;
– la classe dérivée est un sous-type de sa classe parente.

L'héritage n'est pas systématique. Il y a d'abord une condition d'accessibilité, que nous préciserons par la suite. Par exemple, les membres privés ne sont pas hérités. Les méthodes d'instance ne sont héritées que si elles ne sont pas redéfinies dans la classe dérivée. Enfin, les constructeurs ne sont jamais hérités.

À l'exception de la classe `Object`, toute classe dérive d'une autre classe ; si la mention de l'extension est absente, ceci signifie que la classe dérive d'`Object`. Ceci permettra de réaliser une forme de généricité qui permet de traiter tous les objets de façon uniforme.

Héritage de types abstraits Rappelons qu'une méthode abstraite est une méthode déclarée `abstract` et dont le corps est remplacé par un « `;` ». Une classe abstraite est une classe qui a au moins une méthode abstraite (éventuellement héritée d'une classe parente abstraite) ; une telle classe doit être déclarée `abstract`. Les classes abstraites sont donc des classes partiellement implémentées, puisqu'elles peuvent aussi avoir des champs et des méthodes non-abstraites. Une classe abstraite ne peut pas avoir d'instance.

Cependant, les classes abstraites sont des types. On peut déclarer une variable ou un paramètre d'un type abstrait. Le mécanisme de liaison tardive permettra d'invoquer la méthode correcte une fois que la variable ou le paramètre désignera une instance d'une classe concrète dérivée. Supposons par exemple que nous décidions d'attribuer de façon interne à toute forme géométrique un nom, qui est une chaîne de caractères. Au lieu de modifier la classe `Forme` définie plus haut, on peut l'étendre en une classe `FormeNommee`, par l'addition d'un champ de type `String`. Cette nouvelle classe hérite de la méthode abstraite `translater()`, donc est elle-même abstraite :

```
abstract class FormeNommee extends Forme {
  private String nom;
  FormeNommee(String nom) {
    this.nom = nom;
  }
  FormeNommee() {
    this.nom = "";
  }
  String getNom() {
    return nom;
  }
}
```

Notons qu'une classe abstraite peut avoir des constructeurs, mais qu'ils ne peuvent pas être invoqués pour créer des objets et qu'une classe abstraite peut avoir des méthodes non abstraites. Un constructeur d'une classe abstraite est généralement invoqué par un constructeur d'une classe dérivée, explicitement par `super(...)`, ou implicitement, ce qui équivaut à `super()`.

Modifions maintenant les définitions des types concrets `Point`, `Cercle` et `Rectangle` pour les faire dériver de `FormeNommee` au lieu de `Forme`. Ainsi, `Point` dérive de `FormeNommee`, qui dérive de `Forme`. Les types concrets `Point`, `Cercle`, etc., héritent maintenant du champ `nom` et de la méthode `getNom`. Il faut par conséquent ajouter un paramètre de type `String` au constructeur, et invoquer le constructeur de la classe `parente`, au moyen de `super(nom)` :

```
class Point extends FormeNommee {
  double x, y;
  Point(double x, double y, String nom) {
    super(nom);
    this.x = x;
    this.y = y;
  }
  double surface() {
    return 0;
  }
}
```

Un membre privé n'est jamais hérité. Par exemple, le champ privé `nom` défini dans `FormeNommee` n'est pas hérité par `Point` : si `p` est une variable de type `Point`, l'expression `p.nom` n'est pas correcte (figure 1.10). De plus, il n'y a aucun moyen pour contourner ce caractère privé ; il est inutile (contrairement au cas du masquage des champs) d'essayer un transtypage en `FormeNommee` : l'expression `(FormeNommee)p.nom` est également incorrecte, puisque `(FormeNommee)p` est de type `FormeNommee`, et que `nom` en est un champ privé.

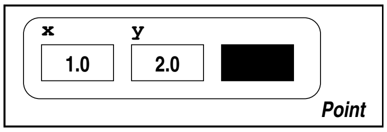

FIG. 1.10 – *Une instance de `Point`, classe dérivée de `FormeNommee` : le champ `nom` n'est pas hérité.*

La classe dérivée ne peut accéder directement aux champs privés de la classe parente, mais peut éventuellement y accéder si elle hérite de méthodes d'accès. Dans l'exemple précédent, le champ `nom` n'est pas hérité, mais la méthode `getNom()` est héritée. Par suite, si `p` est un point, l'expression `p.nom` est incorrecte, mais l'expresssion `p.getNom()` est correcte.

1.9 Liaison tardive Un langage admet un *typage statique* s'il est possible de vérifier à la compilation la cohérence des types de toutes les expressions, et ainsi d'éliminer la possibilité d'erreurs à l'exécution dues à des opérations qui ne respecteraient pas les contraintes de type. Le typage statique est un critère de sécurité essentiel. La plupart des langages de programmation contemporains admettent un typage statique : C, CAML, Java. Il existe cependant des langages non-typés comme Lisp et Prolog.

Le mécanisme de liaison est celui qui associe un objet à un nom. Cette association est déterminée d'une part par le texte du programme, notamment par la *portée lexicale* des noms dans les blocs, et d'autre part par l'environnement d'exécution. C'est ainsi qu'un nom déclaré d'un certain type peut désigner une instance d'un sous-type :

```
Point
  pc = new PointColore(2, 3, java.awt.Color.pink),
  pi = new PointImmobile(2, 3),
  p  = Math.random()>0.5 ? pc : pi;
p.translater(1,1);
```

La troisième affectation est correcte, car de la forme « *type = sous-type* » : le type du nom `p` est `Point`, tandis que le type de la valeur affectée à `p` est

`PointColore` ou `PointImmobile`. La méthode `translater(double, double)` de la classe `Point` est *redéfinie* dans la classe `PointImmobile`. L'évaluation de l'expression `p.translater(1, 2)` consiste à invoquer la méthode `translater(double, double)` définie dans la classe de l'objet désigné par `p`, soit celle de `PointImmobile`, soit celle de `PointColore`. La liaison du nom `translater` à l'une de ces méthodes ne peut être réalisée en général qu'à l'exécution, quand le type réel de `p` est connu. Il s'agit d'une liaison *tardive*, qui est une des particularités des langages orientés objets. Ce mécanisme est propre aux méthodes et ne s'applique pas aux champs.

Redéfinition Une méthode n'est pas héritée si elle est redéfinie dans la classe dérivée, avec le même type de retour et le même profil. Par exemple, la classe `PointImmobile` suivante redéfinit la méthode `translater()` de profil `(double, double)` :

```
class PointImmobile extends Point {
  PointImmobile(double x, double y) {
    super(x, y);
  }
  void translater(double dx, double dy) {}
}
```

La redéfinition des méthodes permet d'utiliser le mécanisme d'extension, non pour enrichir une classe en lui ajoutant des membres, mais pour la spécialiser en modifiant le comportement de certaines méthodes. Le mécanisme de liaison tardive, propre aux langages à objets, permet de tirer parti de ces redéfinitions.

Signalons que la redéfinition d'une méthode peut invoquer la méthode de la classe parente à l'aide du nom `super` : employé dans une méthode, il réfère à l'objet auquel s'applique cette méthode, en tant qu'instance de la classe parente. Il permet ainsi d'accéder aux membres (champs ou méthodes) définis dans la classe parente, même s'ils sont masqués ou redéfinis dans la classe contenant cette utilisation de `super` ; ce serait le cas si nous avions redéfini `translater` ainsi :

```
  void translater(double dx, double dy) {
    super.translater(0, 0);
  }
```

Évaluation d'une invocation De façon générale, l'évaluation d'une invocation de méthode, de la forme *cible.m(arg*$_1$*, ...)*, s'effectue en plusieurs étapes :

1. le type (classe ou interface) t qui doit avoir une méthode m parmi ses

membres est déterminé comme le type de l'expression *cible* (et non le type de sa valeur) ;
2. le type des arguments détermine les méthodes candidates parmi les méthodes de nom *m* de t ;
3. s'il y a plusieurs méthodes candidates (cas de *surcharge*), l'une d'entre-elles est sélectionnée comme étant la plus spécifique, si elle existe ;
4. l'évaluation de l'expression *cible* produit une référence à une instance d'une classe C qui est nécessairement un sous-type de t ;
5. l'évaluation des arguments $arg_1, \ldots$ produit des valeurs $v_1, \ldots$ (le type de ces valeurs n'intervient pas) ;
6. la *liaison tardive* détermine la méthode qui sera invoquée comme étant la méthode sélectionnée si elle n'est pas redéfinie, ou sinon comme la redéfinition de la méthode sélectionnée qui est héritée par la classe C;
7. les valeurs des arguments sont éventuellement convertis en des valeurs du type des paramètres de la méthode déterminée par la liaison tardive ;
8. la création d'un cadre d'invocation de cette méthode, appliquée à l'instance obtenue, avec les arguments évalués.

Sous-typage Voici enfin la définition de la relation de sous-typage (les notions liées aux interfaces seront développées ultérieurement, en § 3.1) :

- tout type est un sous-type de lui-même ;
- chacun des types primitifs `byte`, `short`, `int`, `long`, `float`, `double` est un sous-type des suivants ; `char` est un sous-type de `int` ;
- si t et t' sont des types de références, et si t est un sous-type de t', alors t`[ ]` est un sous-type de t'`[]` ;
- si C et C' sont des classes, si C étend C' et si C' est un sous-type de t, alors C est un sous-type de t ;
- si I et I' sont des interfaces, si I étend I' et si I' est un sous-type de t, alors I est un sous-type de t ;
- si C est une classe implémentant l'interface I, si I est un sous-type de t, alors C est un sous-type de t ;
- si t est un type de références, alors t est un sous-type de la classe `Object` ;
- si t`[]` est un type de tableaux, alors t`[]` est un sous-type de la classe `Object` et des interfaces `Cloneable` et `java.io.Serializable`.

1.10 ***Méthodes, classes et champs finaux*** Une méthode spécifiée `final` est *finale*, c'est-à-dire qu'elle ne peut pas être redéfinie (ou, si elle est statique, elle ne peut pas être masquée) dans une classe dérivée :

```
class A {
```

```
  final void f() { ... }
}

class B extends A {
  void f() { ... }         // INTERDIT
}
```

Une classe spécifiée `final` est finale, c'est-à-dire qu'aucune classe ne peut en être dérivée :

```
final class A { ... }
class B extends A { ... }  // INTERDIT
```

Méthodes et classes finales sont utilisées dans un but de sécurité (un utilisateur de la classe ne peut pas modifier son comportement) et dans un but d'optimisation, puisque la détermination de la méthode invoquée peut être faite dès la compilation, sans que le mécanisme de liaison tardive se produise.

Enfin, un champ peut être déclaré final afin d'interdire toute modification de sa valeur après initialisation :

```
final double n = 3;
n = 4;                    // INTERDIT
```

Une *constante de classe* est déclarée comme un champ `static final`:

```
static final double PI = 3.14;
```

Par exemple, l'expression `System.out.println("Hello")` est l'invocation de la méthode `println()` de la constante (membre statique final) `out` (de type `PrintStream`) de la classe `System`. Les constantes sont souvent désignées par un nom en lettres majuscules (comme `PI`) :

```
static final int EST = 0;
static final int NORD = 1;
static final int OUEST = 2;
static final int SUD = 3;
```

Ces constantes d'énumération sont particulièrement utiles dans une instruction d'aiguillage (`switch`).

1.11 Masquage Si un champ de la classe parente est à nouveau défini, avec le même nom, mais pas nécessairement avec le même type, dans la classe dérivée, le champ hérité de la classe parente est *masqué*.

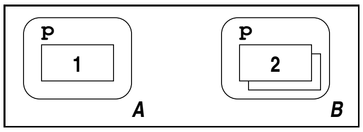

FIG. 1.11 – *Une instance de* `A` *et une instance de* `B`*, avec masquage.*

Le masquage d'un champ est plus une maladresse de programmation qu'une caractéristique utile : il vaut mieux s'abstenir d'utiliser cette possibilité. Par exemple, si l'on a

```
class A {
  int p=1;
}

class B extends A {
  int p=2;           // À ÉVITER
}
```

le champ `p` de `A` est masqué par la définition d'un autre champ de même nom par `B` (il en serait de même si `p` était déclaré d'un autre type dans `B`). Si `b` est une variable (ou une expression) de type `B`, l'expression `b.p` accède au champ `p` défini dans `B`, de valeur `2` (figure 1.11). Cependant, les champs de `A` peuvent être obtenus au moyen d'un transtypage : `(A)b` est une expression de type `A`, donc l'expression `((A)b).p` accède au champ de `A`, qui a pour valeur `1`.

Le mécanisme de la liaison tardive ne concerne ni les champs ni les méthodes de classe : seul le type de l'expression détermine le champ accédé, et non le type de la valeur. Ainsi, si l'on déclare

```
  A a = new B();
```

l'expression `a.p` accède au champ `p` défini dans `A`, pas à celui de `B`, bien que la valeur affectée à `a` soit une référence à une instance de `B`. Contrairement aux méthodes d'instance, qui peuvent être redéfinies, les méthodes de classe peuvent être masquées, de façon analogue aux champs, mais il n'est pas recommandé de le faire :

```
class A {
  static int f() {
    return 1;
  }
}

class B extends A {
  static int f() {   // À ÉVITER
    return 2;
  }
}
```

1.12 Types récursifs Une classe est définie récursivement, ou est récursive si l'un de ses champs a pour type cette classe. Plus généralement, on peut définir un ensemble de types mutuellement récursifs, de façon analogue aux fonctions mutuellement récursives. Cet auto-référencement est permis précisément parce que

– la valeur du champ est une référence à un objet et non l'objet lui-même ;
– une référence peut être nulle.

Les listes et les arbres sont des exemples classiques de types récursifs. Voici la définition (récursive) des arbres binaires étiquetés, qu'on appelera arbres par la suite : un *arbre* est soit l'arbre vide, soit formé d'une étiquette et de deux arbres, appelés fils gauche et fils droit. Nous supposons pour l'instant que l'étiquette est un entier. Un exemple d'arbre est représenté sur la figure 1.12.

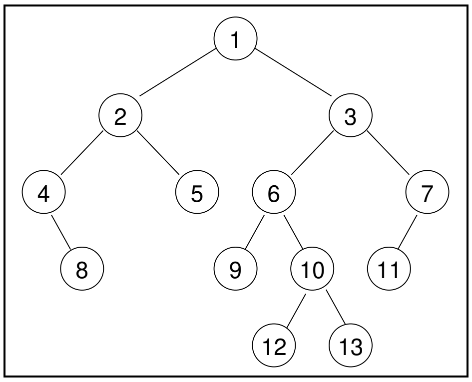

FIG. 1.12 – *Un arbre binaire étiqueté.*

Une traduction possible de cette définition mathématique en une classe est la suivante :

```
class ArbreBinaire {
  int étiquette;
  ArbreBinaire gauche;
  ArbreBinaire droit;
  ArbreBinaire(int étiquette,
               ArbreBinaire gauche,
               ArbreBinaire droit) {
    this.étiquette = étiquette;
    this.gauche = gauche;
    this.droit = droit;
  }
}
```

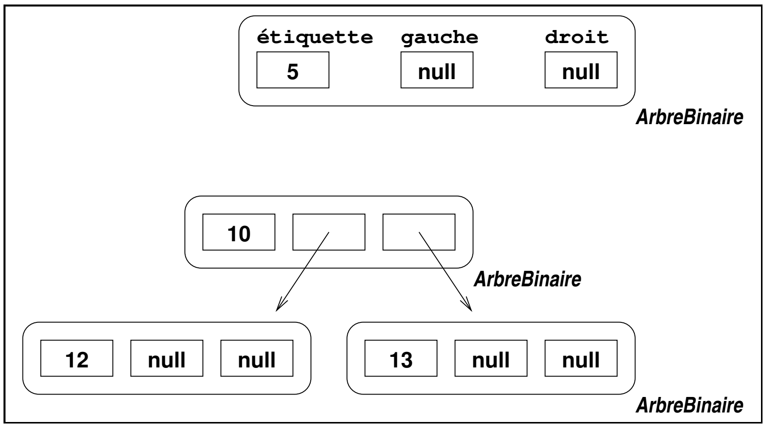

FIG. 1.13 – *Un arbre à un nœud* `new ArbreBinaire(5, null, null)`*; un arbre à trois nœuds.*

L'arbre vide est représenté par la valeur `null`, et un arbre non-vide est créé à l'aide du constructeur `ArbreBinaire`; par exemple, l'arbre de la figure 1.12 peut ainsi être construit par :

```
ArbreBinaire c =
  new ArbreBinaire(1,
    new ArbreBinaire(2,
      new ArbreBinaire(4,
        null,
        new ArbreBinaire(8, null, null)),
    new ArbreBinaire(5, null, null)),
  new ArbreBinaire(3,
    new ArbreBinaire(6,
      new ArbreBinaire(9, null, null),
      new ArbreBinaire(10,
        new ArbreBinaire(12, null, null),
        new ArbreBinaire(13, null, null))),
    new ArbreBinaire(7,
      new ArbreBinaire(11, null, null),
      null)));
```

Cette représentation des arbres est correcte et commode mais a un défaut : l'arbre vide n'est pas un objet. Par suite, on ne peut pas définir de méthode d'instance pour tester le fait qu'un arbre soit vide. On ne peut que recourir au test d'égalité à `null`. L'utilisation d'un type abstrait permettra d'y remédier.

1.13 Arbres et types abstraits Voici une utilisation d'un type abstrait et du sous-typage pour représenter tous les arbres, même l'arbre vide, par des objets. On commence par définir un type abstrait `ArbreBinaire`, puis deux sous-types concrets, un pour les arbres non-vides, et un pour les arbres vides :

```
abstract class ArbreBinaire {
  abstract boolean estVide();
}

class AVide extends ArbreBinaire {
  boolean estVide() {
    return true;
  }
}

class ACons extends ArbreBinaire {
  int étiquette;
  ArbreBinaire gauche, droit;

  ACons(int étiquette,
        ArbreBinaire gauche,
        ArbreBinaire droit) {
   this.étiquette = étiquette;
   this.gauche = gauche;
   this.droit = droit;
  }

  boolean estVide() {
    return false;
  }
}
```

On remarquera que la classe `ACons` n'est plus directement auto-référencée : les champs `gauche` et `droit` ne sont pas de type `ACons`, mais `ArbreBinaire`, de manière à permettre à un sous-arbre d'être vide. Une application pourra déclarer une variable `a` de type abstrait `ArbreBinaire`, et pourra l'initialiser à l'aide des constructeurs des classes concrètes :

```
    ArbreBinaire a =
      new ACons(1,
                new ACons(2, new AVide(), new AVide()),
                new ACons(3, new AVide(), new AVide()));
```

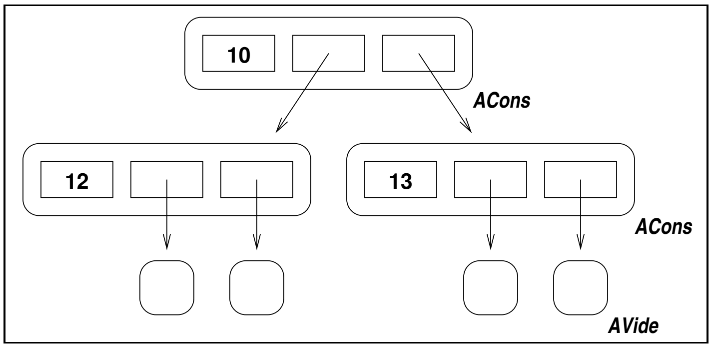

FIG. 1.14 – *Représentation d'un arbre binaire où l'arbre vide est un objet.*

L'exécution de l'appel `a.estVide()` conduit à invoquer la méthode `estVide()` de `ACons`, qui retourne `false`.

La définition de la classe `AVide` a cependant un défaut : chaque invocation du constructeur `AVide()` retourne un nouvel arbre vide, et tous les arbres vides retournés sont distincts. Ce défaut peut être corrigé en interdisant l'invocation de ce constructeur (il suffit de le rendre privé), et en s'assurant de l'existence d'une unique instance de la classe (ce qui s'obtient en en faisant une variable de classe `arbreVide`) : ceci est un exemple de classe *singleton* (voir § 3.5). Pour empêcher toute modification de cette variable, on la rend également privée et on y accède en lecture seulement par une méthode de classe `val()` :

```
class AVide extends ArbreBinaire {
  private static AVide arbreVide = new AVide();
  private AVide(){}
  static AVide val() {
    return arbreVide;
  }

  boolean estVide() {
    return true;
  }
}
```

```
    ArbreBinaire a =
      new ACons(1,
        new ACons(2, AVide.val(), AVide.val()),
        new ACons(3, AVide.val(), AVide.val()));
```

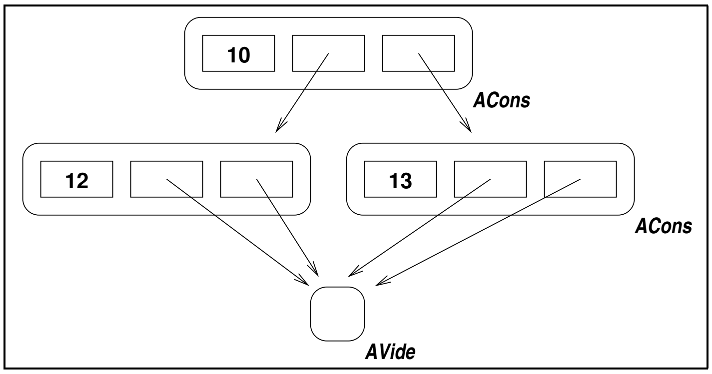

FIG. 1.15 – *Représentation d'un arbre binaire avec l'arbre vide comme l'unique instance de la classe* `AVide`.

1.14 *La classe* `Object` *et la généricité* Toute classe dérive implicitement de la classe `Object`. Ainsi, les deux définitions suivantes sont équivalentes :

```
class A { ... }
class A extends Object { ... }
```

Toute classe est donc un sous-type d'`Object` ; tout type de tableau est aussi un sous-type d'`Object`. Par suite, toute méthode déclarée avec un paramètre de type `Object` accepte en argument un tableau quelconque ou une instance de n'importe quelle classe, mais pas une valeur d'un type primitif. Voici quelques unes des méthodes qui sont définies par la classe `Object` :

```
public class Object {
  public String toString() { ... }
  public final Class getClass() { ... }
  public boolean equals(Object o) { ... }
  public int hashCode() { ... }
  protected Object clone()
    throws CloneNotSupportedException { ... }
  ...
}
```

La méthode `toString()` retourne une représentation de l'objet par une chaîne de caractères. C'est cette méthode qui est invoquée quand un objet figure en argument de la méthode `print()` ou `println()` ou dans une expression de concaténation de chaînes. Il est souvent utile de la redéfinir, pour obtenir une chaîne de caractères plus parlante. Par exemple, la classe `Point` pourrait la redéfinir ainsi :

```
class Point {
  // ...
  public String toString() {
    return "(" + x + ", " + y + ")";
  }
}
```

La méthode `getClass()` retourne une référence à une instance de la classe `Class` qui représente la classe de l'objet. Cette instance permet d'obtenir diverses informations sur cette classe (son nom, ses membres, sa surclasse, etc.), par exemple :

```
void printClassName(Object o) {
   System.out.println("La classe de " + o +
      " est " + o.getClass().getName());
}
```

La méthode `equals()` permet de tester l'égalité de deux objets. La méthode définie dans la classe `Object` est la plus discriminante possible :

`x.equals(y)` retourne `true` si et seulement si `x` et `y` sont des références au même objet, c'est-à-dire `x == y`. Par exemple, si la classe `Point` ne redéfinit pas `equals()`, la valeur de l'expression

```
new Point().equals(new Point())
```

qui compare deux instances différentes, retourne `false`. Il est donc utile de redéfinir `equals()`. Dans le cas de la classe `Point`, un point est égal à un objet `o` si `o` est un point et si les champs correspondants sont égaux. L'expression `o instanceof Point`, de type `boolean`, permet de tester si le type de l'objet désigné par `o` est un sous-type de `Point` :

```
class Point {
  // ...
  public boolean equals(Object o) {
    return o instanceof Point &&
      this.x == ((Point)o).x &&
      this.y == ((Point)o).y;
  }
}
```

Il serait tentant de définir une méthode qui compare seulement des instances de `Point` entre elles :

```
class Point {
  // ...
  boolean equals(Point p) { ... }
}
```

Cependant, ce ne serait pas une redéfinition de la méthode `equals()` de la classe `Object`, car elle n'a pas le même profil. Dans la situation suivante, l'égalité des deux points serait donc testée à l'aide de l'`equals` d'`Object` et non par celle de `Point` :

```
Object o = new Point(), p = new Point(1, 2);
boolean b = o.equals(p);
```

La méthode `clone()` permet de dupliquer un objet. Cette opération importante sera examinée plus loin.

Généricité On peut utiliser la classe `Object` pour transformer une structure de données en une structure générique, par exemple les arbres à noeuds de type `int` en arbres à noeuds de types référencés quelconques. Il suffit de remplacer `int` par `Object`. La définition de la classe devient :

```
class ACons extends ArbreBinaire {
  Object étiquette;
  ArbreBinaire gauche, droit;
```

```
  ACons(Object étiquette,
        ArbreBinaire gauche,
        ArbreBinaire droit) {
   this.étiquette = étiquette;
   this.gauche = gauche;
   this.droit = droit;
  }
  // ...
}
```

On obtient ainsi des arbres génériques hétérogènes (ce qui signifie que les objets aux noeuds ne sont pas nécessairement du même type). Pour passer un argument d'un type primitif à une méthode demandant un objet, on a recours aux classes enveloppantes `Integer`, `Double`, etc. :

```
  ArbreBinaire t = new ACons("toto",
    new ACons(new Integer(3), AVide.val(), AVide.val()),
    new ACons(new Double(0.4), AVide.val(), AVide.val()));
```

Rappelons que les chaînes littérales, comme `"toto"`, sont des objets de classe `String`.

1.15 ***Chaînes de caractères*** Les chaînes de caractères (appelées en anglais *strings*), sont des objets importants en programmation, intervenant dans les entrées-sorties et plus généralement dans la manipulation de fichiers textuels. Une *chaîne littérale* est une suite de caractères entourée par des apostrophes doubles « `"` » (en anglais *double quotes*), comme par exemple `"Knuth"` ; sa *longueur* est le nombre de caractères figurant entre les apostrophes, 5 pour `"Knuth"`. La chaîne vide est `""`, de longueur zéro.

En Java, les chaînes sont représentés par des objets de type `String`. Ce sont des objets de ce type qui sont construits par le compilateur quand il rencontre une chaîne littérale dans le programme source. Ces chaînes littérales peuvent servir à initialiser un champ ou une variable locale de type `String` :

```
   String s1 = "une chaîne de caractères";
   String s2 = "ceci est " + s1;
```

L'opérateur « + » permet de concaténer les chaînes. La longueur d'une chaîne est obtenue par la méthode `length()`, qui retourne un `int`. Chacun des caractères peut être obtenu par la méthode `charAt(int)`, le premier caractère d'une chaîne non vide `s` étant `s.charAt(0)`. D'autres fonctions et méthodes utiles sont disponibles (extraction d'une sous-chaîne, traduction en majuscules, recherche de sous-chaîne, etc). Voici un exemple de ces méthodes :

```
   String s1 = "une chaîne";
```

```
  char c = s1.charAt(1);              // c = 'n'
  String s2 = s1.substring(4, 10);    // s2 = "chaîne"
  String s3 = s1.toUpperCase();       // s3 = "UNE CHAÎNE"
  int n1 = s1.indexOf("ne");          // n1  = 1
  int n2 = s1.lastIndexOf("ne");      // n2  = 8
```

La classe `String` définit également des fonctions de nom `valueOf` qui réalisent la conversion des données de type primitif en `String` : ainsi, `String.valueOf(true)` convertit le booléen `true` en la chaîne `"true"`, et `String.valueOf(12)` convertit l'entier `12` en la chaîne `"12"`.

Les objets de type `String` sont non-modifiables. La classe `StringBuffer` permet la modification de la chaîne, par ajout et insertions de chaînes, ou par modification d'un caractère.

L'API de Java dispose d'une classe `StringTokenizer`, dans le paquet `java.util` qui permet de découper une chaîne en sous-chaînes séparées par des caractères délimiteurs. Cette classe a un constructeur à deux arguments : la chaîne à découper, et la chaîne des caractères délimiteurs. Par exemple,

```
import java.util.StringTokenizer;

class Token {
  public static void main(String[] args) {
    StringTokenizer st =
      new StringTokenizer("/usr/local/java", "/");
    while (st.hasMoreTokens()) {
      System.out.println(st.nextToken());
    }
  }
}
```

imprime sur la sortie standard :

```
usr
local
java
```

1.16 ***Clonage d'objets*** Rappelons que l'affectation d'un objet à une variable écrit dans cette variable une référence à l'objet, mais ne crée pas un nouvel objet. Si l'on veut dupliquer un objet, on peut recourir à la méthode `clone()`, définie dans la classe `Object`. Celle-ci commence par tester si la classe de l'objet implémente l'interface `Cloneable` (voir § 3.1) ; si c'est le cas, elle effectue une copie superficielle de l'objet, et sinon, elle déclenche l'exception `CloneNotSupportedException`. Le transtypage est nécessaire, car `clone()` retourne un `Object`, pas un `A` :

```
  A a1 = new A();
  A a2 = (A)a1.clone();
```

Tous les types de tableaux sont des sous-types d'`Object` (ainsi que des interfaces `Cloneable` et `java.io.Serializable`). Ainsi, un tableau quelconque peut être affecté à une variable de type `Object`. Les types de tableaux héritent de toutes les méthodes publiques de la classe `Object`, à l'exception de la méthode `clone()`, qui est redéfinie (donc utilisable), et qui réalise une copie d'un tableau :

```
int[] a = {1, 2, 3, 4};
int[] b = (int[]) a.clone();
```

Cette méthode est utile quand les objets d'une structure de données sont exactement représentés par l'ensemble des valeurs de leurs champs : ceci est le cas si tous les champs sont d'un type primitif (par exemple, la classe des nombres complexes, qui a deux champs de type `double`), ou pour un tableau d'éléments de type primitif. Dans d'autres cas, par exemple pour des arbres binaires, la copie d'une structure `ArbreBinaire` n'entraine pas une copie de l'arbre complet, mais seulement de la racine :

```
ArbreBinaire a =
  new ACons(0,
            new ACons(1, AVide.val(), AVide.val()),
            new ACons(2, AVide.val(), AVide.val()));
ArbreBinaire a1 = a;
ArbreBinaire a2 = (ArbreBinaire)a.clone();
```

où le type `ArbreBinaire` est une classe abstraite, dont les deux classes dérivées `AVide` et `ACons` représentent respectivement l'arbre vide et les arbres non vides.

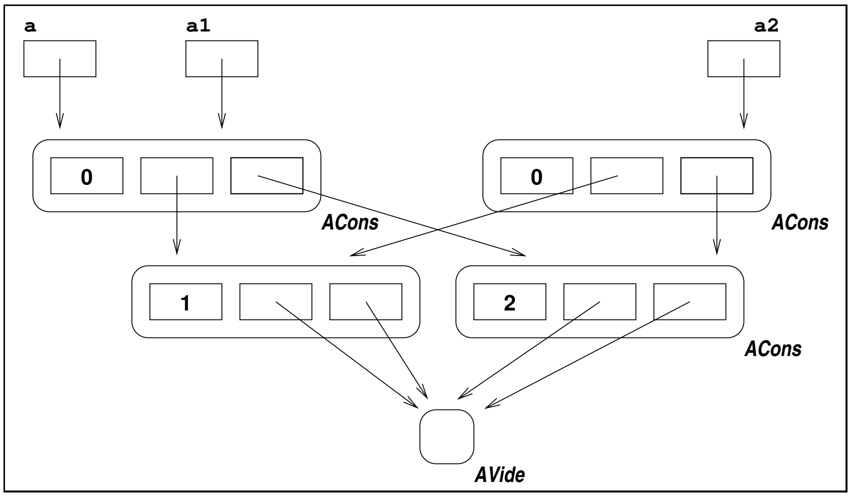

FIG. 1.16 – *Co-référencement par copie superficielle.*

Alors que `a1` et `a` désignent le même objet, la définition de `a2` crée un nouvel objet, mais les objets au deuxième niveau de l'arbre sont identiques : il y a partage d'objets par *co-référencement* (figure 1.16). Une modification d'un nœud de `a2` revient donc à une modification de `a` et de `a1`, ce qui n'est pas toujours souhaité.

Il faut donc redéfinir la méthode `clone()` pour la classe `Arbre` afin qu'elle parcoure l'arbre et effectue des copies de tous les objets, et pas seulement de celui qui figure à sa racine (figure 1.16). Un parcours en profondeur donne la fonction suivante :

```
abstract class ArbreBinaire {
  // ...
  public abstract Object clone();
}

final class AVide extends ArbreBinaire {
  // ...
  public Object clone() {
    return this;
  }
}

class ACons extends ArbreBinaire {
  // ...
  public Object clone() {
    ArbreBinaire a = null, b = null;
    if (gauche != null) a = (ArbreBinaire)gauche.clone();
    if (droite != null) b = (ArbreBinaire)droite.clone();
    return new ACons(étiquette, a, b);
  }
}
```

En général, on souhaite que pour un objet x, la valeur de `x == x.clone()` soit `false`. Le cas de la classe `AVide` est une exception, puisqu'elle est définie afin de n'admettre qu'une seule instance, l'arbre vide : un clone de l'arbre vide sera donc identique à l'arbre vide.

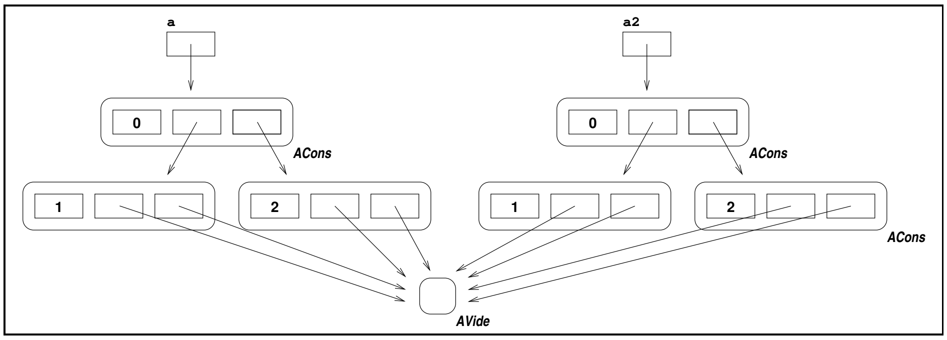

FIG. 1.17 – *Clonage d'un arbre.*

1.17 Tableaux d'objets Les éléments d'un tableau étant nécessairement des valeurs, il n'y a pas de tableaux d'objets, mais seulement des tableaux de références à des objets.

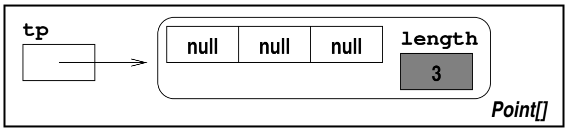

FIG. 1.18 – *Création d'un tableau de* `Point`*; aucune instance de* `Point` *n'est créée.*

Par exemple,

```
Point[] tp = new Point[3];
```

définit `tp` comme une référence à un tableau de longueur 3, dont les éléments sont des références à des instances de `Point` ; la valeur initiale de ces éléments est `null`, et aucune instance de `Point` n'est créée (figure 1.18). Il faut invoquer explicitement un constructeur pour chacun de ses éléments (figure 1.19) :

```
for (int i=0; i<tp.length; i++) tp[i] = new Point();
```

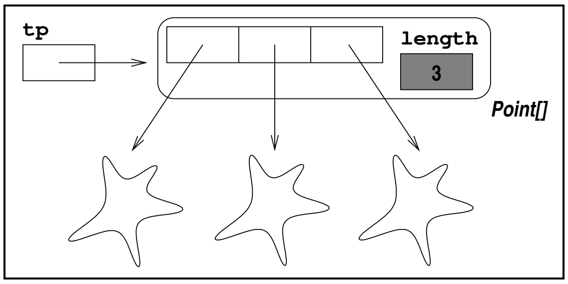

FIG. 1.19 – *Tableaux : création des éléments.*

On peut aussi initialiser explicitement les éléments d'un tableau de la façon suivante (dans ce cas, on ne doit pas spécifier la longueur dans l'expression de construction `new Point[]`):

```
Point[] tp = new Point[] {
               new Point(), new Point(1, 2), new Point(2, 1)
             };
```

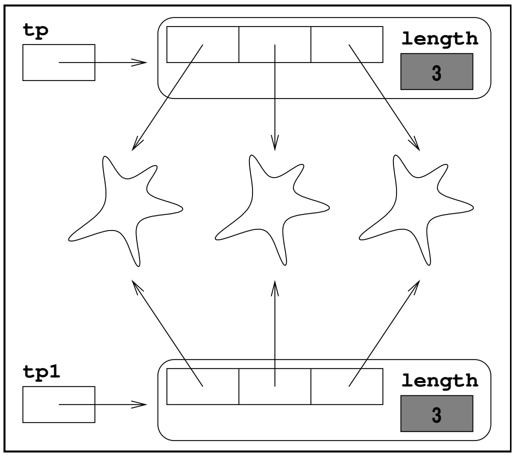

FIG. 1.20 – *Clonage d'un tableau de références : les objets sont partagés.*

Signalons que la méthode `clone()`, redéfinie pour les tableaux, réalise une copie superficielle d'un tableau (le tableau est copié, mais pas les objets auxquels réfèrent ses éléments, figure 1.20) :

```
Point[] tp1 = (Point[]) tp.clone();
```

Tableaux pluridimensionnels Tout ceci s'applique directement aux tableaux pluridimensionnels. Par exemple, un tableau bidimensionnel d'éléments de type t est simplement un tableau dont les éléments sont des références à des tableaux d'éléments de type t : le type de ces tableaux bidimensionnels est t`[][]`. La déclaration du nom du tableau ne crée aucun objet :

```
int[][] t;
```

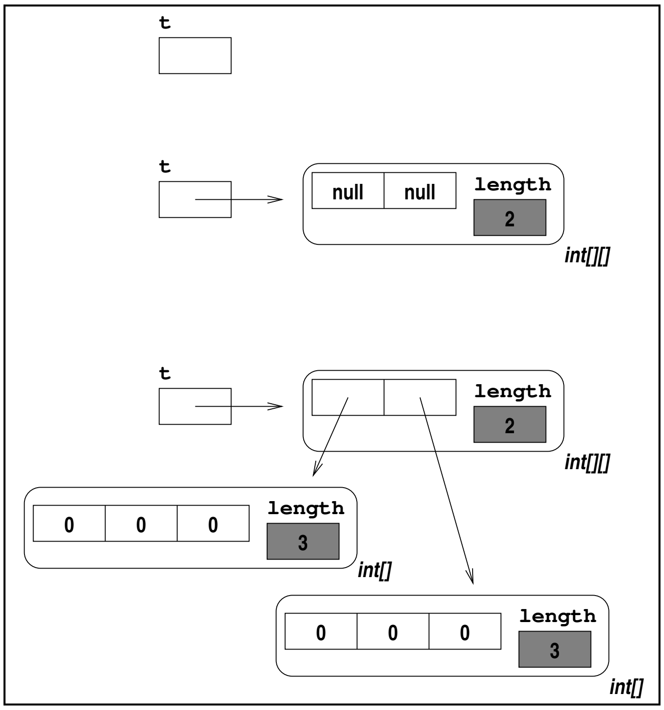

FIG. 1.21 – *Tableau bidimensionnel : déclaration d'une variable et création.*

Deux niveaux de création sont nécessaires :

```
t = new int[2][];
```

crée un tableau de longueur 2, dont les éléments sont des références à des tableaux de `int`, ces éléments étant initialisés à `null`. Ensuite, une boucle permet de créer chacune des lignes du tableau bidimensionnel, les éléments de chaque ligne étant initialisés à 0 (figure 1.21) :

```
for (int i=0; i<t.length; i++) t[i] = new int[3];
```

On accède à ses éléments par double indexation : `t[0][0]`, `t[0][1]` et `t[0][2]` pour la première ligne, `t[1][0]`, `t[1][1]` et `t[1][2]` pour la seconde. Comme `t` est un tableau, une indexation partielle est admise : les expressions `t[0]` et `t[1]` désignent des tableaux de longueur 3. Ces deux niveaux de création peuvent être effectués en une seule fois, par l'affectation suivante, qui crée un tableau rectangulaire à 2 lignes et 3 colonnes, dont tous les éléments sont nuls :

```
t = new int[2][3];
```

Il est aussi possible de créer des tableaux non rectangulaires, chaque ligne ayant son nombre d'éléments :

```
for (int i=0; i<t.length; i++) t[i] = new int[i+1];
```

Enfin, on peut créer un tableau bidimensionnel anonyme en initialisant explicitement ses éléments de la façon suivante :

```
t = new int[][] {
      {1, 2, 3},
      {4, 5, 6}
    };
```

Lors de la définition d'un nom, le constructeur `new int[][]` peut être omis :

```
int[][] t = {
  {1, 2, 3},
  {4, 5, 6}
};
```

Il est également facile d'initialiser un tableau anonyme avec des lignes de longueurs différentes :

```
t =  new int[][] {
  {1},
  {2, 3},
  {4, 5, 6}
};
```

Sous-typage Si t et t' sont des types de références, et si t est un sous-type de t', alors t`[]` est un sous-type de t'`[]`. Par exemple, un tableau de `PointColore` peut être affecté à une variable de type `Point`. Cette affectation ne transforme cependant pas le tableau de `PointColore` en un tableau de `Point`. Ainsi, si l'on tente d'affecter à l'un de ses éléments une instance de `Point`, l'exception `ArrayStoreException` est déclenchée :

```
class Point { int x, y; }
class PointColore extends Point { java.awt.Color couleur; }
class Test {
  public static void main(String[] args) {
    Point[] tp = new PointColore[4];
    tp[0] = new Point();    // ERREUR : ArrayStoreException
  }
}
```

En effet, un `Point` ne peut pas être affecté à un `PointColore`. Cette situation échappe au typage statique : le compilateur accepte ce programme, puisque la variable `tp[0]` est de type `Point` et le membre droit de l'affectation est du même type. La possibilité de cette erreur impose une vérification par la Machine Virtuelle Java, du type de la valeur affectée aux éléments d'un tableau de références.

Signalons que le sous-typage des tableaux ne concerne que les types de références et non les types primitifs : un tableau de `byte` ne peut pas être affecté à une variable de type `int[]`.

Un exemple : la triangulation de systèmes linéaires Les tableaux forment la structure de données de base pour de nombreux algorithmes numériques. En voici un exemple classique.

La méthode de Gauss transforme un système linéaire en un système triangulaire équivalent. L'idée est de résoudre la première équation en x_0, puis d'éliminer x_0 des équations suivantes, ensuite de résoudre la seconde équation en x_1, puis d'éliminer x_1 des équations suivantes, et ainsi de suite. Initialement, on a le système de N équations à N inconnues :

$$\sum_{j=0}^{N-1} a_{ij}x_j = b_i, \qquad 0 \leq i \leq N$$

Pour décrire l'algorithme, on posera $a^0_{ij} = a_{ij}$ et $b^0_i = b_i$. Après k itérations, on obtient le système :

$$\begin{array}{rcl}
a^k_{00}x_0 + a^k_{01}x_1 + \ldots + \quad a^k_{0k}x_k \quad + \ldots + \quad a^k_{0,N-1}x_{N-1} & = & b^k_0 \\
a^k_{11}x_1 + \ldots + \quad a^k_{1k}x_k \quad + \ldots + \quad a^k_{1,N-1}x_{N-1} & = & b^k_1 \\
\ddots \qquad\qquad & & \\
a^k_{kk}x_k \quad + \ldots + \quad a^k_{k,N-1}x_{N-1} & = & b^k_k \\
\vdots \qquad\qquad\qquad & & \\
a^k_{N-1,k}x_k + \ldots + a^k_{N-1,N-1}x_{N-1} & = & b^k_{N-1}
\end{array}$$

Les k premières équations ne seront plus modifiées. Il faut maintenant éliminer x_k. On n'utilise pas nécessairement l'équation $a^k_{kk}x_k + \ldots$ car il se peut que le coefficient a^k_{kk} soit nul ; on va donc chercher un indice l, avec $k \leq l \leq N-1$, tel que a^k_{lk} ait la plus grande valeur absolue (pour minimiser les problèmes d'arrondi). Si la valeur du pivot est non nulle, on échange les équations $a^k_{kk}x_k + \ldots$ et $a^k_{lk}x_k + \ldots$. Après cet échange, les $k+1$ premières équations ne seront plus modifiées. On a donc désormais $a^k_{kk} \neq 0$. On peut donc éliminer x_k de la ligne i $(k+1 \leq i \leq N-1)$ en ajoutant cette ligne à $-a^k_{ik}/a^k_{kk}$ fois la ligne k : on obtient ainsi les coefficients du système à l'issue de cette $k+1$-ème itération, par « pivotage » :

$$\begin{array}{rcll}
a^{k+1}_{ij} & = & a^k_{ij} - a^k_{ik}a^k_{kj}/a^k_{kk}, & k+1 \leq i,j \leq N-1 \\
a^{k+1}_{ik} & = & 0, & k+1 \leq i \leq N-1 \\
b^{k+1}_{i} & = & b^k_i - a^k_{ik}b^k_k/a^k_{kk}, & k+1 \leq i \leq N-1
\end{array}$$

Ceci se transcrit en la procédure `pivoter`. Si, à l'une des itérations, la valeur du pivot est nulle, c'est que le système n'est pas régulier, et la résolution s'arrête (c'est le rôle du `break` dans la fonction `gauss`). Si le système est régulier (c'est-à-dire de rang N), il est transformé en un système triangulaire supérieur en $N-1$ itérations. Il reste à résoudre ce système triangulaire. La complexité de cet algorithme est en $\Theta(N^3)$. Voici les fonctions implémentant cet algorithme :

```
class SystemeLineaire {
```

```
private static int pivot(double[][] a, int k) {
  // cherche l'indice d'un pivot dans k..dim-1
  int l = k;
  double max = Math.abs(a[k][k]);

  for (int i=k+1; i<dim; i++) {
    if (Math.abs(a[i][k]) > max) {
      l = i;
      max = Math.abs(a[i][k]);
    }
  }
  return l;
}

private static void échanger(double[][] a, double[] b,
                             int k, int l) {
  // échange les lignes k et l du système
  double[] ligne = a[k];
  a[k] = a[l];
  a[l] = ligne;
  double temp = b[k];
  b[k] = b[l];
  b[l] = temp;
}

private static void pivoter(double[][] a, double[] b,
                            int k) {
  // élimine x_k des lignes k+1..dim-1
  for (int i=k+1; i<dim; i++) {
    double q = a[i][k]/a[k][k];
    b[i] = b[i] - q * b[k];
    for (int j=k+1; j<dim; j++) {
      a[i][j] = a[i][j] - q * a[k][j];
    }
  }
}

static boolean gauss(double[][] a, double[] b) {

  // si le système "a x = b" est régulier, le transforme
  // en un système triangulaire en dim-1 itérations et
  // retourne true, sinon retourne false

  boolean inversible = false;
  for (int k=0; k<dim-1; k++) {
    int l = pivot(k);
    inversible = (Math.abs(a[l][k]) > EPS);
    if (inversible) {
      if (l > k) {
        échanger(k,l);
      }
      pivoter(k);
    } else break;
```

```
    }
    return inversible;
  }

  static double[] solutionTriangulaire(double[][] a,
                                       double[] b) {

  // calcule la solution x d'un système triangulaire
  // supérieur "a x = b" et retourne x s'il est régulier,
  // et déclenche une exception sinon

    double[] x = new double[dim];
    for (int i=dim-1; i>=0; i--) {
      if (Math.abs(a[i][i])<EPS) {
        throw new ArithmeticException("système irrégulier");
      } else {
        double v = b[i];
        for (int j=i+1; j<dim; j++) {
          v = v - a[i][j]*x[j];
        }
        x[i] = v/a[i][i];
      }
    }
    return x;
  }
}
```

1.18 La fonction **`main`** Si un programme est une collection de types, il doit avoir au moins une classe contenant une définition de fonction principale, de nom `main` ; l'exécution du programme commence par cette fonction. C'est elle qui invoquera éventuellement les autres fonctions. Sa définition minimale est la suivante :

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

La fonction `main()` peut accéder à la *ligne de commande* passée à l'interprète de commandes pour faire exécuter le programme. Cette ligne de commande a pour objet de démarrer une Machine Virtuelle Java, de désigner la classe principale, et de spécifier la valeur d'un certain nombre de paramètres du programme. Ces paramètres sont représentés par un tableau de chaînes de caractères, conventionnellement appelé `args`, qui est l'unique paramètre de la fonction `main()`. Lors de l'invocation de `main()`, les éléments de ce tableau, `args[0]`, ..., `args[args.length-1]`, sont initialisés par les chaînes de caractères (mots séparés par des espaces) figurant sur la ligne de commande après la classe principale[5] Par exemple, le pro-

5. Signalons qu'en C, la fonction principale, également appelée `main`, retourne un `int`, et qu'elle accède à la ligne de commande selon une autre convention : `args[0]` est le nom du programme et les `args[`*i*`]` suivants sont les paramètres du programme.

gramme (traditionnellement appelé echo) qui recopie sur la sortie standard les arguments passés sur sa ligne de commande peut être programmé ainsi :

```
class Echo {
  public static void main(String[] args) {
    for (int i=0; i<args.length; i++) {
      System.out.print(args[i] + " ");
    }
    System.out.println();
  }
}
```

Après avoir compilé cette classe, l'exécution de la commande

```
linux% java Echo un deux trois
```

provoque l'invocation de la la fonction `main()` de la classe `Echo`, et l'initialisation de son paramètre `args` au tableau `{ "un", "deux", "trois" }`, puis produit sur la sortie standard (l'écran) :

```
un deux trois
```

Il est souvent nécessaire de convertir les `args[i]`, qui sont toujours de type `String`, en d'autres types de données. `Integer.parseInt(s)` permet de convertir l'argument `s`, une chaîne qui est la notation d'un entier, en cet entier. Une fonction analogue `Double.parseDouble(s)` est disponible pour les nombres flottants. Par exemple, le programme qui additionne des entiers peut s'écrire la façon suivante.

```
class Add {
  public static void main(String[] args) {
    int somme = 0;
    for (int i=0; i < args.length ; i++) {
      somme = somme + Integer.parseInt(args[i]);
    }
    System.out.println(somme);
  }
}
```

On pourra exécuter

```
linux% java Add 12 23 34
```

et obtenir `69` comme réponse.

La méthode `main()` étant statique, elle ne peut accéder aux membres des instances de sa classe. C'est pourquoi, la classe principale se trouve souvent être instanciée dans la méthode `main()`, dans l'unique but de pouvoir accéder aux membres non-statiques de la classe :

```
class Principale {
  void méthode() { ... }
  public static void main(String[] args) {
    Principale p = new Principale();

    p.méthode();
  }
}
```

L'autre solution est de ne définir dans la classe principale que des membres statiques :

```
class Principale {
  static void méthode() { ... }
  public static void main(String[] args) {

    méthode();
  }
}
```

Rappelons que l'on accède à un membre statique d'une classe en préfixant le nom du membre par le nom de la classe. Par exemple, la classe `Math`, qui n'est pas destinée à être instanciée, ne comporte que des membres statiques, dont les constantes `E` et `PI`, et toutes les fonctions mathématiques usuelles ; on invoquera ainsi `Math.cos(Math.PI)`.

1.19 Flots d'instructions : les threads La programmation à objets provient du besoin d'exprimer des interactions. Jusqu'à présent, deux objets peuvent interagir quand l'un invoque une méthode de l'autre objet : cette forme d'interaction est ainsi limitée à la réponse d'un objet à un autre objet. Un *agent* est un objet doté d'un *comportement* qui lui permet d'agir sans qu'il soit sollicité par d'autres objets. Jusqu'à présent, seule la fonction `main()` de la classe principale a cette capacité, mais, étant statique, elle n'est pas associée à un objet. La fonction `main()` définit le flot d'instructions exécutées par le programme. L'idée est de créer des agents, ayant chacun son propre flot d'instructions.

Un flot d'instructions est modélisé en Java par un *thread*, ou *unité séquentielle d'exécution* (ou encore *brin*, traduction littérale de thread, ou *processus léger*). Quand une application est démarrée, l'environnement Java crée un thread *principal* exécutant la fonction `main()` de l'application. D'autres threads peuvent être créés, soit par l'environnement d'exécution, soit par le programme. Chacun de ces threads exécute une fonction, en *concurrence* avec les autres ; ils communiquent entre eux via des objets partagés. C'est de la programmation multithread.

Un thread est créé par instanciation de la classe `Thread`, ou d'une classe dérivée de `Thread`. La classe `Thread` définit une méthode `run()` qui ne fait rien. C'est en redéfinissant cette méthode dans une classe dérivée que l'on donne un comportement à un objet :

```
class T extends Thread {

  public void run() { ... }

}

 ... new T().start(); ...
```

Un thread, une fois créé, est dans son état *initial*. La méthode `start()`, qui retourne immédiatement, le fait passer dans l'état *actif*, dans lequel il peut être effectivement exécuté (sur un monoprocesseur, il sera exécuté en temps partagé avec les autres unités). Il faut noter que la méthode `run()` n'est jamais appelée explicitement dans le programme (exactement de la même façon que la méthode `main()` d'une application). L'exécution d'un thread peut être suspendue de différentes façons, puis reprise. Le thread passe dans l'état *terminé* quand `run()` termine. L'exemple suivant montre deux agents, l'un émettant un `'a'`, l'autre un `'b'` à des instants aléatoires :

```
class Concurrence {
  public static void main(String[] args) {
    Agent
      a = new Agent('a'),
      b = new Agent('b');
    a.start();
    b.start();
  }
}
```

Chaque agent est une instance de la classe suivante, qui redéfinit la méthode `run()` de `Thread` (cette méthode doit recourir au mécanisme de traitement d'exception, `try ... catch`, qui sera expliqué au § 3.21) :

```
class Agent extends Thread {
  char c;
  Agent(char c) {
    this.c = c;
  }

public void run() {
    while(true) {
      System.out.print(c);
      try {
        Thread.sleep((long)(Math.random()*1000));
      } catch (InterruptedException e) {}
    }
```

```
  }
}
```

L'exécution de ce programme affiche une suite de caractères, résultat de l'entrelacement des comportements de chaque agent :

```
baababab babbaaababbabbbbaabbabaaababbababababaa ...
```

La programmation multithread est un aspect important de la programmation des systèmes logiciels contemporains.

1.20 *La Machine Virtuelle Java* Le résultat de la compilation d'un programme Java (plus exactement, d'unités de compilation Java, c'est-à-dire les fichiers en `.java`) est un ensemble de *fichiers de classe*, un par type défini dans le programme ; le nom d'un tel fichier est formé par le nom du type (classe ou interface) suffixé par `.class`. Le format des fichiers de classe est indépendant de la plate-forme matérielle et logicielle : que l'on soit sur un Pentium, un PowerPC, un Sparc ou sur un Alpha, sous Windows, MacOS, Solaris ou Linux, etc. — dans un navigateur Web, sur un Palm Pilot, voire sur une carte à puce (avec cependant un prétraitement et quelques restrictions, vu la limitation des ressources de ces cartes). Contrairement à d'autres langages (notamment C et C++), Java n'est pas compilé en langage machine : cette indépendance garantit la portabilité des applications écrites en Java, de la même façon qu'une carte SIM peut être insérée dans n'importe quel téléphone GSM.

Une application Java peut être exécutée sur une plate-forme quelconque à la seule condition que celle-ci dispose d'une *Machine Virtuelle Java*, ou JVM. C'est le cas d'un navigateur Web capable d'exécuter des applettes Java, ou des plates-formes comportant un environnement de développement, par exemple le *Java Development Kit* (JDK), ou simplement un environnement d'exécution comme le *Java Runtime Environment* (JRE), l'un et l'autre produits par Sun. Dans ce dernier cas, une JVM est démarrée en exécutant le programme `java` et en lui fournissant le nom d'une classe (et éventuellement, des arguments) :

```
linux% java Application
```

Ceci suppose que la classe Application est une classe principale, c'est-à-dire comporte la définition d'une méthode `main()`. L'exécution de cette commande déclenche une série d'étapes, qui sont pour l'essentiel :

- le démarrage d'une JVM ;
- le chargement de la classe `Application` ;
- la liaison de cette classe aux autres classes ;

– son initialisation ;
– l'exécution de sa méthode `main()`.

Le *chargement* consiste à chercher un fichier de nom `Application.class`, à le lire, et à incorporer à la JVM une représentation binaire de la classe `Application`. Cette étape peut échouer si un fichier de classe contenant la définition de la classe `Application` n'est pas trouvé, ce qui provoque un message signalant l'erreur `NoClassDefFoundError`. Elle peut aussi échouer si le fichier trouvé n'a pas le format d'un fichier de classe : le message signale l'erreur `ClassFormatError`.

L'étape de *liaison* comporte trois phases. Elle commence par une *vérification* de la classe chargée ; en principe, cette étape serait inutile si l'utilisateur était sûr du compilateur ayant produit le fichier de classe, ce qui n'est pas le cas s'il utilise une classe dont il ignore la provenance. La vérification peut échouer et produire un message signalant l'erreur `VerifyError`. La deuxième phase est la *préparation*, qui crée les champs statiques de la classe, les initialise à leurs valeurs par défaut et met en place certaines structures de données utiles à l'exécution (par exemple, la table des méthodes).

La troisième phase est la *résolution* : elle remplace les références externes (à d'autres types, à leurs champs, méthodes et constructeurs) qui apparaissent dans le fichier de classe sous forme de noms par des références internes à la JVM, plus efficaces. Cette phase peut se produire juste après la vérification, ou bien être différée jusqu'à l'exécution, quand une référence externe est effectivement utilisée. Dans le premier cas, elle provoque le chargement d'autres classes, leur vérification, préparation, résolution, etc. La résolution peut échouer et produire un message signalant diverses erreurs, par exemple `IllegalAccessError`, `InstantiationError`, `NoSuchFieldError`, `NoSuchMethodError` : ces erreurs sont généralement dues à des modifications de la classe référencée faites après la compilation de la classe en cours de résolution.

L'*initialisation* d'une classe consiste principalement à initialiser ses champs statiques, mais elle requiert que ses surclasses aient été initialisées auparavant. Ceci induit le chargement de ses surclasses, leur vérification, etc.

Une fois ces opérations réalisées, la JVM crée un *thread principal* qui exécute la méthode `main()`. D'autres threads peuvent être créés en cours d'exécution. L'espace mémoire de la JVM est composé de plusieurs zones (voir figure § 1.22) :

– la *zone des méthodes*, qui contient le code des méthodes et des constructeurs, ainsi que des informations sur la structure de chaque classe, et notamment sa *table des symboles* ;
– le *tas* qui contient les objets (instances de classe et tableaux) ;

– la *pile*, propre à chaque thread, qui contient les cadres d'invocation des méthodes en cours d'exécution ;

– le *compteur d'instruction*, propre à chaque thread, qui désigne l'instruction en cours d'exécution.

Si la zone des méthodes, le tas ou la pile n'est pas assez grande et ne peut être agrandie, l'erreur `OutOfMemoryError` est signalée. Cette organisation est voisine des processus d'exécution des programmes Pascal, C, C++, mais pas de Fortran 77. Les implémentations classiques de Fortran 77 ne disposent ni de tas ni de pile, le processus est uniquement constitué de données statiques (de ce fait, la récursivité n'est pas permise).

Variables et objets ont une existence dans l'espace et dans le temps. Dans l'espace, c'est une zone mémoire qui a été attribuée par *allocation*, dans le temps c'est la durée de vie, qui s'étend de l'allocation à la dés-allocation de cette zone mémoire. La durée de vie d'une donnée est une portion de la durée d'existence d'une JVM.

Une variable locale, c'est-à-dire résultant d'une définition à l'intérieur d'une méthode, est allouée sur la pile et a pour durée de vie celle d'un cadre d'invocation, c'est-à-dire d'une invocation jusqu'au retour de cette méthode. Si une méthode est invoquée plusieurs fois au cours de l'exécution du même programme, chaque invocation donne lieu à une nouvelle instance de chaque variable locale.

Un objet créé par l'opérateur `new` est alloué sur le tas ; il a une durée de vie qui s'étend depuis le retour de `new` jusqu'à une dés-allocation de cet objet qui est réalisée par le système quand l'objet cesse d'être utilisable.

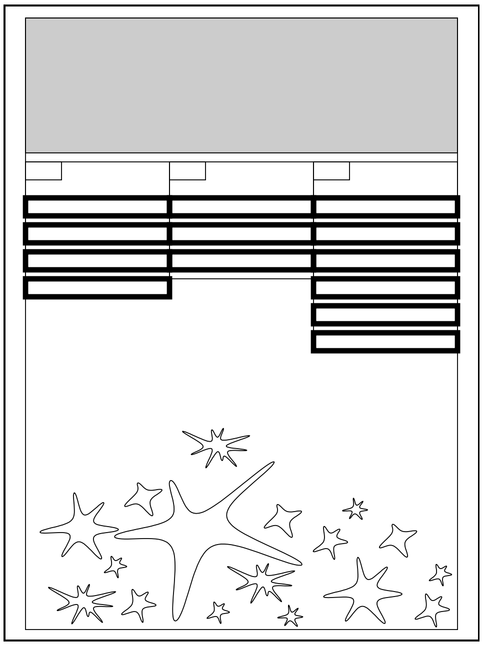

FIG. 1.22 – *Organisation d'un processus ; de haut en bas : la zone des méthodes, le compteur d'instruction et la pile de chaque thread (trois threads sont figurés), et le tas où résident les objets.*

2

Algorithmes

Le premier [précepte] était de ne recevoir jamais aucune chose pour vraie que je ne la connusse évidemment être telle; c'est-à-dire, d'éviter soigneusement la précipitation et la prévention, et de ne comprendre rien de plus en mes jugements que ce qui se présenterait si clairement et si distinctement à mon esprit, que je n'eusse aucune occasion de le mettre en doute.
Le second, de diviser chacune des difficultés que j'examinerais, en autant de parcelles qu'il se pourrait, et qu'il serait requis pour les mieux résoudre.
Le troisième, de 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.
Et le dernier, de faire partout des dénombrements si entiers et des revues si générales, que je fusse assuré de ne rien omettre. DESCARTES

Le mot algorithme provient du nom d'un mathématicien persan du IXe siècle, Mohammed ibn-Musa al-Khuwarizmi, dont le livre d'arithmétique, utilisant la numération arabe et des règles de calcul, montra l'inutilité des tables et abaques. Autant dire que les algorithmes, au sens de solution algorithmique à des problèmes, sont connus et utilisés bien avant les débuts de l'informatique. La notion actuelle d'algorithme a été élaborée par les logiciens des années 1930 (Herbrand, Gödel, Church et Turing), eux-mêmes précurseurs de l'informatique.

L'algorithmique est la branche de l'informatique qui traite des algorithmes. On peut voir cette branche comme l'économie de l'informatique, ou comment mettre en œuvre des ressources de calcul pour résoudre des problèmes au moindre coût. Comment évaluer ce coût ? Le temps d'exécution d'un programme peut être mesuré expérimentalement. Sous Linux, la commande `time` affiche plusieurs estimations de ce temps, en secondes ; par exemple, la commande `java Simulation 1000000` est exécutée et son temps d'exécution est évalué en soumettant la commande suivante au shell :

```
linux% time java Simulation 1000000
13.240u 0.870s 0:19.30 73.1% 0+853k 0+16io 0pf+0w
```

- le premier nombre (suffixé par `u`, pour *user*) est le temps d'exécution des instructions du programme ;
- le second (suffixé par `s`, pour *system*) est le temps d'exécution d'instructions du système nécessaires à l'exécution du programme, mais qui ne figurent pas dans le programme ;
- le troisième est le temps écoulé entre le début et la fin de l'exécution du programme ; ce temps pourrait être mesuré par un utilisateur muni d'un chronomètre ; il dépend non seulement du programme, mais aussi de la charge du système, due notamment aux autres programmes en cours d'exécution ;
- le suivant est le pourcentage du temps consacré au programme, c'est-à-dire $\mathtt{u} + \mathtt{s}$ rapporté au temps écoulé ;
- les trois derniers nombres sont des évaluations des accès à la mémoire (espace utilisé, entrées-sorties, pages).

Il est important d'être capable d'évaluer a priori ce temps `u`, sans devoir exécuter le programme. Si l'on dispose d'information sur la machine, on a l'estimation suivante :

$$\begin{aligned}\text{temps d'exécution} \quad = \quad & \text{nombre d'instructions exécutées du programme} \\ & \times \text{nombre moyen de cycles/instruction} \\ & \times \text{durée d'un cycle d'horloge}\end{aligned}$$

Le ou la programmeur-e peut seulement agir sur le nombre d'instructions ; les deux autres facteurs de cette formule sont respectivement du ressort de l'architecte de la machine et de l'électronicien. Les instructions mentionnées dans cette formule ne sont pas celles du langage source, mais celles de la machine ; en outre, il y a des instructions exécutées qui ne proviennent pas du programme source (par exemple, des opérations de récupération de la mémoire inutilisée). Cependant, une estimation du temps d'exécution peut être obtenue en analysant le programme source. L'usage est de dénombrer certaines instructions significatives : nombre d'opérations arithmétiques dans un algorithme numérique, nombre d'itérations ou d'invocations récursives, nombre de comparaisons dans un algorithme de recherche d'un élément dans une table. Cette évaluation se fait en fonction d'une grandeur caractéristique du problème : valeur d'une donnée, taille d'un tableau, profondeur d'un arbre, etc. Plutôt que de temps d'exécution, on parlera d'une *mesure de complexité*. En outre, il s'agit souvent de comparer plusieurs algorithmes entre eux, indépendamment des machines auxquelles ils sont destinés. On s'intéresse donc surtout à l'ordre de grandeur d'une mesure de complexité, c'est-à-dire à des estimations asymptotiques. On rencontrera notamment des algorithmes *linéaires* quand la complexité est en $\Theta(n)$, *logarithmiques*, en $\Theta(\log n)$, *quadratiques*, en $\Theta(n^2)$, etc. Rappelons que $\Theta(f)$ est l'ensemble des fonctions g pour lesquelles il existe des entiers N, a et b positifs tels que $af(n) \leq g(n) \leq bf(n)$ pour tout $n \geq N$.

2.1 Codes Les systèmes de traitement de l'information ont toujours utilisé des techniques de codage, bien avant l'informatique : les différents systèmes d'écriture, le Braille, le Morse, la sténo. Avec la généralisation de la numérisation de l'information sous toutes ses formes, il existe maintenant une grande variété de codes, adaptés à ses divers supports de transmission et de stockage (cables coaxiaux, ondes hertziennes, disques magnétiques, CD-ROM, Mini-Disc, etc). Les codes sont conçus, outre la simple représentation de l'information, pour satisfaire à certaines exigences sur l'information codée :

– réduire sa taille, c'est la *compression de données* ;
– la rendre incompréhensible par des tiers, par chiffrement, c'est la *cryptographie* ;
– permettre de reconstituer l'information initiale, même si elle a été altérée par un bruit, c'est la *correction d'erreurs*.

Le schéma général est indiqué par la figure 2.1.

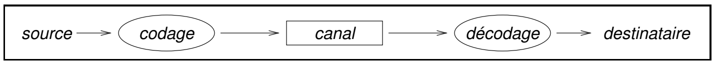

FIG. 2.1 – *Transmission de l'information.*

Les codes font intervenir plusieurs notions, que nous définissons ici brièvement. Un langage est un ensemble de mots, un mot est une suite finie de symboles, un symbole est un élément d'un alphabet, et un alphabet est un ensemble quelconque, de préférence fini. Quand on s'intéresse à des textes, on travaille avec un alphabet formé d'un *jeu de caractères* : par exemple l'alphabet formé des 26 lettres de l'alphabet latin, en minuscules et en majuscules, des 10 chiffres, des signes de ponctuation et de quelques symboles usuels (<, `&`, `@`, etc) que l'on trouve sur un clavier d'ordinateur anglais (un clavier français devrait y ajouter les lettres accentuées). Quand on s'intéresse à ce que contient la mémoire d'un ordinateur, c'est l'alphabet binaire, formé des deux symboles 0 et 1 qui est à considérer. Le traitement des langages a maintenant des applications très au-delà des besoins de l'informatique. Par exemple, l'analyse du génome humain se fait à l'aide des mêmes techniques et requiert des algorithmes élaborés, vu la taille des données en jeu.

Soit A un ensemble dont les éléments seront appelés des *symboles* ; une suite finie $(a_1, \ldots, a_m)$ de symboles est appelée un *mot* sur l'*alphabet* A ; un tel mot est aussi noté $a_1 \ldots a_m$, et l'entier m est sa *longueur*. L'ensemble de tous les mots sur l'alphabet A est noté $A^\star$. Le produit (de *concaténation*) des mots $u = a_1 \ldots a_m$ et $v = b_1 \ldots b_n$ est le mot $uv = a_1 \ldots a_m b_1 \ldots b_n$, de longueur $m + n$. C'est une opération associative, pour laquelle le mot vide ε, de longueur nulle, est élément neutre. Un *langage* sur un alphabet A est un sous-ensemble de $A^\star$.

Un code est un langage C sur un certain alphabet A, muni d'un codage, qui est une application d'un certain ensemble S vers C et d'un décodage qui réalise l'opération inverse. Souvent, l'ensemble codé S est lui-même l'alphabet d'un langage, dit alphabet source et le codage s'étend naturellement en un codage de $S^\star$. Les codes les plus simples sont de *longueur constante*. Le plus connu est le code ASCII, qui contient tous les mots de longueur 7 sur l'alphabet binaire, autrement dit, les 128 ($= 2^7$) mots de 7 bits, et permet ainsi de représenter les caractères usuels anglais. Par exemple, le caractère `A` est codé par le mot binaire `1000001`, écrit plus commodément `0x41` en notation hexadécimale (le préfixe `0x` signale un nombre hexadécimal, c'est-à-dire en base 16). Le codage d'un mot de l'alphabet source se fait à l'aide d'une table, structure de données associant à chaque symbole de l'alphabet source son code :

Symbole	Code
`A`	`1000001`
`B`	`1000010`
`C`	`1000011`
...	...

Il existe des codes ASCII étendus, à 8 bits (un octet), qui représentent aussi les lettres accentuées, par exemple le code ISO8859-1, appelé aussi latin-1, adapté à la plupart des langues de l'Europe de l'ouest et du nord ; le caractère `À` est représenté dans ce code par le mot binaire `11000000`, ou `0xC0`. Un tel code ne peut représenter que 256 ($= 2^8$) caractères, ce qui est insuffisant. Par souci d'internationalisation, les caractères de Java (comme de la plupart des langages conçus pour l'Internet : XML, HTML4, ECMAScript, WML, etc.) utilisent le code *Unicode*, sur 16 bits. Celui-ci permet de représenter au plus 65 536 caractères, ce qui suffit pour un grand nombre de langues (mais pas vraiment pour toutes) ; par exemple, le nouveau symbole de l'euro a récemment reçu le code `0x20AC` (en binaire, `00100000 10101100`). Notons que toutes les valeurs des types primitifs de Java sont également représentées à l'aide de codes de longueur constante sur l'alphabet $\{0, 1\}$: des mots de 32 bits pour le type `int`, de 64 bits pour `double`, etc.

Codes de longueur variable Le Morse, code la lettre *E*, la plus fréquente, par un '`.`' et la lettre *Y*, plus rare, par '`-.--`' : c'est un exemple de code de *longueur variable*, ce qui permet de représenter les lettres les plus fréquentes par des mots plus courts. Une propriété importante est l'unicité du décodage, problème qui ne se pose pas pour les codes de longueur constante. Il peut être résolu, mais de façon trop coûteuse, quand un symbole spécial sépare deux mots successifs du code (le blanc dans le cas du Morse). On peut ne pas recourir à cette technique si aucun mot du code n'est le préfixe d'un autre mot du code (par exemple `01` étant préfixe de `0111`, ces deux mots ne peuvent appartenir simultanément à un code préfixe) ; un code présentant cette propriété est appelé un *code préfixe*. Un exemple de tel code, dans la vie courante, est l'ensemble des numéros de téléphone : que le 18 soit le numéro d'appel des pompiers implique qu'aucun autre numéro ne commence par 18. En informatique, un exemple de code de longueur variable préfixe est le code UTF-8, qui permet de représenter les caractères Unicode sous forme d'un nombre variable d'octets (de 1 à 3) :

– l'intervalle `0x0000 ... 0x007F`, formé des 128 caractères du code ASCII, est codé sur un octet commençant par le bit `0` ;
– l'intervalle `0x0080 ... 0x07FF` est codé sur deux octets, le premier commençant par les bits `110`, le second par `10` ;
– l'intervalle `0x0800 ... 0xFFFF` est codé sur trois octets, le premier commençant par `1110`, les deux autres par `10`.

Le décodage d'un code préfixe se fait à l'aide d'une autre structure de données, un arbre, tel que celui de la figure 2.2. Il suffit de lire les symboles successifs du texte codé : si c'est un `0`, on suit la branche de gauche, si c'est un `1`, celle de droite ; quand on arrive à une feuille de l'arbre, on a décodé une lettre de l'alphabet source ; on remonte alors à la racine et on continue avec le symbole binaire suivant. Ainsi, avec l'arbre de la figure 2.2, `010011110` se décode en `BAC`, sans qu'il soit nécessaire de le décomposer a priori en `010 / 0111 / 10`.

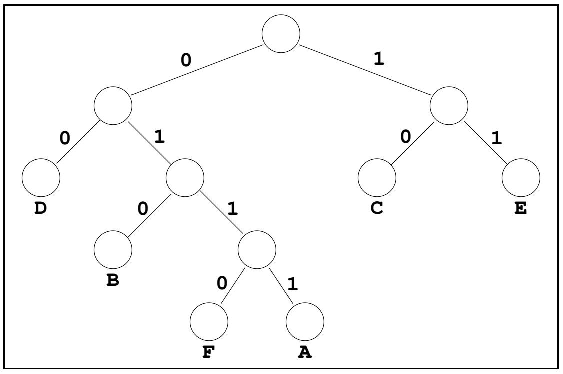

FIG. 2.2 – *Arbre de décodage.*

Nous présenterons trois algorithmes de codage : le codage de Huffman, pour la compression, le codage RSA, pour la confidentialité et le codage de Hamming, pour la correction d'erreurs.

2.2 Compression : le code de Huffman Quand il s'agit de transmettre de l'information sur un canal non bruité, l'objectif prioritaire est de minimiser la taille de la représentation de l'information : c'est le problème de la *compression de données*. Le code de HUFFMAN (1952) est un code de longueur variable optimal, c'est-à-dire tel que la longueur moyenne d'un texte codé soit minimale. On observe ainsi des réductions de taille de l'ordre de 20 à 90%. Ce code est largement utilisé, souvent combiné avec d'autres méthodes de compression.

L'algorithme de Huffman met en œuvre plusieurs structures de données. Il opère sur une *forêt*, c'est-à-dire un ensemble d'arbres. Ceux-ci sont plus précisément des arbres binaires pondérés . Tout nœud est affecté d'un poids, qui est la somme des poids de ses enfants ; le poids de l'arbre est, par définition, le poids de sa racine. La forêt initiale est formée d'un arbre à un nœud pour chaque symbole de l'alphabet source, dont le poids est la probabilité d'occurence de ce symbole. La forêt finale est formée d'un unique arbre, de poids 1, qui est l'arbre de décodage du code. L'algorithme est de type *glouton* : il choisit à chaque étape les deux arbres de poids minimaux, soit a_1 et a_2, et les remplace par l'arbre binaire pondéré d'enfants a_1 et a_2 (ayant donc comme poids la somme du poids de a_1 et du poids de a_2). La structure de données contenant les arbres doit permettre les opérations suivantes : déterminer l'arbre de poids minimum et l'extraire de la forêt, insérer un arbre dans la forêt. La figure 2.3 représente les étapes de la construction d'un

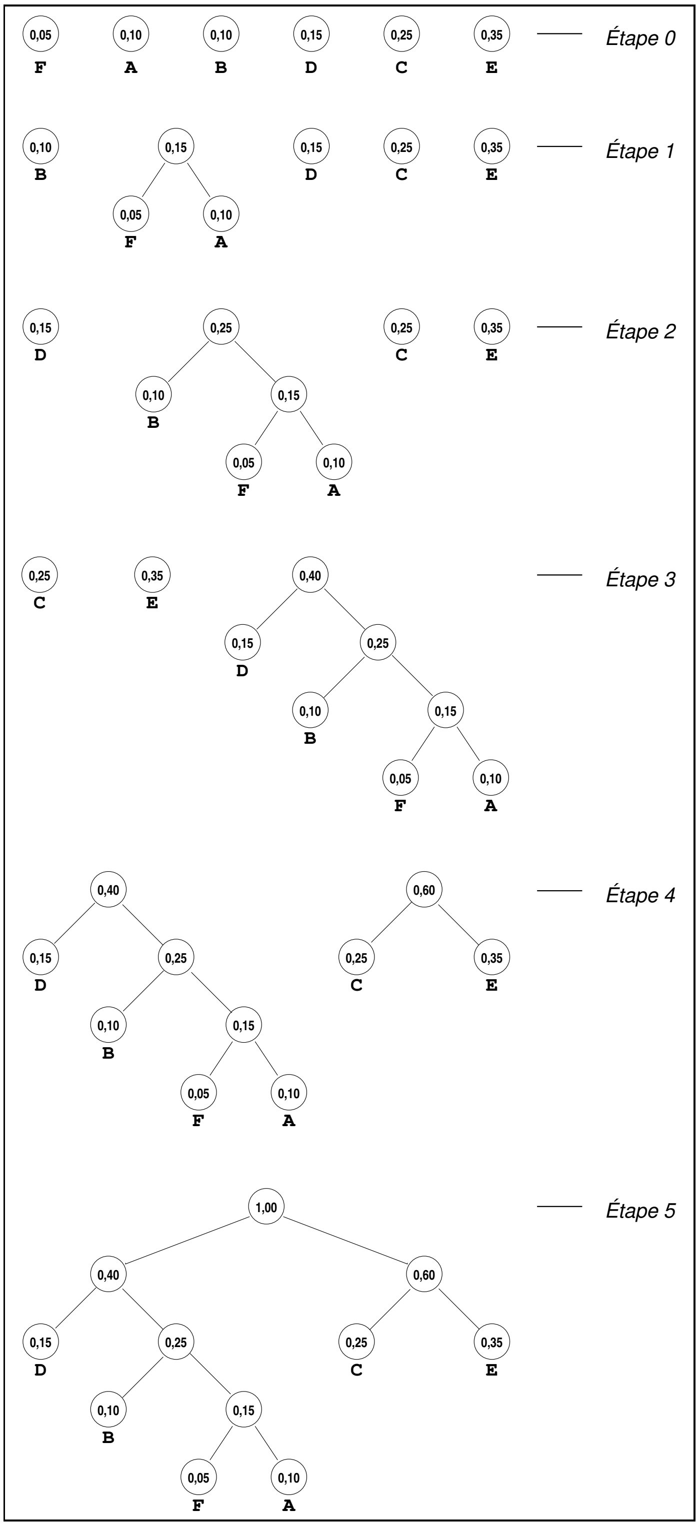

FIG. 2.3 – *Construction d'un code de Huffman.*

code de Huffman pour l'alphabet source $\{\mathtt{A},\mathtt{B},\mathtt{C},\mathtt{D},\mathtt{E},\mathtt{F}\}$, avec les probabilités $\mathbf{P}(\mathtt{A}) = 0,10$, $\mathbf{P}(\mathtt{B}) = 0,10$, $\mathbf{P}(\mathtt{C}) = 0,25$, $\mathbf{P}(\mathtt{D}) = 0,15$, $\mathbf{P}(\mathtt{E}) = 0,35$, $\mathbf{P}(\mathtt{F}) = 0,05$. Si l'on utilise les bonnes structures de données pour représenter les arbres et les forêts (par exemple, une file de priorité), le coût de la construction du code est en $O(n \log n)$ pour un alphabet à n symboles. Le code d'un symbole est alors déterminé en suivant le chemin depuis la racine de l'arbre jusqu'à la feuille associée à ce symbole en concaténant successivement un `0` ou un `1` selon que la branche suivie est à gauche ou à droite.

Le codage se réalise facilement à l'aide d'une table associant à chaque symbole son code ; une fois ce code construit, le décodage se réalise avec un *arbre de décodage* (figure 2.2, page 66), grâce à la propriété de préfixe des codes de Huffman.

symbole	*code*
A	0111
B	010
C	10
D	00
E	11
F	0110

Notons que si l'on utilisait un code binaire de longueur constante, trois bits seraient nécessaires (car l'alphabet-source a 6 symboles, et $2^2 < 6 \leq 2^3$). Compte tenu de ces probabilités, le code obtenu a une longueur moyenne de $2,4$, ce qui représente une compression de 20% par rapport à un codage de longueur constante minimum ; par rapport à un codage standard de chaque symbole sur un octet, la compression est de 70%. Peut-on faire mieux ? La réponse à cette question relève de la *théorie de l'information*, branche commune à l'informatique et au calcul des probabilités (la réponse est oui, mais pas beaucoup mieux : la longueur moyenne minimale d'un code pour ce langage est $2,32$).

Information et entropie L'analyse quantitative de l'information est due à SHANNON (1948). Il définit, dans un cadre probabiliste, une notion d'*entropie*, terme dû à CLAUSIUS (1864), qui a des interprétations diverses en informatique, en thermodynamique et en probabilités, pourtant toutes liées par des intuitions communes.

Considérons une partition $\mathcal{E} = (E_1, \ldots, E_n)$ d'un espace de probabilités $(\Omega, \mathcal{A}, \mathbf{P})$ en un ensemble d'événements : $\Omega = \bigcup_{i=1}^{n} E_i$, avec $E_i \cap E_j = \emptyset$ si $i \neq j$. En posant $p_i = \mathbf{P}(E_i)$, et en utilisant le logarithme en base 2 (par convention, $0 \log 0 = 0$), l'entropie de $\mathcal{E}$ est définie par :

$$\mathbf{H}(\mathcal{E}) = -\sum_{1=1}^{n} p_i \log p_i$$

Dans les applications au codage, Ω est l'alphabet source, et chaque événement est un symbole de l'alphabet. Les probabilités p_i s'obtiennent généralement par la mesure de la fréquence des symboles dans des textes usuels.

L'entropie mesure l'information par l'incertitude qu'elle permet de lever : incertitude *avant*, information *après* la réception d'un message identifiant l'un des événements de $\mathcal{E}$. Elle est ainsi maximale quand tous les événements sont équiprobables, c'est-à-dire $p_i = 1/n$: $\mathbf{H}(\mathcal{E}) = \log n$. On retrouve ainsi le nombre de bits nécessaires pour coder un élément d'un ensemble de cardinal n, en l'absence d'hypothèse probabiliste. Par exemple, pour les 26 lettres de l'alphabet, il faut au moins $\log 26 = 4,7$ bits (on doit donc en utiliser 5). L'entropie est minimale, et nulle, quand un des événements est de probabilité 1 : il n'y a aucune incertitude, et il est inutile de coder des événements de probabilité nulle. Un message codé sur zéro bit suffit à porter une information sûre !

Dans les cas intermédiaires, entre 0 et $\log n$, le rapport $\tau = \mathbf{H}(\mathcal{E})/\log n$ mesure le *taux de compression* idéal que l'on obtiendrait en ne codant que les messages « les plus fréquents », et $1 - \tau$ mesure la redondance intrinsèque, en terme d'information, d'un code représentant tous les messages possibles. Cette redondance est utile car elle permet de décrypter des messages chiffrés alors que la clé de déchiffrement est inconnue, et de façon plus courante, de corriger des fautes d'orthographe. La compression de données, requise pour réduire le coût des supports de stockage et de transmission de l'information, cherche au contraire à diminuer cette redondance.

On peut montrer que la longueur moyenne minimale, L, d'un codage binaire de l'alphabet source vérifie :

$$\mathbf{H}(\mathcal{E}) \leq L < \mathbf{H}(\mathcal{E}) + 1$$

La longueur moyenne d'un code de Huffman peut approcher d'aussi près que voulu la longueur moyenne minimale, à condition de coder non pas des lettres individuellement, mais des blocs de lettres de longueur assez grande. On peut aussi montrer que $\mathbf{H}(\mathcal{E})$ mesure le nombre moyen de questions binaires qu'il suffit de poser pour identifier l'un des événements de $\mathcal{E}$.

2.3 Cryptographie à clé publique : RSA La cryptographie a pour but d'assurer la sécurité des données, en les *chiffrant* afin de les rendre incompréhensibles sans l'usage d'une *clé de déchiffrement*. Pendant longtemps, la cryptographie a reposé sur l'usage d'une clé secrète, qui devait être partagée par l'émetteur et le récepteur. En 1976, Diffie et Hellman suggérèrent la possibilité d'assurer la confidentialité sans recourir à un secret partagé, au moyen d'une clé connue de tous. Cette idée a profondément transformé la cryptographie. Le système de chiffrement à clé publique RSA, proposé en 1977 par RIVEST, SHAMIR et ADLEMAN, est maintenant couramment utilisé par les systèmes de chiffrement, par exemple par PGP, généralement en complément d'un chiffrement à clé secrète à usage unique. Le développement du commerce électronique et l'irruption de l'Internet dans la vie privée ont entraîné une large diffusion des outils cryptographiques, longtemps réservés à des usages militaires. L'exponentielle modulaire intervient dans les algorithmes de la cryptographie à clé publique, car elle est considérablement plus facile à calculer que son inverse, le logarithme modulaire.

Pour construire ses clés, chaque utilisateur de RSA

- choisit deux grands nombres premiers p et q ;
- calcule $n = pq$;
- choisit un entier $e < n$ qui est premier avec $(p-1)(q-1)$;
- calcule l'inverse d de e modulo $(p-1)(q-1)$;
- publie sa clé publique, qui est formée des deux entiers e et n ;
- conserve sa clé privée d ;
- détruit les entiers p et q qui ne doivent pas être divulgués.

Les fonctions de chiffrage et de déchiffrage sont respectivement :

$$\begin{aligned} C(m) &= m^e \bmod n \\ D(m) &= m^d \bmod n \end{aligned}$$

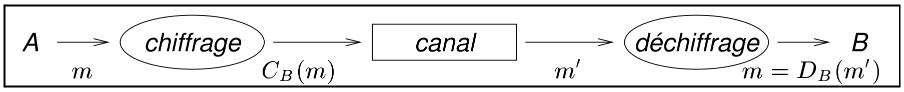

FIG. 2.4 – *Transmission de l'information chiffrée.*

Les fonctions et paramètres d'un utilisateur A sont notés C_A, D_A, n_A, e_A, d_A ; la fonction de chiffrage C_A est connue de tous, tandis que la fonction de déchiffrage D_A n'est connue que de A. Soient A et B deux utilisateurs de RSA. Quand A veut communiquer confidentiellement un entier m ($0 < m < n$) à B, il calcule $C_B(m)$ à l'aide de la clé publique de B, qu'il envoie à B ; à la réception d'un message chiffré c, B calcule $D_B(c)$ à l'aide de sa clé privée. Il s'agit bien d'un déchiffrage car $D_B(C_B(m)) = C_B(m)^{d_B} \bmod n_B = m^{e_B d_B} \bmod n = m$, grâce au théorème d'Euler qui assure que $m^{\phi(n)} = 1 \bmod n$ si m est premier avec n, et au fait que $\phi(n) = (p-1)(q-1)$.

La sécurité de ce schéma provient de la difficulté à factoriser de grands entiers. En effet, déterminer d à partir de e demande la connaissance de $(p-1)(q-1)$; or la publication de $n = pq$ n'est en aucune façon une aide pour calculer $(p-1)(q-1)$, qui ne peut être obtenu qu'à partir de p et q. D'autre part, on ne sait pas calculer efficacement des racines e-ièmes, ce qui permettrait d'avoir le texte clair m à partir du texte chiffré $C(m)$. Contrairement à la plupart des autres problèmes, pour lesquels on cherche des algorithmes efficaces, le chiffrement n'est utile que si le problème de déchiffrement est difficile, c'est-à-dire que si tous les algorithmes que l'on peut proposer sont extrêmement inefficaces.

La méthode RSA est également employée pour l'authentification des données. Si A veut communiquer un entier m $(0 < m < n)$ à B et garantir à B qu'il est l'auteur de ce message, il joint à m sa *signature* $s = D_A(h(m))$ grâce à sa clé privée et à une fonction de hachage h publique (voir § 2.10, par exemple, la fonction MD5). Recevant m et s de la part de A, B calcule à la fois $C_A(s)$ grâce à la clé publique de A et $h(m)$; normalement, $C_A(s) = C_A(D_A(h(m))) = h(m)$, par le même théorème d'Euler ; donc si B obtient le même nombre par dxes deux calculs, le message est authentifié ; sinon, c'est que A n'en est pas l'auteur, ou bien que m a été altéré au cours de la transmission. La sécurité de ce schéma d'authentification repose sur la difficulté, étant donné h, à trouver m tel que $h(m) = h$.

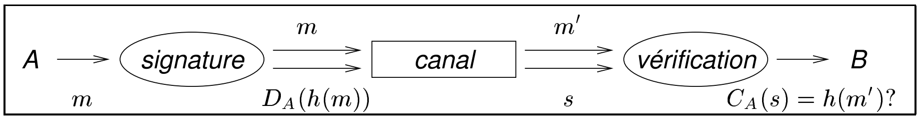

FIG. 2.5 – *Authentification par signature électronique.*

2.4 *Correction d'erreurs : le code Hamming* Un code correcteur d'erreur est utilisé pour transmettre un message dans un canal bruité ; il permet de reconstituer le message émis même si des erreurs (en nombre limité), dues au bruit, ont altéré le message. L'alphabet source, comme l'alphabet du code, est $\{0, 1\}$. On s'intéresse au codage par blocs : chaque mot de longueur m est codé par un mot de longueur n avec $n \geq m$. Le codage est donc une application de $\{0, 1\}^m$ vers $\{0, 1\}^n$. Parmi les n bits du mot-code que nous allons décrire, m reproduisent le mot-source, les $n-m$ autres sont les bits de correction : le taux de transmission est de n/m. On montre que si deux mots distincts du code diffèrent au moins en d bits, alors le code permet de corriger exactement $\frac{d-1}{2}$ erreurs.

Les codes de Hamming, pour lesquels $n = 2^k - 1$ et $m = n - k$, permettent de corriger une erreur ; pour k fixé et n grand, le taux de transmission est voisin de 1. Leur description fait appel à l'algèbre linéaire modulo 2, ou bien aux polynômes modulo 2. Un mot de p bits est représenté par un vecteur binaire de longueur p, c'est-à-dire un élément de l'espace vectoriel $(\mathbf{Z}/2\mathbf{Z})^p$, par exemple pour `110` :

$$\begin{pmatrix} 1 \\ 1 \\ 0 \end{pmatrix}$$

La *matrice de parité* d'un code de Hamming est une matrice binaire à k lignes et n colonnes : les colonnes contiennent les représentations binaires des entiers entre 1 et n, par exemple :

$$H = \begin{pmatrix} 1 & 0 & 0 & 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 & 1 \end{pmatrix}$$

La matrice de parité, H, permet de définir les mots du code : ce sont les vecteurs c de dimension n tels que $Hc = 0$. Comme cette équation définit un sous-espace vectoriel de $(\mathbf{Z}/2\mathbf{Z})^n$, on dit qu'il s'agit d'une code linéaire ; ce sous-espace est de dimension n. Supposons que l'on reçoive c' qui diffère d'un mot du code c de un bit : $c' = c + e$, où e est le vecteur d'erreur, par exemple `0100000`. Comme $Hc = 0$, on a $Hc' = He$; puisque e contient un seul bit (par exemple $e =$ `0100000`), le calcul de Hc' donne l'une des colonnes de H ; si on obtient $Hc' =$ `010`, ce qui est la deuxième colonne de H, c'est que l'erreur est sur le deuxième bit, c'est-à-dire $e =$ `0100000`. La correction est donc très facile. Il reste à dire comment le codage et le décodage s'effectuent, autrement dit quels sont les bits d'information et les bits de correction dans un mot du code.

La détermination d'une base du sous-espace d'équation $Hc = 0$ permet de formuler le codage : si G est la matrice $m \times n$ dont les lignes sont les vecteurs d'une telle base, $Hc = 0$ équivaut à $c = aG$ pour a, un vecteur-ligne de longueur m. Ainsi a est codé par $c = aG$; on dit que G est la *matrice génératrice* du code.

Dans l'exemple précédent, on trouve facilement, par exemple :

$$G = \begin{pmatrix} 1 & 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 & 1 \end{pmatrix}$$

ce qui fait que les 3 premiers bits sont ceux de correction, les 4 suivants étant les bits d'information.

Une autre technique pour construire ce code consiste à utiliser des polynômes *primitifs*. On note ainsi que la matrice H s'obtient en décomposant les monômes X^i, pour $i = 0, \ldots, 6$ dans la base $\{1, X, X^2\}$ de l'espace vectoriel des polynômes à coefficients dans $\mathbf{Z}/2\mathbf{Z}$, et modulo $P = 1 + X + X^3$.

On a ainsi $X^3 = X + 1$, $X^4 = X^2 + X$, $X^5 = X^2 + X + 1$, $X^6 = X^2 + 1$ modulo P (ce polynôme est primitif car les monômes forment une base de cet espace vectoriel). Par construction de H, Hb est égal à $(b_0 + \ldots + b_n X^n)$ modulo P. Il en résulte que les bits de correction sont obtenus en calculant le reste de la division euclidienne de $X^k(a_0 + \ldots + a_m X^m)$ par P. Pour le décodage, supposons qu'une erreur se soit produite sur le bit p. Au lieu de recevoir le polynôme Q, c'est $Q + X^p$ qui est reçu. Le reste de la division euclidienne de $Q + X^p$ par P est égal à X^p modulo P ; ceci permet de déterminer p, donc l'erreur.

Par exemple, un mot du code utilisé par le Minitel comporte 17 octets. Les 16 premiers forment un code de Hamming à 7 bits de correction : $k = 7$, $m = 120$ (soit 15 octets), $n = 127$; le 128ème bit, dit *bit de parité*, est tel que le nombre de 1 dans ces 16 octets soit pair. Le 17ème octet est formé de 8 zéros ; il permet de détecter des incidents importants (par exemple, la foudre). On s'intéresse ici seulement aux 127 premiers bits. Ce code est défini, avec la méthode exposée ci-dessus, avec le polynôme $P = X^7 + X^3 + 1$.

2.5 Problèmes, algorithmes et structures de données Les algorithmes résolvent des problèmes. Voici quelques exemples de problèmes et leur solution algorithmique :

- un entier n ; n est-il premier ? *L'algorithme stochastique de Miller-Rabin*
- un système d'équations linéaires ; le système est-il régulier ? si oui, calculer sa solution. *L'algorithme de triangulation de Gauss*
- un n- uplet de nombres complexes ; calculer la transformée de Fourier discrète de ce n-uplet. *La transformée de Fourier rapide*
- une chaîne de caractères s ; s est-elle un programme Java correct ? *L'analyse syntaxique par un automate à pile*
- un ensemble de villes et de routes les reliant, la longueur de ces routes, et deux villes de cet ensemble ; calculer la longueur du chemin le plus court entre ces deux villes. *L'algorithme du plus court chemin de Floyd, par programmation dynamique*
- un ensemble de tâches, leurs durées et leurs contraintes de précédence ; ces tâches sont-elles réalisables ? si oui, calculer leurs dates de réalisation au plus tôt et au plus tard. *Un algorithme glouton d'affectation de tâches*
- un réseau de transport de marchandises, la capacité de chaque route, et un entrepôt ; quelle est la quantité maximale de marchandises que ce réseau peut écouler à partir de l'entrepôt ? *L'algorithme de flot maximum de Ford-Fulkerson*

– comment rétablir le réseau électrique après une tempête en réparant le minimum de lignes électriques ? *L'algorithme glouton d'arbre couvrant minimum de Prim, ou celui de Kruskal*

Il n'existe pas, hélas, de méthode universelle pour construire des algorithmes. Il y a cependant quelques grandes méthodes :

– la méthode *incrémentale* : c'est l'idée de résoudre un problème $P(n)$ à partir d'une solution de $P(n-1)$; c'est typiquement une méthode par récurrence, qui peut donner lieu aussi bien à des programmes itératifs qu'à des programmes récursifs ; dans le cas d'un problème d'optimisation, la méthode *gloutonne* consiste à construire une solution de $P(n)$ en prolongeant une solution de $P(n-1)$ par un choix localement optimal ;
– la méthode « *diviser pour régner* » : c'est une méthode descendante, qui conduit à décomposer un problème en sous-problèmes et à construire une solution du problème en composant les solutions de ces sous-problèmes, typiquement en transformant un problème $P(n)$ en deux problèmes $P(n/2)$;
– la *programmation dynamique* : c'est une méthode ascendante, utilisable pour des problèmes d'optimisation, qui consiste à construire la solution d'un problème à partir des solutions de *tous* les sous-problèmes ; elle s'applique quand toute sous-solution d'une solution optimale est optimale pour le sous-problème correspondant.

La notion même d'algorithme suppose que les données d'un problème soient représentées de façon finie. Ceci marque la différence entre un entier et un nombre réel (élément de l'ensemble $\mathbf{R}$) : un entier quelconque a une représentation finie, ce n'est pas le cas d'un nombre réel quelconque (mais certains nombres réels, aussi bien $\sqrt{2}$ que π, admettent une représentation finie par un programme qui les calcule). Il suffirait donc d'exiger que les données d'un problème soient représentées par une suite de `0` et de `1`, ce qu'on appelle un mot binaire. Généralement, au terme d'une étape de modélisation, les données apparaissent sous une forme plus structurée : une matrice, un graphe, une table, etc.

La notion de *structure de données* est aussi importante en informatique que les *structures* des mathématiques (corps, espace vectoriel, espace de probabilités, variété différentielle, etc). En informatique comme en mathématiques, il s'agit d'un ensemble de données et de certaines opérations sur celles-ci. D'ailleurs certaines de ces structures sont autant étudiées de part et d'autre : les notions de monoïde, de graphe ou de matroïde figurent dans les cours de mathématiques discrètes. D'autres, les piles, files, tas ou tables sont plus particulièrement étudiées, et surtout utilisées, par les informaticiens. Les structures de données jouent un rôle central dans la conception et la formulation de certains algorithmes : les piles pour le parcours en profondeur d'un graphe, les files pour leur parcours en largeur, les tas pour le codage de Huffman, et les arbres pour le décodage.

En outre, certains problèmes ne peuvent pas être résolus par un algorithme. C'est le cas du problème consistant à déterminer si l'exécution d'un programme quelconque se termine : le plus simple est de l'exécuter, mais si elle ne termine pas, on ne saura jamais si elle termine ou non. Il existe d'autres problèmes, qui peuvent être résolus par des algorithmes, mais tels que tous les algorithmes connus à ce jour pour les résoudre ont un coût excessif (par exemple, un coût exponentiel) : on ne pourra donc les exécuter que sur des données de petite taille. Un exemple en est la détermination d'une tournée d'un voyageur de commerce de longueur minimale. On est donc amené à être moins exigeant : on cherchera un algorithme qui calcule des solutions approchées, ou alors on renoncera au caractère déterministe des algorithmes, et on introduira de l'aléa.

2.6 Recherche d'un élément dans une table Les structures de données servent d'abord à contenir des données. Selon les opérations à réaliser sur ces données, on choisira une structure particulière, par exemple l'une des trois structures les plus usuelles : d'ensemble, de liste ou de table. Il est fréquent que les données soient des associations d'une clé (ou identificateur) et d'une information ; par exemple, l'association nom $\rightarrow$ numéro de téléphone (c'est simplement ce que l'on appelle une *application* en mathématiques). La structure de table offre les trois opérations suivantes :

– l'insertion dans la table d'une clé et de l'information associée ;
– la recherche de l'information associée à une clé ;
– (éventuellement) la suppression d'une association de clé donnée.

L'API Java 2 offre l'interface `java.util.Map` avec ces mêmes fonctionnalités (les types interfaces de java sont décrits en § 3.1). Par souci de généricité, on représente en Java la clé et son information par des instances de la classe `Object` ; souvent, la clé est une chaîne de caractères (par exemple, un nom), et l'information est d'un type quelconque. Plutôt que de retourner un `void`, il est préférable que les méthodes `insérer()` et `supprimer()` retournent un `boolean` qui indique si l'opération est réussie. Le type abstrait de cette structure de données peut être défini en Java comme l'interface suivante (les noms traditionnels de ces opérations en anglais sont *put, get, remove*) :

```
interface Table {
  boolean insérer(Object clé, Object valeur);
  Object rechercher(Object clé);
  boolean supprimer(Object clé);
}
```

Il en existe au moins deux implémentations efficaces : les tables de hachage et les arbres bicolores. Avant de les présenter, il est utile de montrer une implémentation plus simple et comment tenter de l'améliorer. Notons d'ailleurs que les trois opérations ne sont pas toujours réalisées avec la même fréquence. Dans le cas d'un annuaire, les interrogations (ou recherches) sont beaucoup plus fréquentes que les insertions ou les suppressions ; on peut donc chercher à optimiser les recherches plutôt que les deux autres opérations.

2.7 *Recherche séquentielle* La plus simple des implémentations de la structure abstraite de table se fait à l'aide d'un tableau, et l'algorithme de recherche le plus élémentaire est la *recherche séquentielle*, qui parcourt le tableau jusqu'à ce qu'un objet de clé donnée soit trouvé ou bien que la fin du tableau soit atteinte :

```
class TableParTableau implements Table {

  static class Association {
    Object clé;
    Object information;
    Association(Object clé, Object information) {
      if (clé == null || information == null)
        throw new NullPointerException();
      this.clé = clé;
      this.information = information;
    }
  }

  private Association[] tableau;
  private int nbElements;
  TableParTableau(int n) {
    tableau = new Association[n];
  }

  public boolean insérer(Object clé, Object information) {
    if (nbElements < tableau.length) {
      tableau[nbElements] = new Association(clé, information);
      nbElements++;
      return true;
    } else return false;
  }

  public Object rechercher(Object clé) {
    for (int i=0; i<nbElements; i++) {
      if (tableau[i].clé != null &&
          tableau[i].clé.equals(clé)) {
        return tableau[i].information;
      }
    }
    return null;     // objet non trouvé
```

```
  }

  public boolean supprimer(Object clé) {
    for (int i=0; i<nbElements; i++) {
      if (tableau[i].clé != null &&
          tableau[i].clé.equals(clé)) {
        tableau[i].clé = null;
        tableau[i].information = null;
        return true;
      }
    }
    return false;
  }
}
```

La classe `TableParTableau` définit une classe membre statique `Association` pour contenir les couples (clé, information). Java permet en effet la définition d'une classe dans le corps d'une autre classe :

```
class A {
  class B {}
}
```

La classe `B` est dite *imbriquée* dans la classe `A`. Une classe imbriquée peut être déclarée `static`, auquel cas elle ne peut accéder qu'aux membres statiques de la classe englobante, de façon analogue à une méthode statique. Une classe statique est une classe d'intérêt local. Une classe imbriquée non statique est appelée une classe *intérieure* : une instance d'une classe intérieure n'existe qu'au sein d'une instance de la classe englobante. Une classe intérieure peut accéder à tous les membres de la classe englobante.

Le constructeur `Association()` s'assure que ses arguments sont non nuls, de sorte qu'on peut utiliser la valeur `null` pour indiquer qu'un élément a été supprimé. L'élément du tableau occupé par un élément supprimé n'est pas réutilisé par l'insertion, ce qui permet de faire l'insertion en temps constant.

On notera que le « `return tableau[i].information;` » placé dans la boucle de recherche permet de s'en échapper (c'est-à-dire de ne pas exécuter les itérations suivantes) en retournant immédiatement une valeur. Si l'on souhaitait s'échapper de la boucle sans retourner immédiatement, on placerait un « `break;` » à la même place.

Le nombre maximum de comparaisons d'objets (invocation de la méthode `equals()`) effectuées par cet algorithme est de N, le nombre d'éléments insérés, ce maximum étant atteint quand la clé cherchée n'est pas trouvée ; en moyenne, il procède à $N/2$ comparaisons. Cela est très mauvais, et on sait faire beaucoup mieux.

2.8 Recherche dichotomique dans une table ordonnée Un supplément d'information permet souvent de réduire la complexité d'un problème. L'algorithme de recherche séquentielle n'utilisait comme information que le test d'égalité sur les clés, avec deux résultats possibles : égalité, non-égalité. Supposons maintenant que les clés d'une table soient rangées par ordre croissant ; cela a pu être obtenu par l'usage préalable d'un algorithme de tri. On peut alors réaliser un test avec trois résultats : égalité, inférieur, supérieur. On suppose ici que le type de la clé permet ce tri, ce qui est le cas du type `String`, comme de tout sous-type de l'interface `Comparable` dont les instances sont mutuellement comparables.

L'algorithme de recherche séquentielle peut tirer parti de cet ordre en interrompant la recherche dès qu'une clé strictement supérieure à la clé cherchée est rencontrée. Ceci réduit le coût d'une recherche en cas d'échec : au maximum, N, et en moyenne $N/2$. Cette amélioration n'est pas significative.

La bonne idée est de comparer la clé recherchée avec la clé du milieu du tableau ; si la clé recherchée est plus petite, on continue la recherche dans la première moitié du tableau ; si elle est plus grande, on continue dans la seconde moitié du tableau. On suppose ici, pour simplifier, qu'aucune suppression n'a été effectuée, de sorte qu'il n'y a pas de valeurs nulles dans le tableau. La formulation récursive est la plus naturelle :

```
Object rechercheDichotomique(Object clé, int a, int b) {

  // recherche dans t[a],...,t[b]
  if (a<=b) {
    int m = (a+b)/2;
    int c = tableau[m].clé.compareTo(clé);
    if (c == 0) {
      return tableau[m].information;
    } else if (c > 0) {
      return rechercheDichotomique(clé, a, m-1);
    } else {
      return rechercheDichotomique(clé, m+1, b);
    }
  } else {
    return null;
  }
}

public Object rechercher(Object clé) {
  return rechercheDichotomique(clé, 0, nbElements-1);
}
```

Cette définition récursive terminale se transforme facilement en définition itérative :

```
Object rechercheDichotomiqueIter(Object clé, int a, int b) {

  // recherche dans t[a],...,t[b]
```

```
    while (a<=b) {
      int m = (a+b)/2;
      int c = tableau[m].clé.compareTo(clé);
      if (c == 0) {
        return tableau[m].information;
      } else if (c > 0) {
        b = m-1;
      } else {
        a = m+1;
      }
    }
  return null;
  }
```

La recherche dichotomique dans une table de taille N nécessite au plus $\log_2 N$ comparaisons. Cette réduction logarithmique de la complexité est un premier exemple de la méthode « diviser pour régner ». Cependant, l'insertion dans un tableau ordonné se fait en un temps linéaire (il faut $\log_2 N$ comparaisons pour trouver la position de l'élément à insérer, plus une moyenne de $N/2$ décalages vers la droite des éléments de la table plus grands que l'élément inséré). Cette implémentation n'est avantageuse que si les insertions sont plus rares que les recherches. L'insertion est plus facile si l'on remplace le tableau par une liste chaînée : elle se ferait en temps constant une fois la position trouvée.

2.9 Structures de données chaînées : les listes Une *liste chaînée étiquetée* (en anglais *labelled linked list*), appelée désormais liste, est définie récursivement de la façon suivante : une liste est soit vide, soit formée d'une étiquette et d'une liste. De façon analogue aux arbres binaires, une liste chaînée étiquetée non vide peut être implémentée à l'aide d'une classe à deux champs, contenant respectivement une étiquette (objet d'un type `Object` quelconque) et une référence à une liste :

```
class ListeChainee {
  Object étiquette;
  ListeChainee suivant;
  ListeChainee(Object étiquette, ListeChainee suivant) {
    if (étiquette == null)
      throw new NullPointerException();
    this.étiquette = étiquette;
    this.suivant = suivant;
  }
  // méthodes
}
```

Une liste non vide sera représentée par une référence à une instance de `ListeChainee`, et la liste vide par la valeur `null`. Tout traitement d'une liste doit d'abord tester si elle est vide.

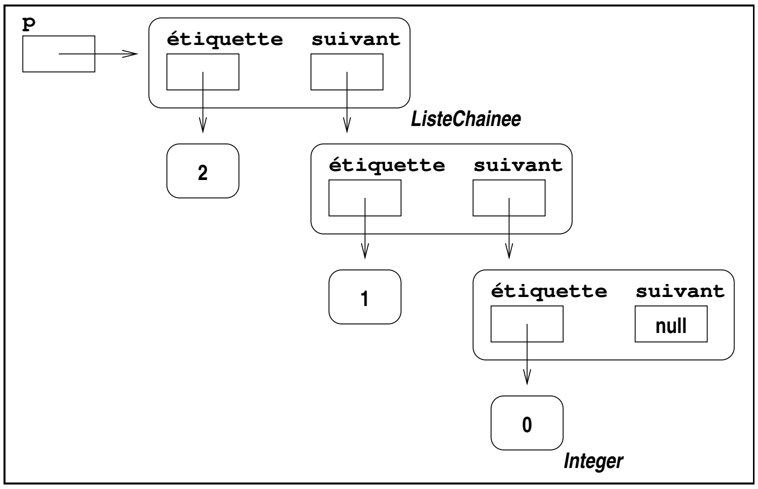

FIG. 2.6 – *Une liste chaînée.*

La liste de la figure 2.6 est désignée par la référence `p` définie par :

```
ListeChainee p =
  new ListeChainee(
    new Integer(2),
    new ListeChainee(
      new Integer(1),
      new ListeChainee(
        new Integer(0),
        null)));
```

La structure de liste chaînée permet d'implémenter le type abstrait `Table`, en étiquetant les listes par des instances du type `Association`. On fait de la classe `ListeChainee` un membre statique privé de `TableParListe`, de manière à cacher les détails d'implémentation. La recherche séquentielle d'une cellule par sa clé dans une liste se fait de façon très classique par une boucle `for` qui parcourt la liste ; la condition d'exécution est `l != null`, où `l` désigne une `ListeChainee` ; l'instruction d'itération remplace un lien par le lien suivant.

```
class TableParListe implements Table {

  static class Association {
    // ...
  }

  private static class ListeChainee {
    // ...
  }

  private ListeChainee liste;

  public boolean insérer(Object clé, Object information) {
```

```
    liste =
      new ListeChainee(new Association(clé, information), liste);
    return true;
  }

  public Object rechercher(Object clé) {
    return rechercher(clé, liste);
  }

  Object rechercher(Object clé, ListeChainee l) {
    if (l == null) {
      return null;
    } else if (((Association)l.étiquette).clé.equals(clé)) {
      return ((Association)l.étiquette).information;
    } else {
      return rechercher(clé, l.suivant);
    }
  }

  public boolean supprimer(Object clé)
  {
    if (liste == null) {
      return false;
    }
    else if (((Association)liste.étiquette).clé.equals(clé)) {
      liste = liste.suivant;
      return true;
    } else {
      return supprimer(clé, liste.suivant, liste);
    }
  }

  boolean supprimer(Object clé,
                    ListeChainee l,
                    ListeChainee pré) {
    if (l == null) {
      return false;
    } else if (((Association)l.étiquette).clé.equals(clé)) {
      pré.suivant = l.suivant;
      return true;
    } else {
      return supprimer(clé, l.suivant, l);
    }
  }
}
```

Il est facile de formuler la recherche de façon itérative :

```
  Object rechercherIter(Object clé) {
    for (ListeChainée l = liste; l!=null; l = l.suivant) {
      if (((Association)l.étiquette).clé.equals(clé)) {
        return ((Association)l.étiquette).information;
      }
    }
```

```
    return null;
  }
```

L'API Java 2 offre l'interface `List` et une implémentation `LinkedList` au moyen de listes doublement chaînées, qui dispose de deux champs (privés) de type `LinkedList`, l'un vers le prédecesseur dans la liste, l'autre vers le successeur.

La recherche dichotomique ne fonctionne plus pour ce type de liste, car on ne peut plus déterminer en temps constant le milieu d'une liste chaînée, mais seulement en temps linéaire. Pour conserver l'idée de la recherche dichotomique et garantir un coût logarithmique à toutes les opérations (insertion, recherche et suppression), il faut implémenter les tables par des arbres. En fait, on peut obtenir encore mieux que le coût logarithmique : le coût constant!

2.10 Le hachage Comment faire des opérations de recherche et d'insertion dans une table en temps constant? Les mémoires associatives, étudiées en intelligence artificielle, auraient cette propriété : la position occupée par un objet est déterminée seulement par cet objet. Il suffit de *calculer* la position en mémoire de cet objet, donc de disposer d'une *fonction* qui associe à un objet x sa position $h(x)$; on veut que le temps de calcul de $h(x)$ soit indépendant de x et aussi petit que possible. On dit que $h(x)$ est la *valeur de hachage* associée à x. C'est l'idée des *tables de hachage*, dont l'organisation s'oppose radicalement à celle des tables ordonnées, dans lesquelles la position d'un objet dépend du nombre d'objets plus petits qui ont déjà été insérés dans la table. Cette structure de données est sûrement la plus utile de toutes celles rencontrées dans ce cours. Son seul inconvénient est que l'ordre dans lequel les données sont stockées n'est pas maîtrisable. Si l'on doit énumérer les données par ordre croissant, il faut recourir à une autre implémentation des tables, à l'aide d'arbres, et renoncer au coût constant des opérations pour un coût logarithmique.

Si l'on devait insérer dans une table les éléments de l'ensemble $[0, N-1]$, il suffirait d'utiliser un tableau t de taille N et de ranger l'objet i dans $t[i]$, solution triviale, avec $h(i) = i$. Cela n'est déjà plus aussi simple si au lieu de l'intervalle $[0, N-1]$, on doit traiter un ensemble quelconque E de N entiers : comment faire pour construire « effectivement » une fonction injective de E dans l'ensemble des indices du même tableau $[0, N-1]$? L'existence d'une fonction injective est sans doute rassurante mais pas suffisante. En effet, les fonctions injectives sont rares : il y a n^m fonctions d'un ensemble de cardinal m dans un ensemble de cardinal n, parmi lesquelles il y a $\frac{n!}{(n-m)!}$ fonctions injectives, si $m \leq n$. Une estimation asymptotique du rapport $\frac{n!}{(n-m)!n^m}$, par la formule de Stirling, quand $n \to \infty$, est $e^{-k^2/2n}$. Le problème se complique encore, car on veut insérer dans une table des objets d'un ensemble E de cardinal plus grand, voire beaucoup plus grand que N. Par exemple, la table de hachage qu'un compilateur construit pour les noms figurant dans un programme donné doit contenir tout au plus quelques centaines de noms, mais ces noms appartiennent au minimum à un ensemble de cardinal 27×37^5, soit près de deux milliards, pour le langage C, dont la norme impose à tout compilateur d'accepter des identificateurs d'au moins 6 caractères, formés de chiffres, de lettres sans distinction de casse et de `'_'`, et commençant par une lettre ou par `'_'`. Le problème est de construire une fonction, qui bien que non injective, a de bonnes propriétés de *dispersion*.

Voici un exemple simple de fonction de hachage sur des chaînes de longueur l à valeurs dans l'intervalle $[0, N-1]$:

$$h(x) = \sum_{i=0}^{l-1} x[i] B^{l-1-i} \bmod N$$

où B est une puissance de 2 (pour faciliter le calcul, par exemple $B = 256$) et N est un nombre premier (pour éviter des collisions « arithmétiques »), ou en Java:

```
private static final int B=256;
private static final int N=311;

private static int h(String x) {
  int v = 0;
  for (int i=0; i<x.length(); i++) {
    v = (v*B + x.charAt(i)) % N;
  }
  return v;
}
```

L'API de Java définit une méthode `hashCode()` dans la classe `Object`, qui peut donc être utilisée pour tous les objets, et qui peut être redéfinie dans n'importe quelle classe.

Comme la fonction de hachage h n'est pas injective, il faut savoir traiter les *collisions*, c'est-à-dire le cas de deux clés $x \neq y$ ayant la même valeur de hachage $h(x) = h(y)$. Il existe deux sortes de techniques de résolution : le hachage ouvert et le hachage par chaînage.

Hachage par adressage ouvert La table de hachage est implémentée par un tableau t dont les cases vont contenir les associations. Initialement, tous les éléments du tableau ont la valeur `null`. Puis des opérations d'insertion sont réalisées ; ni la clé ni l'information ne peut être nulle. Si une association de clé x doit être insérée, et que $t[h(x)]$ est vide, c'est-à-dire contient la valeur nulle, alors l'insertion se fait à cette place ; si par contre $t[h(x)]$

est déjà occupé, et que le contenu a une clé différente de x, alors on calcule des valeurs de hachage *supplétives* $h_1(x)$, $h_2(x)$, … jusqu'à ce que l'on trouve $t[h_i(x)]$ vide ou contenant une association de clé x. De même, pour chercher une association de clé x, on teste la clé de l'objet en $t[h(x)]$, et éventuellement en $t[h_1(x)]$, $t[h_2(x)]$, etc, jusqu'à ce que la clé de l'objet qui s'y trouve soit égale à x, ou bien que la valeur soit nulle. Dans le cas où la table permet aussi des suppressions, il faut remplacer un objet supprimé par un objet spécial *supprimé*, distinct de la valeur nulle ; en insertion, on utilisera la première case vide ou supprimée, tandis qu'en recherche, on ne s'arrêtera qu'à la première case vide. On a choisit de représenter un objet supprimé par l'association d'une clé égale à la chaîne vide `""` et d'une information nulle ; par conséquent, on doit interdire la chaîne vide en tant que clé pour l'insertion. La classe locale `Association` est donc légèrement différente de celle employée précédemment.

Pour générer ces nouvelles valeurs de hachage, la méthode la plus simple est le *sondage linéaire*, qui choisit $h_i(x) = (h(x) + i) \bmod N$, ce qui revient à essayer les cases suivant $t[h(x)]$. Il y a d'autres méthodes (sondage quadratique, double hachage, hachage uniforme) qui ont de meilleures capacités de dispersion.

```
class TableHachageOuvert implements Table {

  static class Association {
    Object clé = "";
    Object information;
    Association(Object clé, Object information) {
      if (clé == null || information == null)
        throw new NullPointerException();
      if (clé == "")
        throw new IllegalArgumentException("clé: \"\"");
      this.clé = clé;
      this.information = information;
    }
    // Garantit que supprimé ne sera utilisable qu'en interne
    private Association() {}
  }

  private Association[] tableau;
  private static final Association supprimé = new Association();

  private int taille;

  TableHachageOuvert(int taille) {
    tableau = new Association[n];
    this.taille = taille;
  }

  public boolean insérer(Object clé, Object information) {
    int v = clé.hashCode();
```

```
    for (int i=0; i<taille; i++) {
      int j = (v+i)%taille;
      if (tableau[j] == null || tableau[j] == supprimé) {
        // la clé n'est pas dans la table
        // on l'y insère avec l'info associée
        tableau[j] = new Association(clé, information);
        return true;
      } else if (tableau[j].clé.equals(clé)) {
        // la clé est déjà dans la table :
        // mise à jour de l'info associée
        tableau[j].information = information;
        return true;
      }
    }
    // i == taille => la table était pleine
    return false;
  }

  public Object rechercher(Object clé) {
    int v = clé.hashCode();
    for (int i=0; i<taille; i++) {
      int j = (v+i)%taille;
      if (tableau[j] == null) {
        return null;
      } else if (tableau[j].clé.equals(clé)) {
        return tableau[j].information;
      }
    }
    return null;
  }

  public boolean supprimer(Object clé) {
    int v = clé.hashCode();
    for (int i=0; i<taille; i++) {
      int j = (v+i)%taille;
      if (tableau[j] == null) {
        return false;
      } else if (tableau[j].clé != null &&
                 tableau[j].clé.equals(clé)) {
        tableau[j] = supprimé;
        return true;
      }
    }
    return false;
  }
}
```

Il est clair que la complexité dans le pire des cas est de N opérations, N étant la taille de la table. On définit le *taux de chargement* d'une table de hachage de taille N contenant n objets comme la fraction $\alpha = n/N$; on a toujours $0 \leq \alpha \leq 1$. Pour donner une estimation asymptotique de la complexité des opérations, on fait tendre n et N vers l'infini, à α constant. On montre que, sous une hypothèse d'uniformité, le nombre moyen d'accès nécessaires pour une recherche infructueuse est d'au plus $1/(1-\alpha)$ et pour une recherche fructueuse de $\frac{1}{\alpha}\log\frac{1}{1-\alpha}+\frac{1}{\alpha}$. Par exemple pour une table à moitié pleine, on doit s'attendre à faire 2 accès pour la recherche d'un objet ne se trouvant pas dans la table, et à faire 3,387 accès s'il s'y trouve. Il s'agit bien d'un algorithme en $\Theta(1)$.

Hachage par chaînage La méthode de hachage par adressage ouvert avec sondage linéaire a l'inconvénient d'effectuer des comparaisons avec plusieurs cases successives de la table, qui n'ont pourtant pas la même valeur de hachage que l'objet recherché. Pour éviter ces comparaisons inutiles, on va *chaîner* les objets ayant la même valeur de hachage. La table de hachage est également implémentée par un tableau, l'élément du tableau d'indice i étant une référence à un objet rassemblant des associations dont les clés ont i pour valeur de hachage. Cet objet peut être une liste chaînée ou plus simplement, un tableau d'`Association`; le type `ArrayList` de l'API Java convient à cette implémentation :

```
List[] tableau = new ArrayList[N];
```

Les opérations d'insertion, de recherche et de suppression se font en deux étapes : le calcul de la valeur de hachage, h, puis la délégation de l'opération à la liste `tableau[h]`.

Le taux de chargement (voir § 2.10) $\alpha = n/N$ est la longueur moyenne des listes chaînées. Sous une hypothèse d'uniformité de la fonction de hachage, on montre que le nombre moyen d'accès nécessaires pour une recherche, négative ou positive, est d'au plus $1+\alpha$.

2.11 Les graphes La théorie des graphes est aujourd'hui un outil majeur de modélisation et de résolution de problèmes dans un très grand nombre de domaines, des sciences fondamentales aux applications technologiques les plus concrètes. Un graphe est défini par ses sommets et ses arcs. Les sommets forment un ensemble S quelconque (quoique fini en pratique), et les arcs un sous-ensemble A de $S \times S$; autrement dit, un graphe est le graphe d'une relation binaire. Si $a = (u, v) \in A$, on dit que a est un arc d'extrémité initiale u et d'extrémité finale v, on le note $a : u \to v$; on dit aussi que v est un sommet adjacent à u, ou est un successeur de u.

La théorie des graphes est pourvue d'un vocabulaire riche mais assez intuitif (figure 2.7). Par exemple, une *chaîne* de longueur $k \geq 0$ d'un graphe est une suite de $k+1$ sommets $v_0, v_1, \ldots, v_k$, avec $v_i \to v_{i+1}$ ou $v_{i+1} \to v_i$ pour $0 \leq i \leq k$. Une chaîne est un *chemin* si ses arcs sont consécutifs : $v_0 \to v_1 \to \ldots v_k$. Un graphe est *connexe* (resp. *fortement connexe*) si deux sommets quelconques sont reliés par une chaîne (resp. un chemin). Les *composantes connexes* sont les classes d'équivalence de la relation « être relié par une chaîne ». S'il existe un chemin de v_0 vers v_k, ont dit que v_k est *accessible* à partir de v_0, ce qu'on note $v_0 \to^\star v_k$; la relation d'accessibilité est un préordre (relation réflexive et transitive). Un chemin est un *circuit* si $v_k = v_0$ et $k \geq 1$. Les *composantes fortement connexes* sont les classes de la relation d'équivalence induite par l'accessibilité, définie entre les sommets u et v quand $u \to^\star v$ et $v \to^\star u$.

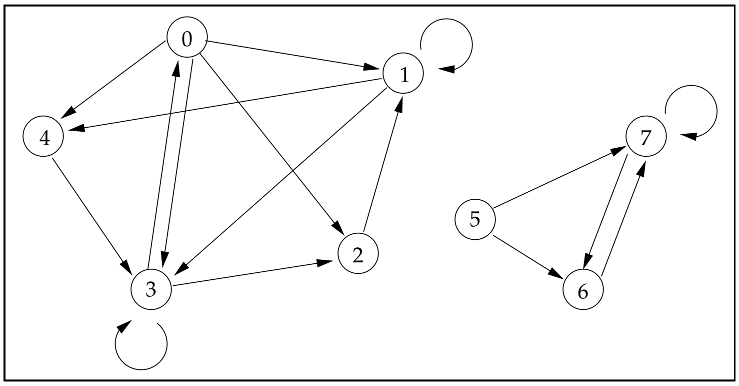

FIG. 2.7 – *Un graphe dont l'ensemble des sommets est* $S = \{0, 1, 2, 3, 4, 5, 6, 7\}$ *;* $0 \to 4 \to 3 \to 2$ *est un chemin,* $0 \to 4 \to 3 \to 0$ *est un circuit,* $0 \leftarrow 3 \leftarrow 1 \to 4$ *est une chaîne ; les composantes connexes sont* $\{0, 1, 2, 3, 4\}$ *et* $\{5, 6, 7\}$ *; les composantes fortement connexes sont* $\{0, 1, 2, 3, 4\}$*,* $\{5\}$ *et* $\{6, 7\}$*.*

Le type `Graphe` Comme pour les autres structures de données, il faut distinguer un type abstrait, l'interface `Graphe`, qui déclare des méthodes publiques, et des types qui implémentent cette interface, par exemple, les classes `GrapheParMatrice` et `GrapheParListe` que nous discuterons bientôt. Ces types seront rassemblés dans un paquet `graphe`, qui pourra être utilisé de la façon suivante :

```
import graphe;

class GrapheDemo {
  public static void main(String[] args) {
    Graphe g = new GrapheParMatrice(4);
    g.ajouteArc(0,1);
    g.ajouteArc(1,2);
    g.ajouteArc(2,3);
    g.ajouteArc(3,0);
    g.ajouteArc(0,2);
    g.ajouteArc(3,1);
    g.ajouteArc(1,1);
    // ...
  }
}
```

Cet exemple déclare une variable `g` de type `Graphe` ; le choix d'une implémentation se fait *via* un constructeur, ici `GrapheParMatrice` ; ce constructeur prend en argument la taille du graphe, c'est-à-dire le nombre de ses sommets. Une fois ce constructeur invoqué, `g` désigne un graphe à 4 sommets, sans arcs. Il est commode, pour spécifier un arc, de disposer d'une numérotation des sommets par un indice commençant par `0` ; les instructions suivantes ajoutent succesivement des arcs au graphe, chaque arc étant spécifié par les indices de ses sommets initial et final.

Deux autres types abstraits seront nécessaires, `Sommet` et `Arc`. Il arrive fréquemment que les sommets ou les arcs soient étiquetés, par exemple, par une couleur, ou par un nombre qui indique une distance ou un coût. Par suite, les sommets et les arcs devront comporter un champ `information` de type `Object` ; si ce champ est privé, les interfaces `Arc` et `Sommet` doivent exporter des méthodes d'accès `valInformation()` (sur les patterns d'accès, voir § 3.4). Comme les sommets peuvent être désignés par un numéro (utilisé en paramètre de la méthode `ajouteArc()`), la méthode `valIndice()` permet d'accéder à cet indice. Le sommet initial et le sommet final d'un arc sont obtenus par les méthodes `valSommetInitial()` et `valSommetFinal()`. Enfin, les méthodes `sommets()`, `arcs()` et `adjacents()` retournent des itérateurs qui auront un rôle crucial dans les algorithmes sur les graphes.

```
public interface Graphe {
  void ajouteArc(int u, int v);
  Iterator sommets();
  Iterator arcs();
  void parcoursProfondeur();
  void parcoursLargeur();
}

public interface Arc {
  Sommet valSommetInitial();
  Sommet valSommetFinal();
  Object valInformation();
}

public interface Sommet
{
  int valIndice();
  Iterator adjacents();
  Object valInformation();
}
```

Implémentation des graphes Deux implémentations des graphes sont couramment utilisées ; le choix se fait en fonction des opérations que l'on veut faire, et de la densité du graphe, c'est-à-dire du rapport entre le nombre d'arcs et le nombre de sommets.

La *matrice d'adjacence* M_G d'un graphe G est une matrice carrée, indicée par les sommets (donc de dimension $|S| \times |S|$), dont les éléments indiquent l'existence d'un arc : $M_G[u][v]$ est différent de `null` si et seulement si $u \to v$. La représentation d'un graphe par sa matrice d'adjacence est préférée quand le graphe est dense (beaucoup d'arcs), car elle comporte toujours les $|S|^2$ éléments d'un tableau bidimensionnel ; elle est bien adaptée aux algorithmes qui s'expriment à l'aide d'opérations matricielles.

```
public class GrapheParMatrice implements Graphe {
  private Arc[][] adjacence;

  public GrapheParMatrice(int taille) {
    adjacence = new boolean[taille][taille];
  }
  // ...
}
```

L'*ensemble d'adjacence*[1] d'un sommet u est l'ensemble $L_G[u]$ des arcs $u \to v$. Un graphe G est défini par l'association, à chaque sommet u de son ensemble d'adjacence $L_G[u]$. Cette représentation est préférée quand le graphe est creux (peu d'arcs).

```
public class GrapheParListe implements Graphe {
  private Set[] adjacence;

  public GrapheParListe(int taille) {
    adjacence = new HashSet[taille];
    for (int i = 0; i < taille; i++)
      adjacence[i] = new HashSet();
  }
  // ...
}
```

2.12 Graphes non orientés et arbres Un graphe est dit *non orienté* si la relation binaire est symétrique (si $u \to v$ alors $v \to u$) et irréflexive (pas de boucle $u \to u$) ; on peut alors considérer ensemble les deux arcs $u \to v$ et $v \to u$ comme la paire de sommets $\{u, v\}$, appelée *arête*. Au lieu de dessiner les deux arcs symétriques $u \to v$ et $v \to u$, on dessine seulement une arête, sans flèche, entre u et v, ce qu'on peut noter $u \leftrightarrow v$.

Dans un graphe non orienté, une chaîne $v_0, v_1, \ldots, v_k$ (avec $v_i \leftrightarrow v_{i+1}$ pour tout i) telle que $v_k = v_0$ et qui ne contient pas deux fois la même arête est appelée un *cycle* ; sa longueur est nécessairement ≥ 3 ; un graphe non orienté sans cycle est appelé *acyclique*. Les propriétés suivantes sont équivalentes, si $G = (S, A)$ est non orienté :

1. G est acyclique et connexe ;

1. On parle plus souvent de la *liste* d'adjacence, ce qui correspond en fait à un *multigraphe*, dans lequel plusieurs arcs peuvent avoir le même sommet initial et le même sommet final.

2. G est acyclique et $|A| = |S| - 1$;
3. G est connexe et $|A| = |S| - 1$;
4. G est connexe et cesse de l'être si on lui retire une arête quelconque ;
5. G est acyclique et cesse de l'être si on lui ajoute une arête quelconque ;
6. deux sommets quelconques sont reliés par une chaîne dont tous les sommets sont distincts.

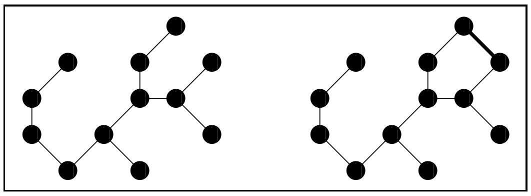

FIG. 2.8 – *Un arbre ; le graphe obtenu en lui ajoutant une arête contient un cycle.*

Un graphe vérifiant ces propriétés est appelé un *arbre* (figure 2.8); un graphe non orienté acyclique qui n'est pas connexe est appelé une *forêt* ; chaque composante connexe d'une forêt est un arbre.

Un arbre est *enraciné* dès qu'un sommet a été désigné comme sa *racine* (n'importe quel sommet peut être désigné comme racine). Une racine r détermine alors une relation de descendance : si $r \to^\star u \to^\star v$, on dit que u est *ancêtre* de v et que v est *descendant* de u ; si $r \to^\star u \to v$, on dit que u est le *parent* de v et que v est un *enfant* de u ; le nombre d'enfants d'un sommet est appelé son degré. La racine n'a pas de parent ; un sommet sans enfant est appelé une *feuille*.

Un arbre enraciné peut alors être considéré comme un graphe orienté, avec un arc $u \to v$ quand u est le parent de v ; l'orientation inverse peut aussi être utile pour certaines applications. On dit que ce sont des *arborescences* (figure 2.9).

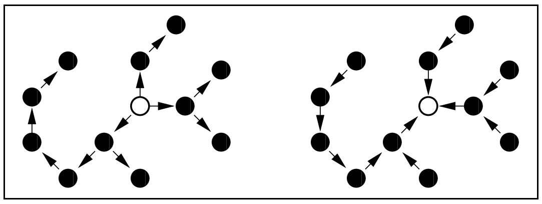

FIG. 2.9 – *Les deux arborescences correspondant à un arbre enraciné.*

2.13 Parcours en profondeur des graphes Comme pour les structures de données élémentaires (ensembles, listes, tables), un graphe peut être sujet à l'énumération de ses sommets ou de ses arcs ; un sommet peut aussi être sujet à l'énumération des sommets adjacents. Les interfaces des graphes et des sommets doivent comporter les itérateurs suivants, chacun devant implémenter les méthodes `hasNext()` et `next()` (le pattern d'itération est décrit au § 3.11) :

```
interface Graphe {
  // ...
  Iterator sommets();
  Iterator arcs();
}

interface Sommet {
  // ...
  Iterator adjacents();
}
```

L'implémentation de la méthode `adjacents()` dépend évidemment de la structure de données utilisée pour représenter un graphe : matrice d'adjacence ou ensemble d'adjacence.

Parcourir un graphe consiste à choisir un sommet et à énumérer à partir de celui-ci ses sommets en suivant ses arcs autant que possible ; chaque sommet énuméré peut donner lieu à un traitement (par exemple, imprimer une information associée au sommet). Il y deux parcours classiques. Si un parcours énumère un sommet u, puis un sommet adjacent v, un parcours en profondeur va énumérer les sommets adjacents à v avant d'énumérer les sommets adjacents à u autres que v, et un parcours en largeur adopte le choix inverse. Comme un sommet peut être adjacent à plusieurs sommets, il faut prendre garde à ne pas énumérer plusieurs fois le même sommet ; en outre, comme un graphe peut comporter des circuits, ceci risquerait de produire une énumération infinie. Il faut donc marquer les sommets déjà énumérés. Quand on réalise un parcours sur une feuille de papier, on colorie ces sommets au fur et à mesure du parcours ; dans un programme, on utilise une structure de données, par exemple un ensemble, pour stocker les sommets énumérés.

Nous allons rassembler les algorithmes sur les graphes dans une classe `Algorithmes` du package `graphe`. Le parcours en profondeur est très facile à programmer dans sa version récursive ; s'il s'agit simplement d'imprimer les valeurs des nœuds, un par ligne, on écrira la fonction suivante :

```
package graphe;
import java.util.Set;

public class Algorithmes {

  static void parcoursProfondeur(Sommet origine,
                                 Set sommetsVisités) {
```

```
    sommetsVisités.add(origine);
    System.out.println(origine.getIndice());
    Iterator i = origine.adjacents();
    while (i.hasNext()) {
      Sommet suivant = (Sommet)i.next();
      if (!sommetsVisités.contains(suivant)) {
        parcoursProfondeur(suivant, sommetsVisités);
      }
    }
  }
}
```

Un parcours en profondeur construit implicitement une *arborescence* qui a la même structure que l'arbre d'invocation de la fonction récursive `parcoursProfondeur()`. Pour rendre cette arborescence explicite, il suffirait d'ajouter un champ `parent` aux sommets pour enregistrer le prédécesseur de chaque sommet énuméré, et d'insérer juste avant l'invocation récursive l'affectation (ceci suppose que l'interface `Sommet` soit enrichie d'une méthode `chgParent()`):

```
  suivant.chgParent(origine);
```

Ce parcours en profondeur n'atteint que les sommets accessibles depuis l'origine. Pour parcourir l'ensemble des sommets d'un graphe `g`, il faut être capable d'énumérer tous les sommets (dans un ordre quelconque), et il faut relancer un parcours en profondeur à partir d'une origine qui n'a pas encore été énumérée, tant qu'il en existe :

```
package graphe;
import java.util.Set;
import java.util.HashSet;

public class Algorithmes {

  public static void parcoursProfondeur(Graphe g) {
    Iterator j = g.sommets();
    Set sommetsVisités = new HashSet();
    while (j.hasNext()) {
      Sommet s = (Sommet) j.next();
      if (!sommetsVisités.contains(s)) {
        parcoursProfondeur(s, sommetsVisités);
      }
    }
  }
  static void parcoursProfondeur(Sommet origine,
                                 Set sommetsVisités) {
  // comme ci-dessus
  }
}
```

On invoquera `Algorithmes.parcoursProfondeur(g)`, pour un graphe `g`.

Pré-traitement et post-traitement Le parcours précédent effectue en fait un traitement *préfixe*, où chaque sommet est traité *avant* ses sommets adjacents. Ceci produit une énumération préfixe des sommets, ce qui donne, dans le cas du graphe de la figure 2.10 le résultat : 0, 1, 2, 4, 5, 3. Dans l'énumération *postfixe*, chaque sommet est traité *après* tous ses sommets adjacents, soit dans cet exemple : 5, 4, 2, 3, 1, 0. Le traitement *postfixe* est obtenu en déplaçant l'invocation de la méthode de traitement (ici, l'impression) après le `while`;

```
static void parcoursProfondeur(Sommet origine,
                               Set sommetsVisités) {
  sommetsVisités.add(origine);
  Iterator i = origine.adjacents();
  while (i.hasNext()) {
    Sommet suivant = (Sommet) i.next();
    if (!sommetsVisités.contains(suivant)) {
      parcoursProfondeur(suivant, sommetsVisités);
    }
  }
  System.out.println(origine.valInformation());
}
```

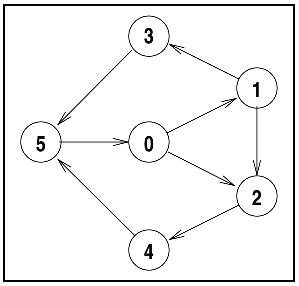

FIG. 2.10 – *Un graphe à parcourir.*

Un même parcours peut comporter un pré-traitement et un post-traitement du sommet énuméré. Java ne disposant pas de variable de type fonctionnel, on doit ajouter à la méthode de parcours deux paramètres `pré` et `post` d'un type abstrait `Traitement` déclarant juste une méthode `traite()` :

```
package graphe;

public interface Traitement {
  void traite(Objet o);
}
```

La fonction de parcours en profondeur devient :

```
static void parcoursProfondeur(Sommet origine,
```

```
                             Set sommetsVisités,
                             Traitement pré,
                             Traitement post) {
    sommetsVisités.add(origine);
    pré.traite(origine);
    Iterator i = origine.adjacents();
    while (i.hasNext()) {
      Sommet suivant = (Sommet)i.next();
      if (!sommetsVisités.contains(suivant)) {
        parcoursProfondeur(suivant, sommetsVisités, pré, post);
      }
    }
    post.traite(origine);
  }

  public static void parcoursProfondeur(Graphe g,
                                        Traitement pré,
                                        Traitement post) {
    Iterator j = g.sommets();
    Set sommetsVisités = new HashSet();
    while (j.hasNext()) {
      Sommet s = (Sommet)j.next();
      if (!sommetsVisités.contains(s)) {
        parcoursProfondeur(s, sommetsVisités, pré, post);
      }
    }
  }
```

Pour l'utiliser, il faut d'abord implémenter l'interface `Traitement`, par exemple :

```
class Pre implements Traitement {
  public void traite(Object o) {
    System.out.print(o + " [");
  }
}

class Post implements Traitement {
  public void traite(Object o) {
    System.out.print("] " + o);
  }
}
```

Il faudra maintenant invoquer :

```
Algorithmes.parcoursProfondeur(g, new Pre(), new Post())}
```

Notons que pour des traitements simples, une implémentation anonyme suffit (voir § 3.10). Si un seul des deux traitements est souhaité, par exemple le pré-traitement, il suffira que l'autre argument soit l'implémentation minimale suivante :

```
Algorithmes.parcoursProfondeur(g,
```

```
                              new Pre(),
                              new Traitement() {
                                public void traite(Object o) {}
                              });
```

2.14 ***Piles*** On a vu au § 1.6, page 20 que les cadres d'invocation des méthodes sont des blocs de mémoire « empilés » les uns sur les autres, le dernier empilé étant le premier retiré (en anglais *last in, first out* ou *LIFO*), comme une vulgaire pile d'assiettes. Les piles permettent également de parcourir un graphe en profondeur sans récursivité. La figure 2.11 montre le fonctionnement d'une pile. Les quatre opérations sur une pile sont : ajouter un élément au sommet de la pile (ou empiler), lire la valeur se trouvant au sommet d'une pile non-vide, tester si une pile est vide, retirer l'élément au sommet de la pile (ou dépiler). En anglais, « pile » se dit *stack* et les opérations portent respectivement les noms *push*, *top*, *isEmpty*, *pop*. Voici l'interface des piles, en Java :

```
interface Pile {
  boolean estVide();
  void empiler(Object o);
  Object sommet();
  Object dépiler();
}
```

On va implémenter les piles en utilisant les fonctionnalités des listes et leur implémentation sous forme de tableau fournie par l'API Java :

```
import java.util.List;
import java.util.ArrayList;

class PileParListe implements Pile {

  private List contenu;

  PileParListe() {
    contenu = new ArrayList();
  }

  public boolean estVide() {
    return contenu.isEmpty();
  }

  public void empiler(Object o) {
    contenu.add(o);
  }

  public Object sommet() {
      return contenu.get(contenu.size()-1);
  }
```

```
  public Object dépiler() {
      return contenu.remove(contenu.size()-1);
  }
}
```

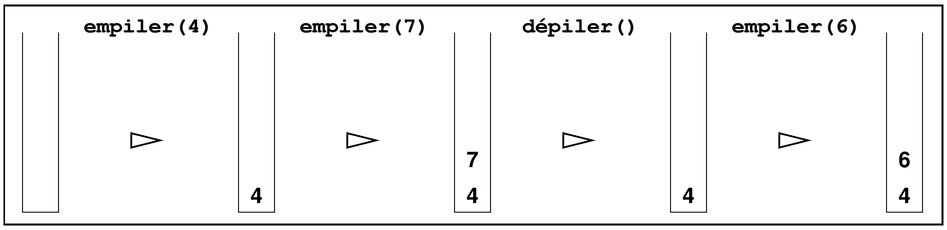

FIG. 2.11 – *Fonctionnement d'une pile.*

Parcours en profondeur Il est possible de transformer la version récursive du parcours en profondeur en une version itérative, à l'aide d'une pile. Les sommets du graphe sont empilés et traités au dépilement. Seules importent les fonctionnalités des piles, une implémentation quelconque des piles peut être utilisée. L'ensemble des sommets visités est la réunion de l'ensemble des sommets visités une première fois (qui sont dans la pile) et de l'ensemble des sommets traités (qui ont été dépilés).

```
static void parcoursProfondeur(Sommet origine,
                               Set sommetsVisités,
                               Traitement pré,
                               Traitement post) {
  Pile pile = new PileParListe() ;
  pile.empiler(origine);
  sommetsVisités.add(origine);
  while (!pile.estVide()) {
    Sommet s = (Sommet)pile.dépiler();
    pré.traite(s);
    Iterator i = s.adjacents();
    while (i.hasNext()) {
      Sommet suivant = (Sommet)i.next();
      if (!sommetsVisités.contains(suivant)) {
        pile.empiler(suivant);
        sommetsVisités.add(suivant);
      }
    }
    post.traite(s);
  }
}
```

Tri topologique d'un graphe sans circuit La relation d'accessibilité d'un graphe est un préordre qui n'est pas nécessairement un ordre (partiel): il se peut que $u \to^\star v$ et $v \to^\star u$ sans que $u = v$. Cette relation est un ordre si et seulement si le graphe est sans circuit. Les graphes sans circuit, qui constituent une généralisation des arborescences, interviennent dans de nombreuses applications, par exemple pour représenter une relation de précédence entre tâches, ou une relation de partage de sous-expressions. Une linéarisation de la relation d'ordre $\rightarrow^\star$ est une relation d'ordre total contenant cet ordre, c'est-à-dire une relation réflexive, anti-symétrique et transitive $\leq$ telle que $u \rightarrow^\star v$ implique $u \leq v$.

Un *tri* (ou un *parcours*) *topologique* est une énumération des sommets d'un graphe sans circuit selon une linéarisation de sa relation d'accessibilité (figure 2.12) ; ce terme de topologique est traditionnel et mal choisi. Un parcours topologique peut être obtenu en modifiant le parcours en largeur (en calculant et utilisant le degré entrant de chaque sommet), ou en appliquant le parcours en profondeur. On peut montrer que l'énumération postfixe des sommets, au cours d'un parcours en profondeur, produit les sommets dans l'ordre inverse d'un ordre topologique. Par conséquent, il suffit d'utiliser une pile `t` et de choisir comme post-traitement l'empilement de l'objet à traiter sur cette pile ; une fois le parcours terminé, les dépilements successifs de `t` produiront une énumération des sommets dans l'ordre topologique.

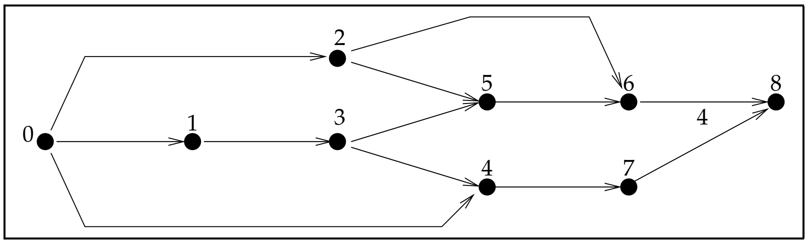

FIG. 2.12 – *Les sommets de ce graphe sans circuit sont numérotés selon un parcours topologique.*

2.15 *Files* Les files (en anglais *queue*) sont une autre structure linéaire dont l'usage est typique des « files d'attente » d'un service (d'un guichet, d'une liste d'attente, etc) : le premier arrivé est le premier servi (en anglais *first in, first out* ou *FIFO*). Contrairement aux piles, une file est accessible par ses deux extrémités, la tête et la queue. Les quatre opérations sont : tester si la file est vide, enfiler, c'est-à-dire entrer dans la file (« à la queue »), défiler, c'est-à-dire sortir de la file en tête, et enfin prendre la valeur de tête. Voici leur interface en Java :

```
interface File {
  boolean estVide();
  void enfiler(Object o);
  Object tête();
```

```
  Object défiler();
}
```

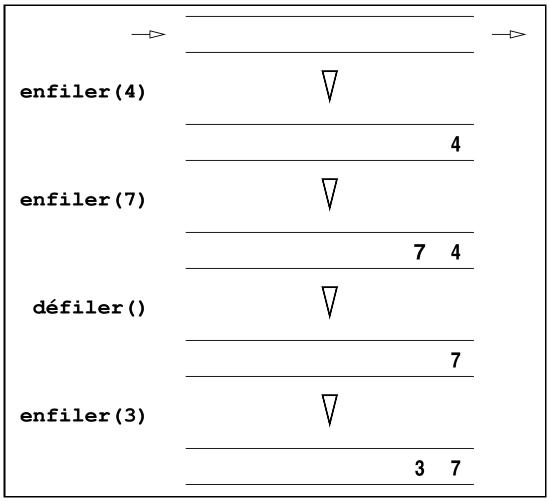

FIG. 2.13 – *Fonctionnement d'une file.*

Il existe plusieurs implémentations des files : par un tableau dont l'index est géré de façon circulaire, par une liste doublement chaînée, l'entrée se faisant à l'une des extrémités de la liste et la sortie à l'autre extrémité. C'est le choix retenu dans l'implémentation suivante, les opérations étant déléguées à une liste doublement chaînée de type `java.util.LinkedList`.

```
import java.util.LinkedList;

public class FileParListe implements File {
  private LinkedList contenu;

  FileParListe() {
    contenu = new LinkedList();
  }

  public boolean estVide() {
    return contenu.isEmpty();
  }

  public void enfiler(Object o) {
    contenu.addFirst(o);
  }

  public Object tête() {
    return contenu.getLast();
  }

  public Object défiler() {
```

```
    return contenu.removeLast();
  }
}
```

Parcours en largeur des graphes L'autre façon de parcourir un graphe est en largeur (d'abord). Les sommets de la figure 2.10 ont été numérotés dans l'ordre du parcours en largeur. On ne peut pas programmer récursivement ce parcours de façon naturelle. Sa version itérative utilise la structure de file au lieu de celle de pile.

```
public static void parcoursLargeur(Graphe g,
                                   Traitement v) {
  Iterator j = g.sommets();
  Set sommetsVisités = new HashSet();
  while (j.hasNext()) {
    Sommet s = (Sommet) j.next();
    if (!sommetsVisités.contains(s)) {
      parcoursLargeur(s, sommetsVisités, v);
    }
  }
}

static void parcoursLargeur(Sommet origine,
                            Set sommetsVisités,
                            Traitement v) {
  File file = new FileParListe() ;
  file.enfiler(origine);
  sommetsVisités.add(origine);
  while (!file.estVide()) {
    Sommet s = (Sommet) file.défiler();
    v.traite(s);
    Iterator i = s.adjacents();
    while (i.hasNext()) {
      Sommet suivant = (Sommet) i.next();
      if (!sommetsVisités.contains(suivant)) {
        file.enfiler(suivant);
        sommetsVisités.add(suivant);
      }
    }
  }
}
```

Cette fonction a exactement la même forme que celle implémentant le parcours itératif en profondeur, les piles étant simplement remplacées par les files, avec leurs opérations : ceci est un indice du pouvoir *structurant* des structures de données.

Comme pour le parcours en profondeur, le parcours en largeur construit implicitement une arborescence couvrante (ou une forêt d'arborescences) ; il suffirait d'insérer l'affectation suivante après l'entrée dans la file du sommet suivant :

```
suivant.chgParent(s);
```

En outre, le parcours en largeur permet d'énumérer les sommets par ordre de niveau d'accessibilité (c'est-à-dire de nombre d'arcs depuis l'origine) croissant, l'origine ayant le niveau 0. Il suffirait d'ajouter à l'interface `Sommet` une méthode qui incrémente le niveau d'un sommet, et d'insérer l'instruction suivante :

```
suivant.incrémenteNiveau();
```

2.16 Arbres binaires étiquetés Un arbre binaire est soit l'arbre vide, soit formé de deux arbres binaires, appelés fils gauche et fils droit. On considère souvent des arbres étiquetés (dans les exemples qui suivent, ces étiquettes seront des entiers, mais elles peuvent être de n'importe quel type) : un *arbre binaire étiqueté* (en anglais *labelled binary tree*), ce qu'on abrégera ici en « arbre », est soit l'arbre vide, soit formé d'une étiquette et de deux arbres, appelés fils gauche et fils droit. Un sous-arbre d'un arbre est soit un fils, soit un sous-arbre d'un fils. Un arbre vide n'a pas de sous-arbre. Les *feuilles* d'un arbre sont des sous-arbres dont les deux fils sont des arbres vides. La figure 2.14 représente un arbre binaire étiqueté par des entiers.

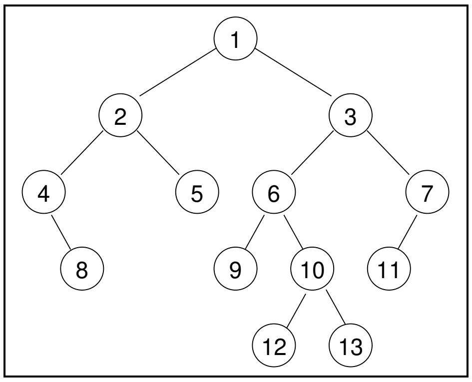

FIG. 2.14 – *Arbre binaire étiqueté.*

Comme le suggère cette représentation, un arbre binaire a une structure de graphe non orienté : l'ensemble de ses sommets, ou *nœuds*, est formé de l'arbre et de tous ses sous-arbres non vides, et il y a une arête entre u et v si v est un fils de u. Ce graphe est connexe et sans circuit, autrement dit c'est un arbre, au sens de la théorie des graphes (§ 2.12). Un arbre binaire a en fait une structure plus riche : il est aussi enraciné et ordonné. Sa racine est le sommet associé à l'arbre lui-même, et chaque nœud est la racine du sous-arbre correspondant ; les nœuds qui ne sont pas des feuilles sont dits *internes*. Par exemple, l'arbre de la figure 2.14 est non vide ; il a donc une *racine*, étiquetée par 1, et deux fils ; ses nœuds internes sont étiquetés par 1, 2, 3, 4, 6, 7, 10 ; ses feuilles sont étiquetées par 8, 5, 9, 12, 13, 11 ; il a en tout 13 nœuds. De plus, un arbre binaire est ordonné au sens où l'on peut distinguer le sous-arbre gauche et le sous-arbre droit.

Arbres bicolores Un *arbre binaire de recherche* est un arbre binaire étiqueté, dont les étiquettes sont comparables, tel que l'étiquette de la racine est plus grande que les étiquettes du sous-arbre gauche et plus petite que les étiquettes du sous-arbre droit. Cette structure permet de réaliser des opérations d'insertion, de recherche et de suppression avec en coût en $O(h)$, où h est la hauteur de l'arbre (la longueur maximale d'un chemin de la racine à une feuille). Or, la hauteur d'un arbre binaire comportant n nœuds est comprise entre $\log_2 n$ et $n+1$; les opérations ne sont donc pas toujours efficaces. De plus, l'insertion de nouveaux nœuds dans un arbre peut dégrader les performances des opérations ultérieures. Par exemple, si l'on insère successivement n nœuds d'étiquettes croissantes, l'arbre obtenu sera linéaire de hauteur $n + 1$, ce qui est le pire des cas. Il est possible de réarranger l'arbre après chaque insertion ou suppression, de manière à conserver une configuration favorable. Si l'on adopte une définition stricte de ce qu'est un arbre *équilibré*, les réarrangements peuvent être très coûteux. Une définition plus tolérante des déséquilibres permet de réaliser ces réarrangements moins souvent, donc à un moindre coût global : on fait en sorte qu'aucun chemin de la racine vers une feuille ne soit plus de deux fois plus long qu'un autre. Cette contrainte est respectée grâce à un coloriage des nœuds. C'est l'idée des arbres bicolores (ou rouges et noirs), qui sont des sont des arbres binaires de recherche approximativement équilibrés introduits par Bayer en 1972.

Un *arbre bicolore* est une arbre binaire de recherche dont les nœuds sont colorés en rouge ou en noir et vérifiant les propriétés suivantes :

– chaque feuille est noire ;
– si un nœud est rouge, ses deux enfants sont noirs ;
– pour chaque nœud, tous les chemins menant de ce nœud à une feuille ont le même nombre de nœuds noirs.

Les arbres bicolores forment la base des implémentations `TreeSet` `TreeMap` des interfaces `Set` et `Map` du paquet `java.util` de Java 2.

2.17 Algorithmes gloutons Les algorithmes gloutons (ou voraces, en anglais : *greedy*) construisent une solution de façon incrémentale, en choisissant à chaque étape la direction qui est la plus prometteuse. Ce choix localement optimal n'a aucune raison de conduire à une solution globalement optimale. Cependant, certains problèmes peuvent être résolus ainsi. La construction d'un code de Huffman est un exemple classique d'algorithme glouton. Dans d'autres cas, la méthode gloutonne est seulement une heuristique qui ne conduit qu'à une solution sous-optimale, mais qui peut être utilisée quand on ne connaît pas d'algorithme exact efficace. Notons qu'on ne peut recourir à un algorithme glouton que si une propriété d'optimalité locale est vérifiée. Les deux exemples suivants illustreront cette forme d'algorithme.

Arbre couvrant minimum Le problème est de connecter par un réseau (de câbles) un ensemble de points ; on connaît les distances entre tous les couples de points entre lesquels on peut installer un câble, et on veut minimiser la longueur totale du câble qui doit être utilisé. On modélise le problème par un graphe non orienté $\mathcal{G} = (S, A)$, muni d'une fonction de coût $c : A \to \mathbf{N}$. Une solution du problème est un sous-graphe $\mathcal{G}' = (S, A')$ connexe sans circuit, c'est-à-dire un arbre, de coût minimum, ce coût étant $c(\mathcal{G}') = \sum_{a \in A'} c(a)$; une telle solution est appelée un *arbre couvrant de coût minimum* (voir figure 2.15). On suppose que le problème a une solution, c'est-à-dire que le graphe $\mathcal{G}$ est connexe. Il existe plusieurs algorithmes pour résoudre ce problème. Ce sont des algorithmes gloutons qui construisent la solution de façon incrémentale en ajoutant une arête à chaque étape.

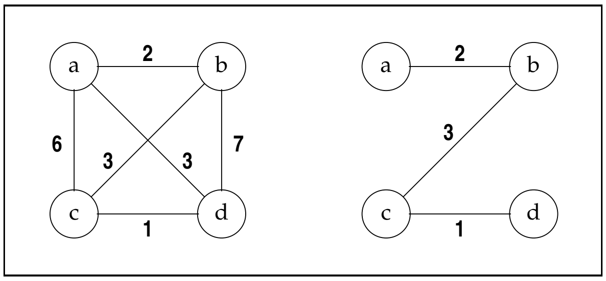

FIG. 2.15 – *Un graphe et un arbre couvrant de coût minimum 6.*

L'*algorithme de Kruskal* (1956) utilise une *forêt* $\mathcal{F}$ (c'est-à-dire un ensemble d'arbres disjoints). Initialement, la forêt est formée de tous les sommets du graphe, sans aucune arête. À chaque étape, on ajoute à la forêt une arête choisie de coût minimum, mais seulement si cette arête ne crée pas de circuit. Pour cela, on range d'abord les arêtes par ordre de coût croissant et on les considère dans cet ordre ; une arête dont les deux extrémités appartiennent au même arbre de $\mathcal{F}$ est refusée et ne sera plus considérée ; une arête qui relie deux arbres $a_1 \neq a_2$ de $\mathcal{F}$ est acceptée, elle est ajoutée à $\mathcal{F}$, ce qui a pour effet de joindre les arbres a_1 et a_2. L'arbre couvrant est cosntruit quand toutes les arêtes ont été examinées (figure 2.16).

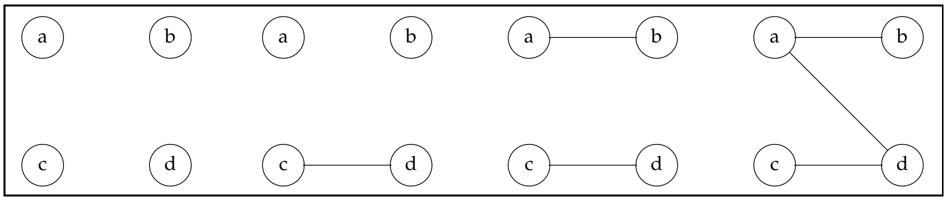

FIG. 2.16 – *Construction d'un arbre couvrant par l'algorithme de Kruskal ; les arêtes sont considérées dans l'ordre croissant cd, ab, ad, bc, ac, bd ; les trois premières sont acceptées, les suivantes sont refusées.*

Pour s'assurer que l'on ne crée pas de circuit, c'est-à-dire que l'arête choisie relie deux arbres distincts de $\mathcal{F}$, on gère une structure de données de *partition* (ou *d'ensembles disjoints*). En effet, la forêt $\mathcal{F}$ détermine une partition de l'ensemble S des sommets : S est réunion disjointe des ensembles de sommets de chaque arbre de $\mathcal{F}$. Les trois opérations de cette structure de données sont : initialiser, représenter, fusionner. L'initialiser consiste à créer une partition de S consistant en $|S|$ sous-ensembles, chacun à un élément ; `représenter()` associe à tous les éléments d'un sous-ensemble un même objet de sorte que des objets appartenant à des sous-ensembles distincts soient représentés par des objets distincts ; on peut alors déterminer si deux éléments de S appartiennent au même sous-ensemble en comparant leurs représentants ; `fusionner()` consiste à remplacer deux sous-ensembles par leur réunion.

```
interface Partition {
  Object représenter(int i);
  void fusionner(int i, int j);
}
```

Une implémentation de cette interface est réalisée par la classe `PartitionParArborescences`, à l'aide d'arborescences dont les sommets comportent un lien qui pointe vers le parent (contrairement aux arbres binaires où les liens pointent vers les enfants), la racine de chaque arborescence pointant vers elle-même. Les sommets de ces arborescences sont décrits par la classe membre `Arborescence`. Dans l'opération `fusionner()`, on peut choisir de faire de la racine du plus petit des deux arbres un enfant du plus grand ; un champ `rang` est utilisé à cet effet. On peut aussi profiter de chaque opération `représenter()` pour *comprimer* le chemin menant d'un élément à la racine en faisant de tous les sommets sur ce chemin des enfants de la racine (figure 2.17). On peut montrer que la complexité d'une suite de m opérations est $O(\alpha(m)m)$ où α est une fonction très lentement croissante et en pratique majorée par 4 (c'est un résultat de Tarjan qui date de 1975). Grâce à cette implémentation, la complexité de l'algorithme de Kruskal est en $O(|A| \log |S|)$.

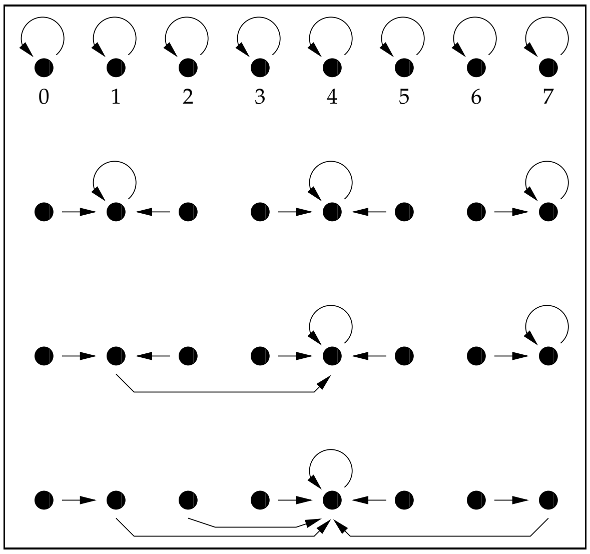

FIG. 2.17 – *Une structure de partition d'un ensemble à 8 éléments ; la partition initiale ; après les fusions de 0 et 1, 0 et 2, 2 et 3, 3 et 4, 6 et 7 ; après la fusion de 2 et 3 ; après la fusion de 6 et 7, le chemin $2 \to 1 \to 4$ est comprimé en $2 \to 4$.*

```
class PartitionParArborescences implements Partition {

  Arborescence[] t;

  // PartitionParArborescences(Set s) { ... }

  public Object représenter(int i) {
    return représenter(t[i]);
  }

  Arborescence représenter(Arborescence a) {
    if (a.parent != a) {
      a.parent = représenter(a.parent);
    }
    return a.parent;
  }

  public void fusionner(int i, int j) {
    relier(représenter(t[i]), représenter(t[j]));
  }

  void relier(Arborescence a, Arborescence b) {
    if (a.rang > b.rang)
      b.parent = a;
```

```
    else {
      a.parent = b;
      if (a.rang == b.rang) b.rang++;
    }
  }

  static class Arborescence {
    Arborescence parent;
    Object étiquette;
    int rang;

    Arborescence(Object étiquette) {
      this.parent = this;
      this.étiquette = étiquette;
    }
  }
}
```

L'*algorithme de Prim* (1957) construit incrémentalement un arbre en ajoutant à chaque étape une arête, choisie de coût minimum. Initialement, l'arbre est formé d'un unique sommet r, quelconque, du graphe.

L'algorithme utilise comme structure de données une *file de priorité* contenant à chaque étape tous les sommets qui n'appartiennent pas (encore) à l'arbre. Une file de priorité est une structure de données abstraite avec les opérations `insérer()`, `minimum()` et `extraireMin()`, et les objets insérés doivent implémenter l'interface `Comparable` :

```
interface FilePriorite {
  void insérer(Object o);
  Object minimum();
  Object extraireMin();
}
```

Une file de priorité peut être implémentée par un *tas-min* (Williams 1964), arbre binaire presque complet (tous les niveaux de l'arbre sont remplis, sauf peut-être le niveau plus bas, dont les nœuds sont tassés à gauche) ayant la propriété d'ordre suivante : un nœud est inférieur ou égal à ses enfants.

Chaque sommet du graphe inséré dans le tas-min est affecté d'un entier `val` qui est le coût minimum d'une arête reliant ce sommet à un sommet de l'arbre courant et d'un lien `parent` vers un sommet de l'arbre courant qui réalise ce minimum ; pour un sommet qui n'est pas reliable par une arête à l'arbre courant, l'entier `val` vaut ∞ et `parent` vaut `null`. Initialement, le tas-min est formé de tous les sommets, `val` étant ∞, sauf pour r, `val` étant 0 et `parent` étant `null`. Ensuite, tant que le tas-min est non vide, on en retire un sommet avec `val` minimum ; soit x ce sommet, qui appartiendra désormais à l'arbre couvrant ; pour chaque sommet y adjacent à x, si y est dans le tas-min et si $c(x, y) < y.\mathtt{val}$, alors on met à jour `val` et on fait de x

le parent de y, en exécutant les affectations $y.\texttt{parent} = x$ et $y.\texttt{val} = c(x, y)$. La complexité de cet algorithme dépend de l'implémentation du tas-min ; on sait réaliser l'opération `extraireMin()` en un temps $O(\log |S|)$. L'algorithme de Prim peut donc être implémenté en $O(|A| \log |S|)$.

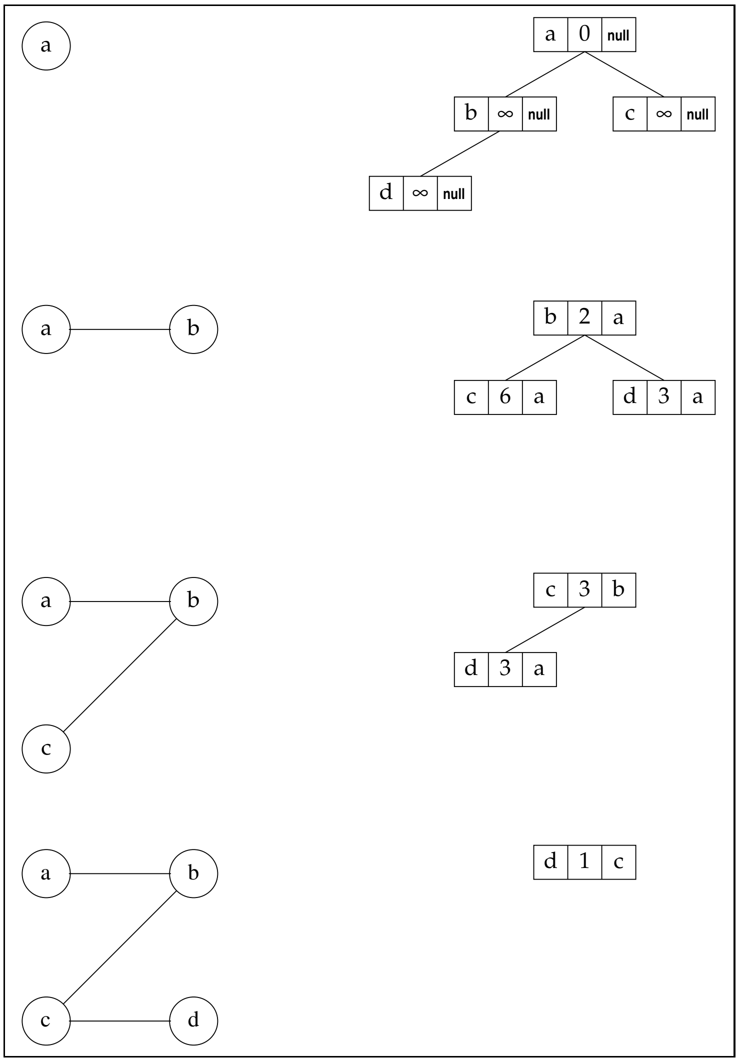

FIG. 2.18 – *Construction d'un arbre couvrant par l'algorithme de Prim ; sur la droite, le tas-min des sommets à ajouter à l'arbre couvrant.*

La construction d'un arbre couvrant de poids minimum a bien d'autres applications. Par exemple, les sommets du graphe peuvent représenter des données (mots, images, plantes, etc.), une arête ayant un coût mesurant une notion de proximité entre ces données (entre deux mots, deux images, deux plantes, etc.). On veut répartir ces données en deux classes telles que deux données de la même classe soient plus proches que deux données de deux classes différentes, et obtenir une classification en itérant ce partage (comme en botanique). On commence par construire un arbre couvrant de poids minimum de ce graphe ; on sélectionne une arête de plus grand coût dans cet arbre ; cette arête relie deux sous-arbres, dont les ensembles de sommets constituent les deux classes cherchées, au premier niveau de la classification. On itère ensuite la sélection d'une arête de plus grand coût dans les deux sous-arbres pour avoir les niveaux successifs de la classification.

2.18 Programmation dynamique Les problèmes d'optimisation dynamique ont des applications importantes aussi bien dans l'industrie qu'en gestion. Il s'agit de minimiser le coût d'une trajectoire dans un espace d'états. On dispose d'une loi d'évolution, qui détermine l'état suivant à partir de l'état courant et d'une « commande » ; les trajectoires sont construites à partir d'un état initial et d'une suite de commandes, suivant cette loi d'évolution ; on se donne également une fonction d'objectif, définie sur les trajectoires, qu'il s'agit de minimiser. La *programmation dynamique* est une méthode de résolution, pour les problèmes qui satisfont au principe d'optimalité de Bellman (1955) : une sous-trajectoire d'une trajectoire optimale est elle-même optimale pour la fonction d'objectif restreinte aux trajectoires ayant pour origine celle de cette sous-trajectoire. Ce principe permet une méthode de résolution ascendante[2], qui détermine une solution optimale d'un problème à partir des solutions de tous les sous-problèmes.

Une approche descendante pourrait être tentée : à partir du problème initial, générer ses sous-problèmes, les résoudre (récursivement), et déterminer la trajectoire optimale à partir des trajectoires optimales obtenues pour les sous-problèmes. Cette approche conduit en général à la génération d'un nombre exponentiel de sous-problèmes (par exemple, 2 à la première étape, 4 à la seconde, . . . , 2^n à la n-ème étape) ; cependant, quand l'ensemble des sous-problèmes a une taille inférieure à cette estimation du nombre de sous-problèmes générés, les sous-problèmes générés ne peuvent pas être tous distincts. Par exemple, pour le problème « trouver le plus court chemin entre deux sommets d'un graphe », quand le graphe a n sommets, il y a n^2 sous-problèmes « trouver le plus court chemin de u à v », pour chaque couple (u, v) de sommets. Pourtant, à chaque étape, il faudrait générer $2n$ sous-problèmes, « trouver le plus court chemin de u à w » et « trouver le plus court chemin de w à v », pour chacun des n sommets du graphe. Dans ces situations, un même sous-problème sera résolu plusieurs fois. On peut combiner cette approche descendante avec une mise en mémoire (ou *tabulation*) du résultat d'un sous-problème : ainsi, quand le même sous-problème

2. C'est l'idée du troisième précepte de la *méthode* de Descartes.

se présente une deuxième fois, il suffit de lire en mémoire le résultat, ce qui évite de tenter de le résoudre à nouveau.

L'approche descendante avec mise en mémoire peut être utilisée indépendamment du principe d'optimalité. Par exemple, le calcul de la fonction de Fibonacci peut en bénéficier, comme le montre le programme suivant, où la mise en mémoire utilise une liste :

```
import java.util.List;
import java.util.ArrayList;

class Fibonacci {

  private static List mémoire = new ArrayList(20);

  public static int fibonacci(int n) {
    if (n<=1)
      return 1;
    else {
      if (mémoire.get(n)!=null) {
        return ((Integer) mémoire.get(n)).intValue();
      } else {
        int f = fibonacci(n-1) + fibonacci(n-2);
        mémoire.set(n, new Integer(f));
        return f;
      }
    }
  }

  static void main(String[] args) {
    int max = Integer.parseInt(args[0]);
    for (int n=max-1; n>=0; n--) {
      System.out.println("fibonacci(" + n +") = " +
                         fibonacci(n));
    }
  }
}
```

La programmation dynamique est au contraire une approche ascendante, non récursive. Signalons que le terme « programmation » n'a pas ici de signification informatique, mais désigne plutôt une technique de « tabulation » (comme par exemple, la programmation linéaire).

2.19 ***L'algorithme de Floyd*** Un exemple simple de programmation dynamique, mais classique, est celui du calcul des plus courts chemins dans un graphe, par l'algorithme de Floyd (1962) ; on doit plutôt parler de chemins à moindre coût, ce problème n'ayant aucune signification métrique.

Représentons un graphe à N sommets comme un tableau c d'entiers positifs de taille $N \times N$; l'élément c_{ij} est le coût de l'arc $i \to j$; si l'arc n'existe pas, on pose $c_{ij} = \infty$, de sorte qu'on travaille avec un graphe complet. Le coût d'un chemin est la somme des coûts des arcs de ce chemin. Considérons un chemin de coût minimal entre i et j et $k \neq i, j$ un sommet intermédiaire sur ce chemin : les sous-chemins de ce chemin, de i à k et de k à j sont aussi de coût minimal (sinon, en remplaçant un de ces sous-chemins par un chemin de moindre coût de mêmes extrémités, on diminuerait le coût du chemin de i à j, ce qui est impossible). Ce problème satisfait donc au principe d'optimalité.

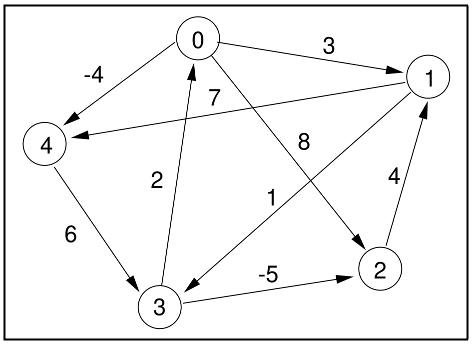

FIG. 2.19 – *Un graphe valué par les coûts des arcs.*

L'algorithme de Floyd est une méthode ascendante, qui calcule successivement pour k croissant de 1 à N, pour tous les couples de sommets i, j, les chemins minimaux de i à j parmi ceux dont les sommets intermédiaires sont dans $\{1, \ldots, k\}$, chemins appelés les k-chemins. Notons d_{ij}^k le coût d'un chemin de i à j minimal parmi les k-chemins. On doit calculer d_{ij}^n à partir de $d_{ij}^0 = c_{ij}$. Considérons un k-chemin de coût minimum. S'il ne contient pas k, c'est un $(k-1)$-chemin de coût minimum ; s'il contient k, on peut le diviser en deux sous-chemins de i à k et de k à j, qui sont eux-mêmes des $k-1$-chemins, et par le principe d'optimalité, chacun de ces sous-chemins est un $(k-1)$-chemin de coût minimal ; comme l'un des deux cas se produit, on a la relation :

$$d_{ij}^k = \min(d_{ij}^{k-1}, d_{ik}^{k-1} + d_{kj}^{k-1}).$$

Voici sur l'exemple de la figure 2.19, les matrices d^k successives, calculées d'après cette relation (on a utilisé ∞ pour indiquer la non-existence d'un arc):

$$d^0 = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \infty & -5 & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad d^1 = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix}$$

$$d^2 = \begin{pmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad d^3 = \begin{pmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & -1 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix}$$

$$d^4 = \begin{pmatrix} 0 & 3 & -1 & 4 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix} \quad d^5 = \begin{pmatrix} 0 & 1 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix}$$

On peut interpréter cette formule récurrente comme une fonction récursive en i, j, k ; une exécution récursive conduirait à des invocations multiples, comme dans le cas de la définition récursive de la suite de Fibonacci. L'algorithme de Floyd procède au contraire par une évaluation ascendante. On l'appliquera par exemple à `c` pour calculer les coûts minimaux dans `d`:

```
static final int M = Integer.MAX_VALUE;

int[][] c = {
  {0, 3, 8, M, -4},
  {M, 0, M, 1, 7},
  {M, 4, 0, M, M},
  {2, M, -5, 0, M},
  {M, M, M, 6, 0}
};
```

La fonction `floyd()` implémente cet algorithme :

```
static int[][] floyd(int[][] c) {
  int n = c.length;
  int[][] d = new int[n][n];
  for (int i=0; i<n; i++) {
    for (int j=0; j<n; j++) {
      d[i][j] = c[i][j];
      }
  }
  for (int k=0; k<n; k++) {
    for (int i=0; i<n; i++) {
      for (int j=0; j<n; j++) {
        int dik = d[i][k], dkj = d[k][j];
        if (dik == M || dkj == M) continue;
        int u = dik + dkj;
        if (u < d[i][j]) {
          d[i][j] = u;
        }
      }
    }
  }
```

```
    return d;
  }
```

Il est facile de modifier cet algorithme pour être capable de construire les chemins minimaux : il suffit de conserver dans un autre tableau `p` le prédecesseur de j dans un k-chemin de coût minimal et d'ajouter l'affectation `p[i][j] = p[k][j]` juste après `d[i][j] = u`. Initialement, $p_{ij} = i$, si $i \neq j$ et $c_{ij} < \infty$, et une fois l'algorithme terminé, un chemin de coût minimal de i à j est un chemin de coût minimal de i à p_{ij}, suivi de l'arc $p_{ij} \to j$.

2.20 Ordonnancement de projet Certains problèmes d'ordonnancement de projet sont représentables à l'aide d'un graphe dont les sommets sont des événements et les arcs des tâches, ceux-ci étant étiquetés par leur durée ; le sommet initial et final d'un arc représente, respectivement, l'événement de début et de fin d'exécution de la tâche (figure 2.20). On cherche à déterminer les tâches critiques, c'est-à-dire celles dont un retard est répercuté sur l'ensemble du projet. Ce problème revient à trouver dse chemins *les plus longs* dans un graphe.

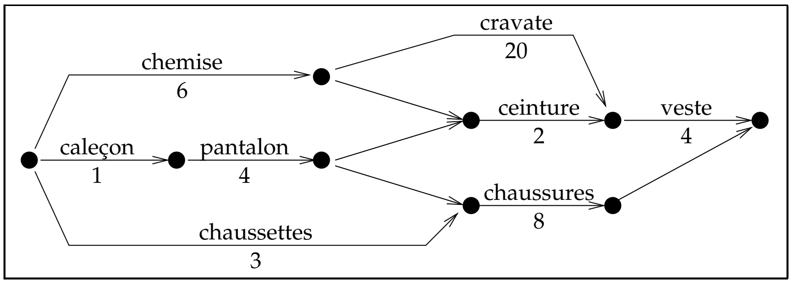

FIG. 2.20 – *Un graphe représentant un projet d'habillement standard.*

On modélise un projet par un graphe $\mathcal{G} = (S, A)$ et une fonction de durée $d : A \to \mathbf{N}$. On suppose qu'un sommet initial s et un sommet final t ont été désignés. La longueur d'un chemin est la somme des durées des arcs le composant. Un chemin *critique* est un chemin de s à t de longueur maximale. S'il n'existe pas de chemin critique, le projet n'est pas réalisable. Le temps minimal requis pour l'exécution d'un projet réalisable est égal à la longueur d'un chemin critique ; c'est la *durée du projet*. Les tâches figurant sur un chemin critique sont aussi appelées critiques, ce qui s'explique par le fait que tout retard d'exécution d'une tâche critique retarde d'autant l'exécution du projet.

Une *exécution* du projet est définie par une fonction de date de début d'exécution des tâches $t : A \to \mathbf{N}$. Si le projet est réalisable, on définit l'exécution *au plus tôt*, $t_{\min}$, et l'exécution *au plus tard*, $t_{\max}$: $t_{\min}(x, y)$ est la longueur maximale des chemins de s à x (elle ne dépend pas de y), $t_{\max}(x,y)$ est la date la plus tardive pour commencer la tâche (x,y) telle que la durée totale de l'exécution soit la durée du projet. On notera qu'une tâche (x,y) est critique si $t_{\min}(x,y) = t_{\max}(x,y)$. Il est plus simple de calculer des fonctions $t_{\min}, t_{\max} : S \to \mathbf{N}$ définies par :

$$\begin{aligned} t_{\min}(y) &= \max_{x \to y}(t_{\min}(x) + d(x,y)) \\ t_{\max}(x) &= \min_{x \to y}(t_{\max}(y) - d(x,y)) \end{aligned}$$

Ces fonctions sur les sommets sont reliées aux fonctions sur les arcs de la façon suivante :

$$\begin{aligned} t_{\min}(x,y) &= t_{\min}(x) \\ t_{\max}(x,y) &= t_{\max}(y) - d(x,y) \end{aligned}$$

L'algorithme consiste en un double parcours du graphe. Dans une première phase, on calcule les dates $t_{\min}(x))$, initialisées à 0, à partir du sommet initial s, en procédant par un tri topologique du graphe (voir § 2.14, p. 96). Dans une seconde phase, les $t_{\max}(x)$ sont calculées à partir du sommet final, en l'initialisant à l'aide de $t_{\min}(t)$ (la durée du projet, maintenant connue), par un tri topologique en ordre inverse. Ceci permet de déterminer si le projet est réalisable et dans le cas positif, effectuer le calcul de sa durée, des dates au plus tôt et au plus tard, et dire quelles sont les tâches critiques (figure 2.21).

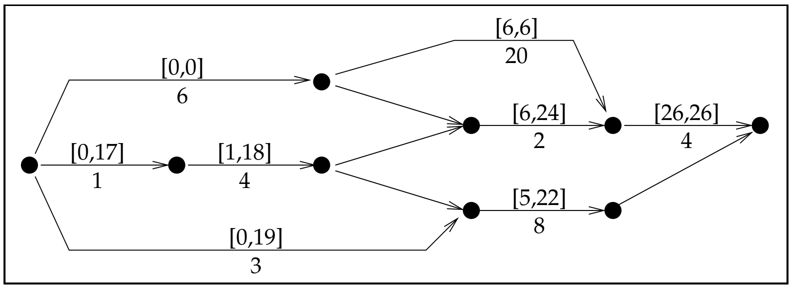

FIG. 2.21 – *Les dates au plus tôt et au plus tard $[t_{\min}(x,y), t_{\max}(x,y)]$ de chaque tâche : la durée du projet est 30, les tâches critiques sont la chemise, la cravate et la veste.*

2.21 *Réseaux de transport* Certains problèmes de transport (de marchandises, de fluides, d'énergie, etc) dans un réseau (routier, de distribution, etc) sont représentables de façon naturelle par des graphes dont les sommets sont des points de passage (sans stockage) et les arcs des trajets entre ces points. Le problème consiste à maximiser l'utilisation de tels réseaux, c'est-à-dire à calculer la quantité maximale transportable compte tenu des contraintes. Ce problème a bien d'autres réalisations, par exemple pour déterminer un appariement maximum dans un graphe biparti.

On modélise un réseau de transport à l'aide d'un graphe $\mathcal{G} = (S, A)$, muni d'une fonction de capacité $c : S \times S \to \mathbf{N}$, telle que $c(x, y) = 0$ si $(x, y) \notin A$. On suppose qu'un sommet *source* s et un sommet *puits* t ont été désignés (figure 2.22).

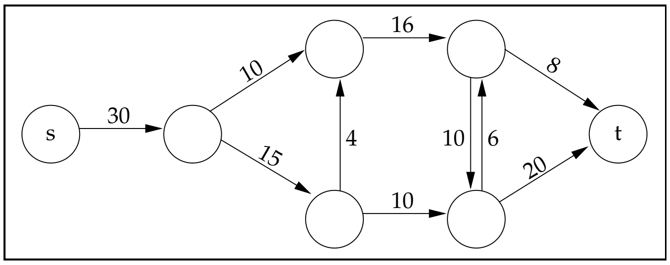

FIG. 2.22 – *Un réseau de transport* $(\mathcal{G}, c)$.

On appelle *flot* une fonction $f : S \times S \to \mathbf{Z}$ telle que (contraintes de capacité et de conservation, Cf. les lois de Kirchhoff) :

1. quels que soient $x, y \in S$, $f(x, y) \leq c(x, y)$
2. quels que soient $x, y \in S$, $f(x, y) = -f(y, x)$
3. quel que soit $x \neq s, t$, $\sum_{y \in S} f(x, y) = 0$

Un flot est représenté sur la figure 2.23 par la notation $f(x, y)/c(x, y)$ en étiquette des arcs. La *valeur* du flot f est le nombre $|f| = \sum_{y \in S} f(s, y)$, qui est aussi égal à $\sum_{x \in S} f(x, t)$; elle mesure la quantité totale transportée par le flot depuis la source vers le puits. Un arc est *saturé* par le flot s'il est utilisé au maximum de sa capacité, c'est-à-dire si $f(x, y) = c(x, y)$.

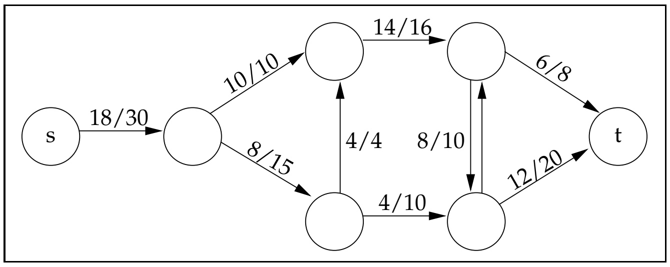

FIG. 2.23 – *Un flot* f *de valeur 18 dans* $(\mathcal{G}, c)$ *; deux arcs (*$10/10$ *et* $4/4$*) sont saturés par* f*, mais ce flot n'est pas maximal.*

Le problème est de déterminer un flot de valeur maximale. Sa résolution fait appel à la notion de réseau résiduel d'un flot. Étant donné un réseau de graphe $\mathcal{G} = (S, A)$ et de capacité c et un flot f, on définit $c_f(x,y) = c(x,y) - f(x,y)$, pour $(x,y) \in S \times S$; le *réseau résiduel* de f est défini par le graphe $\mathcal{G}_f = (S, A_f)$ dont l'ensemble des arcs est $A_f = \{(x,y) \in S \times S; c_f(x,y) > 0\}$, et par la capacité c_f (figure 2.24) ; on notera que A_f n'est pas nécessairement inclus dans A, car il peut contenir aussi l'arc réciproque d'un arc de A.

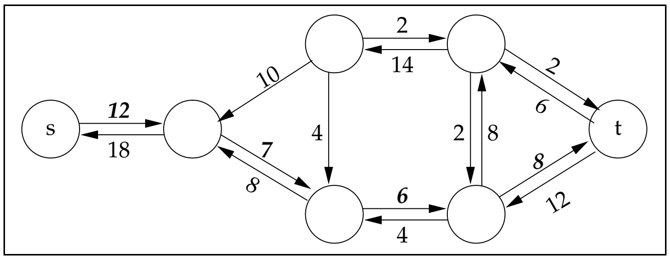

FIG. 2.24 – *Le graphe résiduel $\mathcal{G}_f$ du flot de la figure 2.23. Les arcs saturés par f ne figurent plus dans $\mathcal{G}_f$, mais des arcs réciproques sont apparus ; un chemin d'augmentation p est représenté en étiquetant les arcs en italique ; ce chemin est de capacité 6.*

Un chemin d'*augmentation* de f dans un réseau est un chemin simple (c'est-à-dire sans boucle) de s vers t dans le graphe résiduel $\mathcal{G}_f$. La capacité d'un chemin p est la quantité $c(p)$ maximale transportable le long de p, soit $c(p) = \min\{c_f(x,y); (x,y) \in p\}$. Un chemin d'augmentation permet d'augmenter f en définissant le flot noté $f + p$ (figure 2.25) :

$$\begin{array}{rcl} (f+p)(x,y) & = & f(x,y) + c(p) \quad \text{si } (x,y) \in p \\ (f+p)(x,y) & = & f(x,y) - c(p) \quad \text{si } (y,x) \in p \\ (f+p)(x,y) & = & f(x,y) \quad \text{sinon} \end{array}$$

Il s'agit bien d'un flot et c'est une augmentation de f au sens où $|f+p| > |f|$.

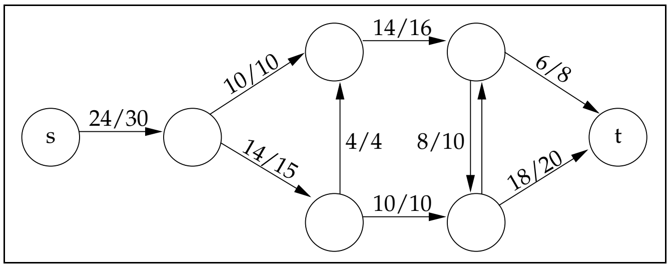

FIG. 2.25 – *Le flot augmenté $f + p$ défini par le chemin p de $\mathcal{G}_f$; un nouvel arc est saturé (10/10). Le graphe résiduel de $f + p$ ne comporte plus de chemin de s vers t. Le flot est donc maximal.*

On montre alors qu'un flot est maximal si et seulement si son graphe résiduel ne contient aucun chemin d'augmentation. L'algorithme de Ford-Fulkerson (1956) consiste à augmenter le flot selon un chemin d'augmentation s'il en existe :

– le flot f est initialisé à 0 ;
– (itération) tant qu'il existe un chemin d'augmentation dans $\mathcal{G}_f$, en choisir un de longueur (nombre d'arcs) minimum, p (cette recherche se fait par un parcours en largeur), augmenter f selon p, puis calculer le nouveau graphe résiduel $\mathcal{G}_f$;
– s'il n'existe pas de chemin d'augmentation dans $\mathcal{G}_f$, alors f est un flot de valeur maximale.

Il faut choisir une structure de données adéquate ; on notera que les graphes résiduels partagent le même ensemble de sommets, mais que l'ensemble des arcs est variable. Correctement implémenté, cet algorithme a une complexité en $O(|S||A|^2)$.

2.22 *Automates finis* Étudiés par Kleene au début des années 50, les automates finis furent auparavant utilisés par McCulloch et Pitts comme un modèle « neuronal » de calcul et par Shannon pour décrire le comportement d'un canal de communication. Ils constituent la classe la plus simple de *machines abstraites*.

Un automate fini est défini par la donnée d'un alphabet A, d'un ensemble fini d'*états* E, d'une relation de *transition*, sous-ensemble de $E \times A \times E$, d'un état initial $I \in E$ et d'un ensemble d'états finaux $F \subseteq E$. Une transition (e, a, e'), dite de l'état e vers l'état e' et étiquetée par le symbole a, est notée $e \xrightarrow{a} e'$. En termes de graphes, les automates sont des multigraphes étiquetés, les sommets étant les états, les arcs étant étiquetés par les symboles de l'alphabet. Un *calcul* de cet automate est une suite de transitions $I = e_1 \xrightarrow{a_1} e_2 \xrightarrow{a_2} \cdots \xrightarrow{a_{n-1}} e_n$; ce calcul est *réussi* si le dernier état, e_n, est un état final, et on dit alors que le mot $a_1 a_2 \cdots a_{n-1}$ est *reconnu* par l'automate fini.

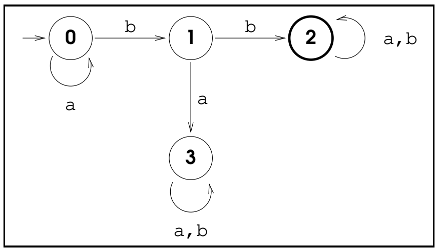

FIG. 2.26 – *Un automate déterministe sur l'alphabet $\{a, b\}$, à quatre états, 0 est l'état initial, 2 est final.*

Un langage X est *régulier* s'il existe un automate fini dont l'ensemble des mots reconnus est exactement X. On montre qu'il existe alors un automate fini *déterministe* et *complet* (c'est-à-dire, pour tout état, il existe exactement une transition par symbole de l'alphabet) reconnaissant X. Ces automates finis peuvent être vus comme des machines abstraites, parmi les plus simples, puisqu'elles n'utilisent pas de mémoire, dont le fonctionnement est facile à simuler à l'aide d'un programme.

Un automate fini déterministe dont les transitions sont étiquetées par des valeurs de type `Object` peut être implémenté au moyen des deux classes suivantes :

```
class Etat {
  private boolean acceptant;
  private Map transitions;

  Etat(boolean acceptant) {
    this.acceptant = acceptant;
    this.transitions = new HashMap();
  }

  void ajouteTransition(Object c, Etat q) {
    transitions.put(c, q);
  }
  boolean accepte() {return acceptant;}
  Etat transition(Object c) {
    return (Etat) transitions.get(c);
  }
}

class Automate {

  private Etat initial;

  Automate(Etat initial) {
    this.initial = initial;
  }

  boolean accepte(List s) {
    Etat q = initial;
    for (Iterator i=s.iterator(); i.hasNext() && q!=null;)
      q = q.transition(i.next());
    return q!=null ? q.accepte() : false;
  }
}
```

Si `q` est un état, `q.accepte()` indique si `q` est final, et `q.transition(c)` retourne l'état résultant de la transition issue de `q` et étiquetée par l'objet `c`. L'unique constructeur de la classe `Automate` spécifie un état initial. Si `a` est un automate, et si `l` est une liste d'objets, `a.accepte(l)` indique s'il existe un calcul étiqueté par les objets successifs de `l`, menant de l'état initial à un état final. Un automate est défini à l'aide de son état initial. Voici l'exemple correspondant à la figure 2.26 :

```
class AutomateTest {

  public static void main(String[] args) {
    Etat
      q0 = new Etat(false),
      q1 = new Etat(false),
      q2 = new Etat(true),
      q3 = new Etat(false);
    q0.ajouteTransition(new Character('a'), q0);
    q0.ajouteTransition(new Character('b'), q1);
    q1.ajouteTransition(new Character('a'), q3);
    q1.ajouteTransition(new Character('b'), q2);
    q2.ajouteTransition(new Character('a'), q2);
    q2.ajouteTransition(new Character('b'), q2);
    q3.ajouteTransition(new Character('a'), q3);
    q3.ajouteTransition(new Character('b'), q3);
    Automate a = new Automate(q1);

    String s = args.length==1 ? args[0] : "";
    List l = new ArrayList();
    for (int i=0; i<s.length(); i++)
      l.add(new Character(s.charAt(i)));
    System.out.println("chaîne \"" + s + "\" acceptée : "
                       + a.accepte(l));
  }
}
```

Expressions rationnelles Le produit de concaténation des mots s'étend naturellement aux langages : si X et Y sont des langages d'alphabet A, le produit XY est l'ensemble des mots uv tels que $u \in X$ et $v \in Y$. On définit aussi les *puissances*, l'*étoile* et l'*étoile propre* d'un langage :

$$X^0 = \{\varepsilon\}, \quad X^{n+1} = X^n X, \text{pour } n \geq 0, \quad X^* = \bigcup_{n\geq 0} X^n, \quad X^+ = \bigcup_{n\geq 1} X^n$$

Les langages d'alphabet A, en tant que sous-ensembles de $A^\star$, peuvent aussi être combinés par les opérations ensemblistes : union, intersection, complémentation.

On dit qu'un langage est *rationnel* s'il peut être obtenu à partir des langages singletons (à un seul mot) par un nombre fini d'unions, de produits et d'étoiles. Par exemple $\{a\}^\star\{b\}\{b\}(\{a\} \cup \{b\})^\star$ est un langage rationnel formé des mots commençant par un nombre quelconque de a, suivi de deux occurrences de b et se terminant par un nombre quelconque de a et de b. Ce langage peut être défini au moyen de l'*expression rationnelle* $a^\star bb(a|b)^\star$. Un important résultat de la théorie des langages est dû à Kleene (1956) : un langage est reconnaissable par un automate fini si et seulement s'il est rationnel. Par exemple, le langage défini par l'expression rationnelle $a^\star bb(a|b)^\star$ est reconnu par l'automate déterministe de la figure 2.26. C'est ainsi que sont réalisées les opérations de recherche de la commande `egrep` d'Unix.

Une telle expression est un objet formel, dont la valeur est un langage, de même que la valeur d'une expression arithmétique est un nombre. Les expressions rationnelles (ou *régulières*, parfois dénommées en anglais informatique *regexp*) sont largement utilisées à la fois dans les fonctions de recherche des éditeurs de texte et dans l'environnement Unix. Ainsi, la commande `egrep` *exp f* de Unix recherche toutes les lignes d'un fichier *f* contenant une chaîne de caractères appartenant au langage valeur de l'expression rationnelle *exp*.

Par exemple, les chaînes de caractères représentant un identificateur valide en Java (et dans la plupart des langages de programmation) forment un langage rationnel sur l'alphabet Unicode. Les langages de ce type sont considérés comme les langages les plus simples, après les langages finis. On dispose ainsi, avec la notion d'expression rationnelle, d'un moyen formel pour définir certains langages. Cependant, ces expressions ne donnent pas directement d'algorithme résolvant le problème d'appartenance d'un mot à un langage d'expression rationnelle donnée.

2.23 ***Analyse lexicale*** Un compilateur comporte toujours une première phase qui détermine les unités lexicales à l'aide d'un automate fini.

Considérons le texte suivant :

```
int f(int arg) {
  return 2*arg+1;
}
```

Ce texte est d'abord découpé en *lexèmes* : dans l'exemple ci-dessus, `int`, `(`, `return` et `1` sont des lexèmes, tandis que `in` ou `int f` n'en sont pas. Chaque lexème appartient à une *unité lexicale*, qui comporte un ou plusieurs lexèmes. Voici les unités lexicales correspondant aux exemples précédents : **ident** (qui contient tous les identificateurs), **par-gauche** (qui contient seulement la parenthèse ouvrante), **return** (qui contient seulement le mot-clé `return`), **nombre** (qui contient tous les lexèmes numériques).

L'analyse lexicale d'un texte, c'est-à-dire d'un mot sur l'alphabet Unicode, consiste à le transformer en une suite d'unités lexicales, c'est-à-dire en un mot sur un autre alphabet. Par exemple, le texte précédent `int f ...` est transformé en **ident ident par-gauche ident ident par-droite acc-gauche return nombre mult ident plus nombre point-virgule acc-droite**. Certaines unités lexicales sont affectées d'une valeur (par exemple, la valeur numérique d'un **nombre**, ou la valeur textuelle d'un **ident**).

L'API de Java dispose d'une classe `StreamTokenizer`, dans le paquet `java.io` qui aide à effectuer l'analyse lexicale d'un flot d'entrée. Selon le type de langage qui doit être analysé, il peut être utile de spécialiser cette classe ; voici par exemple une classe adaptée à l'analyse lexicale de programmes Java :

```
class JavaTokenizer extends StreamTokenizer {
   JavaTokenizer(InputStreamReader in) {
     super(in);
     slashSlashComments(true);
     slashStarComments(true);
     ordinaryChar('/');
     ordinaryChar('.');
     wordChars('_', '_');
   }
}
```

La méthode `nextToken()` retourne l'unité lexicale suivante sous forme d'un entier ou bien la constante `TT_EOF` si la fin du flot est atteinte ; le champ `ttype` contient soit la valeur de l'unité lexicale (la constante de classe `TT_WORD` ou `TT_NUMBER`) si c'est un identificateur ou un nombre, ou quand le lexème est un caractère ordinaire, le code de ce caractère, ou, quand le lexème est une chaîne littérale, le caractère de citation (`"` ou `'`).

```
class Lexer {
  public static void main(String[] args) throws IOException {

    JavaTokenizer jt =
      new JavaTokenizer(
         new FileReader(args[0]));

    while (jt.nextToken() != StreamTokenizer.TT_EOF) {
      switch(jt.ttype) {
      case StreamTokenizer.TT_WORD:
        System.out.print("ident "); break;
      case StreamTokenizer.TT_NUMBER:
        System.out.print("nombre "); break;
      default: System.out.print((char)jt.ttype + " ");
      }
    }
    System.out.println();
  }
}
```

L'exécution de ce programme, sur le texte `int f ...` ci-dessus produit la suite d'unités lexicales suivantes :

```
ident ident ( ident ident )
{ ident nombre * ident + nombre ; }
```

Le lexème correspondant à une unité lexicale **ident** est obtenu à l'aide de la variable `jt.sval`, de type `String` ; le lexème correspondant à une unité lexicale **nombre** est converti en un nombre, obtenu dans la variable `jt.nval`, de type `double`.

Pour vérifier que le texte `int f ...` est un programme Java correct, on doit vérifier que le mot **ident ident ...** vérifie les règles de grammaire de Java, ce qu'on appelle l'*analyse syntaxique*. Cette vérification ne peut pas se faire avec un automate fini, car elle nécessite de garder en mémoire les états passés. On doit donc combiner un automate fini avec une *pile*, mais la description de cet algorithme dépasse le cadre de ce cours.

2.24 ***Graphes de jeu et arbres minimax*** Certains jeux (échecs, Othello, Awélé) peuvent être décrits à l'aide d'un graphe dont les sommets sont les *configurations* possibles (positions des pièces, etc) du jeu et les arcs sont les *coups* permis par les règles du jeu ; on notera $s \to s'$ s'il existe un coup changeant la configuration s en s'. On s'intéressera aux jeux à deux joueurs à information complète : on suppose qu'ils jouent à tour de rôle, que chacun connaît la configuration du jeu et que le hasard n'intervient pas.

Une configuration est *finale* si aucun coup n'est permis à partir de celle-ci. Dans un certain nombre de jeux, une configuration finale est perdante pour le joueur qui s'y trouve. Si l'on connaît l'intégralité du graphe du jeu, on peut décider de proche en proche, à partir des configurations finales, si une configuration est gagnante ou perdante, en appliquant les deux règles suivantes :

- une configuration est gagnante s'il existe un coup menant à une configuration perdante,
- une configuration est perdante si tout coup mène à une configuration gagnante.

Un jeu est *à somme nulle* si les gains d'un joueur représentent exactement les pertes de l'autre joueur. Supposons maintenant que la perte du joueur qui se trouve dans la configuration finale s est donnée par l'entier $p_f(s) \in \mathbf{Z}$. Appelons les deux joueurs Xavier et Yvette : si Xavier perd $p_f(s)$, Yvette perd $-p_f(s)$, c'est-à-dire gagne $p_f(s)$. Toujours si l'on connaît l'intégralité du graphe du jeu, on peut étendre p_f en une fonction p définie pour toutes les configurations, par les règles suivantes :

- si s est une configuration finale, $p(s) = p_f(s)$,
- si s est une configuration gagnante, $p(s) = \max_{s \to s' \text{ perdante}} p(s')$,
- si s est une configuration perdante, $p(s) = \min_{s \to s' \text{ gagnante}} p(s')$.

La fonction p est une évaluation complète du jeu, pour des joueurs cherchant à maximiser leurs gains et minimiser leurs pertes. La construction de p se fait donc de façon *ascendante*, en remontant à partir des feuilles du graphe : chaque noeud s se trouve affecté d'un attribut $p(s)$, qui est *synthétisé*, c'est-à-dire calculé de façon ascendante à partir des attributs $p(s')$ des successeurs s' de s.

L'exploration exhaustive d'un graphe de jeu n'est généralement pas faisable, du moins pour les jeux « intéressants ». La plupart des joueurs ne décident pas du coup à jouer en remontant à partir des configurations finales, mais en descendant à partir de la configuration courante. Au lieu de construire le graphe du jeu, on va travailler avec un *arbre de recherche* qui représente une exploration de ce graphe jusqu'à une certaine profondeur P. Plus P est grand, plus l'arbre de recherche est grand et donc long à explorer et meilleur (en principe) est le coup joué.

Au lieu de mesurer la perte du joueur en configuration finale, on utilise une mesure heuristique h indiquant l'intérêt d'une configuration quelconque pour Xavier : h peut prendre en compte le nombre des pièces, leurs positions, etc. (plus $h(s)$ est grand, plus la configuration s est bonne). Si c'est au tour de Xavier de jouer et que le jeu est dans la configuration s, il est dans l'intérêt immédiat de Xavier de choisir le coup qui fera passer le jeu dans la configuration s', telle que $h(s')$ est maximale parmi les configurations accessibles en un coup à partir de s. Si c'est au tour d'Yvette, elle cherchera au contraire s' telle que $h(s')$ est minimale.

Un arbre *minimax* de profondeur P a pour racine la configuration courante et des noeuds qui sont, par niveaux alternants, des noeuds de maximisation et des noeuds de minimisation. La figure 2.27 montre un arbre minimax de profondeur 2. L'évaluation avec la fonction h se fait aux feuilles à la profondeur P.

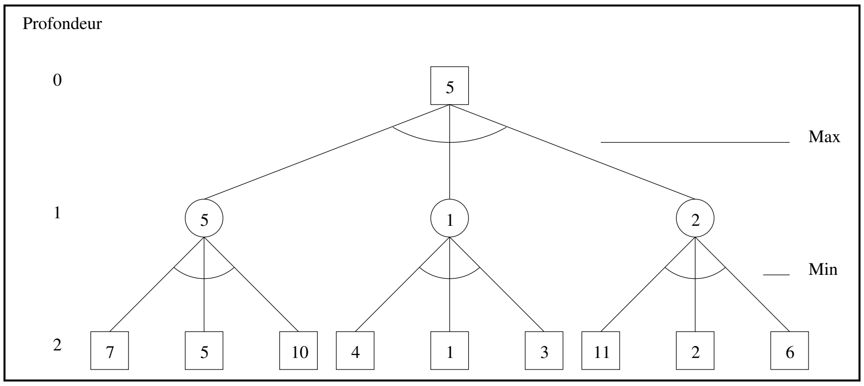

FIG. 2.27 – *Arbre minimax de profondeur 2.*

Nous supposons que le type `Configuration` offre les méthodes :

– `boolean estFeuille()`, qui teste si la configuration est finale ;

- `int type()` qui détermine le type, minimisant ou maximisant de la configuration, sous forme d'une des deux constantes de classe `Configuration.MIN` ou `Configuration.MAX`;
- `Iterator succs()`, qui retourne un itérateur sur les configurations accessibles en un coup ;
- `int h()`, qui retourne une évaluation heuristique d'une configuration en position de feuille.

La fonction `minimax()` remonte la meilleure valeur jusqu'à la racine ce qui détermine le meilleur coup à jouer :

```
static int minimax(Configuration s) {
  if (s.estFeuille()) return s.h();
  switch(s.type()) {
    case Configuration.MAX :
      return max{minimax(s′) | s → s′} ;
    case Configuration.MIN :
      return min{minimax(s′) | s → s′} ;
  }
}
```

La fonction `minimax()` est une évaluation heuristique du jeu, pour des joueurs cherchant à maximiser leurs gains et minimiser leurs pertes. La construction de `minimax()`, comme celle de p se fait donc de façon *ascendante*, en remontant à partir des feuilles de l'arbre : $h(s)$ est également un attribut synthétisé.

2.25 *L'algorithme* $\alpha\beta$ L'algorithme minimax effectue une exploration complète de l'arbre de recherche jusqu'à un niveau donné, alors qu'une exploration partielle de l'arbre pourrait suffire. Il suffit en effet, dans l'exploration en profondeur d'abord et de gauche à droite, d'éviter d'examiner des sous-arbres qui conduiront à des configurations dont la valeur ne contribuera sûrement pas au calcul du gain à la racine de l'arbre.

Dans les exemples de la figure 2.28 certains nœuds ont une valeur définitive alors que les autres (étiquetés avec un nom de variable) n'en ont pas encore reçu. D'après la définition de la fonction `minimax()` la valeur de la configuration racine de l'arbre *(a)* est obtenue par

$$u = \max\{5, v\}, \text{ où } v = \min\{4, \ldots\}.$$

Il est clair que $u = 5$ indépendamment de la valeur de v. Il en résulte que l'exploration des branches filles du nœud étiqueté par v peut être omise : on réalise ainsi une *coupure superficielle*. En appliquant récursivement le même raisonnement à l'arbre *(b)*, on en déduit que la valeur de u peut être obtenue sans connaître la valeur finale de y. De même que précédemment, l'exploration des branches filles du nœud étiqueté par y n'est pas nécessaire : on parle alors de *coupure profonde*.

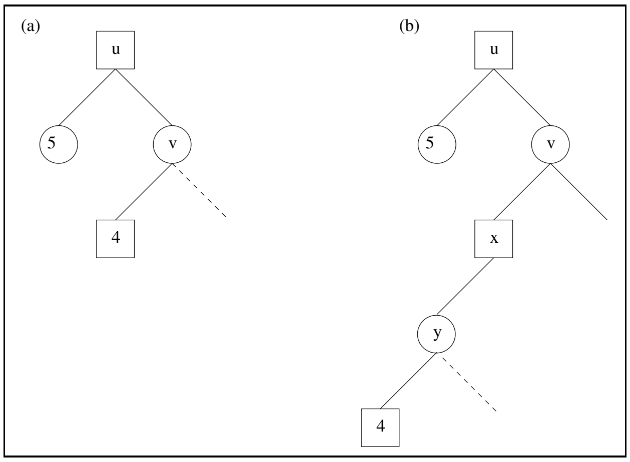

FIG. 2.28 – *Coupure superficielle (a) et profonde (b).*

Plus généralement, lorsque dans le parcours de l'arbre minimax il y a modification de la valeur courante d'un nœud, si cette valeur franchit un certain seuil, il devient inutile d'explorer la descendance encore inexplorée de ce nœud. On distingue deux seuils, appelés α (pour les nœuds Min) et β (pour les nœuds Max) :

– le seuil α, pour un nœud Min s, est égal à la plus grande valeur (déjà déterminée) de tous les nœuds Max ancêtres de s ; si la valeur de s devient inférieure ou égale à α, l'exploration de sa descendance peut être arrêtée ;
– le seuil β, pour un nœud Max s, est égal à la plus petite valeur (déjà déterminée) de tous les nœuds Min ancêtres de s. Si la valeur de s devient supérieure ou égale à β, l'exploration de sa descendance peut être arrêtée.

L'algorithme $\alpha\beta$ peut être décrit informellement par la fonction suivante, qui maintient ces deux seuils pendant le parcours de l'arbre. Une invocation de `alphabeta(s, −∞, +∞)` détermine une évaluation du jeu issu de la configuration s.

```
static int alphabeta (Configuration s, int α, int β) {
   if (s.estFeuille()) return s.h();
   Iterator i = s.succs();
   switch(s.type()) {
     case Configuration.MAX : {
```

```
        int m = α;
        while(i.hasNext()) {
          Configuration s1 = (Configuration)i.next();
          int t = alphabeta(s1, m, β);
          if (t >  m) m = t;
          if (m >= β) return m;
        }
        return m;
      }
      case Configuration.MIN : {
        int m = β;
        while(i.hasNext()) {
          Configuration s1 = (Configuration)i.next();
          int t = alphabeta(s1, α, m);
          if (t <  m) m = t;
          if (m <= α) return m;
        }
        return m;
      }
    }
  }
```

Contrairement aux fonctions p et `minimax()` (§ 2.24), le calcul des valeurs de `alphabeta()` se fait de façon à la fois *ascendante* et *descendante*. Chaque noeud de l'arbre se trouve affecté de trois attributs : la valeur m de la configuration et les seuils α et β. L'attribut m est synthétisé, c'est-à-dire calculé de façon ascendante à partir des successeurs s' de s. Par contre, les seuils sont des attributs *hérités*, c'est-à-dire calculés de façon descendante, à partir des parents : par exemple, si s est un noeud de maximisation avec deux successeurs, soit $s \to s_1, s_2$, alors le seuil α de s_2 est la valeur m de s qui a été calculée à partir de s_1. L'information a donc circulé de s_1 à s, puis de s à s_2.

La construction d'arbres comportant des attributs et l'évaluation de ces attributs sont des techniques importantes utilisées, non seulement dans les arbres de jeu, mais aussi dans la compilation des programmes.

2.26 *Diviser pour régner* Un algorithme *diviser pour régner* a la structure suivante : pour résoudre un problème de taille n, l'algorithme consiste à décomposer le problème en a sous-problèmes ayant tous la taille n/b (peut-être approximativement), à appliquer l'algorithme à tous les sous-problèmes, puis à construire une solution du problème en composant les solutions des sous-problèmes. Non seulement cette méthode est naturelle (et voisine du second précepte de la *méthode* de Descartes), mais elle permet souvent une réduction importante de $T(n)$, la complexité dans le pire des cas de l'algorithme appliqué à un problème de taille n. On suppose que la complexité des opérations de décomposition du problème et de recomposition des solutions des sous-problèmes a une complexité en $d(n)$. Enfin, on se donne la complexité $T(1)$ sur un problème de taille 1. La complexité de l'algorithme est donc déterminée par l'équation de récurrence suivante :

$$T(n) = aT(n/b) + d(n)$$

Voici quelques exemples :

- recherche dichotomique dans une table ordonnée ($a = 1$, $b = 2$) : $T(n) = T(n/2) + \Theta(1)$, d'où $T(n) = \Theta(\log n)$, alors qu'une recherche séquentielle dans une table ordonnée est en $\Theta(n)$;
- tri d'une table par fusion ($a = 2$, $b = 2$) : $T(n) = 2T(n/2) + \Theta(n)$, d'où $T(n) = \Theta(n \log n)$, alors qu'un tri ordinaire est typiquement en $\Theta(n^2)$;
- transformée de Fourier rapide ($a = 2$, $b = 2$) : $T(n) = 2T(n/2) + \Theta(n)$, d'où $T(n) = \Theta(n \log n)$, alors qu'une évaluation directe des formules est en $\Theta(n^2)$;
- multiplication de deux matrices par l'algorithme de Strassen ($a = 7$, $b = 2$) : $T(n) = 7T(n/2) + \Theta(n^2)$, d'où $T(n) = \Theta(n^{\log 7}) \approx \Theta(n^{2,81})$ alors que la multiplication ordinaire est en $\Theta(n^3)$.

Résolvons ces équations quand n est une puissance de b ; dans ce cas l'équation s'écrit :

$$T(b^p) = aT(b^{p-1}) + d(b^p)$$

Sa solution est

$$T(b^p) = a^p T(1) + \sum_{j=0}^{p-1} a^j d(b^{p-j})$$

Le premier terme, $a^p = a^{\log_b n} = n^{\log_b a}$ correspond à la complexité due à la résolution de tous les sous-problèmes ; le second terme est la complexité due à toutes les opérations de décomposition/recomposition.

Examinons ce deuxième terme quand $d(n) = n^\delta$. On a

$$\sum_{j=0}^{p-1} a^j d(b^{p-j}) = b^{p\delta} \sum_{j=0}^{p-1} (\frac{a}{b^\delta})^j$$

Discutons selon les valeurs relatives de a et b^δ.

- Si $a = b^\delta$, cette somme vaut $b^{p\delta} p = n^\delta \log_b n$; donc $T(n) = n^{\log_b a} T(1) + n^\delta \log_b n$; ainsi, si $\delta = 1$ et $a = b$ (cas très usuel, en particulier de la FFT, du tri par fusion et du tri rapide), on a $T(n) = nT(1) + n \log_b n = \Theta(n \log_b n)$. Si $\delta = 0$ et $a = 1$ (cas de la recherche dichotomique), on obtient $T(n) = \log_b n$;

– Si $a \neq b^\delta$,
$$b^{p\delta}\sum_{j=0}^{p-1}(\frac{a}{b^\delta})^j = b^{p\delta}\frac{(a/b^\delta)^p - 1}{a/b^\delta - 1} = \frac{a^p - b^{p\delta}}{a/b^\delta - 1};$$
– Si $a > b^\delta$, le terme dominant est $a^p = n^{\log_b a}$, qui est indépendant de δ et qui est du même ordre que le premier terme, donc $T(n) = \Theta(n^{\log_b a})$. Ainsi, si $\delta = 1$ et $a > b$ (cas de la multiplication matricielle de STRASSEN, où $a = 7$ et $b = 2$), on a $T(n) = \Theta(n^{\log_b a})$;
– Si $a < b^\delta$, le terme dominant est $b^{p\delta} = n^\delta$, donc $T(n) = \Theta(n^\delta)$. La décomposition du problème ne conduit dans ce dernier cas à aucune accélération, les opérations de décomposition et recomposition étant dominantes. C'est le cas de l'équation $T(n) = 2T(n/2) + n^2$.

2.27 La transformée de Fourier rapide La FFT, ou transformée de Fourier rapide, est l'un des quelques algorithmes dont la publication a provoqué une véritable révolution dans le champ technique. Généralement associé aux noms de J.W. COOLEY et J.W. TUCKEY qui l'ont publié en 1965, cet algorithme de calcul de la transformée de Fourier discrète avait été maintes fois « redécouvert » depuis GAUSS, notamment par DANIELSON et LANCZOS en 1942. La FFT permet de ramener le calcul de la transformée de Fourier discrète de N^2 à $N \log N$ opérations ; cette réduction de complexité suffit à faire passer d'impossibles à facilement résolubles nombre de problèmes.

La transformée de Fourier discrète d'un n-uplet de nombres complexes $a = (a_0, \ldots, a_{n-1})$ est le n-uplet $\hat{a} = (\hat{a}_0, \ldots, \hat{a}_{n-1})$ défini par
$$\hat{a}_q = \sum_{p=0}^{n-1} a_p \omega_n^{pq}$$
où $\omega_n = e^{2\pi i/n}$. Notons que $\hat{a}_{q+n} = \hat{a}_q$ et que pour $n = 1$, ceci se réduit en $\hat{a} = a$. La transformée inverse se calcule ainsi :
$$a_p = \frac{1}{n}\sum_{q=0}^{n-1} \hat{a}_q \omega_n^{-pq}$$
Supposons que $n = 2m$ et séparons les éléments de a d'indices pairs de ceux d'indices impairs :
$$\begin{aligned} a^{[0]} &= (a_0, a_2, \ldots, a_{n-2}) \\ a^{[1]} &= (a_1, a_3, \ldots, a_{n-1}) \end{aligned}$$
Alors, en séparant la somme en deux termes regroupant d'une part les $p = 2j$ et d'autre part les $p = 2j + 1$, et en utilisant l'égalité $\omega_{2m}^{2jq} = \omega_m^{jq}$:
$$\hat{a}_q = \sum_{p=0}^{n-1} a_p \omega_n^{pq} \qquad (0 \leq q \leq n-1)$$

$$\begin{aligned}
&= \sum_{j=0}^{m-1} a_{2j}\omega_{2m}^{2jq} + \sum_{j=0}^{m-1} a_{2j+1}\omega_{2m}^{(2j+1)q} \\
&= \sum_{j=0}^{m-1} a_{2j}\omega_{m}^{jq} + \omega_{2m}^{q} \sum_{j=0}^{m-1} a_{2j+1}\omega_{m}^{jq} \\
&= \hat{a}_q^{[0]} + \omega_n^q \hat{a}_q^{[1]} \quad \text{, si } 0 \leq q \leq m-1 \\
\hat{a}_q &= \hat{a}_{q-m}^{[0]} + \omega_n^q \hat{a}_{q-m}^{[1]} \quad \text{, si } m \leq q \leq n-1
\end{aligned}$$

Cette dernière expression utilise la propriété de périodicité :

$$\begin{aligned}
\hat{a}_{q+m}^{[0]} &= \hat{a}_q^{[0]} \\
\hat{a}_{q+m}^{[1]} &= \hat{a}_q^{[1]}
\end{aligned}$$

Ainsi, pour calculer la transformée de Fourier sur n points, il suffit de calculer deux transformées de Fourier sur $n/2$ points, de faire n multiplications et n additions. Si n est une puissance de 2, cette décomposition peut s'appliquer récursivement, ce qui donne pour équation de complexité : $T(n) = 2T(n/2) + \Theta(n)$. Il en résulte une complexité en $\Theta(n \log n)$ opérations.

Il existe aussi une FFT sur l'anneau $\mathbf{Z}/N\mathbf{Z}$ des entiers modulo N, quand $N = 2^{tn/2} + 1$, t est un entier quelconque ; on pose $\omega = 2^t$, qui est une racine n-ème principale de l'unité dans $\mathbf{Z}/N\mathbf{Z}$; on a toujours $n = 2^h$. Par exemple, quand $t = 2$ et $n = 2^3 = 8$, on a $N = 257$, et $\omega = 2^2 = 4$; un n-uplet et sa transformée sont :

$$\begin{aligned}
a &= (255, 8, 0, 226, 37, 240, 3, 0) \\
\hat{a} &= (255, 85, 127, 200, 78, 255, 194, 75)
\end{aligned}$$

Pour traiter de façon identique le cas complexe et le cas modulo N, on utilise l'interface suivante, la première méthode servant à injecter un entier dans l'anneau (`entier(0)` est le zéro et `entier(1)` est l'unité de l'anneau) :

```
interface Nombre {
    Nombre entier(int i);
    Nombre add(Nombre y);
    Nombre sub(Nombre y);
    Nombre mult(Nombre y);
}
```

On supposera que l'on dispose d'implémentations `Complexe` et `Entier-Modulo` de cette interface.

La définition récursive de la FFT résulte directement de cette décomposition. La figure 2.29 représente l'arbre des invocations de ce calcul récursif

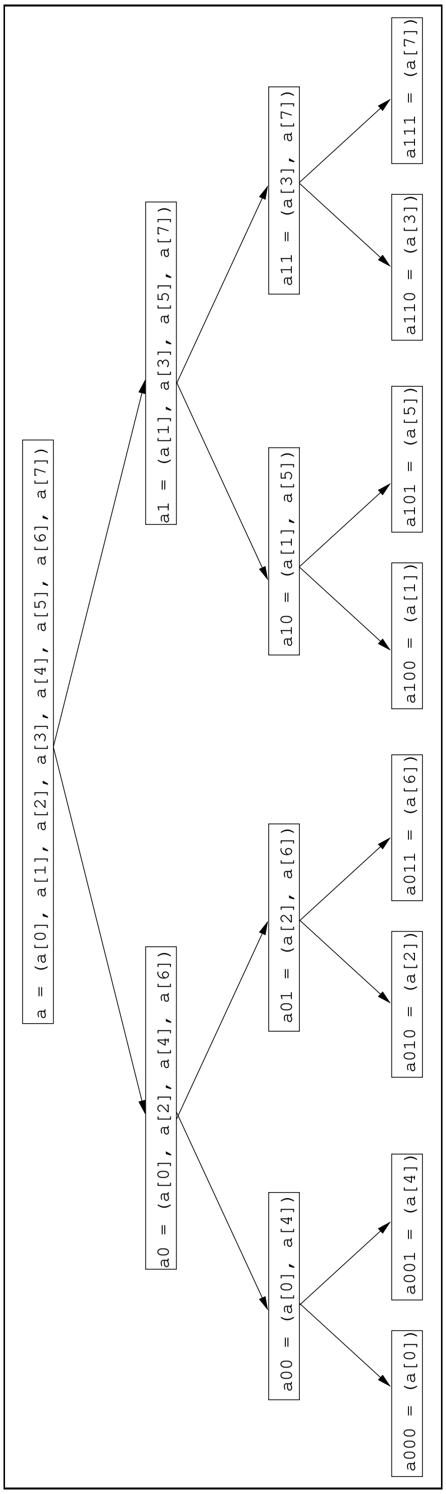

FIG. 2.29 – *Arbre des invocations de la FFT sur 8 points.*

pour $n = 8$ points : aux feuilles, la FFT sur 1 point, qui est l'identité. L'ordre dans lequel les a_p sont rangés dans les feuilles, de gauche à droite, est l'inverse de la représentation binaire de p. La fonction `miroir(h,p)` calcule cet inverse binaire de p, pour $0 \leq p \leq 2^h - 1$; par exemple, $\texttt{miroir}(3, 3) = \texttt{miroir}(3, \texttt{011}_2) = \texttt{110}_2 = 6$; on utilise les opérateurs binaires `<<`, `>>`, `&`, `|` et `^`, plus efficaces pour opérer au niveau des bits d'un entier ; pour h fixé, cette fonction est une involution : `miroir(h, miroir(h, n))` = `n`.

```
static int miroir(int h, int n) {
  int r;                     // Résultat
  for (r = 0; h > 0; h--) {
    // A chaque itération le bit de droite de n est
    // est tranféré dans r comme bit de droite
    r <<= 1;          // Un nouveau bit à droite
    r |= (n & 1);     // Positionnement de ce bit
    n >>= 1;          // Effacement du bit de droite de n
  }
  return r;
}
```

On va donc commencer par permuter le n-uplet a pour ranger ses éléments dans cet ordre, ce qui va permettra de construire un algorithme itératif, qui parcourt l'arbre des feuilles vers la racine :

```
static void permuteTableau(Object[] a) {
  int h = log2(a.length);     // On suppose a.length = 2^h
  int i,j;

  for (i = 0; i < a.length; i++) {
    if ((j = miroir(h, i)) < i) {
      Object tmp = a[i];
      a[i] = a[j];
      a[j] = tmp;
    }
  }
}
```

En utilisant la périodicité de la transformée de Fourier, et l'égalité $\omega_n^m = -1$, on peut réécrire la dernière expression de $\hat{a}$:

$$\begin{array}{rcl} \hat{a}_q & = & \hat{a}_q^{[0]} + \omega_n^q \hat{a}_q^{[1]} \\ \hat{a}_{m+q} & = & \hat{a}_q^{[0]} - \omega_n^q \hat{a}_q^{[1]} \end{array} \qquad (0 \leq q \leq m-1)$$

Cette transformation, de la forme $(x, y) \mapsto (x + \alpha y, x - \alpha y)$, est appelée un *papillon* (en anglais, *butterfly*) ; on l'appliquera successivement avec $\alpha = 1$, $\omega, \omega^2, \ldots, \omega^{m-1}$. La fonction `butterfly()` s'écrit :

```
static void butterfly(Nombre[] a, int i, int j, Nombre alpha) {
  Nombre u = a[i];
  Nombre v = alpha.mult(a[j]);
```

```
    a[i] = u.add(v);
    a[j] = u.sub(v);
  }
```

Voici enfin la fonction de calcul de la FFT ; le paramètre `a` est un tableau dont la longueur n doit être une puissance de 2, et `racine` est une racine n-ième de l'unité :

```
  public static Nombre[] fft(Nombre[] a, Nombre racine) {
    int h = log2(a.length);
    int i, j, l;
    int pas = 1;

    // Calcul des racine^{2^l} pour 0 <= l < h
    Nombre[] puissancesRacine = new Nombre[h];
    puissancesRacine[0] = racine;
    for(l = 1; l< h; l++)
      puissancesRacine[l] =
        puissancesRacine[l-1].mult(puissancesRacine[l-1]);

    // Permutation du tableau
    permuteTableau(a);

    // Itération
    for(l = h-1, pas = 1; l >=0; l--, pas *= 2) {
      Nombre alpha = racine.Entier(1); // Unité
      for (i = 0; i < pas; i++) {
        for (j = i; j < a.length; j += 2*pas) {
          butterfly(a, j, j+pas, alpha);
        }
        alpha = alpha.mult(puissancesRacine[l]);
      }
    }
    return a;
  }
```

2.28 Tri d'un tableau De nombreux algorithmes ont été proposés et étudiés pour résoudre le problème du tri, qui se présente fréquemment dans les problèmes de gestion de données : par exemple, on dispose d'un annuaire alphabétique des abonnés au téléphone, et on veut produire un annuaire classé par adresses. On présentera seulement trois algorithmes : le tri par insertion, de type incrémental, puis le tri par fusion et le tri rapide, tous deux de type « diviser pour régner ».

Le tri par insertion procède de façon incrémentale, à la manière d'un joueur de carte qui range les cartes, au fur et à mesure qu'il les reçoit, à leur place parmi les précédentes.

```
  static void triInsertion(int[] t) {
    for (int j=1; j<t.length; j++) {
      int x = t[j];
```

```
      int i = j-1;
      // x doit être inséré dans le tableau ordonné 0..j-1
      while (i>=0 && t[i]>x) {
        t[i+1] = t[i];
        i = i-1;
      }
      t[i+1] = x;
    }
  }
```

La complexité de cet algorithme est en $\Theta(n^2)$. S'il est facilement utilisable pour des tableaux de taille réduite (≤ 100), cette complexité est prohibitive pour une utilisation sur de grandes bases de données.

Tri par fusion En appliquant la méthode « diviser pour régner » au problème du tri d'un tableau, on obtient facilement le tri par fusion (en anglais, *mergesort*) : on divise une table de longueur n en deux tables de longueur $n/2$, on trie, récursivement, ces deux tables, puis on fusionne les tables triées. Comme la fusion de deux tables ordonnées de longueur $n/2$ se fait en $\Theta(n)$, l'équation de complexité est $T(n) = 2T(n/2) + \Theta(n)$, dont la solution est en $\Theta(n \log n)$. Ce tri est implémenté dans les fonctions `Collections.sort(List l)` et `Arrays.sort(Object[] t)` de l'API, pour trier respectivement des listes et des tableaux d'objets. Voici une version optimisée de ce tri. Pour trier un tableau t, on commence par en créer une copie c, puis on appelle la fonction récursive `triFusion()` qui écrit dans t le résultat du tri de c ; chaque invocation récursive échange le rôle des tableaux source et destination :

```
  static void triFusion(Object[] t) {
    Object[] copie = (Object[])t.clone();
    triFusion(copie, t, 0, t.length);
  }

  static void triFusion(Object source[], Object destination[],
                        int début, int fin) {
    int longueur = fin - début;

    if (longueur <= 1) return;
    else {
      int milieu = (début + fin)/2;
      triFusion(destination, source, début, milieu);
      // source est trié de début à milieu
      triFusion(destination, source, milieu, fin);
      // source est trié de milieu à fin

      // Fusion des tableaux triés dans destination
      for (int i = début, p = début, q = milieu;
           i < fin;
           i++) {
```

```
      if (q>=fin || p<milieu &&
          ((Comparable)source[p]).compareTo(source[q])<=0) {
        destination[i] = source[p];
        p++;
      } else {
        destination[i] = source[q];
        q++;
      }
    }
  // destination est trié de debut à fin
  }
}
```

Outre sa complexité en $\Theta(n \log n)$, le tri par fusion a l'avantage d'être stable, c'est-à-dire que des éléments de même clé ne sont pas échangés. Ceci permet de trier successivement un tableau selon plusieurs clés et de conserver l'ordre obtenu lors des tris précédents.

Tri rapide Il existe cependant un meilleur algorithme de tri (Hoare, 1962) appelé *tri rapide*, en anglais *quicksort*, qui est utilisé également par l'API Java, pour le tri des tableaux dont les éléments sont de type primitif, par les fonctions `Arrays.sort(int[] t)`, etc. ; il diffère du tri par fusion en ce que la décomposition en deux tables est calculée, la recomposition des tables triées étant immédiate. Cette décomposition se fait par une fonction `partition()`, qui choisit un élément de la table, appelé *pivot*, réorganise la table en déplaçant tous les éléments plus petits que le pivot à la gauche du pivot et tous les éléments plus grands à sa droite, et retourne l'indice de l'élément pivot après réorganisation.

```
static void échangerÉléments(int[] t, int m, int n) {
  int temp = t[m];

  t[m] = t[n];
  t[n] = temp;
}

static int partition(int[] t, int m, int n) {
  int v = t[m];                // valeur pivot
  int i = m-1;
  int j = n+1;                 // indice final du pivot

  while (true) {
    do {
      j--;
    } while (t[j] > v);
    do {
      i++;
    } while (t[i] < v);
    if (i<j) {
      échangerÉléments(t, i, j);
```

```
      } else {
        return j;
      }
    }
  }

  static void triRapide(int[] t, int m, int n) {
    if (m<n) {
      int p = partition(t, m, n);
      triRapide(t, m, p);
      triRapide(t, p+1, n);
    }
  }
```

La fonction `partition()`, qui est pourtant itérative, est la partie difficile à écrire : celle-ci choisit comme pivot le premier élément de chaque tableau. Sa complexité est en $\Theta(n)$. Testé sur trois tableaux de taille 1000 dont les éléments sont respectivement, générés aléatoirement, générés par ordre croissant, générés par ordre décroissant, le nombre d'échanges d'éléments observé est respectivement : 5477, 999, et 250 999. On montre en effet, sous des hypothèses d'uniformité, que la complexité moyenne du tri rapide est en $\Theta(n \log n)$; en pratique, son comportement est même meilleur que celui du tri par fusion (car la constante devant $n \log n$ est plus petite). Mais comme il n'y aucune raison pour que la partition décompose une table de longueur n en deux tables de longueur $n/2$, la complexité dans le pire des cas est très mauvaise, puisqu'elle est en n^2.

2.29 Algorithmes stochastiques La définition classique des algorithmes en fait des processus de calcul déterministes : avec les mêmes données, un algorithme exécutera toujours la même suite d'opérations. Cependant, l'hypothèse de déterminisme est restrictive, voire contraignante. Un algorithme non-déterministe admettrait des instructions du genre « faire ceci ou faire cela», « choisir un élément dans un ensemble », etc ; deux exécutions différentes d'un tel algorithme pourraient réaliser des choix différents. Une façon de réaliser ces choix est de les rendre aléatoires : on obtient ainsi les algorithmes stochastiques, ou probabilistes. Il y a plusieurs usages de l'aléatoire :

– les algorithmes numériques simulent une variable de loi uniforme, afin d'obtenir une approximation numérique de la solution du problème (qui pourrait être obtenue par un algorithme déterministe) ;
– les algorithmes de Monte-Carlo peuvent calculer une solution incorrecte en un temps déterministe avec une probabilité d'erreur qui peut être rendue arbitrairement petite en répétant l'algorithme ;
– les algorithmes de Las Vegas ne terminent pas nécessairement, mais calculent toujours une solution correcte quand ils terminent, le temps

d'exécution étant aléatoire ;
– enfin, les algorithmes de randomisation garantissent une complexité moyenne.

Un générateur de nombres pseudo-aléatoires est un algorithme implémenté par une fonction, qui retourne à chaque invocation une nouvelle valeur numérique[3], et telle que la suite des valeurs retournées ait de bonnes propriétés statistiques : ces propriétés permettent de supposer qu'il s'agit d'une suite de variables aléatoires indépendantes de loi uniforme dans un intervalle spécifié. Ces nombres sont utilisés dans deux situations :

– pour tester un programme quelconque, en lui soumettant comme données des « jeux de test » générés aléatoirement, et en comparant le résultat calculé au résultat attendu ;
– pour implémenter des algorithmes stochastiques, soit pour des problèmes numériques (intégration de Monte-Carlo), soit pour améliorer le comportement d'algorithmes par randomisation, soit pour donner un résultat avec une probabilité d'erreur (test de primalité), ou pour briser des symétries (élection d'un chef), par obstination.

La conception d'un générateur de nombres pseudo-aléatoires est délicate (ce n'est pas que ces algorithmes soient difficiles à écrire, c'est qu'il est difficile d'échapper à des régularités arithmétiques qui s'opposent à l'uniformité souhaitée). Dans la pratique, il est préférable d'utiliser un générateur fourni dans la bibliothèque standard : la fonction `Math.random()` de l'API Java retourne un `double` dans l'intervalle $[0, 1[$.

Pour obtenir un entier dans l'intervalle $[a, b]$, on pourra définir la fonction :

```
static int irand(int a, int b) {
  return a + (int)(Math.random() * (b-a+1));
}
```

On obtiendra un tirage à pile ou face en invoquant `irand(0,1)`.

Comme la fonction `Math.random()` n'a en soi rien d'aléatoire, chaque exécution du programme verra la même suite générée. Si l'on doit faire des traitements statistiques portant sur les résultats de plusieurs exécutions, il est alors nécessaire d'obtenir une suite différente à chaque exécution, pour assurer l'indépendance des résultats. Il faut alors initialiser la *graine* (anglais *seed*) du générateur.

L'API Java offre la classe `Random` dans le paquet `java.util` qui permet cette initialisation à la création de l'instance. Sans argument, le constructeur utilise le temps courant pour l'initialiser. Une instance de cette classe

3. On notera combien cette notion informatique de « fonction » est distante de celle des mathématiciens, bien qu'elle ne diffère guère de la « variable aléatoire » des probabilistes, qui omettent souvent l'argument ω dans $X(\omega)$.

est un générateur de nombres pseudo-aléatoires dont les valeurs successives sont obtenues par l'une des méthodes `nextBoolean()`, `nextInt()`, `nextLong()`, `nextFloat()` et `nextDouble()` qui simulent des loi uniformes sur l'ensemble des valeurs de type, respectivement, `boolean`, `int`, `long`, `float` et `double`; la méthode `nextInt(int n)`, avec un argument entier positif n simule une loi uniforme sur l'intervalle $[0, n[$; enfin, la méthode `nextGaussian()` simule une gaussienne de moyenne 0 et de variance 1.

On obtiendra par exemple une suite de `doubles` pseudo-aléatoires de la façon suivante :

```
public static void main(String[] argv) {

  Random g = new Random();
  while (true)
    System.out.println(g.nextDouble());
}
```

2.30 *Un algorithme de Monte-Carlo : test de primalité* Le dernier des préceptes de Descartes est un obstacle à la conception de certains algorithmes : il faut savoir renoncer à l'exhaustivité et accepter des résultats « presque » sûrs.

L'un des meilleurs tests de primalité est celui de MILLER et RABIN, publié en 1976. Il appartient à la catégorie des algorithmes stochastiques de Monte-Carlo, et repose sur deux propriétés assez élémentaires d'arithmétique :

– Si n est premier et ne divise pas a, alors $a^{n-1} = 1 \bmod n$; par conséquent, si $a^{n-1} \neq 1 \bmod n$, avec $1 < a < n$, alors n est nécessairement composé (mais il y a une infinité d'entiers composés, dits de Carmichael, qui échappent à ce test, quand a et n sont premiers entre eux, le plus petit étant 561)
– Si n est premier, alors les racines carrées de 1 modulo n sont 1 et -1 ; par conséquent, s'il existe une racine carrée de l'unité modulo n non triviale (c'est-à-dire, s'il existe r tel que $r^2 = 1 \bmod n$, $r \neq 1 \bmod n$ et $r \neq n - 1 \bmod n$), alors n est composé.

Le principe de l'algorithme de Miller-Rabin est de tirer aléatoirement a dans $[2, n-1]$ et de calculer $a^{n-1} \bmod n$ au moyen de l'algorithme d'exponentiation modulaire rapide. Comme il procède par carrés successifs, ce dernier algorithme donne l'opportunité de découvrir au passage une racine carrée non-triviale de 1. La fonction `témoin()` qui en dérive retourne `true` dès qu'une racine carrée non-triviale est obtenue ou si $a^{n-1} \neq 1 \bmod n$, ce qui permet de conclure que n est composé ; on dit alors que a est un *témoin* de Miller, c'est-à-dire une preuve, du fait que n est composé ;

elle retourne `false` sinon, ce qui est un indice de primalité, qui devra être confirmé ou infirmé par d'autres tirages aléatoires.

```
static boolean témoin(int a, int n) {
  int m = n-1;
  int y = 1;
  while (m != 0) {
    if (m%2 == 1) {
      y = (a*y) % n;
      m = m-1;
    } else {
      int b = a;

      a = (a*a) % n;
      if (a==1 && b!=1 && b!=n-1) {
        // b est une racine carre non triviale de 1
        return true;        // n est composé
      }
      m = m/2;
    }
  }
  if (y != 1) {
    return true;            // n est composé
  } else {
    return false;           // ?
  }
}
static boolean millerRabin(int n, int t) {
  for (int i=0; i<t; i++) {
    int a = irand(2, n-1);
    if (témoin(a,n)) {
      return true;          // n est composé
    }
  }
  return false;             // n est probablement premier
}
```

On notera que le « `return true;` » placé dans la boucle `while` de `témoin` et dans la boucle `for` de `millerRabin()` permet de s'en échapper (c'est-à-dire de ne pas exécuter les itérations suivantes) en retournant immédiatement une valeur.

La fonction `millerRabin()` est invoquée avec deux arguments, l'entier n à tester et le nombre t de tirages ; elle retourne `true` dès qu'un témoin est trouvé, auquel cas n est composé, et `false` sinon, auquel cas n est probablement premier.

Combien de tirages sont nécessaires ? On montre que si n est composé et impair, au moins $3/4$ des $n-2$ entiers a tels que $1 < a < n$ sont des témoins de Miller pour n ; donc quand a est tiré aléatoirement avec une probabilité uniforme, une invocation de `témoin(a,n)` a une probabilité $\geq 3/4$ de retourner 1, et une probabilité $< 1/4$ de retourner 0, c'est-à-dire de laisser croire que n est premier. Les tirages étant indépendants, la fonction `Miller_Rabin(n,t)` a une probabilité $< (\frac{1}{4})^t$ de retourner 0 alors que n est composé. Ainsi, la réponse « n est composé » est toujours exacte, et une réponse « n est probablement premier » est exacte avec une probabilité d'erreur $< (\frac{1}{4})^t$: l'obstination permet de réduire la probabilité d'erreur à un nombre aussi petit qu'on le souhaite. Quel que soit n, 50 tirages suffisent largement (la probabilité d'erreur est alors de l'ordre de 10^{-31}). Abelson[1] fait remarquer qu'on obtient ainsi une probabilité inférieure à la probabilité d'une erreur à l'exécution due à l'incidence d'une radiation cosmique sur la machine ; et il ajoute « juger de l'adéquation d'un algorithme en prenant en compte le premier type d'erreur mais pas le second illustre la différence entre un mathématicien et un ingénieur ».

L'algorithme de Miller-Rabin a une complexité en $O(t \log n)$ opérations. Aucun algorithme déterministe n'est en revanche capable de résoudre le problème de primalité en un temps raisonnable pour des entiers à plusieurs centaines de chiffres. Ce problème est pourtant important pour garantir la sécurité des systèmes de chiffrement, donc finalement la sécurité de nombre d'applications informatiques ainsi que la confidentialité des données. Le problème de factorisation des grands entiers est encore plus difficile et encore plus crucial pour la sécurité ; aucun algorithme, ni déterministe, ni de Monte-Carlo ne parvient à le résoudre.

2.31 Un algorithme de Las Vegas : l'élection d'un chef Quand deux personnes d'une égale politesse et sans la moindre imagination s'apprêtent à franchir une porte au même moment, on déplore une interminable suite d'« après vous, je vous en prie ». Quand le même phénomène se produit dans un réseau de machines, on constate un blocage irrémédiable. Par exemple, si deux machines d'un réseau local émettent simultanément un paquet de données sur le réseau et constatent une collision, il ne faut pas que ces deux machines réémettent à l'issue d'un *même* délai. La solution est de briser la symétrie en introduisant de l'aléatoire, en recourant à une catégorie d'algorithmes stochastiques, dits de *Las Vegas*.

Considérons le problème de l'élection d'un chef parmi n candidats ; ce problème généralise celui du passage de portes et trouve des applications très importantes aux protocoles dans les réseaux de communication qui ne comportent pas de contrôle centralisé. L'algorithme d'élection consiste à éliminer par tours successifs les candidats votants ; au début, les n candidats sont votants, et à la fin, le seul candidat qui reste votant est le chef élu. À chaque tour, chaque candidat votant tire un nombre au hasard compris entre 1 et le nombre de votants, et on compte le nombre t de 1 qui ont été tirés. Si $t = 0$, on recommence ; si $t > 1$, seuls les candidats qui ont tiré un 1 restent actifs pour le tour suivant ; si $t = 1$, on s'arrête, le chef étant celui qui a tiré ce 1. Voici une trace d'exécution de cet algorithme :

```
candidat 0   1   2   3   4   5   6   7   8   9
tire     1   2   3   9  10   3   8   5   1   1
         1                               1   2
         2                               2
         2                               2
         2                               1
```

La seule difficulté de programmation provient du cas $t = 0$ qui oblige à conserver en mémoire l'activité d'un candidat afin de recommencer le vote. La méthode `tour()` définit ce que fait chaque candidat. La méthode `chef()` définit le « protocole » de l'élection et retourne le chef élu :

```
class Election {
  private Random g;
  private int nbCandidats;
  private int nbVotants;
  Candidat[] candidats;
  private int nbActifs;

  Election(int n) {
    nbCandidats = n;
    candidats = new Candidat[n];
    for (int i=0; i<nbCandidats; i++) {
      candidats[i] = new Candidat();
    }
    nbVotants = n;
    g = new Random();
  }

  class Candidat {
    boolean votant;
    boolean éliminé;
    Candidat() {
      votant = true;
      éliminé = false;
    }
    void tour() {
      int choix = g.nextInt(1+nbVotants);
      if (choix != 1) {
        votant = false;  // devient inactif
      } else {
        nbActifs ++;
      }
    }
  }

  int chef() {
    int élu = 0;
    while (nbVotants != 1) {
      nbActifs = 0;
      for (int i=0; i<nbCandidats; i++) {
```

```
        if (candidats[i].votant) {
          candidats[i].tour();
        }
      }
      for (int i=0; i<nbCandidats; i++) {
        if (!candidats[i].éliminé) {
          if (nbActifs == 0) candidats[i].votant = true;
          else if (nbActifs>1 && !candidats[i].votant)
            candidats[i].éliminé = true;
          else if (nbActifs==1 && candidats[i].votant)
            élu = i;
        }
      }
      if (nbActifs != 0) nbVotants = nbActifs;
    }
    return élu;
  }

  public static void main(String[] args) {
    Election élection = new Élection(10);
    System.out.println(élection.chef());
  }
}
```

Cet algorithme ne termine pas nécessairement, mais quand il termine, le résultat est toujours correct, un chef est élu : ce comportement est différent des algorithmes de Monte-Carlo qui terminent toujours, mais dont le résultat n'est correct qu'avec une certaine probabilité d'erreur. Les probabilités interviennent ici seulement pour estimer le nombre de tours probable.

La probabilité pour que k candidats parmi n actifs ($0 \leq k \leq n$), tirent une valeur donnée entre 1 et n (chacune ayant la même probabilité $1/n$) est donnée par une loi binômiale : $p(n,k) = C_n^k(\frac{1}{n})^k(1-\frac{1}{n})^{n-k}$. On peut montrer que l'espérance du nombre de tours nécessaires pour choisir un chef parmi n candidats est asymptotiquement constante ; la probabilité que l'algorithme ne termine pas est nulle, autrement dit, la terminaison est presque sûre. L'obstination conduit presque sûrement à un résultat correct.

2.32 *Randomisation* Qu'un algorithme ait une complexité moyenne satisfaisante n'est pas une garantie absolue d'efficacité. D'abord, parce que les données ne vérifient pas nécessairement l'hypothèse d'uniformité à partir de laquelle cette complexité moyenne a été calculée. Ensuite, l'algorithme n'est pas toujours destiné à être utilisé sur un grand nombre de données, parmi lesquelles les cas extrêmes seraient statistiquement minoritaires : un utilisateur peut systématiquement employer l'algorithme sur des mauvais cas. C'est la situation d'un jeu qui oppose le programmeur et un utilisateur malicieux de l'algorithme : le programmeur cherche à minimiser le temps d'exécution et l'utilisateur à le maximiser (pour prouver au programmeur qu'il est incompétent). Les techniques de *randomisation* assurent que l'utilisateur malicieux ne peut pas gagner : comme les conditions d'uniformité sont effectivement réalisées (et pas seulement supposées), les cas extrêmes sont garantis rares, inconditionnellement. Les algorithmes randomisés sont stochastiques.

Une première technique consiste à brasser le tableau au début de l'algorithme en lui appliquant une permutation aléatoire (les $n!$ permutations doivent être équiprobables). Cette technique est évidemment correcte pour un problème de tri, mais s'applique difficilement à d'autres types de problèmes.

L'autre technique a un champ d'applications plus large. Il arrive souvent qu'à une certaine étape d'un algorithme, un choix soit à faire entre plusieurs opérations (dans le tri rapide, il s'agit du choix du pivot) a priori équivalentes. Quand ce choix a été figé dans un algorithme déterministe, il se trouve que c'est un bon choix pour certaines données, et que c'est un mauvais choix pour d'autres. Un algorithme randomisé fera ce choix au hasard. Voici comment randomiser la fonction `partition()` du tri rapide :

```
int randomPartition(int m, int n) {
  int p = irand(m, n);
  échangerÉléments(m, p);
  return partition(m,n);
}
```

Il suffit de remplacer dans `triRapide()` l'invocation de `partition()` par une invocation de `randomPartition`.

Une autre application est le *hachage universel* : au lieu de choisir à l'avance une fonction de hachage qui risque de se révéler mauvaise (en termes de collisions) pour certaines applications, on tire au hasard, pour chaque application, une fonction dans une famille de fonctions de hachage. On dit qu'une famille $\mathcal{H}$ de fonctions d'un univers U de clés vers l'intervalle $[0, m-1]$ est *universelle* si pour toutes les clés x et $y \neq x$, le nombre de fonctions $f \in \mathcal{H}$ telles que $f(x) = f(y)$ est $|\mathcal{H}|/m$. Il est facile, avec très peu d'arithmétique, de vérifier que la famille suivante est universelle. Soit p tel que toute clé x puisse être décomposée en $p + 1$ entiers $x_0, \ldots, x_p$ dans l'intervalle $[0, m - 1]$; la famille $\mathcal{H}$ est composée des fonctions $f_\alpha(x) = \sum_{i=0}^{p} \alpha_i x_i \bmod m$, où $\alpha = (\alpha_0, \ldots, \alpha_p) \in [0, m - 1]^{p+1}$. L'algorithme de hachage universel consiste à tirer α aléatoirement et à utiliser f_α comme fonction de hachage (il est bien sûr indispensable de stocker α avec la table pour les opérations ultérieures de recherche).

Les algorithmes randomisés sont appelés algorithmes de Sherwood par G. BRASSARD et P. BRATLEY, en hommage à Robin des Bois, qui pratiquait une redistribution équitable des richesses.

3

Patterns

Une des exigences de l'ingénierie du logiciel est la production de *composants* assemblables et utilisables dans plusieurs réalisations, comme dans toute autre activité industrielle. La mise en œuvre de cette exigence dans un langage de programmation est permise, d'une part par la notion de *module*, d'autre part par des règles d'écriture et d'usage de ces modules.

Plusieurs langages, notamment Modula 2 et Ada, ont été conçus autour de la notion de module. L'idée est qu'un programme est formé à partir de plusieurs modules ; les entités définies dans chaque module sont classées en deux catégories : *publiques* ou *privées*. Les entités publiques d'un module sont déclarées dans son *interface* et sont *exportables* vers les autres modules, lesquels déclarent dans leurs interfaces les entités qu'ils *importent*. Les modules exportant des entités offrent des *services* dont sont *clients* les modules qui les importent. En outre, la distinction public/privé permet une discipline de noms : les noms privés, n'étant pas connus à l'extérieur de leur module, peuvent être réutilisés sans risque par d'autres modules. Le langage disposant actuellement de la meilleure notion de module semble être Ocaml. Java met en œuvre la modularité à travers des mécanismes d'accessibilité et d'abstraction portant sur ses paquets et ses types.

Loin d'être simplement un nouveau langage, Java est au cœur d'une nouvelle technologie, qui se déploie à la fois de façon matérielle (futures cartes à puce, systèmes embarqués) et de façon logicielle sous forme d'API spécialisée (*Application Programming Interface*), par exemple pour l'accès aux bases de données, la programmation réseau ou pour le graphique. L'utilisation des APIs fait que de nombreuses applications traditionnellement difficiles à programmer deviennent très simples. Cependant, leur utilisation demande une certaine compréhension de leur organisation, ce qui est l'un des objectifs de cette partie. Cette organisation résulte d'une part de traits du langage Java (les `packages`, les `interfaces`, les règles d'accessibilité, les hiérarchies de types) et d'autre part, de l'application systématique de certaines méthodes de conception, qui identifient des situations typiques et des façons de les aborder et de les résoudre. Le besoin de recourir à de telles méthodes de conception n'est pas propre à l'informatique. La notion de *pattern* a été développée à la fin des années 1970 par un architecte, Christopher Alexander, qui les définit ainsi :

> «A pattern is a careful description of a perennial solution to a recurring problem within a building context, describing one of the configurations which brings life to a building.
>
> « A pattern language is a network of patterns that call upon one another. An individual house might, for example, call upon the patterns described under the names of half-hidden garden, light from two sides in every room, variation of ceiling height, bed alcove, etc. Patterns help us remember insights and knowledge about design and can be used in combination to create designs.»

Cette notion s'est vue réappropriée par des informaticiens, une dizaine d'années plus tard, et constitue actuellement l'une des approches les plus intéressantes de l'architecture des systèmes logiciels. Tant ces traits du langage que les patterns sont utiles à la programmation d'applications, pour peu que l'on s'efforce de les programmer proprement en respectant certains critères : indépendance de l'interface et de l'implémentation, facilité de modification ou de réutilisation, etc. Les choix que l'on fait quand on écrit un programme peuvent souvent se lire comme le choix d'un pattern contre un autre. Cette partie présentera quelques uns de ces patterns : fabrication, itération, décoration, visitation (?), etc.

3.1 Interfaces Une programmation modulaire peut être obtenue d'une part au moyen des règles d'accessibilité concernant les paquets, les classes et leurs membres, et d'autre part grâce au système de types.

Rappelons d'abord que les classes de Java ont trois rôles :

1. de typage, au sens usuel, ce qui permet de déclarer des noms d'un certain type et de vérifier que les expressions où ils apparaissent sont correctement typées ;
2. d'implémentation, pour définir le comportement des objets, en implémentant les méthodes ;
3. de moule, pour construire leurs instances (seulement si la classe n'est pas abstraite).

On peut regretter que Java confonde ces rôles en une seule construction syntaxique et laisse au programmeur le soin de les utiliser à bon escient. Il est possible, par un usage judicieux des classes abstraites (qui remplissent le premier, et éventuellement, le second de ces trois rôles), de programmer de façon assez modulaire. Aucune instance d'une classe abstraite ne peut être construite, mais on peut déclarer une variable ou des paramètres de méthode d'un tel type.

Java offre cependant à côté des classes une autre catégorie de types, les *interfaces* qui, ne remplissant que le premier de ces trois rôles, permet une meilleure modularité. Une interface est un type purement abstrait au sens où il ne définit aucune implémentation et ne comporte pas de constructeur : une interface déclare uniquement des méthodes publiques (comme dans les classes abstraites, le corps de la méthode est remplacé par un « `;` »). Les interfaces portent souvent des noms se terminant en `able`. Par exemple, l'interface `Comparable` du paquet `java.lang` déclare une méthode de comparaison des objets :

```
interface Comparable {
  int compareTo(Object o);
}
```

La documentation indique que la méthode `compareTo()`, appliquée à un objet x, avec pour argument un objet y, compare les objets x et y et retourne un entier < 0, ou 0 ou un entier > 0, selon que x est plus petit que y, lui est égal ou lui est supérieur. La documentation ne dit pas comment cette comparaison s'effectue, ce qui n'aurait aucun sens pour des objets quelconques : une telle méthode de comparaison n'est d'ailleurs définie que pour certaines classes, dont on dit qu'elles *implémentent* l'interface `Comparable`. Par exemple, les classes `Character`, `Double`, `String`, `Integer` implémentent cette interface, mais la classe `Object` ne l'implémente pas.

Une classe *implémente* une interface si elle contient une implémentation publique pour chacune des méthodes de l'interface. Une classe dont la définition spécifie `implements`, suivi de noms d'interfaces, doit implémenter ces interfaces ; elle est alors considérée comme un sous-type de ces interfaces.

Toute interface est un sous-type d'`Object`. Une classe qui implémente une interface est un sous-type de cette interface. On peut donc affecter à une variable de type l'interface une expression d'une classe l'implémentant, par exemple :

```
Comparable c = new Integer(3);
```

On peut aussi passer à une méthode un argument de type `Integer` si le paramètre correspondant est déclaré de type `Comparable`. Par exemple, si nous voulons définir une fonction `min()` qui calcule le minimum de deux objets, nous devons supposer que son premier argument est *comparable* à son second argument : il suffit de déclarer le premier argument de type `Comparable` :

```
static Object min(Comparable x, Object y) {
  return x.compareTo(y) <=0 ? x : y;
}
```

On pourra alors invoquer cette fonction, par exemple, sur des instances d'`Integer`[1] :

```
Object m = min(new Integer(3), new Integer(2));
```

ou, si l'on veut obtenir un `Integer`, à l'aide d'un transtypage :

```
Integer m = (Integer) min(new Integer(3), new Integer(2));
```

Extension d'une interface Comme pour les classes, il y a une notion de sous-interface et d'héritage. Comme une interface déclare des méthodes, étendre une interface permet de déclarer des méthodes supplémentaires. Ceci se fait également au moyen du mot-clé `extends` :

```
interface I1 { ... }
interface I2 extends I1 { ... }
```

On dit alors que `I2` est une sous-interface directe de `I1`, ou qu'elle dérive de `I1`, et que `I1` est une sur-interface directe de `I2`. L'effet de cette extension est que les déclarations de méthodes de `I1` sont héritées par `I2`[2]. Une sous-interface d'une interface est considérée comme un sous-type de cette interface, ce qui permet d'utiliser une expression disposant de méthodes supplémentaires, là où seulement certaines de ces méthodes sont requises ; les affectations suivantes sont donc correctes :

```
I2 y = ...;
I1 x = y;
```

Dans ce cas, seules les méthodes déclarées par `I1` pourront être invoquées sur `x`, alors que l'objet désigné par `x` implémente probablement d'autres méthodes. On dispose ainsi d'une notion d'abstraction graduelle.

1. Bien que la déclaration de `min()` n'impose pas de restriction sur le type du second argument, l'implémentation de `compareTo()` dans la classe `Integer` impose que cet argument soit aussi de type `Integer`, ce qui est raisonnable, mais ne permet pas, par exemple, de comparer un `Integer` et un `Double`.

2. Il n'y a cependant pas d'héritage d'une méthode qui est redéclarée dans la sous-interface avec une clause d'exceptions comportant strictement moins d'exceptions que la déclaration de la sur-interface.

3.2 *Une discipline d'abstraction* L'usage conjoint des classes et des interfaces permet de découpler l'implémentation de la déclaration : aux interfaces la déclaration des méthodes publiques (de leur nom, des types de leurs paramètres, de leur type de retour), aux classes l'implémentation des méthodes (leur corps, c'est-à-dire ce qu'elles font) et la construction des objets. Pour assurer ce découplage, on observera autant que possible les deux règles suivantes, qui constituent une *discipline d'abstraction* pour l'utilisateur, c'est-à-dire du côté du client :

– les champs, les paramètres des méthodes, les variables locales sont toujours déclarées avec comme type une interface ;
– les classes d'implémentation ne sont utilisées que par l'intermédiaire de leurs constructeurs, dans des expressions de création d'objets.

L'idée est de choisir séparément les fonctionnalités souhaitées, ce qui détermine l'interface (par exemple, a-t-on besoin d'une méthode de comparaison ?), et la représentation des données et l'implémentation des méthodes, ce qui détermine les classes d'implémentation.

Ce découplage est particulièrement utile quand plusieurs implémentations existent ; nous en verrons des exemples à propos de la famille des collections. Voici l'exemple des piles, dont le type abstrait est formé des déclarations suivantes :

```
interface Pile {
  boolean estVide();
  void empiler(Object o);
  Object sommet();
  Object dépiler();
}
```

Notons qu'il s'agit d'un type *générique* au sens où n'importe quel objet (c'est-à-dire instance de la classe `Object`) peut être empilé. Un programme utilisant des piles doit connaître ce type abstrait, et peut ignorer la nature de l'implémentation des piles (par un tableau, une liste chaînée, etc.). Il *doit* même ignorer cette implémentation, afin d'être indépendant de l'implémentation choisie, laquelle doit être modifiable sans remettre en cause les modules qui l'utilisent. Il ne faut donc pas qu'un programme utilisant des piles comporte des expressions du style `p.tableau[p.hauteur - 1]`, qui n'ont de sens que pour une implémentation particulière des piles. Il suffit au programme de connaître le nom d'une classe d'implémentation. Cela ne signifie pas que le choix d'une classe d'implémentation est arbitraire ; des considérations d'efficacité, mais aussi de disponibilité des classes d'implémentation guident généralement ce choix. Cependant, ce découplage met en œuvre la liaison tardive, qui est moins efficace qu'une liaison déterminée à la compilation.

Voici par exemple une méthode `toString()` que l'on pourrait ajouter à la classe `ArbreBinaire`, afin d'imprimer les étiquettes d'un arbre parcouru en profondeur d'abord ; on utilise la première implémentation, pour laquelle l'arbre vide est représenté par la valeur `null` :

```
class ArbreBinaire {
  int étiquette;
  ArbreBinaire gauche;
  ArbreBinaire droit;
  // ...
  public String toString() {
    StringBuffer tampon = new StringBuffer();
    Pile p = new PileParListe();
    p.empiler(this);
    while (!p.estVide()) {
      ArbreBinaire t = (ArbreBinaire) p.depiler();
      if (t != null) {
        tampon.append(t.étiquette).append(" ");
        p.empiler(t.droit);
        p.empiler(t.gauche);
      }
    }
    return tampon.toString();
  }
}
```

Ici, `PileParListe` est une implémentation de l'interface `Pile`. Il est inutile d'en connaître le contenu pour définir la méthode `imprimer()`. Si l'on préférait utiliser une autre implémentation de `Pile`, soit `PileParTableau`, il suffirait de remplacer l'appel au constructeur `PileParListe()` par un appel au constructeur `PileParTableau()`. Du fait de la généricité de `Pile`, la valeur retournée par `dépiler()` est de type `Object` ; avant de l'affecter à `t`, on lui applique un transtypage vers le type `ArbreBinaire`, de façon à pouvoir accéder aux membres de la classe `ArbreBinaire` à travers `t`.

3.3 Paquets et accessibilité Les programmes Java sont des familles de types, dont certains sont *publics*, c'est-à-dire destinés à être accessibles à d'autres utilisateurs que leur auteur. Ces familles de types sont organisées en *paquet* (ou *package*), dans le but de garantir une désignation non-ambiguë de chaque type, si possible dans le monde entier. La spécification de Java propose d'utiliser le système des noms de domaine Internet pour assurer cette unicité. Par exemple, tous les types conçus à l'ENPC devraient figurer dans un paquet `FR.enpc` (en renversant le nom de domaine `enpc.fr`, et en écrivant le nom de premier niveau en majuscules). Ensuite, de façon interne, l'École pourrait décider de désigner le type public `List`, dont l'auteur est l'élève `toto`, par le nom `FR.enpc.eleves.toto.List`, ce qui le distingue du type `java.util.List` de l'API de Java.

Les *membres* d'un paquet sont des types qui sont définis dans des unités de compilation, et des sous-paquets. Par exemple, le paquet `java` est composé des sous-paquets `applet`, `awt`, `beans`, `io`, `lang`, `math`, `net`, `rmi`, `security`, `sql`, `text` et `util` ; chacun de ses sous-paquets est désigné par son nom complet : `java.applet`, `java.awt`, etc.

Une *unité de compilation* se compose de définitions de types (classes ou interfaces). Par exemple, le paquet `java.util` contient les définitions des types `Collection`, `Set`, `HashSet`, etc. L'usage est de désigner les paquets par des noms commençant par une minuscule, les classes par des noms commençant par une majuscule, et les membres d'une classe à nouveau par des noms commençant par une minuscule. Ainsi, le nom complet `java.lang.System.out` désigne le membre `out` de la classe `System` du paquet `java.lang`.

Les paquets permettent de réaliser une forme de modularité basée sur la visibilité des noms, donc l'accessibilité des entités qu'ils désignent. Les noms dont la visibilité est contrôlable sont ceux des types (classes et interfaces) et ceux des membres et constructeurs de ces types. Les types d'un paquet sont déclarés `public` ou bien n'ont pas d'indication de visibilité. Les types publics sont accessibles de tout paquet (à condition que le paquet contenant ces types publics soit lui-même accessible) ; les types non publics ne sont accessibles que du paquet où ils sont définis. Seuls les types publics sont documentés. La règle (d'usage) est qu'une unité de compilation contient au plus un type public et qu'elle est placée dans un fichier dont le nom est celui du type public, suffixé par `.java`. L'unité de compilation suivante doit donc être placée dans un fichier de nom `A.java` :

```
package p;

public class A { ... }       // accessible partout
class A1 { ... }             // accessible seulement dans p
class A2 { ... }             // accessible seulement dans p
```

Les membres d'un type ou les constructeurs d'une classe sont déclarés `public`, `protected` ou `private` ou n'ont pas d'indication de visibilité.

```
package p;

public class C {
  public int a;              // accessible partout
  protected int b;           // accessible dans p et
                             // les sous-classes de C
  int c;                     // accessible dans p
  private int d;             // accessible dans C
}
```

Les membres ou constructeurs *privés* d'une classe ne sont accessibles qu'à l'intérieur de la classe où ils sont définis ; en particulier, ils ne peuvent pas être hérités par les types dérivés. Une méthode privée ne peut pas être redéfinie dans une sous-classe ; il est bien sûr possible de définir dans une sous-classe une méthode de même profil, mais elle n'aura aucune relation avec la méthode correspondante de la sur-classe, et le mécanisme de liaison tardive ne s'appliquera pas.

Les membres ou constructeurs *sans indication de visibilité* (ni publics, ni privés, ni protégés) d'une classe sont accessibles de toute classe du même paquet ; en particulier, ils ne sont hérités que par les classes dérivées appartenant au même paquet. Une méthode sans indication de visibilité peut être redéfinie dans une sous-classe par une méthode qui ne doit pas être privée.

Les membres ou constructeurs *protégés* d'une classe sont accessibles à partir d'une classe dérivée, à travers une référence qui est un sous-type de cette classe dérivée, ainsi que de toute classe du même paquet ; en particulier, ils peuvent être hérités par les classes dérivées. Une méthode protégée ne peut être redéfinie, dans une sous-classe, que par une méthode publique ou protégée. On notera qu'un membre protégé est davantage visible (ou moins protégé ?) qu'un membre sans indication de visibilité.

Les membres ou constructeurs *publics* d'un type public sont accessibles de tout paquet ; en particulier, ils peuvent être hérités par les types dérivés. Une méthode publique ne peut être redéfinie, dans une classe dérivée, que par une méthode publique. Les membres d'une interface sont implicitement déclarés publics.

Les membres et constructeurs publics et protégés doivent être documentés. Java prévoit des commentaires d'une forme particulière (entre `/**` et `*/`) qui permettent la génération automatique d'une documentation HTML avec la commande `javadoc` du Java Development Kit. Voici l'exemple de la documentation d'une méthode d'une interface ; le commentaire précède la déclaration de la méthode, comporte des mots-clés spécifiques (`@param`, `@returns`) et des balises HTML (`<tt>...</tt>`) :

```
/**
 * Returns <tt>true</tt> if this collection contains
 * the specified element.  More formally, returns
 * <tt>true</tt> if and only if this collection contains
 * at least one element <tt>e</tt> such that
 * <tt>(o==null ? e==null : o.equals(e))</tt>.
 *
 * @param o element whose presence in this collection
 * is to be tested.
 * @return <tt>true</tt> if this collection contains the
 * specified element
 */
boolean contains(Object o);
```

3.4 Patterns d'accès et discipline d'encapsulation L'*encapsulation* est une technique de génie logiciel qui permet de cacher les détails d'implémentation d'un objet pour ses utilisateurs. Java donne plusieurs moyens pour assurer cette encapsulation, et il appartient au programmeur de les utiliser. Afin d'empêcher l'utilisateur d'accéder directement aux champs de l'objet, on peut déclarer ses champs `private` et offrir éventuellement des méthodes publiques de lecture ou d'écriture de ces champs, formés selon l'usage par `get` ou `set` (en français, `val` ou `chg`) suivi du nom du champ capitalisé, selon le *pattern d'accès* suivant :

```
class A {
 private int champ;

 public int valChamp() {
  return champ;
 }

 public void chgChamp(int champ) {
  this.champ = champ;
 }
}
```

Le fait de ne définir que `val...` (resp. `chg...`) interdit l'accès en écriture (resp. lecture).

L'usage des méthodes d'accès permet également de contrôler la valeur des champs dans les cas où elle doit respecter certaines contraintes. Par exemple, si l'on sait qu'un champ numérique doit être compris entre deux valeurs `champMIN` et `champMAX`, la méthode `chgChamp()` s'assurera du respect de cette contrainte :

```
  private int champMIN, champMAX;

  public void chgChamp(int champ) {
    if (champ<champMIN)
      this.champ = champMIN;
    else if (champ>champMAX)
      this.champ = champMAX;
    else
      this.champ = champ;
  }
```

Comme autre exemple, il y a des situations où le nombre de fois qu'un champ est modifié doit être limité (par exemple, la zone d'un lecteur de DVD peut être changée au plus 5 fois) : un champ supplémentaire, évidemment privé, et initialisé implicitement à 0, permet de contrôler le nombre de modifications.

```
  private static final int N = 5;
```

```
  private int champModifié;

  public void chgChamp(int champ) {
    if (champModifié<N)
      this.champ = champ;
      champModifié++;
  }
```

Tous ces exemples constituent des « patterns », c'est-à-dire des solutions typiques à des problèmes qui se rencontrent fréquemment, et qui peuvent être adaptées à chaque problème. La *discipline d'encapsulation* consiste à rendre systématiquement les champs privés, et à fournir, selon les besoins, l'une ou l'autre des méthodes d'accès `chgXxx()` et `valXxx()` à chaque champ `xxx`, selon le pattern souhaité.

Une méthode peut aussi être déclarée privée en faisant précéder sa déclaration du mot-clé `private`. Une méthode privée peut être invoquée par une méthode de la classe où elle est définie, mais pas par une méthode d'une autre classe. Une méthode privée n'est pas héritée et ne peut pas être redéfinie.

3.5 Un pattern de création : les classes singletons Une classe singleton est une classe qui ne peut avoir qu'une seule instance. La réalisation d'une telle classe met en œuvre :

- un champ `instance` privé statique du type de la classe désignant l'instance ;
- une méthode publique statique de création, qui teste si l'instance n'a pas encore été créée, et si c'est le cas, qui appelle un constructeur ;
- un constructeur, qui doit être privé pour ne pas être invoqué librement de l'extérieur de la class ;
- les autres champs (`champ`, etc) sont privés par sécurité.

```
class Singleton {
  private Object champ;
  // ...
  private static Singleton instance;
  private Singleton(Object champ) {
    this.champ = champ;
    // ...
  }
  Object valChamp() { return champ; }
  // ...
  static Singleton uniqueInstance(Object champ) {
   if (instance == null) {
      instance = new Singleton(champ);
   }
   return instance;
```

```
  }
}
```

Cette classe sera utilisée ainsi :

```
    Singleton s1 = Singleton.uniqueInstance(new Double(2.3));
```

Remarquons que si la méthode `uniqueInstance()` n'est pas appelée, l'instance n'est pas créée : dans une application donnée, cette classe a au plus une instance. La classe `AVide` dont l'unique instance est l'arbre binaire vide est programmée de façon légèrement différente, l'instance étant créée par initialisation du champ statique, même si la méthode d'accès à l'instance n'est pas invoquée.

3.6 Unités de compilation En ingénierie du logiciel, on cherche non seulement à produire des logiciels modulaires, mais aussi à réaliser leur production d'une façon modulaire. Typiquement, on ne doit pas être obligé de recompiler tout le programme quand une modification locale du source a été réalisée. La solution, qui existe depuis Fortran, est la *compilation séparée*.

Le Java Development Kit de Sun fait du fichier l'unité de compilation (on ne peut pas compiler une partie d'un fichier). Les définitions de types peuvent être réparties entre plusieurs fichiers, et chacun peut être compilé séparément. La spécification du langage n'impose pas comment les paquets et les unités de compilation sont stockés : dans une base de données, dans un système de fichiers, local ou distribué. Elle n'indique pas non plus quels sont les paquets accessibles.

L'implémentation couramment utilisée, sous Unix, représente chaque paquet comme un répertoire, les sous-paquets comme des sous-répertoires, et les unités de compilation comme des fichiers de nom suffixé en `.java` ; chaque définition de type figurant dans une unité de compilation donne lieu à un *fichier de classe* dont le nom est suffixé en `.class` qui contient sa définition compilée. Par exemple, la classe `projet.util.Liste` a un fichier de classe `Liste.class` qui se trouve dans un sous-répertoire `util` d'un répertoire `projet`.

Les *types accessibles* sont ceux situés sous l'un des répertoires spécifiés par la variable d'environnement `CLASSPATH`, ainsi que ceux contenant les types standards de l'environnement Java ; ces répertoires et leurs fichiers sont explorés dans l'ordre où ils figurent dans cette variable, pourvu qu'ils soient autorisés en lecture. Par exemple, supposons que l'on ait défini, sous Linux :

```
linux% setenv CLASSPATH .:$(HOME)/java/classes
```

Rappelons que « `.` » désigne le répertoire courant, vraisemblablement celui où le développement du programme s'effectue, et `$(HOME)` le répertoire personnel. Par exemple, le type `projet.util.Liste` sera alors recherché successivement dans

- `./projet/util/Liste.class`,
- `$(HOME)/java/classes/projet/util/Liste.class`,
- et finalement dans les types standards (où il ne devrait pas se trouver).

Pour faire exécuter une classe principale figurant dans un paquet, on doit utiliser le nom complet de la classe. Par exemple, pour charger dans la Machine Virtuelle Java la classe `projet.Main`, c'est-à-dire la classe `Main` du paquet `projet`, dont le fichier de classe est `projet/Main.class`, on exécute la commande suivante, sous Linux :

```
linux% java projet.Main
```

Une *unité de compilation* se compose de trois parties :

- la déclaration du paquet, avec son nom complet ;
- des déclarations d'importation de types;
- des définitions de type (classes et interfaces).

Une *déclaration de paquet* a la forme :

```
package projet.util;
```

Elle indique que l'unité de compilation appartient au paquet `projet.util`. Si l'unité ne commence pas par une déclaration de paquet, elle est considérée comme faisant partie d'un *paquet anonyme*. L'usage des paquets anonymes, commode pour développer de petites applications, est contraire aux ambitions du langage en matière de génie logiciel. C'est pourquoi la spécification de Java ne précise pas comment ces paquets anonymes doivent être traités. Sur les implémentations courantes, sous Linux, les unités de compilation sans déclaration de paquet d'un même répertoire constituent un même paquet anonyme. Il est recommandé que les types d'un paquet anonyme ne soient pas déclarés public, afin qu'ils ne puissent pas être importés par un autre paquet, même accidentellement.

Les *déclarations d'importation* permettent à un type public d'un autre paquet d'être désigné par son nom simple au lieu de son nom qualifié complet : cette déclaration doit spécifier le nom complet du paquet qui contient ce type. Il est possible d'importer un seul type :

```
import java.awt.Graphics;
```

ou tous les types publics d'un paquet :

```
import java.awt.*;
```

Cette forme étoilée n'importe pas les sous-paquets d'un paquet. Il est donc nécessaire de déclarer à la fois :

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

Toute unité de compilation importe implicitement le paquet `java.lang` : tous ses types (`Integer`, `Exception`, `Cloneable`, etc.) peuvent donc être désignés par leur nom simple dans tout programme.

3.7 Compatibilité binaire Une application Java consiste en un ensemble de fichiers de classe, qui peuvent être d'origines diverses : produits par le programmeur de l'application, membres d'une API standard et documentée, ou obtenus par d'autres voies. Quand les programmes sources sont disponibles, la construction de l'application se fait en compilant l'ensemble des sources. Cette compilation doit respecter la relation de *dépendance* entre unités : on dit qu'une unité A dépend d'une unité B si A utilise un type défini par B. Dans ce cas, le compilateur, après avoir lu l'unité A`.java`, doit chercher des informations sur B qui se trouvent dans le fichier de classe B`.class`. Si ce fichier de classe n'existe pas, mais que l'unité B`.java` existe, le compilateur doit procéder à sa compilation, pour produire B`.class`. Le compilateur est capable de suivre ces relations de dépendance pour chercher les fichiers de classe nécessaires et éventuellement pour les produire, si les unités de compilation correspondantes sont disponibles. Dans le cas où plusieurs unités forment un circuit pour la relation de dépendance (c'est-à-dire, sont mutuellement dépendantes), ces unités, après avoir été lues successivement, sont compilées ensemble.

Si l'une des unités de compilation est modifiée, il faut seulement recompiler toutes les unités qui dépendent de celle-ci. Si le compilateur dispose à la fois du fichier de classe C`.class` et de l'unité de compilation C`.java` dont il provient, il ne recompile C`.java` que si cette unité est plus récente que C`.class`.

Cette recompilation n'est pas toujours possible quand le programmeur ne dispose pas de tous les programmes sources. C'est toujours le cas dès qu'il ou elle utilise une API, ou des fichiers de classe distribués sur l'Internet. L'auteur d'une API ne peut pas exiger que tous les utilisateurs d'une API recompilent leurs application à chaque fois qu'une nouvelle version de l'API est distribuée. En fait, le même problème se pose aussi quand des applications de très grande taille sont développées, car il n'est pas réaliste de recompiler une partie, voire l'ensemble de l'application dès qu'une modification mineure est faite à un fichier. Il faut donc se résoudre à travailler avec un ensemble de fichiers de classe qui n'est pas nécessairement cohérent. C'est pourquoi la spécification du langage Java décrit un certain nombre de modifications qui *préservent la compatibilité binaire*. Cette propriété garantit que la liaison du fichier de classe modifié se fera sans erreur si c'était le cas avant la modification. Elle ne garantit pas que le comportement ou le résultat de l'application seront identiques. Ces modifications, considérées comme sûres, sont les transformations suivantes :

- réimplémenter des méthodes ou des constructeurs existants pour améliorer leurs performances ;
- modifier des méthodes ou des constructeurs afin de les faire retourner au lieu de déclencher une exception inutilement, de boucler indéfiniment ou de se bloquer ;
- ajouter des champs, des méthodes ou des constructeurs dans un type ;
- supprimer des champs, méthodes ou constructeurs qui sont privés dans un type ;
- supprimer des champs, méthodes ou constructeurs qui sont accessibles seulement au paquet qui les contient, si l'ensemble du paquet est mis à jour ;
- réordonner des champs, des méthodes ou des constructeurs dans un type ;
- déplacer une méthode d'une classe vers une sur-classe ;
- réordonner la liste des sur-interfaces d'une classe ou d'une interface ;
- insérer un type dans la hiérarchie des types.

L'idée générale de ces transformations est que l'on peut toujours modifier un type pour en faire un « sous-type » (en ajoutant des champs, par exemple), mais que si l'on en fait un « sur-type » (en supprimant des champs, par exemple), il faut alors mettre à jour les types pour lesquelles ces modifications sont visibles : ainsi, il faut mettre à jour tous les types du même paquet si on supprime un champ sans indication de visibilité, mais il n'y a pas de mise à jour nécessaire si on supprime un champ privé.

3.8 Les collections La conception d'une bonne architecture modulaire est loin d'être évidente. La notion de structure de données permet souvent de construire des modules de base d'une telle architecture. Quelques unes des structures de données les plus utiles sont rassemblées dans un ensemble cohérent de types figurant dans le paquet `java.util`, autour du concept de collection. Une *collection* représente un groupe d'objets, qui sont ses *éléments*. De façon précise, les éléments d'une collection ne sont pas des données, mais des références à des objets (instances de classe ou tableaux).

Les principales opérations sur une collection sont :

- ajouter un objet (`add(Object o)`);

- tester l'appartenance d'un objet (`contains(Object o)`);
- retirer un élément (`remove(Object o)`) ou tous les éléments (`clear()`);
- retirer tous les éléments (`clear()`);
- obtenir le nombre d'éléments (`size()`);
- tester si la collection ne contient aucun élément (`isEmpty()`).

Comme les éléments d'une collection ne peuvent pas être des valeurs d'un type primitif (`char`, `int`, `double`, etc.), il faut recourir à un constructeur d'une classe enveloppante (`Character`, `Integer`, `Double`, etc.) pour en faire des instances de classe : par exemple, si `c` est une collection, on ne pourra pas écrire `c.add(2)`, mais `c.add(new Integer(2))`.

Certaines collections acceptent des éléments dupliqués, certaines sont ordonnées. Le paquet `java.util` se compose d'interfaces et de classes ; l'interface `Collection`, qui spécifie toutes les collections, a plusieurs sous-interfaces spécialisées :

- `List` spécifie les *suites*, c'est-à-dire les collections ordonnées ; la méthode `int indexOf(Object o)` retourne l'indice de la première occurrence de `o` dans la suite ; la méthode `List subList(int initial, int final)` retourne une référence vers la sous-liste formée des éléments d'indices $\geq$ `initial` et $<$ `final` ; la méthode `Object set(int i, Object o)` remplace l'élément d'indice `i` par l'objet `o` ; la méthode `Object get(int index)` retourne l'élément d'indice `i` ;
- `Set` spécifie les *ensembles*, c'est-à-dire les collections dont les éléments ne sont pas dupliqués ;
  - `SortedSet`, sous-interface de `Set` spécifie les ensembles ordonnés (par ordre croissant) par une relation d'ordre sur ses éléments ; la méthode `SortedSet subSet(Object initial, Object final)` retourne une référence vers le sous-ensemble formé des éléments $\geq$ `initial` et $<$ `final` ; `first()` et `last()` retournent respectivement le plus petit et le plus grand élément de l'ensemble ordonné.

Les collections disposent d'opérations ensemblistes extrêmement utiles permettant de tester l'appartenance de, d'ajouter ou de retirer tous les éléments d'une collection à une autre collection. Ce sont les méthodes :

- `boolean containsAll(Collection c)`, pour l'inclusion ;
- `boolean addAll(Collection c)`, pour la réunion ;
- `boolean removeAll(Collection c)`, pour la différence ;
- `boolean retainAll(Collection c)`, pour l'intersection.

Les trois dernières retournent `true` si l'opération a modifié la collection.

Les collections sont naturellement des structures de données hétérogènes, c'est-à dire que leurs éléments ne sont pas nécessairement du même type. On peut ainsi ajouter les éléments suivants à une liste `l` :

```
l.add(new Integer(2));
l.add("Java");
l.add(java.awt.Color.red);
```

D'autre part, les collections comprennent une autre interface, `Map` (qui n'est pas une sous-interface de `Collection`):

- `Map` spécifie les *tables*, c'est-à-dire les collections associant une valeur à une clé, les clés ne pouvant être dupliquées, au plus une valeur étant associée à chaque clé ; la méthode `put(Object clé, Object valeur)` insère une association `clé` $\rightarrow$ `valeur` dans la table et retourne l'objet précédemment associé à cette clé ou bien `null`; `get(Object clé)` retourne l'objet associé à `clé` par cette table, ou bien `null`;
  - `SortedMap`, sous-interface de `Map` spécifie les tables dont l'ensemble des clés est ordonné ; la méthode `subMap(Object initial, Object final)` retourne une référence vers la sous-table formé des associations de clé $\geq$ `initial` et $<$ `final`; `firstKey()` et `lastKey()` retournent respectivement la plus petite et le plus grande clé de la table ordonnée.

3.9 Implémentations d'une collection Les interfaces spécialisées spécifiant les collections et les tables disposent d'une ou de plusieurs implémentations :

- les classes `ArrayList` (recommandée) et `LinkedList` implémentent l'interface `List`;
- la classe `HashSet` (recommandée) implémente l'interface `Set`;
- la classe `TreeSet` implémente l'interface `SortedSet`;
- la classe `HashMap` (recommandée) implémente l'interface `Map`;
- la classe `TreeMap` implémente l'interface `SortedMap`.

Ces deux niveaux permettent de découpler l'implémentation de l'interface. Il faut d'abord choisir les fonctionnalités souhaitées, ce qui détermine l'interface (par exemple, souhaite-t-on utiliser une relation d'ordre sur les éléments ?), puis choisir la représentation de la structure de données (quelles sont les opérations qui doivent être les plus efficaces ?), ce qui détermine l'implémentation. Il faut savoir que dans une `ArrayList`, les opérations `get()` et `set()` se font en temps constant, mais l'opération `add()` peut se faire en temps linéaire dans le pire des cas (elle se fait cependant se fait en temps amorti constant, c'est-à-dire que n opérations se font en temps $O(n)$) ; dans une `LinkedList`, les opérations `get()` et `set()` se font en temps linéaire, mais l'opération `add()` se fait en temps constant. On adoptera donc la discipline d'abstraction suivante :

- les champs, les paramètres des méthodes, les variables locales sont toujours déclarées avec comme type une interface (`Set`, `List`, etc.) ;
- les classes d'implémentation ne sont utilisées que par l'intermédiaire de leurs constructeurs (`HashSet`, `ArrayList`, etc.).

Ainsi, on déclarera par exemple :

```
  List l = new ArrayList();
  Set s = new HashSet();
  Map m = new HashMap();
```

Voici un exemple d'utilisation qui insère dans un ensemble les mots figurant sur la ligne de commande, ce qui permet de détecter les mots apparaissant plusieurs fois comme étant ceux qui ont déjà été insérés ; ce sont les mots `m` tels que `s.add(m)` retourne `false` :

```
import java.util.Set;
import java.util.HashSet;

class Test {
  public static void main(String[] args) {
    Set s = new HashSet();
    for (int i=0; i<args.length; i++)
      if (!s.add(args[i]))
        System.out.println("mot dupliqué : "+args[i]);
    System.out.println(s.size() + " mots distincts : " + s);
  }
}
```

Les implémentations des collections servent à leur tour à implémenter certaines structures de données. Voici par exemple une implémentation de l'interface `Pile` par la classe `PileParListe` ; un objet de classe `PileParListe` a un champ privé de type `java.util.Liste` :

```
import java.util.List;
import java.util.ArrayList;

class PileParListe implements Pile {

  private List contenu;

  PileParListe() {
    contenu = new ArrayList();
  }
```

```
  public boolean estVide() {
    return contenu.isEmpty();
  }

  public void empiler(Object o) {
    contenu.add(o);
  }

  public Object sommet() {
      return contenu.get(contenu.size()-1);
  }

  public Object dépiler() {
      return contenu.remove(contenu.size()-1);
  }
}
```

Collections et tableaux Les collections constituent un moyen à la fois beaucoup plus souple et plus puissant que les tableaux pour stocker des données. Il est souvent commode de réaliser des "conversions" entre tableaux et collections, et de façon plus générale, d'établir des *vues* selon différents modèles.

La méthode `Object[] toArray()` retourne un *nouveau* tableau d'objets contenant tous les éléments d'une collection :

```
Collection c = ...;
Object[] t = c.toArray();
```

Si l'on souhaite spécifier le type du tableau ainsi créé, on doit utiliser une autre méthode `Object[] toArray(Object[] a)`. Par exemple, pour créer un tableau de `String` (et non un tableau d'objets), il suffit de passer comme argument à cette méthode un tableau de `String` (même de taille nulle), et ensuite de faire un transtypage vers `String[]` :

```
String[] t = (String[]) c.toArray(new String[0]);
```

Cette forme retorse est due au fait que le type d'une référence à un tableau n'est jamais déterminé par le type de ses éléments : par exemple, le tableau désigné par `a` dans l'exemple suivant est de type `Object[]`, et non `String[]`, alors que l'instance définie par `s` est de type `String` :

```
Object[] a = {"p", "q"};
Object s = "p";
```

De plus, si le tableau passé en argument à `toArray()` est de taille suffisante, il est utilisé pour copier les éléments de la collection, de sorte qu'aucun autre objet n'est créé en mémoire :

```
String[] s = new String[c.size];
Object[] x =  (String[]) c.toArray(s);  // x == s
```

Inversement, la méthode statique `static List asList(Object[] a)` de la classe `java.util.Arrays` construit une suite basée sur un tableau d'objets :

```
Object[] t = ...;
List l = Arrays.asList(t);
```

La documentation précise que la suite `l` est de taille fixe (ce n'est donc ni une `ArrayList`, ni une `LinkedList`) et que le tableau et la suite ont les mêmes éléments (toute modification faite à un élement de `l` modifie l'élément correspondant de `t`).

3.10 Les relations d'ordre Plusieurs structures de données (notamment `SortedSet` et `SortedMap`) et algorithmes sur celles-ci (tri d'une collection ou d'un tableau, recherche binaire dans un tableau trié) supposent que leurs éléments peuvent être comparés par une relation d'ordre. Java exprime cette hypothèse en demandant que la classe des éléments implémente l'interface `Comparable` ; c'est le cas d'un certain nombre de classes usuelles, dont les éléments sont comparables par un ordre usuel, par exemple : `Byte`, `Short`, `Integer`, `Long`, `Float`, `Double`, `BigInteger`, `BigDecimal` (ordre numérique), `Character` (ordre alphabétique), `File` (ordre lexicographique sur le chemin d'accès), `String` (ordre lexicographique), `Date` (ordre chronologique).

Rappelons que l'interface `Comparable` déclare une méthode `int compareTo(Object o)`, qui retourne un entier < 0, nul ou > 0 selon que l'objet auquel elle est appliquée précède, est égal ou suit `o`. Voici un exemple de classe qui implémente cette interface ; le constructeur vérifie que ses deux arguments sont non nuls, et déclenche l'exception `NullPointerException` si ce n'est pas le cas, afin de garantir que les méthodes qui s'appliqueront aux champs `prénom` et `nom` ne déclencheront pas cette exception :

```
class Nom implements Comparable {
  private String prénom, nom;

  String valNom() {return nom;}
  String valPrénom() {return prénom;}

  public Nom(String prénom, String nom) {
    if (prénom==null || nom==null)
      throw new NullPointerException();
    this.prénom = prénom;
    this.nom = nom;
  }

  public boolean equals(Object o) {
    return
      o instanceof Nom &&
```

```
      ((Nom)o).prénom.equals(prénom) &&
      ((Nom)o).nom.equals(nom);
  }

  public int hashCode() {
    return 31*prénom.hashCode() + nom.hashCode();
  }

  public int compareTo(Object o) {
    Nom n = (Nom)o;
    int compNom = nom.compareTo(n.nom);
    return
      compNom!=0 ? compNom : prénom.compareTo(n.prénom);
  }

  public String toString() {
    return prénom + " " + nom;
  }
}
```

On notera que les méthodes `equals()` et `compareTo()` se comportent différemment si l'objet n'a pas le type requis, ici `Nom`: le test `o instanceof Nom` permet à `equals(Object)` de retourner `false`, tandis que `compareTo(Object)`, ne procédant pas à ce test, peut déclencher l'exception `ClassCastException` due au transtypage `(Nom)o`. D'autre part, toute classe qui redéfinit `equals()` doit aussi redéfinir `hashCode()`; en effet, deux objets égaux par `equals()` doivent avoir la même valeur de hachage par `hashCode()`. Ces contraintes (sur `equals()`, `compareTo()`, `hashCode()`, etc.) doivent être respectées, afin d'assurer à l'utilisateur que les méthodes qui les utilisent (par exemple, `Collections.sort`, etc.) font bien ce qu'elles sont sensées faire.

Enfin, il arrive que des données doivent être comparées selon plusieurs relations : parfois, selon le nom, parfois selon le prénom, etc. Dans ce cas, associer à la classe un ordre naturel en lui faisant implémenter l'interface `Comparable` n'est pas suffisant. L'API offre une autre interface, `Comparator`, à cette fin :

```
interface Comparator {
  int compare(Object o1, Object o2);
}
```

Une implémentation de cette interface sera par exemple :

```
class PrénomComparator implements Comparator {
   public int compare(Object o1, Object o2) {
     Nom r1 = (Nom) o1;
     Nom r2 = (Nom) o2;
     int prénomComp =
       r1.valPrénom().compareTo(r2.valPrénom());
     if (prénomComp != 0)
```

```
        return prénomComp;
      else
        return r1.valNom().compareTo(r2.valNom());
    }
}
```

On peut alors créer un objet comparateur, et le passer en argument à certaines méthodes qui l'utilisent, par exemple la fonction de tri `Collections.sort()` :

```
class Test {
  public static void main(String[] args) {
    Comparator prénomComparator = new PrénomComparator();
    List l = new ArrayList(...);
    Collections.sort(l, prénomComparator);
    System.out.println(l);
  }
}
```

Implémentations anonymes L'utilisation de l'interface `Comparator` serait lourde si l'on devait définir une classe d'implémentation pour chaque méthode de comparaison ; ceci peut être évité, car Java permet d'instancier des classes anonymes :

```
class Test {

  public static void main(String[] args) {
    List l = new ArrayList(...);
    Comparator prénomComparator = new Comparator() {
      public int compare(Object o1, Object o2) {
        Nom r1 = (Nom) o1;
        Nom r2 = (Nom) o2;
        int prénomComp = r1.valPrénom().compareTo(r2.valPrénom());
        if (prénomComp != 0)
          return prénomComp;
        else
          return r1.valNom().compareTo(r2.valNom());
      }
    };

    Collections.sort(l, prénomComparator);
    System.out.println(l);
  }
}
```

L'expression `new Comparator() { ... }`, qui utilise le nom de l'interface, permet à la fois de créer une instance et d'implémenter la méthode `compare()`. Cette expression pourrait apparaître directement dans l'invocation de `sort()`, bien que ce ne soit plus très lisible, ce qui évite de définir la variable de classe `prénomComparator` :

```
    Collections.sort(l, new Comparator() { ... });
```

3.11 ***Itérations*** Le *pattern d'itération* permet de généraliser à certaines structures de données le parcours itératif d'un intervalle d'entiers : `for(i=0; i<n; i++)` ; les trois opérations de ce parcours étant l'initialisation, le test de fin de parcours et l'incrémentation qui donne l'élément suivant. Un objet de type `Iterator` permet de parcourir une structure de données en gardant en mémoire un *curseur*, qui désigne non un élément, mais une position avant, ou après un élément. Au début du parcours, le curseur est avant le premier élément, et à la fin du parcours, il est après le dernier élément (figure 3.1).

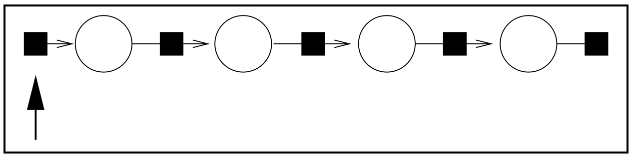

FIG. 3.1 – *Un itérateur sur une structure de données : les éléments sont représentés par des disques, les positions du curseur par des carrés, et initialement, le curseur est avant le premier élément.*

L'interface `Iterator` déclare les méthodes suivantes :

- `boolean hasNext()`, qui teste s'il existe un élément suivant le curseur;
- `Object next()`, qui retourne l'élément suivant le curseur (s'il n'y en a pas, il déclenche une exception de type `NoSuchElementException`), et avance le curseur d'une position ;
- `void remove()`, qui supprime le dernier élément retourné par `next()`, c'est-à-dire l'élément précédant le curseur (son implémentation est optionnelle, elle peut consister à déclencher une exception de type `UnsupportedOperationException`).

Toutes les collections disposent d'une méthode `Iterator iterator()`, qui permet d'initialiser un itérateur, de façon analogue au `int i=0` qui initialise un indice de boucle entier. Par exemple, la procédure suivante permet de supprimer d'une collection tous les éléments qui ne satisfont pas une condition représentée par une méthode `boolean cond(A a)`, à l'aide d'une boucle `while` :

```
static void filtre(Collection c) {
  Iterator i = c.iterator();
  while (i.hasNext()) {
    if (!cond((A) i.next())) i.remove();
  }
}
```

ou d'une boucle `for` :

```
static void filtre(Collection c) {
  for (Iterator i = c.iterator(); i.hasNext(); )
    if (!cond((A) i.next())) i.remove();
}
```

Itération sur les listes La sous-interface `ListIterator` est spécialisée pour l'itération sur les listes, permettant de les parcourir dans l'un ou l'autre sens et de les modifier au cours du parcours (figure 3.2).

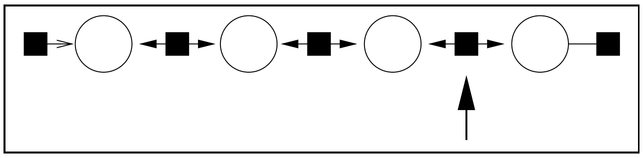

FIG. 3.2 – *Le curseur d'un itérateur de liste peut se déplacer dans les deux sens et être initialisé à une position quelconque.*

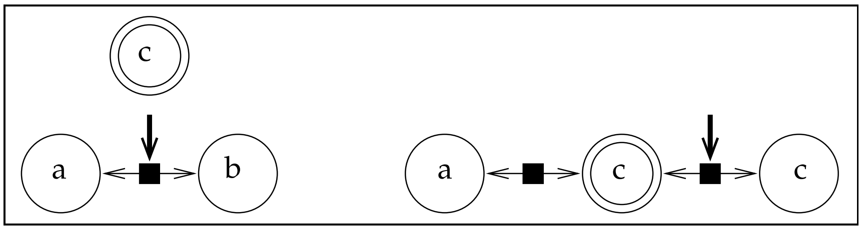

FIG. 3.3 – *Insertion d'un objet par invocation de la méthode* `add(c)`.

Outre les méthodes héritées d'`Iterator`, elle déclare les méthodes suivantes :

– `hasPrevious()` et `previous()`, analogues à `hasNext()` et `next()` et permettant un parcours arrière ;
– `add(Object o)` insère `o` dans la liste juste avant le prochain élément retourné par `next()`, et juste après le prochain élément retourné par `previous()`, ou encore comme seul élément si la liste était vide, et avance le curseur d'une position (figure 3.3) ;
– `set(Object o)` remplace le dernier élément retourné par `next()` ou `previous()` par l'objet `o` (figure 3.4), à moins que `add()` ou `remove()` n'ait été appelé auparavant, auquel cas une exception de type `IllegalStateException` est déclenchée.

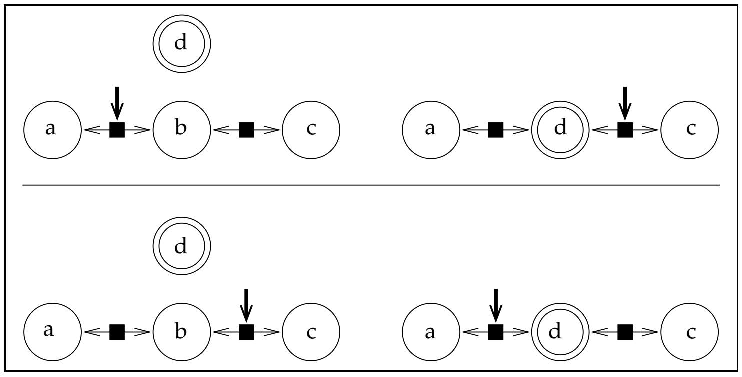

FIG. 3.4 – *Remplacement d'un élément par invocation de la méthode* `set(d)`, *invoquée après un* `next()` *(en haut) et après un* `previous()` *(en bas).*

Les listes disposent aussi d'une méthode `ListIterator listIterator(int n)`, qui retourne un itérateur de liste, c'est-à-dire respectant l'ordre des éléments de la liste et positionne le curseur devant l'élément d'indice `n`. Le parcours arrière d'une liste `l` se ferait ainsi :

```
for (ListIterator i = l.listIterator(l.size());
     i.hasPrevious();) {
  if (!cond((A) i.previous())) i.remove();
}
```

Itérations sur les tables À la différence des collections, les tables ne disposent pas directement d'un mécanisme d'itération. Cependant, trois méthodes permettent de *voir* une table comme un ensemble :

– `keySet()` retourne l'ensemble des clés ;
– `values()` retourne la collection des valeurs ;
– `entrySet()` retourne l'ensemble des associations clé/valeur ; les éléments de cet ensemble sont de type `Map.Entry` (il s'agit d'une interface interne, membre de `Map`) ; les méthodes `getKey()` et `getValue()` permettent d'obtenir les deux composantes d'une association.

```
Map m = ...;
Set
  clés = m.keySet(),
  associations = m.entrySet();
Collection valeurs = m.values();
```

Chacune des trois collections obtenues dispose alors d'un mécanisme d'itération, et ce sont les seules façons d'itérer sur une table :

```
for (Iterator i=clés.iterator(); i.hasNext();)
```

```
    System.out.println(i.next());

  for (Iterator i=valeurs.iterator(); i.hasNext();)
    System.out.println(i.next());

  for (Iterator i=associations.iterator(); i.hasNext();) {
    Map.Entry e = (Map.Entry) i.next();
    System.out.println(e.getKey() + " -> " + e.getValue());
  }
```

Outre la simple énumération des éléments (par `next()`), ces trois vues permettent l'opération `remove()`.

3.12 *Implémentation d'un itérateur* Les algorithmes sur les graphes ont souvent besoin d'énumérer les sommets adjacents à un sommet. Par exemple, un parcours en profondeur utilise l'idiome suivant, où `origine` est de type `Sommet` :

```
    Iterator i = origine.adjacents();
    while (i.hasNext()) {
      Sommet suivant = (Sommet) i.next();
      ...
    }
```

La méthode `adjacents()`, qui retourne un itérateur, est déclarée dans l'interface `Sommet` :

```
public interface Sommet
{
  int getIndice();
  Iterator adjacents();
}
```

Cette méthode ne peut être implémentée que dans le contexte d'une implémentation concrète des graphes. Par exemple, si les graphes sont implémentées comme des matrices d'arcs, par la classe `GrapheParMatrice`, la classe interne privée `_Sommet` qui implémente l'interface `Sommet` définit un itérateur à l'aide d'une implémentation anonyme d'`Iterator`. C'est la méthode `hasNext()` qui fait l'essentiel du travail, pour chercher l'élément suivant non nul dans la ligne de la matrice correspondant au sommet considéré. Les variables booléennes `trouvé` et `terminé` évitent à la méthode `next()` de refaire cette recherche ; cependant comme un utilisateur de l'itérateur n'est pas contraint à invoquer `hasNext()` juste avant `next()`, il faut éventuellement que `next()` invoque `hasNext()` pour faire cette recherche. L'implémentation de la méthode `remove()` (qui est optionnelle) se réduit ici au déclenchement de l'exception `UnsupportedOperationException`.

```
import java.util.Iterator;
```

```
import java.util.NoSuchElementException;

public class GrapheParMatrice extends GrapheAbstrait {
  private Arc[][] matrice;

  private class _Sommet implements Sommet {
    protected int indice;
    protected Object info;

    _Sommet(int indice, Object info) {
      this.indice = indice;
      this.info = info;
    }

    public Iterator adjacents() {
      return new Iterator() {
        private int j=0;
        private boolean trouvé, terminé;

        public boolean hasNext() {
          while (j<nbSommets && matrice[indice][j] == null) j++;
          if (j < nbSommets) {
            trouvé = true;
            return true;
          } else {
            trouvé = false;
            terminé = true;
            return false;
          }
        }

        public Object next() {
          if (terminé) throw new NoSuchElementException();
          if (trouvé || hasNext()) {
            Object suivant = sommets[j];
            j++;
            return suivant;
          } else throw new NoSuchElementException();
        }

        public void remove() {
          throw new UnsupportedOperationException();
        }
      };
    }
  }
}
```

Itération préfixe d'un graphe La structure de pile permet de définir un itérateur préfixe sur les graphes, qui réalise un parcours en profondeur du graphe avec une énumération préfixe des sommets. La définition de cet itérateur utilise deux autres itérateurs, l'un, `j`, pour énumérer tous les sommets (ce qui est nécessaire si tous les sommets ne sont pas accessibles), l'autre, `i`, pour énumérer les sommets adjacents à un sommet :

```
package graphe;
import java.util.Iterator;
import java.util.Set;
import java.util.HashSet;
import java.util.NoSuchElementException;

public class Prefixe implements Iterator {
  Set sommetsVisités;
  Pile pile;
  Iterator j;
  public Prefixe(Graphe g) {
    sommetsVisités = new HashSet();
    pile = new PileParListe();
    j = g.sommets();
    if (j.hasNext()) pile.empiler(j.next());
  }

  public boolean hasNext() {
    if (pile.estVide()) {
      while (j.hasNext()) {
        Sommet s = (Sommet) j.next();
        if (!sommetsVisités.contains(s)) {
          pile.empiler(s);
          return true;
        }
      }
      return false;
    } else {
      return true;
    }
  }

  public Object next() {
    if (pile.estVide()) {
      while (j.hasNext()) {
        Sommet s = (Sommet) j.next();
        if (!sommetsVisités.contains(s)) {
          sommetsVisités.add(s);
          Iterator i = s.adjacents();
          while (i.hasNext()) {
            Sommet suivant = (Sommet) i.next();
            if (!sommetsVisités.contains(suivant)) {
              pile.empiler(suivant);
            }
          }
          return s;
        }
      }
      throw new NoSuchElementException();
    } else {
      Sommet s = (Sommet) pile.dépiler();
```

```
      sommetsVisités.add(s);
      Iterator i = s.adjacents();
      while (i.hasNext()) {
        Sommet suivant = (Sommet) i.next();
        if (!sommetsVisités.contains(suivant)) {
          pile.empiler(suivant);
        }
      }
      return s;
    }
  }

  public void remove() {
    throw new UnsupportedOperationException();
  }
}
```

On l'utilisera de la façon suivante :

```
    Iterator i = new Prefixe(g);
    while (i.hasNext())
      System.out.println((Sommet) i.next());
```

De façon analogue, la structure de file permet de définir un itérateur en largeur sur les graphes :

```
package graphe;
import java.util.Iterator;
import java.util.Set;
import java.util.HashSet;
import java.util.NoSuchElementException;

public class EnLargeur implements Iterator {
  Set sommetsVisités;
  File file;
  Iterator j;
  public EnLargeur(Graphe g) {
    sommetsVisités = new HashSet();
    file = new FileParListe();
    j = g.sommets();
    if (j.hasNext()) {
      Sommet s = (Sommet) j.next();
      file.enfiler(s);
      sommetsVisités.add(s);
    }
  }

  public boolean hasNext() {
    if (file.estVide()) {
      while (j.hasNext()) {
        Sommet s = (Sommet) j.next();
        if (!sommetsVisités.contains(s)) {
          file.enfiler(s);
```

```
          sommetsVisités.add(s);
          return true;
        }
      }
      return false;
    } else {
      return true;
    }
  }

  public Object next() {
    if (file.estVide()) {
      while (j.hasNext()) {
        Sommet s = (Sommet) j.next();
        if (!sommetsVisités.contains(s)) {
          Iterator i = s.adjacents();
          while (i.hasNext()) {
            Sommet suivant = (Sommet) i.next();
            if (!sommetsVisités.contains(suivant)) {
              file.enfiler(suivant);
              sommetsVisités.add(s);
            }
          }
          sommetsVisités.add(s);
          return s;
        }
      }
      throw new NoSuchElementException();
    } else {
      Sommet s = (Sommet) file.défiler();
      Iterator i = s.adjacents();
      while (i.hasNext()) {
        Sommet suivant = (Sommet) i.next();
        if (!sommetsVisités.contains(suivant)) {
          file.enfiler(suivant);
          sommetsVisités.add(suivant);
        }
      }
      return s;
    }
  }

  public void remove() {
    throw new UnsupportedOperationException();
  }
}
```

3.13 Délégation Alors que l'héritage permet de réutiliser certaines méthodes de la sur-classe, la délégation consiste à créer de nouveaux objets pour utiliser leurs fonctionnalités. Nous en verrons plusieurs exemples.

L'exemple des threads Nous avons vu au § 1.19 comment définir un agent par dérivation de la classe `Thread`, et en redéfinissant sa méthode `run()`. Il existe une autre façon pour créer un thread, qui est indispensable quand on travaille déjà dans une classe dérivée, et qui consiste à déléguer à une instance de `Thread` l'exécution de la méthode définissant le comportement de l'agent. Ceci se fait en implémentant l'interface `Runnable`, qui déclare une méthode appelée également `run()`. L'argument `this` du constructeur `Thread` permet au thread créé de savoir de quel objet il doit endosser le comportement, c'est-à-dire d'accéder à la méthode `run()` de l'agent.

```
Class A implements Runnable {

  Thread t;

  A() {
    t = new Thread(this);
    t.start();
  }

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

Un pattern de délégation : les visiteurs On doit représenter les expressions arithmétiques et effectuer un certain nombre de traitements sur celles-ci, par exemple, les évaluer, les imprimer de façon infixe, ou suffixe, etc. Une expression est soit une constante, soit l'addition de deux expressions, soit la multiplication de deux expressions, etc. On transcrit cette définition en une hiérarchie de classes : la classe parente est une classe abstraite `Expr`, ses classes dérivées sont des classes concrètes `Const`, `Plus`, `Mult`, etc. Supposons ces types définis ; il sera naturel de définir, par exemple, l'objet suivant :

```
    Expr expr =                          // expr = 2 + (3+6)
      new Plus(new Const(2),
               new Plus(new Const(3),
                        new Const(6)));
```

Les classes concrètes `Const`, `Plus`, etc., sont dérivées de `Expr` :

```
class Const extends Expr {
  private int c;
  Const(int c) {
    this.c = c;
  }
   ...
}

class Plus extends Expr {
  private Expr expr1, expr2;
```

```
  Plus(Expr expr1, Expr expr2) {
    this.expr1 = expr1;
    this.expr2 = expr2;
  }
   ...
}
```

Une façon de procéder serait de définir des méthodes d'évaluation (`eval()`) et d'affichage (`infixe()`) dans la classe `Expr` et ses classes dérivées.

```
    Expr expr = ...                      // expr = 2 + (3+6)
    Object valeur = expr.eval();         // valeur = 11
    Object chaine = expr.infixe();       // chaine = "2+(3+6)"
```

Une autre façon est de déléguer ces fonctions à un autre objet, appelé *visiteur*. On introduit donc une interface pour typer ces visiteurs d'expression, avec autant de méthodes que de classes concrètes d'expression :

```
interface ExprVisiteur {
  Object visiterConst(int c);
  Object visiterPlus(Expr expr1, Expr expr2);
}
```

Chacune de ces méthodes doit traiter une instance d'une classe d'expressions et produire un objet. Il y aura autant d'implémentations de cette interface que de traitements demandés :

```
class EvalVisiteur implements ExprVisiteur { ... }
class InfixeVisiteur implements ExprVisiteur { ... }
```

La classe `Expr` n'a plus besoin que d'une unique méthode pour associer un de ses objets à un traitement. On appelle cette méthode `déléguer()`, car l'expression « délègue » le traitement à un visiteur, et on l'utilise ainsi :

```
    Expr expr = ...                          // expr = 2 + (3+6)
    ExprVisiteur eval = new EvalVisiteur();
    ExprVisiteur infixe = new InfixeVisiteur();
    Object valeur = expr.déléguer(eval);    // valeur = 11
    Object chaine = expr.déléguer(infixe);  // chaine = "2+(3+6)"
```

Ainsi, les données (les expressions) et les traitements (évaluations, etc.) sur ces données sont découplés en deux objets distincts. On doit donc définir une méthode abstraite dans la classe `Expr` :

```
abstract class Expr {
  abstract Object déléguer(ExprVisiteur v);
}
```

Il faut l'implémenter dans chaque classe dérivée, en appelant la méthode du visiteur spéciale à cette classe :

```
class Const extends Expr {
```

```
  private int c;
  Const(int c) {
    this.c = c;
  }
  Object déléguer(ExprVisiteur v) {
    return v.visiterConst(c);
  }
}

class Plus extends Expr {
  private Expr expr1, expr2;
  Plus(Expr expr1, Expr expr2) {
    this.expr1 = expr1;
    this.expr2 = expr2;
  }
  Object déléguer(ExprVisiteur v) {
    return v.visiterPlus(expr1, expr2);
  }
}
```

Il reste à implémenter les méthodes des visiteurs, pour chaque classe d'expression et pour chaque traitement demandé. Comme tous les traitements doivent retourner un `Object`, l'évaluation retourne un `Integer` (pas un `int`!) et l'impression retourne un `String`, qui sont des `Objects`:

```
class EvalVisiteur implements ExprVisiteur {
  public Object visiterConst(int c) {
    return new Integer(c);
  }
  public Object visiterPlus(Expr expr1, Expr expr2) {
    return
      new Integer(((Integer)expr1.déléguer(this)).intValue() +
                  ((Integer)expr2.déléguer(this)).intValue());
  }
}

class InfixeVisiteur implements ExprVisiteur {
  public Object visiterConst(int c) {
    return Integer.toString(c);
  }
  public Object visiterPlus(Expr expr1, Expr expr2) {
    return
      "(" +
      expr1.déléguer(this) +
      "+" +
      expr2.déléguer(this) +
      ")";
  }
}
```

Les données et les traitements étant découplés, si un nouveau traitement doit être programmé, il suffit d'écrire une nouvelle classe dérivée de `ExprVisiteur`, sans toucher aux autres ni toucher aux différentes classes d'expressions. Si par contre, on ajoute une nouvelle classe d'expressions (les produits, divisions, etc.), il faut aussi ajouter une méthode pour cette classe dans chacune des classes concrètes de visiteurs.

Java ne permet pas, à la différence de certains langages (C, C++, et bien sûr les langages fonctionnels comme Ocaml) de passer en argument une fonction. Ceci s'avère pourtant utile : passer en argument une fonction de comparaison à une fonction de tri, ou une fonction de traitement des éléments à une fonction de parcours d'une structure de données. L'usage des interfaces permet d'y suppléer, comme nous l'avons vu avec l'interface `Comparator`, et de façon plus souple, avec la notion de visiteur.

3.14 *Les flots* Les entrées et les sorties sont organisées en Java autour du modèle des *flots* (anglais *stream*), à l'aide d'un ensemble très développé de types, mettant en œuvre divers patterns. Un flot est une suite de données (octets, caractères, objets quelconques) successivement lues ou écrites. Ces flots sont ainsi classifiés en flots d'entrée (qui comportent des méthodes de lecture), et en flots de sortie (qui comportent des méthodes d'écriture). Outre le programme, un flot d'entrée est connecté à une source, et un flot de sortie à une cible. La source ou la cible d'un flot peut être un fichier, un tampon en mémoire, une chaîne de caractères, un autre flot, une ressource Web ou bien un port Internet. Un type de flot est donc caractérisé par trois éléments : le type des données, sa direction et sa connectivité.

Les flots les plus élémentaires sont des flots d'octets. Les classes des flots d'octets en écriture ont un nom en `...OutputStream`; celles des flots d'octets en lecture ont un nom en `...InputStream`. Pour travailler avec des flots de caractères Unicode, on utilisera des classes en `...Writer` (en écriture) et `...Reader` (en lecture). Toutes ces classes font partie du paquet `java.io`; les unités de compilation faisant appel à ces classes commenceront donc, par commodité, par :

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

Les méthodes générales de lecture sur un flot d'octets, déclarées dans la classe abstraite `InputStream` sont :

- `int read()`, qui lit l'octet suivant disponible sur le flot (et se bloque en l'attendant), et le retourne dans un `int` entre `0` et `255`, ou bien retourne `-1` si la fin du flot est atteinte, ou déclenche l'exception `IOException` en cas d'erreur de lecture ;
- `int read(byte b[])`, qui lit au plus `b.length` octets du flot, les place dans le tableau `b`, et retourne le nombre d'octets lus ou bien -1 si la fin du flot est atteinte, ou déclenche l'exception `IOException` en cas d'erreur de lecture (par exemple, si le flot a été fermé) ; les éléments du tableau `b` qui n'ont pas été écrits ne sont pas modifiés.

Les méthodes générales d'écriture sur un flot d'octets, déclarées dans la classe abstraite `OutputStream`, sont :

- `void write(int c)`, qui écrit un octet, représenté par un `int`
- `void write(byte[] b)`, qui écrit les `b.length` octets de `b`
- `void flush()`, qui vide le flot sur sa cible (mémoire, fichier, etc.)

Comme premier exemple d'utilisation de ces méthodes, la procédure suivante lit sur un flot d'octets `in`, octet par octet, et les écrit sur un flot d'octets `out` ; on notera que les types des paramètres sont abstraits :

```
static void copier(InputStream in, OutputStream out)
  throws IOException {
  int c;
  while ((c = in.read()) != -1) out.write(c);
}
```

En l'absence de récupération de l'exception `IOException`, cette procédure doit déclarer qu'elle est susceptible de déclencher (c'est-à-dire de propager) cette exception. Une autre façon d'écrire cette itération est la suivante :

```
int c = in.read();
while (c != -1) {
  out.write(c);
  c = in.read();
}
```

Toute application peut accéder aux *flots standards* d'entrée `System.in` (de type `InputStream`, connecté par défaut au clavier), de sortie `System.out` et de sortie d'erreur `System.err` (tous deux de type `PrintStream`, sous-type de `OutputStream`, et connectés par défaut à l'écran). La procédure `copier()` peut servir à recopier du clavier à l'écran les caractères tapés par l'utilisateur :

```
copier(System.in, System.out);
```

3.15 Fichiers La lecture et l'écriture d'octets sur un fichier se fait à l'aide des classes `FileInputStream` et `FileOutputStream`, sous-types respectifs de `InputStream` et `OutputStream`. L'exemple suivant montre une application qui copie un fichier dans un autre (dont les noms sont donnés sur la ligne de commande), octet par octet ; si les deux noms de fichiers ne sont pas donnés sur la ligne de commande, on utilise les flots standards `System.in`, et `System.out`, ce qui est possible car `FileOutputStream` et `PrintStream` sont des sous-types de `OutputStream`.

```
public static void main(String[] args)
```

```
    throws IOException {
    InputStream in = System.in;
    OutputStream out = System.out;
    if (args.length > 0)
      in = new FileInputStream(args[0]);
    if (args.length > 1)
      out = new FileOutputStream(args[1]);
    copier(in, out);
  }
```

En l'absence de récupération de l'exception `IOException`, la méthode `main()` doit déclarer qu'elle est susceptible de déclencher cette exception.

Pour concaténer des octets à la fin d'un fichier (au lieu d'écrire en écrasant éventuellement son contenu), on utilise un autre constructeur de `FileOutputStream`, avec l'argument supplémentaire `true` :

```
    if (args.length > 1)
      out = new FileOutputStream(args[1], true);
```

La classe `File` a pour objet des *chemins d'accès* à des fichiers ou à des répertoires (et non les fichiers eux-mêmes). Cette classe est utile pour obtenir diverses propriétés des fichiers (savoir si un chemin désigne un fichier ordinaire ou un répertoire, est accessible en lecture ou en écriture, etc.) :

```
File cheminRepertoire =
  new File("/usr/local/www/doc/java/jdk1.1.5/docs");
File cheminFichier =
  new File(cheminRepertoire, "index.html");
...
  if (cheminRepertoire.isDirectory() &&
      cheminFichier.canRead()) {
    ...
  }
```

La méthode `length()`, appliquée à un objet de type `File` et qui retourne un `long`, détermine la taille d'un fichier en nombre d'octets.

Modes d'accès à un fichier Si un accès direct à une position quelconque du fichier est nécessaire, on devra utiliser la classe `RandomAccessFile`, qui fonctionne à la fois en écriture et en lecture, permettant de sauvegarder, puis de restituer, la position d'une opération dans le fichier. Le second d'argument du constructeur est une chaîne qui spécifie le mode d'accès : `"rw"` indique un accès en lecture (`"r"` pour « read ») et en écriture (`"w"` pour « write »), et `"r"` indique un accès en lecture seulement.

On obtient la position courante à l'aide de la méthode `getFilePointer()`, qui retourne un `long`. On peut se positionner à l'aide de la méthode `seek()`, qui a un paramètre entier de type `long` indiquant un déplacement dans le fichier.

```
RandomAccessFile inOut =
    new RandomAccessFile("out.txt", "rw");
inOut.seek(inOut.length());       // positionnement à la fin
inOut.writeBytes("etc., ...");    // ajout de caractères
inOut.seek(0);                    // positionnement au début
inOut.writeBytes("Au début");     // écrase le début !
inOut.seek(0);                    // positionnement au début
String s = inOut.readLine();      // lecture d'une ligne
inOut.close();
```

Cette classe implémente les interfaces `DataInput` et `DataOutput`.

3.16 Le pattern de décoration Les méthodes `read()` et `write()` sont les plus primitives. Il est généralement nécessaire, tant par commodité que par efficacité, de recourir à d'autres méthodes.

Par exemple, pour *imprimer* une représentation textuelle d'une valeur quelconque sur un flot d'octets, on invoque `print()` et `println()`, qui ne sont définies que dans `PrintStream`, qui est une sous-classe de `OutputStream`. Si l'on veut imprimer ces représentations dans un fichier, on ne peut pas utiliser simplement `FileOutputStream` car cette classe est dépourvue de ces méthodes d'impression. Comment faire pour disposer à la fois des fonctionnalités de `PrintStream` et de `FileOutputStream` ? Il n'est pas possible, en Java, de définir une classe qui soit sous-classe directe de deux classes : le mécanisme d'extension, souvent utilisé pour ajouter des fonctionnalités à une classe est ici insuffisant. On peut par contre procéder par délégation.

On ajoute les fonctionnalités offertes par `PrintStream`, en faisant d'une instance de `FileOutputStream` une instance de `PrintStream` :

```
PrintStream out =
  new PrintStream(
    new FileOutputStream("out.txt"), true);
out.println(2);
out.println(new Integer(2));
```

L'argument `true` permet de vider automatiquement le tampon d'écriture à la fin des lignes.

Ce mécanisme est très courant parmi les classes du paquet `java.io` : le constructeur prend en argument un flot et lui ajoute des fonctionnalités. Il s'agit d'un pattern dit de *décoration*, également très employé dans les classes graphiques (par exemple, pour décorer une fenêtre).

Les classes suivantes, sous-classes de `FilterOutputStream`, elle-même sous-classe de `OutputStream`, ont toutes un constructeur qui prend en argument un `OutputStream` et lui ajoutent des fonctionnalités.

- `BufferedOutputStream`: utilise un tampon pour grouper une série d'opérations d'écritures consécutives, ce qui optimise les écritures ;
- `DataOutputStream`: implémente l'interface `DataOutput` qui déclare des méthodes spécialisées pour l'écriture de données primitives (`writeInt()`,`writeDouble()`,`writeChar()`,etc..);
- `DeflaterOutputStream`: permet d'écrire des données comprimées ;
- `PrintStream`: permet d'écrire une représentation textuelle des données.

L'exemple suivant, qui empile trois constructeurs, permet d'ajouter successivement un tampon d'écriture et les méthodes d'impression textuelle à un flot d'écriture sur fichier :

```
PrintStream out =
  new PrintStream(
    new BufferedOutputStream(
      new FileOutputStream("out.txt")));
out.println(2);
out.println(new Integer(2));
```

Tampons Chaque opération de lecture ou d'écriture peut être très coûteuse sur certains flots ; c'est notamment le cas des accès à un fichier, ou des accès à l'Internet. Pour éviter des opérations individuelles (sur un octet ou sur un caractère à la fois), on préfère souvent travailler sur un *tampon* (anglais *buffer*). Par exemple, pour écrire sur un fichier, on écrira sur un flot-tampon, lequel est attaché à un flot d'écriture sur un fichier. Les classes qui mettent en œuvre ces tampons sont :

- `BufferedInputStream`, pour les flots d'octets en entrée
- `BufferedOutputStream`, pour les flots d'octets en sortie
- `BufferedReader`, pour les flots de caractères en entrée
- `BufferedWriter`, pour les flots de caractères en sortie

Un flot de caractères de la classe `BufferedReader` permet des opérations supplémentaires (par exemple, lecture d'une ligne de texte, par la méthode `readLine()`). Il est très courant de connecter un tel tampon à un flot de lecture sur un fichier :

```
BufferedReader in =
  new BufferedReader(new FileReader("toto"));
String s = in.readLine();
```

Symétriquement, pour écrire sur un fichier, il est préférable de travailler avec un tampon :

```
PrintWriter out =
  new PrintWriter(
    new BufferedWriter(
      new FileWriter("toto")));
out.println("un long texte");
```

Flots de caractères Les flots de caractères sont des objets de classe `Reader` (flots de caractères d'entrée) ou `Writer` (flots de caractères de sortie). Les méthodes `read()` et `write()` de ces classes sont analogues à celles opérant sur des flots d'octets, à la différence que c'est un caractère 16 bits qui est lu ou écrit, et non un octet.

La conversion entre un flot d'octets et un flot de caractères se fait à l'aide des classes `OutputStreamWriter` et `InputStreamReader`. Cette conversion se fait, par décoration d'un flot d'octets :

```
InputStreamReader isr = new InputStreamReader(System.in);
```

Cette conversion permet éventuellement de spécifier le codage utilisé (par exemple, par la chaîne "MacSymbol" s'il s'agit d'un codage MacIntosh) pour lire un fichier `"toto"`:

```
InputStreamReader isr =
  new InputStreamReader(
    new FileInputStream("toto"),
    "MacSymbol"
  );
```

Pour connecter un flot de caractères à un fichier, si la conversion par défaut est appropriée, il est plus simple de recourir aux classes `FileWriter` et `FileReader`, qui s'emploient de façon analogue à `FileOutputStream` et `FileInputStream`.

La classe `PrintWriter` permet d'écrire sur un flot de sortie des données en les représentant à l'aide de chaînes de caractères Unicode (16 bits), à l'aide des méthodes `print()` et `println()` (la représentation textuelle d'un objet est obtenue par la méthode `toString()`). Pour bénéficier de ces méthodes, on doit procéder par décoration d'un objet de type `Writer` :

```
PrintWriter pw =
  new PrintWriter(
    new FileWriter("toto"));
...
pw.println("ici, un texte en caractères Oriya");
```

3.17 Flots de données Si l'on veut lire et écrire, non pas des octets, mais des valeurs d'un type connu, il faut utiliser les classes `DataInputStream` et `DataOutputStream`. Ces classes disposent de méthodes spécialisées pour divers types de valeurs : `readInt()`, `readDouble()`, `readChar()`, `readBoolean()`, etc., et les méthodes `write...()` correspondantes.

On doit connecter un flot de données à un autre flot pour bénéficier de ces méthodes supplémentaires, lors de sa création :

```
DataOutputStream dos =
  new DataOutputStream(
    new FileOutputStream("toto"));
...
dos.writeBoolean(true);
dos.writeInt(4);
dos.close();
```

On connecte ainsi un flot d'entrée à un autre flot d'entrée, ou un flot de sortie à un autre flot de sortie. Par exemple, pour lire un entier sur l'entrée standard :

```
DataInputStream dis =
  new DataInputStream(new FileOutputStream("toto"));
int n = dis.readInt();
```

Le modèle des flots peut être utilisé afin de lire ou d'écrire sur un tableau d'octets. Supposons que l'on dispose d'un tableau d'octets `data` (reçu par exemple par une communication UDP sur l'Internet). La classe `ByteArrayInputStream` permet de construire un flot de lecture à partir de ce tableau `data`.

```
byte[] data = ...;

ByteArrayInputStream in =
  new ByteArrayInputStream(data);
```

Si l'on sait qu'un tableau d'octets contient la représentation d'un booléen et d'un entier, on doit le lire à l'aide des méthodes déclarées dans `DataInput`. On ajoute ces fonctionnalités par décoration du flot, à l'aide du constructeur `DataInputStream`:

```
byte[] data = ...;
DataInputStream dis =
  new DataInputStream(
    new ByteArrayInputStream(data));

boolean b = dis.readBoolean();
int n = dis.readInt();
dis.close();
```

Ceci suppose que ce tableau contient des données codées de façon compatible avec le décodage réalisé par les méthodes `readBoolean()`, etc. Ce sera le cas, symétriquement, si le tableau d'octets a été obtenu par les classes `DataOutputStream` et `ByteArrayOutputStream` :

```
byte[] data;
ByteArrayOutputStream baos = new ByteArrayOutputStream();
DataOutputStream dos = new DataOutputStream(baos);
dos.writeBoolean(true);
dos.writeInt(4);
data = baos.toByteArray();
dos.close();
```

De même, la classe `StringReader` permet de construire un flot de lecture à partir d'une chaîne de caractères :

```
String s = ...;

StringReader in =
  new StringReader(s);
```

Persistance et sérialisation Les classes `ObjectOutputStream` et `ObjectInputStream` permettent de rendre *persistants* les objets de Java en les sauvegardant sur un flot (qui peut être écrit sur un fichier), puis en les relisant. Seuls les objets dont la classe implémente l'interface `Serializable` peuvent bénéficier de ce mécanisme, appelé *sérialisation*. Si un objet comporte des champs qui sont des références à d'autres objets sérialisables, ces objets sont aussi sérialisés ; un objet partagé (par exemple, figure 1.16, p. 46) ne sera sauvegardé qu'en un seul exemplaire. Ces fonctionnalités sont encore obtenues par décoration d'un flot.

```
ObjectOutputStream oos =
  new ObjectOutputStream(new FileOutputStream("toto"));
oos.writeObject("Aujourd'hui");
oos.writeObject(new java.util.Date());
```

La lecture de ces objets, qui se fait dans le même ordre que leur écriture, doit opérer un transtypage d'`Object` vers la classe que l'on veut restituer :

```
ObjectInputStream ois =
  new ObjectInputStream(new FileInputStream("toto"));
String chaîne = (String) ois.readObject();
java.util.Date date = (java.util.Date) ois.readObject();
```

Pour faire bénéficier une classe de ce mécanisme, il suffit de déclarer que la classe implémente `Serializable`, interface qui ne déclare aucune méthode.

3.18 Les flots et l'Internet L'accès à des ressources sur le World Wide Web, ou plus généralement, sur l'Internet, s'intègre naturellement dans le modèle des flots. Voici deux exemples de *connexion* d'un programme à un service Internet.

Le programme suivant connecte un flot d'entrée à une ressource Web spécifiée par son URL, transforme ce flot d'octets en un flot de caractères et le place dans un tampon ; les lignes successivement lues sur ce flot d'entrée sont copiées sur la sortie standard du programme (jusqu'à ce que `readLine()` retourne `null`).

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

class URLReader {
  public static void main(String[] args)
    throws MalformedURLException, IOException {
    URL url = new URL("http://www.enpc.fr/");
    BufferedReader in =
      new BufferedReader(
        new InputStreamReader(
          url.openStream()));
    String ligne;

    while ((ligne = in.readLine()) != null)
      System.out.println(ligne);

    in.close();
  }
}
```

Le programme suivant connecte un flot d'entrée à un port Internet spécifié par un nom de machine et un numéro de port ; ce numéro, 13, est celui d'un serveur dont la réponse est une ligne contenant la date et l'heure courante ; la connexion à ce port utilise le protocole TCP. Ce flot d'entrée est ensuite transformé en un flot de caractères, puis placé dans un tampon ; la ligne lue sur ce flot d'entrée est simplement copiée sur la sortie standard du programme.

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

class DateReader {
  public static void main(String[] args)
    throws UnknownHostException, IOException {

    String nomHote = args.length>0 ? args[0] : "localhost";
    Socket s = new Socket(nomHote, 13);
    BufferedReader reponse =
```

```
        new BufferedReader(
          new InputStreamReader(
            s.getInputStream()));
    String date = reponse.readLine();
    System.out.println(nomHote + " : " + date);
    reponse.close();
    s.close();
  }
}
```

3.19 Communication entre agents par tubes Il s'agit d'une technique importante pour connecter entre eux deux programmes, en connectant un flot de sortie du premier programme avec un flot d'entrée du second programme. Cette technique, dite de *tube*, issue d'une pratique courante dans le système Unix (appelée en anglais *pipe*), est appliquée ici, non à deux programmes, mais à deux agents implémentés par des *threads*. Les classes permettant ces connexions sortie/entrée sont :

- `PipedInputStream` et `PipedOutputStream`, pour les tubes d'octets
- `PipedReader` et `PipedWriter`, pour les tubes de caractères.

Un tube d'entrée doit être connecté à un tube de sortie. Considérons une classe `Producteur`, dont un constructeur prend en argument un flot de sortie `out`, et une classe `Consommateur`, dont un constructeur prend en argument un flot d'entrée `in`. Comme on doit connecter `in` et `out`, ces flots doivent être des tubes, par exemple de caractères :

```
  try {
    PipedWriter out = new PipedWriter();
    PipedReader in = new PipedReader(out);
  } catch (IOException e) {}
```

On peut alors connecter des agents `p` et `c` à l'aide de ces flots.

```
  Producteur p = new Producteur(out);
  Consommateur c = new Consommateur(in);
```

L'écriture dans `out` et la lecture dans `in` seront réalisés par les agents eux-mêmes, dont on suppose qu'ils ont chacun leur propre thread (l'un des deux pouvant être le thread du programme principal) :

```
  p.start();
  c.start();
```

Dans l'exemple suivant, le producteur écrit le caractère `'a'` sur son flot de sortie toutes les 1000 millisecondes ; le consommateur est continuellement en attente d'un caractère sur son flot d'entrée.

```
class Producteur extends Thread {
```

```
 private Writer out;

 Producteur(Writer out) {
   this.out = out;
 }

 public void run() {
   while (true) {
     try {
       out.write('a');
       Thread.sleep(1000);
     }
     catch(InterruptedException e) {}
     catch(IOException e) {}
   }
 }
}

class Consommateur extends Thread {
  private Reader in;

  Consommateur(Reader in) {
    this.in = in;
  }

  public void run() {
    while (true) {
      try {
        char c = (char) in.read();
        System.out.println(c);
      }
      catch(IOException e) {}
    }
  }
}
```

3.20 Un pattern de création : les fabriques Même si l'on peut déclarer des variables d'un type abstrait, les constructeurs font nécessairement partie d'un type concret. Une application qui contient un grand nombre d'appels à des constructeurs est donc très dépendante des classes concrètes choisies. Si l'on décide de changer l'implémentation de certains types abstraits, il faut remplacer toutes les invocations des constructeurs des classes concrètes correspondantes, ce qui n'est pas commode et sujet à erreurs. Il est préférable de rassembler tous les choix d'implémentation dans un objet particulier, appelé *fabrique*, auquel sera déléguée la construction des objets. Cette technique constitue le *pattern de fabrication*.

Prenons l'exemple d'une application qui doit utiliser des listes, des ensembles et des tables. Les variables et les paramètres des méthodes seront déclarées à l'aide des interfaces `List`, `Set` et `Map`.

La construction de listes, d'ensembles et de tables est déléguée à une fabrique `f`, déclarée d'un type abstrait `Fabrique`. La fabrique est créée à l'aide d'un constructeur d'une classe concrète qui rassemble les choix d'implémentations. Ici la classe `MaFabrique` indique que les implémentations `ArrayList`, `HashSet` et `HashMap` sont choisies. Dans la suite du programme, là où l'on aurait invoqué un constructeur, `new ArrayList()`, `new HashSet()` ou `new HashMap()`, on invoque une méthode déclarée dans le type abstrait `Fabrique` sur l'objet fabrique : respectivement `f.fabriquerList()`,`f.fabriquerSet()` ou `f.fabriquerMap()` :

```
  Fabrique f = new MaFabrique();
  List l = f.fabriquerList();
  Set s = f.fabriquerSet();
  Map m = f.fabriquerMap();
```

`Fabrique` est une interface, et c'est en l'implémentant qu'on décide quelles implémentations vont être utilisées :

```
interface Fabrique {
  List fabriquerList();
  Set fabriquerSet();
  Map fabriquerMap();
}

class MaFabrique implements Fabrique {
  List fabriquerList(){
    return new ArrayList();
  }
  Set fabriquerSet(){
    return new HashSet();
  }
  Map fabriquerMap(){
    return new HashMap();
  }
}
```

Si d'autres choix sont à faire, il suffit de modifier la classe concrète `MaFabrique` ; mieux, on définit une autre classe concrète implémentant `Fabrique` et on modifie la seule ligne `Fabrique f = new MaFabrique()`. Par exemple, si l'on préfère, pour des raisons de préservation de l'ordre des données, des implémentations des ensembles et des tables par des arbres plutôt que par hachage, on définira :

```
class FabriqueParArbres implements Fabrique {
  List fabriquerList(){
    return new ArrayList();
  }
  Set fabriquerSet(){
    return new TreeSet();
  }
```

```
  Map fabriquerMap(){
    return new TreeMap();
  }
}
```

Il suffira de définir une fabrique par :

```
  Fabrique f = new FabriqueParArbres();
```

Le reste du programme n'a pas besoin d'être modifié. Dans une approche plus systématique, l'application figurerait dans une classe `Application`, dont un constructeur aurait un paramètre de type `Fabrique`. L'instanciation de l'application se ferait par :

```
 ... new Application(new MaFabrique()) ...
```

3.21 ***Erreurs et exceptions*** La notion d'exception décrit des situations où la procédure normale d'évaluation des expressions n'est pas pertinente. Cette procédure suppose que l'évaluation d'une expression (par exemple, d'une invocation de méthode) résulte en la production d'une valeur. Il y a des cas où une valeur n'est pas obtenue, quand la procédure d'évaluation conduirait à exécuter une opération qui ne peut pas ou ne doit pas être réalisée :

– parce qu'une ressource demandée n'est pas disponible : de la mémoire ne peut pas être allouée, un fichier ne peut pas être ouvert, le code d'une classe ne peut pas être chargé, etc ;
– parce qu'une opération n'est pas permise par la sémantique du langage : division entière par zéro, accès à un tableau hors de ses bornes ;
– parce qu'une opération n'est pas permise par la sémantique de l'application : dépiler une pile vide.

Il est donc possible que l'évaluation d'une expression, au lieu de retourner une valeur, déclenche une *exception*. Certaines de ces exceptions sont des erreurs, qui conduisent fatalement à une fin prématurée du programme. D'autres peuvent être *récupérées* pour permettre la poursuite du programme.

En Java, les exceptions sont représentées par des objets de classe `Throwable`. La sous-classe `Error` est formée des exceptions qui ne sont pas considérées comme récupérables ; elles concernent les opérations de la machine virtuelle Java.

La sous-classe `Exception` est formée des exceptions considérées comme récupérables. Cependant, les objets de la sous-classe `RuntimeException` d'`Exception` ne sont pas obligatoirement récupérables : cette sous-classe comporte les exceptions

– `ArithmeticException`,

– `ArrayStoreException`,
– `NullPointerException`,
– `IndexOutOfBoundsException`, etc.

Les autres sous-classes d'`Exception` sont formées d'exceptions qui doivent être récupérées : par exemple,

– `java.io.FileNotFoundException`,
– `java.net.UnknownHostException`, ou
– `InterruptedException`.

Ces exceptions sont dites *contrôlées* parce que le compilateur vérifie comment elles sont traitées. Les exceptions définies par le programmeur sont des sous-classes d'`Exception` ; elles sont donc contrôlées.

Considérons l'implémentation d'une pile par un tableau. Les opérations d'empilage et de dépilage peuvent conduire à exécuter une écriture ou une lecture en dehors des bornes du tableau. Ces opérations ne seront pas réalisées et déclencheront une `IndexOutOfBoundsException`, qui n'est pas obligatoirement récupérable.

Cependant, une écriture en dehors du tableau, demandée par un empilage, n'est une erreur que dans la mesure où la taille du tableau, choisie *a priori*, n'est pas assez grande. Plutôt que de sortir brutalement du programme, on peut créer un nouveau tableau de taille double, copier le contenu du tableau précédent dans le nouveau, et continuer avec celui-ci ; cette exception est donc récupérable par la méthode d'empilage. Mieux : on n'utilise pas de tableau, mais le type `java.List`.

Par contre, la lecture en dehors du tableau (à l'indice `-1`), qui provoque la même exception, est due à une erreur de conception de l'algorithme utilisant la pile ; il ne revient donc pas aux opérations de la pile de récupérer cette exception ; par contre, la fonction qui demande un dépilage devrait, éventuellement, récupérer cette exception.

```
public class PileVideException extends Exception {
  PileVideException(Pile p) {
    super("Pile vide");
  }
}
```

Une méthode dont le corps est susceptible de lever une exception contrôlée doit :

– soit intercepter l'exception et la traiter (par un `try` ... `catch`)
– soit indiquer que cette exception est propagée (par un `throws`)

Cette indication, sous la forme d'une liste d'exceptions, intervient dans la relation de typage. La récupération minimale, qui gobe n'importe quelle exception, sans rien dire, est :

```
try {
...
}
catch (Exception e) {}
```

Une version plus informative de cette récupération minimale consiste à imprimer la chaîne de caractères décrivant l'exception sur le flot de sortie en erreur standard, par `System.err.println(e)`, ou mieux, à imprimer l'état de la pile :

```
try {
  ...
}
catch (Exception e) {
  e.printStackTrace()
}
```

La déclaration d'une exception se fait dans l'en-tête de la méthode, à la fois dans sa déclaration, dans une interface :

```
Object sommet() throws PileVideException;
```

et dans son implémentation :

```
public Object sommet() throws PileVideException {
  if (!estVide())
    return contenu.get(contenu.size()-1);
  else throw new PileVideException(this);
}
```

L'exécution du `throw` provoque d'une part la sortie immédiate de la méthode, et d'autre part le dépilage des appels jusqu'à ce qu'apparaisse un cadre d'invocation d'une méthode contenant un `catch`: l'exception est donc propagée le long de la chaîne invocations tant qu'elle n'est pas récupérée. Ce mécanisme de transmission est distinct du mécanisme usuel de retour d'une méthode. Si l'exception n'est jamais récupérée, le programme termine anormalement.

La récupération et le traitement sont réalisés en plaçant les appels susceptibles de déclencher une exception dans un `try ... catch`:

```
try {                             //
  p.empiler(a);                   // invocation protégée
} catch (PileVideException e) {   // récupération d'exception
  e.printStackTrace();            // traitement de l'exception
}
```

Ce style de programmation est nécessaire dans des logiciels qui doivent être robustes.

3.22 *Indications bibliographiques* Ecrit il y a déjà une dizaine d'années, le *SICP*[1] reste un des livres les plus inspirés sur la programmation, enseignée au moyen de Scheme ([13]), un langage fonctionnel dérivé de Lisp.

De même que le meilleur ouvrage consacré à C a toujours été et reste le KERNIGHAN-RITCHIE[10], on ne peut qu'indiquer le STROUSTRUP[14] comme référence pour C++, et [4] pour Java.

La programmation fonctionnelle est très clairement exposée dans [8]. La programmation dans un langage à objets est présentée dans [14] (C++) et [4] (Java) par les auteurs de ces langages.

Pour comprendre la compilation des langages de programmation et leur environnement d'exécution, le grand classique est le « dragon » [2].

La notion de pattern, issue des recherches de l'architecte Christopher Alexander [3], est développée dans [9], avec de nombreux exemples (en C++).

L'architecture des machines contemporaines est expliquée avec brio par deux des concepteurs du modèle RISC, HENNESSY et PATTERSON[11], qui enseignent à Stanford et à Berkeley.

Pour l'algorithmique, on préférera la côte est, avec l'excellent cours du MIT de CORMEN, LEISERSON et RIVEST[7], ou sur les rives de la Seine, le livre de BEAUQUIER, BERSTEL et CHRÉTIENNE[5]. Pour des algorithmes numériques, les *Numerical Recipes*[12] constitue un matériel de référence dont un ingénieur doit connaître l'existence. Pour des algorithmes plus algébriques que numériques, les deux petits volumes [6] de BERSTEL, PIN et POCCHIOLA sont une mine de problèmes intéressants.

Références

[1] H. Abelson, G.J. Sussman, and J. Sussman. *Structure and Interpretation of Computer Programs*. The MIT Press, seconde edition, 1996.

[2] A. Aho, R. Sethi, and J. Ullman. *Compilateurs, Principes, techniques et outils*. Collection iia. InterÉditions, Paris, 1989.

[3] C. Alexander, S. Ishikawa, and M. Silverstein. *A Pattern Language : Towns, Buildings, Construction*. Oxford University Press, 1977.

[4] K. Arnold and J. Gosling. *Java Programming Language*. Addison Wesley, 2nd edition, 1998.

[5] D. Beauquier, J. Berstel, and P. Chrétienne. *Élements d'algorithmique*. Manuels informatique Masson. Masson, Paris, 1992.

[6] J. Berstel, J.E. Pin, and M. Pocchiola. *Mathématiques et Informatique, Problèmes résolus*. McGraw-Hill, Paris, 1992. 2 volumes.

[7] T.H. Cormen, C.E. Leiserson, and R.L. Rivest. *Introduction à l'algorithmique*. Dunod, 1994.

[8] G. Cousineau and M. Mauny. *Approche Fonctionnelle de la Programmation*. Collection Informatique. Ediscience international, Paris, 1995.

[9] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns. Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[10] B.W. Kernighan and D.M. Ritchie. *Le Langage C, C ANSI*. Manuels informatiques Masson. Masson, Paris, seconde edition, 1990.

[11] D.A. Patterson and J.L. Hennessy. *Organisation et conception des ordinateurs : l'interface matériel/logiciel*. Dunod, Paris, 1994.

[12] W.H. Press, S.A. Teukolsky, W.T. Vetterling, and B.P. Flannery. *Numerical Recipes in C, The Art of Scientific Computing*. Cambridge University Press, seconde edition, 1992.

[13] G. Springer and D.P. Friedman. *Scheme and the Art of Programming*. The MIT Press, Cambridge, MA, 1989.

[14] B. Stroustrup. *C++ Programming Language*. Addison Wesley, 3r edition, 1997.

A

À-côtés

Ce chapitre rassemble des éléments de programmation qui sont probablement déjà connus car ils ne concernent pas spécifiquement la programmation objet : ce sont les types primitifs, les traits impératifs et applicatifs de la plupart des langages de programmmation.

A.1 Types entiers Java, comme la plupart des langages, mais contrairement à quelques uns (comme CAML ou Maple), ne dispose pas d'un type primitif représentant des entiers de taille quelconque, mais de quatre types entiers de taille fixe : `byte` (un octet), `short` (deux octets), `int` (quatre octets), `long` (huit octets), le plus courant étant `int`. Ce sont tous des types d'entiers *signés* (c'est-à-dire positifs ou négatifs).

Quand un type entier est codé sur N bits, 2^N entiers peuvent être représentés. Pour des types non-signés, on peut ainsi représenter l'intervalle $[0, 2^N - 1]$; la suite de bits $b_{N-1} \ldots b_0$ représente l'entier $b_{N-1}2^{N-1} + \ldots + 2^1 b_1 + b_0$. Pour des types signés, les entiers dans l'intervalle $[-2^{N-1}, 2^{N-1} - 1]$ sont représentables ; les entiers ≥ 0 sont représentés de la même façon que dans le cas non-signé et l'on a $b_{N-1} = 0$; les entiers < 0 sont représentés en *complément à deux* et l'on a $b_{N-1} = 1$; cela signifie que $b_{N-1} \ldots b_0$ représente l'entier $-b_{N-1}2^{N-1} + b_{N-2}2^{N-2} + \ldots + 2^1 b_1 + b_0$; par exemple $\texttt{111..1}_2$ représente -1.

La notation décimale peut être utilisée pour noter les valeurs de type `int` ; un `L` (ou un `l`, moins lisible) doit être suffixé pour représenter une valeur de type `long` : `12` est un `int`, et `12L` est un `long`.

À chacun des types primitifs `byte`, `short`, `int` et `long` est associée une classe enveloppante, respectivement `Byte`, `Short`, `Integer` et `Long` , qui contiennent des constantes et fonctions très utiles. Les entiers représentés par le type `int` forment l'intervalle

$$[\texttt{Integer.MIN\_VALUE}, \texttt{Integer.MAX\_VALUE}].$$

Il y a des constantes analogues pour les trois autres types entiers. Le plus grand entier du type `int`, `Integer.MAX_VALUE`, est de l'ordre de 2×10^9. Les opérations sont réalisées modulo 2^N, et aucun test de débordement n'est fait à l'exécution. Il ne faut pas s'étonner de voir des résultats négatifs en cas de débordement ; par exemple l'évaluation de la constante `Integer.MAX_VALUE + 1` donnera la valeur de `Integer.MIN_VALUE`.

La conversion d'un type primitif vers son type enveloppant se fait à l'aide d'un constructeur :

```
Integer i = new Integer(3);
```

Inversement, des méthodes `byteValue()`, `shortValue()`, `intValue()` et `longValue()` permettent d'en extraire la valeur primitive :

```
int n = i.intValue();  // n = 3
```

D'autre part, une chaîne de caractères (provenant par exemple d'une lecture sur l'entrée standard) peut être convertie en entier de la façon suivante :

```
String s = "12";
int m = Integer.parseInt(s); // m = 12
```

L'API Java offre la classe `java.math.BigInteger` représentant les entiers de taille arbitraire.

A.2 Types flottants Les flottants sont des représentations en mémoire d'une partie des nombres rationnels ; il est évidemment impossible de représenter des nombres réels quelconques, pour des raisons de cardinalité. Seuls les rationnels dont la forme irréductible est $n/2^q$, peuvent avoir une représentation exacte ; les autres ont nécessairement une représentation approchée (par exemple, le nombre décimal $1/10$ a $0,00011\underline{0011}\ldots$ comme représentation en base 2, la partie soulignée étant répétée indéfiniment). Cette représentation fait l'objet de la norme IEEE 754, proposée par William KAHAN (Turing Award 1989), publiée en 1985 et adoptée par la plupart des fabricants d'ordinateurs. Cette norme distingue deux niveaux de précision : simple (sur 4 octets) et double (sur 8 octets), qui sont implémentés en Java par les types primitifs `float` et `double`. Il est souhaitable que le numéricien programmeur ait une idée de l'implémentation des flottants, sans qu'il doive nécessairement en connaître tous les détails.

Un nombre flottant est caractérisé par trois blocs de bits, qui déterminent respectivement son signe, son exposant et sa mantisse. Chacun de ses blocs est de taille fixe :

	simple précision	double précision
taille	32 bits	64 bits
signe	1 bit, b_{31}	1 bit, b_{63}
exposant	8 bits, $b_{30} \dots b_{23}$	11 bits, $b_{62} \dots b_{52}$
mantisse	23 bits, $b_{22} \dots b_0$	52 bits, $b_{51} \dots b_0$

La valeur d'un flottant (s, e, m) est $(-1)^s \times f \times 2^e$, où, dans le cas de la simple précision :

- $s = b_{31}$
- $e = b_{30}2^7 + b_{29}2^6 + \dots + 2^1 b_{24} + b_{23} - 127$
- $f = 1 + b_{22}2^{-1} + b_{21}2^{-2} + \dots + b_0 2^{-23}$

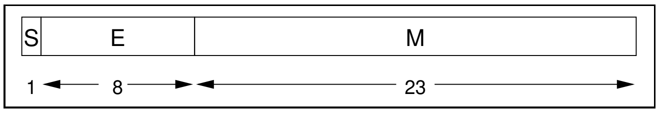

FIG. A.1 – *Flottants*

Dans l'exposant, soustraire 127 permettrait de représenter le plus petit exposant, -127, par les bits `0000 0000` et le plus grand exposant, 128, par les bits `1111 1111` ; 2^0 est par exemple représenté par `0111 1111`. En fait, `0000 0000` et `1111 1111` ont des significations spéciales, pour représenter 0, $+/-\infty$, et *NaN* (c'est-à-dire *Not a Number*) ; les exposants extrêmes des flottants normalisés sont donc -126 (environ 10^{-38}, par `0000 0001`) et 127 (environ 10^{38}, par `1111 1110`). La valeur de la mantisse étant toujours $1 + \dots$, zéro n'est pas représentable dans ce schéma ; par convention, le bit de signe suivi de 31 bits nuls représentent la valeur $(-1)^s \times 0$ (et non $(-1)^s \times 2^{-127}$)[1]. Si les bits d'exposant valent `1`, et les bits de mantisse valent `0`, la valeur est $(-1)^s\infty$, qui est obtenue dans le cas d'une division par 0. Enfin, si les bits d'exposant valent `1`, et les bits de mantisse ne sont pas tous nuls, la valeur est *NaN*, qui peut être obtenue comme le résultat d'opérations illicites, comme $0/0$, $\log(-1)$, ou $\infty - \infty$.

Bien que portant les mêmes noms (addition, multiplication), les opérations flottantes ne sont pas ces opérations mathématiques, et n'ont pas les mêmes propriétés. Par exemple, l'addition n'est pas associative : on peut vérifier

1. Il y aurait alors un « trou » entre 0 et le plus petit nombre normalisé, qui a comme bits d'exposant `0000 0001` et les bits de mantisse nuls. Pour combler ce trou, on convient que si les 8 bits d'exposant sont nuls, la valeur est un flottant *dénormalisé*, de valeur $(b_{22}2^{-1} + b_{21}2^{-2} + \dots + b_0 2^{-23})2^{-126}$.

que $(10000003.0 - 10000000.0) + 7.501 = 10.501$, tandis que $10000003.0 + (-10000000.0 + 7.501) = 11.0$. Autre exemple : la série $1.0 + 1.0 + 1.0 + \cdots$ converge !

Aux types primtifs `float` et `double` sont associées les classes enveloppantes `Float` et `Double`, qui définissent diverses constantes du type primitif correspondant :

- `MIN_VALUE`, la plus petite valeur > 0 ;
- `MAX_VALUE`, la plus grande valeur > 0 ;
- `POSITIVE_INFINITY`, `NEGATIVE_INFINITY` et `NaN`.

Les valeurs de type `double` peuvent être notées avec un signe, un point décimal et un exposant optionnel, par exemple `-2.3e+4`. Pour le type `float`, la constante doit être terminée par `F` (ou `f`) : `3.141592653` est un `double`, `6.02e23F` est un `float`. D'autre part, une chaîne de caractères (provenant par exemple d'une lecture sur l'entrée standard) peut être convertie en flottant de la façon suivante :

```
String s = "12.3";
double x = Double.parseDouble(s); // x = 12.3
```

A.3 Caractères Les caractères sont représentés par les valeurs du type `char`, codé sur deux octets. La plupart des alphabets des langues occidentales peuvent être représentés dans un jeu de caractères codés sur un octet, mais un même jeu de caractères ne peut évidemment coder tous les alphabets existants ; dans un souci d'internationalisation, rendu nécessaire à cause de l'Internet, Java utilise deux octets pour coder les caractères : c'est le codage *Unicode*, qui autorise $2^{16} = 65\,536$ caractères.

Les constantes de type caractère sont notées entre deux apostrophes (en anglais *single quote*) : `'A'`, `'Z'`, `'a'`, `';'`, `'4'`, etc. S'y ajoutent des caractères spéciaux comme `'\n'` pour le *retour à la ligne* et `'\t'` pour une tabulation.

Au type primitif `char` est associée une classe enveloppante `Character`, qui contient des fonctions très utiles :

- les fonctions de test `isDigit()`, `isLowerCase()`, `isUpperCase()`, `isSpaceChar()`, qui prennent un `char` en argument et retournent un `boolean`
- les fonctions de traduction `toUpperCase()` et `toLowerCase()`, qui prennent un `char` en argument et retournent un `char`

Les conversions entre `char` et `Char` se font respectivement à l'aide du constructeur et de la méthode suivante :

```
Character c = new Character('5');
char d = c.charValue();           // d = '5'
```

A.4 Opérateurs et expressions arithmétiques Les expressions arithmétiques sont formées à partir des opérateurs arithmétiques. Ceux-ci sont les opérateurs additifs `+`, `-`, les opérateurs multiplicatifs `*`, `/` et `%` (modulo) et les opérateurs unaires + et −.

L'analyse syntaxique d'une expression est déterminée par des règles de précédence , dont on ne peut s'affranchir (et donc les ignorer) que si les expressions sont complètement parenthésées. On préfère cependant écrire `2*x/3 - 5*y` plutôt que `((2*x)/3) - (5*y)`, conformément à l'usage courant ; mais sans règle de précédence, l'expression non parenthésée pourrait tout aussi bien être analysée comme `(2*x)/((3-5)*y)`, qui n'a sûrement pas la même valeur.

opérateurs	parenthésés
$+_{\mathsf{un}}$ $-_{\mathsf{un}}$	par la droite
`*` `/` `%`	par la gauche
`+` `-`	par la gauche

En termes de précédence, les opérateurs unaires (+, −) sont les plus forts, puis viennent les multiplicatifs (`*`, `/`, `%`) et les additifs (+, −). Par exemple, −1+3 est analysée comme `(-1) + 3`, et `2*x+3*y` comme `(2*x) + (3*y)`. Les opérateurs binaires sont *parenthésés par la gauche* (en anglais, *left associative*) ; par exemple, `x/2*3` est analysée comme `(x/2)*3`, et `x-y-z` comme `(x-y)-z` (s'ils étaient parenthésés par la droite, ces expressions seraient analysées respectivement comme `x/(2*3)` et comme `x-(y-z)`). Les opérateurs unnaires sont *parenthésés par la droite* : par exemple, `--x` est analysée comme `-(-x)` et non comme `(-)x`.

L'évaluation d'une expression arithmétique additive ou multiplicative comporte une phase de conversion :

– si l'un des deux opérandes est de type `double`, l'autre est converti en `double;`
– sinon, si l'un des opérandes est de type `float`, l'autre est converti en `float;`
– sinon, si l'un des opérandes est de type `long`, l'autre est converti en `long;`
– sinon (les types des opérandes sont `byte`, `short` ou `char`), les deux opérandes sont promus en `int`.

L'évaluation des opérations unaires + et − comporte aussi une promotion de leur opérande en `int` si son type est `byte`, `short` ou `char`. Par conséquent, le type d'une expression arithmétique comportant l'une de ces opérations n'est jamais `byte`, `short` ou `char`.

Il arrive souvent que l'on range le résultat d'une opération binaire dans le premier de ses opérandes : `a` = `a` *op* `b`. Il est alors possible d'avoir recours à la notation abrégée: `a` *op*= `b`. On écrira ainsi : `x+=y` au lieu de `x=x+y` et `x/=2` à la place de `x=x/2`. Outre la simplification d'écriture, l'utilisation de cette syntaxe peut avoir des effets importants sur le comportement et la rapidité du programme :

- `a[i][j]+=b` au lieu de `a[i][j]=a[i][j]+b` ne calcule qu'une fois l'emplacement mémoire de `a[i][j]`.
- `a[f(x,y,z)]*=b` au lieu de `a[f(x,y,z)]=a[f(x,y,z)]*b` n'invoque la fonction `f` qu'une seule fois.

A.5 *Opérations bit à bit* Pour manipuler directement l'écriture binaire d'un entier, on utilise les opérateurs suivants :

- `&` effectue un *et* sur chacun des bits de ses deux opérandes ; par exemple, `6&3` = 0110_2 `&` 0011_2 =0010_2 = `2` ;
- `|` effectue un *ou* sur chacun des bits de ses deux opérandes ; par exemple, `6|3` = 0110_2 `|` 0011_2 = 0111_2 = `7` ;
- `^` effectue un *ou exclusif* sur chacun des bits de ses deux opérandes ; par exemple, `6^3` = 0110_2 `^` 0011_2 = 0101_2 = `5` ;
- `>>n` effectue un *décalage à droite* de `n` crans de tous les bits (ceux le plus à droite tombent, le bit le plus à gauche est reproduit sur les n bits de gauche) ; i>>$n = \lfloor i/2^n \rfloor$; par exemple, `14>>2` = 1110_2 `>>2` = 11_2 = 3, tandis que `-15>>2` = $1..10001_2$ `>>2` = $1..100_2$ = -4 ;
- `>>>n` effectue un *décalage à droite non signé* de `n` crans de tous les bits (ceux le plus à droite tombent, des zéros rentrent par la gauche) ; si $i \geq 0$, i>>$n = i$>>n ; par exemple, `-15>>>2` = $1..10001_2$ `>>>2` = $001..100_2$ = 1073741820 (calcul sur 32 bits);
- `<<n` effectue un *décalage à gauche* de `n` crans de tous les bits (ceux le plus à gauche tombent, des zéros rentrent par la droite) ; par exemple, `3<<2` = 11_2 `<<2` = 1100_2 = `12` ;
- `~` effectue un *complément* de tous les bits (les zéros deviennent des 1 et réciproquement) ; par exemple, `~5` = `~`$0..0101_2$ = $1...1010_2$ = `-6` ; de façon générale, $-n = ($~$n) + 1$.

Ces opérations sont utilisées pour travailler sur les bits de la façon suivante :

- Lever le n^{eme} bit : `i |= (1<<n);`
- Baisser le n^{eme} bit : `i &= ~(1<<n);`
- Complémenter le n^{eme} bit : `i ^= (1<<n);`
- Tester si le n^{eme} bit est levé : `if (i&(1<<n)!=0) ....`

et aussi dans les cas fréquents suivants où elles accélèrent le programme :

- Reste de la division par une puissance de 2, `i&`$(2^n - 1) = i \bmod 2^n$;
- Quotient de la division par une puissance de 2, i`>>`$n = i/2^n$, si $i \geq 0$;
- Multiplication par une puissance de 2, i`<<`$n = i2^n$.

A.6 Booléens et expressions logiques Java, comme beaucoup d'autres langages, mais contrairement à C, dispose d'un type booléen, `boolean`, dont les valeurs sont `true` et `false`. Ce type n'est un sous-type ou sur-type d'aucun autre type primitif : on ne peut pas convertir un entier en booléen ni un booléen en entier.

Les expressions logiques sont formées à partir des expressions relationnelles à l'aide des opérateurs logiques.

Les *opérateurs relationnels* sont : <, <=, >, >=, ==, `!=`. Ces opérateurs retournent un booléen : la valeur de `1 == 2` est `false` (faux), celle de `2 == 2` est `true` (vrai).

Les opérateurs logiques sont la négation `!`, le « et » `&&` et le « ou » `||`. Les deux derniers ont la particularité d'être *séquentiels*, c'est-à-dire de donner lieu à une évaluation de gauche à droite :

- A `&&` B est faux si A est faux, et vrai si A et B sont vrais : B n'est pas évalué si A est faux
- A `||` B est vrai si A est vrai, et faux si A et B sont faux : B n'est pas évalué si A est vrai

Ce comportement des opérateurs `&&` et `||` est différent de celui des `and` et `or` de Pascal qui évaluent toujours leurs deux arguments. Il permet d'écrire des tests de la forme :

```
if (x != 0 && 1/x < epsilon) { ... }
if (i > N || t[i] > A) { ... }          // t de taille N
```

Une autre expression dont l'évaluation est séquentielle est l'*expression conditionnelle*, présente également en CAML, en C et en C++, mais pas en Pascal ou en Fortran : l'évaluation de A `?` B `:` C commence par évaluer A ; si sa valeur est vraie, alors B est évalué, sinon C est évalué. Par exemple, l'expression

```
x >= 0 ? x : -x
```

a pour valeur la valeur absolue de la valeur de `x`.

Voici un extrait du tableau des précédences pour les opérateurs arithmétiques, logiques, conditionnels :

opérateurs	parenthésés
`!` $+_{un}$ $-_{un}$	par la droite
`*` `/` `%`	par la gauche
`+` `-`	par la gauche
`<` `<=` `>` `>=`	par la gauche
`==` `!=`	par la gauche
`&&`	par la gauche
`||`	par la gauche
`? :`	par la droite

On retiendra l'ordre : opérateurs arithmétiques, opérateurs de comparaison, opérateurs logiques, qui permet d'écrire sans parenthèse l'expression logique

```
x+y>z || x<0 && y<0
```

qui est analysée en :

```
((x+y)>z) || ((x<0) && (y<0))
```

L'opérateur conditionnel ayant la plus faible précédence, il n'est pas nécessaire de placer ses trois sous-expressions entre parenthèses. Il suffira d'écrire

```
n<=1 ? 1 : fib(n-1) + fib(n-2)
```

au lieu de

```
(n<=1) ? 1 : (fib(n-1) + fib(n-2))
```

Signalons qu'au type primitif `boolean` est associée une classe enveloppante `Boolean`.

A.7 Instructions Enfin, les expressions servent à construire des instructions (en anglais *statement*), autre notion syntaxique importante dont le rôle est d'opérer sur la mémoire et de contrôler l'exécution du programme. Celles-ci ne peuvent figurer que dans le corps d'une méthode (ou dans un bloc statique). Une *instruction* est soit une instruction simple, soit une instruction composée.

Toute expression suivie d'un « `;` » est une instruction simple. Notamment :

– une affectation suivie d'un « `;` »
– un « `return` » suivi d'une expression et d'un « `;` »

– une invocation de méthode (généralement dont le type de retour est `void`) suivi d'un « `;` »

Les instructions d'*échappement* `break`, hors d'un `switch` ou d'une itération, et `continue`, hors d'une itération sont aussi des instructions simples.

Les instructions composées sont :

– les instructions conditionnelles : `if`, `switch`
– les instructions d'itération (`for`, `while`, `do`)
– les blocs d'instructions

Un *bloc* d'instructions est formé d'une suite de déclarations et d'instructions (simples ou composées), encadrée par « `{` » et « `}` ». Par exemple, le bloc suivant est formé de la déclaration de la variable locale `t` et de deux affectations :

```
{
  int t = x%y;
  x = y;
  y = t;
}
```

Notons que le corps d'une méthode est un bloc, qu'un bloc peut être vide, on l'écrit alors simplement `{}`.

A.8 Portée lexicale Les programmes sont peuplés de noms servant à désigner diverses entités : paquets, types, membres et variables. La *portée* d'une déclaration d'un nom est la région du programme dans laquelle ce nom est connu. Elle est déterminée par les règles suivantes :

Cette dernière clause concerne le *masquage* d'une déclaration par une autre : une déclaration d'un nom masque toute autre déclaration de ce nom qui lui est externe. Par exemple, dans

```
int n = 3;                    // n est un champ

int f(int x) {                // x paramètre
  int n=2;                    // n est une variable locale
  return x+n ;
}
```

la déclaration locale de `n` masque la déclaration du champ `n` : la valeur de `f(0)` est 2. Le masquage d'un nom est à éviter.

On voit donc qu'un même nom peut apparaître à plusieurs endroits dans un programme : on parle des *occurrences* de ce nom. Il y a des occurrences d'utilisation et des occurrences de déclaration. Par exemple, dans

```
int n = 4;
```

l'occurrence de n est une déclaration, tandis que dans

```
return x+n;
```

l'occurrence de n est une utilisation. Quand on voit une occurrence d'utilisation d'un nom, il faut pouvoir remonter à sa déclaration.

A.9 Instruction conditionnelle **`if`** Cette instruction a deux formes. La plus simple a une condition et une branche ; la condition doit figurer entre des parenthèses, et son type doit être `boolean` ; la branche est une instruction, simple ou composée, généralement un bloc d'instructions (donc placé entre les accolades { et }) ; elle est exécutée si la valeur de la condition est `true`.

```
if (delta < 0) {
  System.out.println("Pas de solution réelle");
}
```

Dans la forme à deux branches, la première branche est exécutée quand la valeur de la condition est `true` ; la seconde est exécutée quand cette valeur est `false`.

```
if (delta < 0) {
  System.out.println("Pas de solution réelle");
} else {
  System.out.println("Au moins une solution réelle");
}
```

Enfin, la branche `else` peut elle-même contenir un `if` (et ainsi de suite), ce qui conduit à l'imbrication suivante :

```
if (delta < 0) {
  System.out.println("Pas de solution réelle");
} else if (delta == 0) {
  double x1 = -b/(2*a);
  System.out.println("Une solution x1 = " + x1);
} else {
  double
    r = Math.sqrt(delta),
    x1 = (-b - r)/(2*a),
    x2 = (-b + r)/(2*a);
  System.out.println(
    "Deux solutions" +
    "x1 = "  + x1 + ", x2 = " + x2);
}
```

A.10 Instruction d'aiguillage **`switch`** Il est souvent utile d'exécuter une instruction en fonction de la valeur d'un entier ou d'un caractère ; Java offre l'instruction composée `switch` à cette fin. L'exemple suivant pourrait servir, avec quelques lignes supplémentaires, à compter les voyelles et les consonnes d'un texte :

```
switch(c) {
  case 'a':
  case 'e':
  case 'i':
    voyelles = voyelles + 1;
    break;
  case 'b':
  case 'c':
  case 'd':
    consonnes = consonnes + 1;
    break;
  default:
    autres = autres + 1;
}
```

L'instruction d'aiguillage `switch` est formée à partir d'une expression *exp* et d'un bloc d'instructions étiquetées (par un `case` ou par `default`). L'expression doit être de l'un des types `byte`, `short`, `int`, `long` ou `char`. La valeur de l'expression *exp* est successivement comparée à la valeur de chacune des expressions constantes figurant à droite de `case`. Dès qu'il y a un cas d'égalité, l'instruction suivant ce `case` est exécutée, ainsi que toutes les instructions suivantes, jusqu'à la fin du bloc ou jusqu'à un *échappement* : les instructions `break` ou `return`. Si aucune égalité n'est vérifiée, ce sont les instructions qui suivent le cas `default` qui sont exécutées.

Dans les usages courants du `switch`, chaque cas (non-vide) est terminé par un échappement :

```
switch(c) {
  case ...: ... ; break ;
  case ...: ... ; break ;
  default : ... ;
}
```

A.11 Itération **`for`** La structure d'itération `for` est spécialement adaptée au *parcours* de certaines structures de données, notamment un intervalle d'entiers, un tableau, une liste, etc. C'est l'un des piliers du style impératif.

Par exemple, pour calculer la somme des entiers entre 1 et N on écrira

```
int N = 10;
int somme = 0;
for (int i=1; i<=N; i++) {
  somme = somme + i;
}
```

Une boucle `for` est spécifiée par trois expressions d'en-tête qui sont, dans l'ordre, une initialisation, une condition d'exécution, et une itération. Le *corps* d'une boucle est un bloc quelconque ; le corps est exécuté un certain nombre de fois, ceci étant contrôlé par les expressions d'en-tête ; chaque exécution du corps est appelée une *itération* de la boucle.

Dans cet exemple, la *variable d'itération* `i` est déclarée comme une variable locale à la boucle et initialisée à 1 ; si la valeur de `i` est $\leq N$, le corps de la boucle est exécuté, puis `i` est incrémentée par `i++`, instruction équivalente à `i = i+1` ; si la valeur `i` n'est pas $\leq N$, la boucle est terminée ; il y a ici 10 itérations. La nature impérative de cette structure provient de l'utilisation de la variable `somme` qui est affectée à chaque itération d'une nouvelle valeur.

***A.12 Itération* `while`** La structure d'itération `while`, moins structurée que `for`, est plutôt adaptée à l'itération d'une transformation, tant qu'une certaine condition est vérifiée. C'est cette condition qui est spécifiée dans l'en-tête du `while`, tandis que le corps, qui est un bloc, décrit cette transformation.

Un des plus anciens algorithmes connus, dû à Euclide, permet de calculer le plus grand commun diviseur de deux entiers a et b, par itération. L'algorithme utilise deux variables `x` et `y`, initialisées aux valeurs a et b, puis opère itérativement sur ces variables. L'idée de la transformation est de maintenir l'*invariant* $\mathrm{pgcd}(x, y)$ et de s'arrêter quand $x = y$ auquel cas, $\mathrm{pgcd}(x, y) = x = y$. Euclide savait que si $x > y$, $\mathrm{pgcd}(x, y) = \mathrm{pgcd}(x - y, y)$ et que si $y > x$, $\mathrm{pgcd}(x, y) = \mathrm{pgcd}(x, y - x)$. D'où l'itération suivante, qu'on peut décrire ainsi en français : « tant que x et y sont différents, soustraire le plus petit du plus grand » :

```
while (x != y) {
  if (x > y) {
    x = x-y;
  } else {
    y = y-x;
  }
}
```

Contrairement au `for` de l'exemple précédent, il n'est pas évident que cette boucle *termine*[2], c'est-à-dire qu'il n'y ait qu'un nombre fini d'itérations. En effet si a ou b est ≤ 0, la boucle ne termine pas. Supposons que $a > 0$ et $b > 0$; après initialisation, on a donc $x > 0$ et $y > 0$; à chaque itération, si $x \neq y$, $\max(a, b)$ décroît strictement, mais reste strictement positif. Il y a donc au plus $\max(a, b) - 1$ itérations avant que la condition ne devienne fausse, c'est-à-dire que $x = y$. Ce n'est pas très efficace.

2. Cette utilisation intransitive du verbe « terminer » semble être propre aux informaticiens, alors que l'usage courant serait de dire cette boucle « se termine », ou « est terminée ».

Tout en conservant le même invariant $\mathrm{pgcd}(x, y)$, on peut choisir d'autres transformations de x et y de sorte que la boucle termine en moins d'itérations ; c'est l'algorithme d'Euclide, dans sa version moderne, où l'on remplace la soustraction par le calcul du reste par division entière (opérateur %), en utilisant l'identité $\mathrm{pgcd}(x, y) = \mathrm{pgcd}(x \bmod y, y)$; voici ce que donne directement ce remplacement :

```
while (x != y) {
  if (x > y) {
    x = x%y;
  } else {
    y = y%x;
  }
}
```

Comme $x \bmod y < y$, les tests peuvent être éliminés en faisant en sorte que la valeur de x soit supérieure à celle de y.

```
static int pgcd(int x, int y) {
  while (y > 0) {
     int t = x%y;
     x = y;
     y = t;
  }
  return x;
}
```

Rappelons que les arguments étant passés par valeur, les affectations aux paramètres x et y ne modifient que des variables locales et en aucune façon les arguments eux-mêmes.

Il existe également une itération `do ... while (...);`, plus rarement utilisée, qui exécute au moins une fois son corps et s'arrête quand la condition n'est plus vérifiée (on veillera à ne pas oublier le « ; » final).

A.13 Définitions récursives Une définition de méthode est *récursive* si son corps contient une expression d'invocation d'elle-même, dite invocation récursive :

```
static int fact(int n) {
  if (n == 0) {
    return 1;
  } else {
    return n*fact(n-1);
  }
}
```

Une invocation `fact(3)` dans le corps de `main` provoque la suite d'invocations :

main() → fact(3) → fact(2) → fact(1) → fact(0) et la suite de retours, qui retourne finalement 6 à `main()` :

$$\texttt{fact(0)} \xrightarrow{1} \texttt{fact(1)} \xrightarrow{1} \texttt{fact(2)} \xrightarrow{2} \texttt{fact(3)} \xrightarrow{6} \texttt{main()}$$

L'intérêt des définitions récursives est double. D'une part, elles permettent de transcrire de façon quasiment littérale certaines définitions mathématiques, souvent appelées récurrentes. D'autre part, on obtient facilement des définitions récursives à partir de la conception récursive d'un algorithme : pour résoudre un problème par un algorithme, on applique ce même algorithme à un ou plusieurs sous-problèmes. Cette méthode, appelée *diviser pour régner* permet d'écrire assez facilement des algorithmes parmi les plus intéressants, voire les plus efficaces.

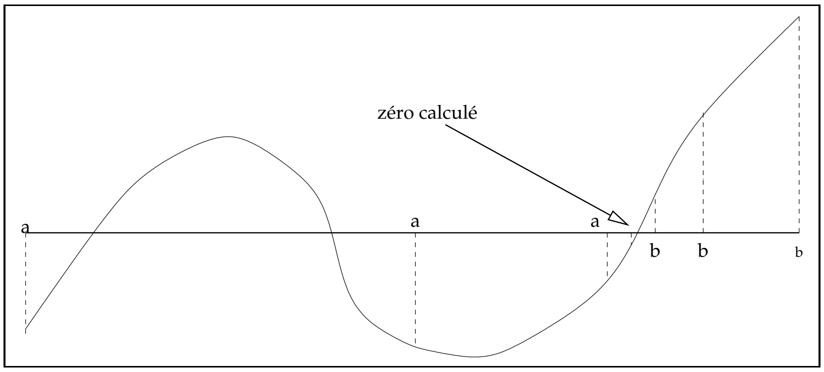

FIG. A.2 – *Recherche d'un zéro par dichotomie*

La *dichotomie* est un exemple simple de cette méthode. On cherche à calculer un zéro d'une fonction réelle continue f sur un intervalle $[a, b]$, prenant des valeurs de signes opposés aux extrémités. Le théorème des valeurs intermédiaires assure l'existence d'un zéro. L'idée de la dichotomie est de chercher un zéro sur $[a, (a+b)/2]$ ou bien sur $[(a+b)/2, b]$, selon le signe de $f((a+b)/2)$.

```
static final double EPS = 1e-5;

static double zero_dicho(double a, double b)
  // on suppose que f(a)*f(b) < 0
{
  double m = (a+b)/2;
  double fm = f(m);

  if (Math.abs(fm)<EPS) {
    return m;
  } else {
    if (f(a)*fm<0) {
      return zero_dicho(a,m);
```

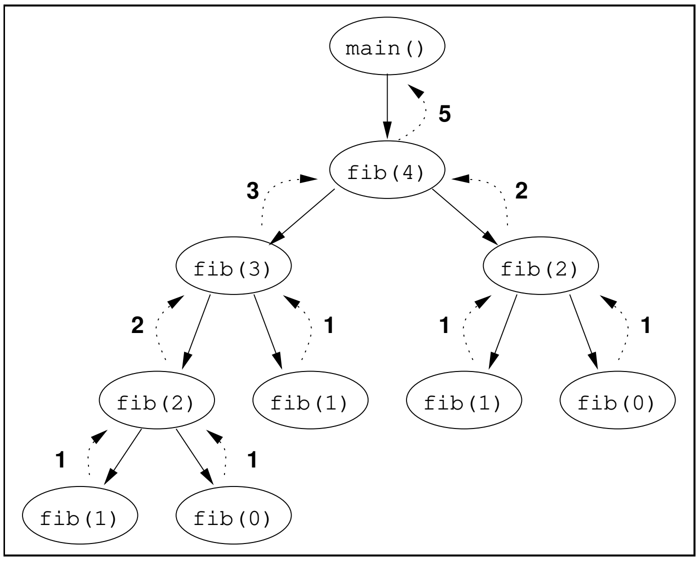

FIG. A.3 – *Arbre d'invocation de la suite de Fibonacci*

```
      } else {
        return zero_dicho(m,b);
      }
    }
  }
```

La condition d'arrêt est `Math.abs(fm)<EPS`, `EPS` étant une variable statique, et `Math.abs` étant la fonction déclarée dans la classe `Math` calculant la valeur absolue d'un `double`.

Quand le corps d'une fonction comporte plusieurs invocations récursives, leur évaluation est organisée sous la forme d'un arbre. L'exemple classique est celui de la définition récursive de la suite de Fibonacci :

```
static int fib(int n) {
  return n<=1 ? 1 : fib(n-1) + fib(n-2);
}
```

La figure A.3 représente l'arbre des invocations de `fib(4)`, et la figure A.4 représente les états successifs de la pile d'exécution, pour une invocation de `fib(4)` dans `main()` (pour alléger, les cadres d'invocation de `main()`, `fib(4)`, `fib(3)`, etc, y sont désignés par `m`, `4`, `3`, etc). On remarquera que certaines invocations sont exécutées plusieurs fois : `fib(2)` est exécutée deux fois, `fib(1)` trois fois, `fib(0)` deux fois.

Il n'y a aucune différence de principe, du point de vue de l'exécution, entre des méthodes définies récursivement ou pas. Le mécanisme d'allocation

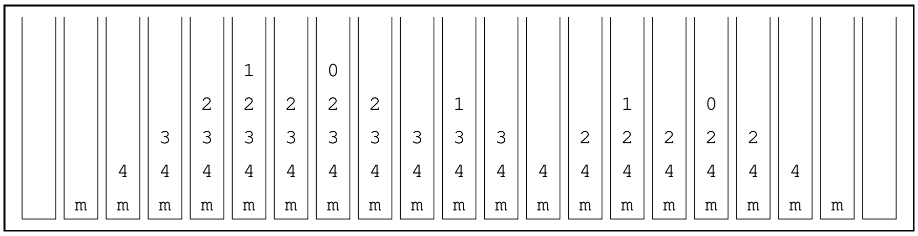

FIG. A.4 – *Pile d'exécution pour la suite de Fibonacci*

sur la pile s'applique de façon identique. Cependant, en l'absence de définitions récursives, chaque méthode a au plus un seul cadre d'invocation en cours. Il en résulte que la pile d'exécution est bornée ; elle peut être allouée initialement et ne croît pas en cours d'exécution. Le programme opère par transformations de cette zone mémoire fixe. On dit que le programme est *itératif*. S'il existe des définitions récursives, la pile d'exécution n'est pas bornée, croissant et décroissant pendant l'exécution. C'est le principal reproche adressé aux programmes utilisant des définitions récursives : ils consomment de l'espace mémoire, et bien qu'automatique, la gestion de la pile prend aussi du temps. Les programmeurs soucieux de l'efficacité de leurs programmes préfèrent donc les programmes itératifs, basés sur des structures d'itération. Ceux qui privilégient la facilité d'écriture et de compréhension des programmes n'hésitent pas à écrire des programmes récursifs.

Récursivité mutuelle Un ensemble de définitions est *mutuellement récursif* si la relation « f invoque g » admet un cycle : f_1 invoque … invoque f_n invoque f_1. L'exemple suivant est classique (et sans grand intérêt) :

```
  static boolean impair(int n) {
    if (n == 0) {
      return false;
    } else {
      return pair(n-1);
    }
  }

  static boolean pair(int n) {
    if (n == 0) {
      return true;
    } else {
      return impair(n-1);
    }
}
```

Les fonctions `pair()` et `impair()` s'invoquent mutuellement :

$$\texttt{pair}(3) \to \texttt{impair}(2) \to \texttt{pair}(1) \to \texttt{impair}(0)$$

La dernière invocation retourne `false` :

$$\texttt{impair}(0) \overset{\texttt{false}}{\to} \texttt{pair}(1) \overset{\texttt{false}}{\to} \texttt{impair}(2) \overset{\texttt{false}}{\to} \texttt{pair}(3) \overset{\texttt{false}}{\to} \texttt{main()}$$

Les définitions mutuellement récursives sont surtout utiles pour travailler sur des structures de données mutuellement récursives, par exemple en analyse syntaxique.

Récursivité terminale Une invocation récursive d'une fonction `f` est dite *terminale* si elle est de la forme `return f(...);` ; autrement dit, la valeur retournée est directement la valeur obtenue par l'invocation récursive, sans qu'il n'y ait d'opération sur cette valeur. Par exemple, l'invocation récursive de la factorielle

```
return n*f(n-1);
```

n'est pas terminale, puisqu'il y a multiplication par `n` avant de retourner. Par contre, l'invocation récursive dans

```
static int f(int n, int a) {
  if (n<=1) {
    return a;
  } else {
    return f(n-1,n*a);
  }
}
```

est terminale. Dans cette version, le paramètre `a` joue le rôle d'un accumulateur ; l'évaluation de `f(5,1)` conduit à la suite d'invocations

$$\texttt{f}(5,1) \to \texttt{f}(4,5) \to \texttt{f}(3,20) \to \texttt{f}(2,60) \to \texttt{f}(1,120) \to$$

dont la suite de retours

$$\texttt{f}(1,120) \overset{120}{\to} \texttt{f}(2,60) \overset{120}{\to} \texttt{f}(3,20) \overset{120}{\to} \texttt{f}(4,5) \overset{120}{\to} \texttt{f}(5,1) \overset{120}{\to}$$

est en fait une suite d'égalités

$$\texttt{f}(5,1) = \texttt{f}(4,5) = \texttt{f}(3,20) = \texttt{f}(2,60) = \texttt{f}(1,120) = 120$$

Une définition de méthode est *récursive terminale* quand toute invocation récursive est terminale. La plupart des langages fonctionnels, notamment CAML, exécutent un programme à récursivité terminale comme s'il était itératif, c'est-à-dire en espace constant. Certains compilateurs d'autres langages ont partiellement cette capacité. Sinon, il est facile de transformer une définition récursive terminale en itération pour optimiser l'exécution. Le programme dérécursivé est :

```
static int factIter(int n) {
  int a = 1;
  while (n>1) {
    a = n*a;
    n = n-1;
  }
  return a;
}
```

A.14 Un exemple : l'exponentiation Java ne disposant pas d'un opérateur d'exponentiation entière, il faut programmer cette opération (dans le cas des nombres flottants, utiliser `Math.exp()`). Le programme suivant calcule x^e, pour des entiers x et e avec $e \geq 0$, en accumulant (par multiplication) x dans y, ceci e fois, l'expression $y * x^e$ étant un invariant de boucle :

```
static int exp(int x, int e) {
  int y = 1;
  while (e != 0) {
    y = y*x;
    e = e-1;
  }
  return y;
}
```

Chaque itération effectue une transformation $(y, e) \mapsto (y', e')$; par exemple, le calcul de `exp(5,8)` effectue les transformations successives suivantes de (y, e) :

$$\begin{aligned} (1,8) \mapsto (5,7) \mapsto (25,6) \mapsto (125,5) \mapsto (625,4) \\ \mapsto (3125,3) \mapsto (15625,2) \mapsto (78125,1) \mapsto (390625,0) \end{aligned}$$

On vérifie l'invariant

$$y' x^{e'} = yxx^{e-1} = yx^e$$

On a donc l'égalité de valeur de cet invariant au début de la boucle (quand $y = 1$) et à la fin de la boucle (quand $e = 0$) :

$$x^{e_{\text{initial}}} = y_{\text{final}}$$

On notera que la version récursive équivalente à ce programme itératif n'est pas celle que l'on écrirait directement à partir de la définition par récurrence de la fonction puissance, mais est la suivante (la boucle du `while` modifiant les variables y et e, la fonction récursive doit prendre ces valeurs en argument) :

```
private static int exp_aux(int x, int y, int e) {
  if (e == 0) {
    return y;
  } else {
    return exp_aux(x, y*x, e-1);
  }
}

static int exp(int x, int e) {
  return exp_aux(x,1,e);
}
```

La fonction `exp_aux` est une fonction auxiliaire qui ne sera appelée que par `exp` ; c'est pourquoi elle est déclarée `private`.

Pour les trois programmes précédents, le nombre d'itérations ou d'appels récursifs est l'entier e. Il est possible d'accélérer significativement ce calcul en ramenant ce nombre de e à au plus $2\log_2 e$, grâce à la propriété suivante :

$$\begin{aligned} x^{2e+1} &= x * x^{2e} \\ x^{2e} &= (x^2)^e \end{aligned}$$

Par exemple, $x^{10} = (x^2)^5 = x^2(x^2)^4 = x^2((x^2)^2)^2$ en quatre multiplications au lieu de 9. On obtient ainsi la définition

```
private static int exp_fastrec_aux(int x, int y, int e) {
  if (e == 0) {
    return y;
  } else if (e % 2 == 1) {
    return exp_fastrec_aux(x, x*y, e-1);
  } else {
    return exp_fastrec_aux(x*x, y, e/2);
  }
}
```

La version itérative s'écrit facilement à partir de cette version récursive terminale, en remplaçant la liste des arguments des appels récursifs par des affectations appropriées :

```
static int exp_fastiter(int x, int e) {
  int y = 1;
  while (e!=0) {
    if (e%2 == 1) {
      y = x*y;
      e = e-1;
    } else {
      x = x*x;
      e = e/2;
```

```
    }
  }
  return y;
}
```

Le cas bénéficiant de la plus forte accélération est celui où l'exposant est une puissance de 2 ; voici la suite des transformations de (x, y, e) pour le calcul de 5^8 (en trois itérations au lieu de 8):

$$(5, 1, 8) \mapsto (25, 1, 4) \mapsto (625, 1, 2) \mapsto (390625, 1, 0)$$

La situation est moins favorable quand l'exposant n'est pas une puissance de 2 ; le calcul de 5^7 se fait en 5 itérations, soit moins de $2 \log_2 7$:

$$\begin{aligned} (5, 1, 7) \mapsto (5, 5, 6) \mapsto (25, 5, 3) \\ \mapsto (25, 125, 2) \mapsto (625, 125, 1) \mapsto (625, 78125, 0) \end{aligned}$$

Cet algorithme est décrit dans le *Chandah Sutra* d'Acharya PINGALA (écrit avant 200 ans avant J.C.).

B Grammaire LALR(1)

B.1 The Syntactic Grammar

Goal:
  CompilationUnit

B.2 Lexical Structure

Literal:
  IntegerLiteral
  FloatingPointLiteral
  BooleanLiteral
  CharacterLiteral
  StringLiteral
  NullLiteral

B.3 Types, Values, and Variables

Type:
  PrimitiveType
  ReferenceType
PrimitiveType:
  NumericType
  `boolean`

NumericType:
  IntegralType
  FloatingPointType
IntegralType: one of
  `byte short int long char`
FloatingPointType: one of
  `float double`
ReferenceType:
  ClassOrInterfaceType
  ArrayType
ClassOrInterfaceType:
  Name
ClassType:
  ClassOrInterfaceType
InterfaceType:
  ClassOrInterfaceType
ArrayType:
  PrimitiveType `[ ]`
  Name `[ ]`
  ArrayType `[ ]`

B.4 Names

Name:
  SimpleName
  QualifiedName
SimpleName:
  Identifier
QualifiedName:
  Name `.` *Identifier*

B.5 Packages

CompilationUnit:
  PackageDeclaration$_{\text{opt}}$ *ImportDeclarations*$_{\text{opt}}$ *TypeDeclarations*$_{\text{opt}}$
ImportDeclarations:
  ImportDeclaration
  ImportDeclarations ImportDeclaration
TypeDeclarations:
  TypeDeclaration
  TypeDeclarations TypeDeclaration
PackageDeclaration:
  `package` *Name* `;`
ImportDeclaration:
  SingleTypeImportDeclaration
  TypeImportOnDemandDeclaration
SingleTypeImportDeclaration:
  `import` *Name* `;`
TypeImportOnDemandDeclaration:
  `import` *Name* `. * ;`
TypeDeclaration:
  ClassDeclaration
  InterfaceDeclaration
  `;`

B.6 Modificateurs

Modifiers:
  Modifier
  Modifiers Modifier
Modifier: one of
  `public protected private`
  `static`
  `abstract final native synchronized transient volatile`

B.7 Class Declaration

ClassDeclaration:
  Modifiers$_{\text{opt}}$ `class` *Identifier Super*$_{\text{opt}}$ *Interfaces*$_{\text{opt}}$ *ClassBody*
Super:
  `extends` *ClassType*
Interfaces:
  `implements` *InterfaceTypeList*
InterfaceTypeList:
  InterfaceType
  InterfaceTypeList `,` *InterfaceType*
ClassBody:
  `{` *ClassBodyDeclarations*$_{\text{opt}}$ `}`
ClassBodyDeclarations:
  ClassBodyDeclaration
  ClassBodyDeclarations ClassBodyDeclaration
ClassBodyDeclaration:
  ClassMemberDeclaration
  StaticInitializer
  ConstructorDeclaration
ClassMemberDeclaration:
  FieldDeclaration
  MethodDeclaration

B.8 Field Declarations

FieldDeclaration:
  Modifiers$_{\text{opt}}$ *Type VariableDeclarators* `;`
VariableDeclarators:
  VariableDeclarator
  VariableDeclarators `,` *VariableDeclarator*
VariableDeclarator:
  VariableDeclaratorId
  VariableDeclaratorId = *VariableInitializer*
VariableDeclaratorId:
  Identifier
  VariableDeclaratorId `[ ]`
VariableInitializer:
  Expression
  ArrayInitializer

B.9 Method Declarations

MethodDeclaration:

MethodHeader MethodBody
MethodHeader:
Modifiers$_{\text{opt}}$ *Type MethodDeclarator Throws*$_{\text{opt}}$
Modifiers$_{\text{opt}}$ `void` *MethodDeclarator Throws*$_{\text{opt}}$
MethodDeclarator:
Identifier `(` *FormalParameterList*$_{\text{opt}}$ `)`
MethodDeclarator `[ ]`
FormalParameterList:
FormalParameter
FormalParameterList `,` *FormalParameter*
FormalParameter:
Type VariableDeclaratorId
Throws:
`throws` *ClassTypeList*
ClassTypeList:
ClassType
ClassTypeList `,` *ClassType*
MethodBody:
Block
`;`

B.10 Static Initializers

StaticInitializer:
static Block

B.11 Constructor Declarations

ConstructorDeclaration:
Modifiers$_{\text{opt}}$ *ConstructorDeclarator Throws*$_{\text{opt}}$ *ConstructorBody*
ConstructorDeclarator:
SimpleName `(` *FormalParameterList*$_{\text{opt}}$ `)`
ConstructorBody:
`{` *ExplicitConstructorInvocation*$_{\text{opt}}$ *BlockStatements*$_{\text{opt}}$ `}`
ExplicitConstructorInvocation:
`this (` *ArgumentList*$_{\text{opt}}$ `) ;`
`super (` *ArgumentList*$_{\text{opt}}$ `) ;`

B.12 Interface Declarations

InterfaceDeclaration:
Modifiers$_{\text{opt}}$ `interface` *Identifier ExtendsInterfaces*$_{\text{opt}}$ *InterfaceBody*
ExtendsInterfaces:
`extends` *InterfaceType*
ExtendsInterfaces `,` *InterfaceType*
InterfaceBody:
`{` *InterfaceMemberDeclarations*$_{\text{opt}}$ `}`
InterfaceMemberDeclarations:
InterfaceMemberDeclaration
InterfaceMemberDeclarations InterfaceMemberDeclaration
InterfaceMemberDeclaration:
ConstantDeclaration
AbstractMethodDeclaration

ConstantDeclaration:
  FieldDeclaration
AbstractMethodDeclaration:
  MethodHeader **;**

B.13 Arrays

ArrayInitializer:
  { *VariableInitializers*$_{\mathsf{opt}}$ **,**$_{\mathsf{opt}}$ }
VariableInitializers:
  VariableInitializer
  VariableInitializers **,** *VariableInitializer*

B.14 Blocks and Statements

Block:
  { *BlockStatements*$_{\mathsf{opt}}$ }
BlockStatements:
  BlockStatement
  BlockStatements BlockStatement
BlockStatement:
  LocalVariableDeclarationStatement
  Statement
LocalVariableDeclarationStatement:
  LocalVariableDeclaration **;**
LocalVariableDeclaration:
  Type VariableDeclarators
Statement:
  StatementWithoutTrailingSubstatement
  LabeledStatement
  IfThenStatement
  IfThenElseStatement
  WhileStatement
  ForStatement
StatementNoShortIf:
  StatementWithoutTrailingSubstatement
  LabeledStatementNoShortIf
  IfThenElseStatementNoShortIf
  WhileStatementNoShortIf
  ForStatementNoShortIf
StatementWithoutTrailingSubstatement:
  Block
  EmptyStatement
  ExpressionStatement
  SwitchStatement
  DoStatement
  BreakStatement
  ContinueStatement
  ReturnStatement
  SynchronizedStatement
  ThrowStatement
  TryStatement
EmptyStatement:
  ;

LabeledStatement:
  Identifier `:` *Statement*
LabeledStatementNoShortIf:
  Identifier `:` *StatementNoShortIf*
ExpressionStatement:
  StatementExpression `;`
StatementExpression:
  Assignment
  PreIncrementExpression
  PreDecrementExpression
  PostIncrementExpression
  PostDecrementExpression
  MethodInvocation
  ClassInstanceCreationExpression
IfThenStatement:
  `if (` *Expression* `)` *Statement*
IfThenElseStatement:
  `if (` *Expression* `)` *StatementNoShortIf* `else` *Statement*
IfThenElseStatementNoShortIf:
  `if (` *Expression* `)` *StatementNoShortIf* `else` *StatementNoShortIf*
SwitchStatement:
  `switch (` *Expression* `)` *SwitchBlock*
SwitchBlock:
  `{` *SwitchBlockStatementGroups*$_{\mathsf{opt}}$ *SwitchLabels*$_{\mathsf{opt}}$ `}`
SwitchBlockStatementGroups:
  SwitchBlockStatementGroup
  SwitchBlockStatementGroups SwitchBlockStatementGroup
SwitchBlockStatementGroup:
  SwitchLabels BlockStatements
SwitchLabels:
  SwitchLabel
  SwitchLabels SwitchLabel
SwitchLabel:
  `case` *ConstantExpression* `:`
  `default :`
WhileStatement:
  `while (` *Expression* `)` *Statement*
WhileStatementNoShortIf:
  `while (` *Expression* `)` *StatementNoShortIf*
DoStatement:
  `do` *Statement* `while (` *Expression* `) ;`
ForStatement:
  `for (` *ForInit*$_{\mathsf{opt}}$ `;` *Expression*$_{\mathsf{opt}}$ `;` *ForUpdate*$_{\mathsf{opt}}$ `)`
  Statement
ForStatementNoShortIf:
  `for (` *ForInit*$_{\mathsf{opt}}$ `;` *Expression*$_{\mathsf{opt}}$ `;` *ForUpdate*$_{\mathsf{opt}}$ `)`
  StatementNoShortIf
ForInit:
  StatementExpressionList
  LocalVariableDeclaration
ForUpdate:
  StatementExpressionList
StatementExpressionList:
  StatementExpression

StatementExpressionList `,` *StatementExpression*
BreakStatement:
`break` *Identifier*$_{\text{opt}}$ `;`
ContinueStatement:
`continue` *Identifier*$_{\text{opt}}$ `;`
ReturnStatement:
`return` *Expression*$_{\text{opt}}$ `;`
ThrowStatement:
`throw` *Expression* `;`
SynchronizedStatement:
`synchronized (` *Expression* `)` *Block*
TryStatement:
`try` *Block Catches*
`try` *Block Catches*$_{\text{opt}}$ *Finally*
Catches:
CatchClause
Catches CatchClause
CatchClause:
`catch (` *FormalParameter* `)` *Block*
Finally:
`finally` *Block*

B.15 Expressions

Primary:
PrimaryNoNewArray
ArrayCreationExpression
PrimaryNoNewArray:
Literal
`this`
`(` *Expression* `)`
ClassInstanceCreationExpression
FieldAccess
MethodInvocation
ArrayAccess
ClassInstanceCreationExpression:
`new` *ClassType* `(` *ArgumentList*$_{\text{opt}}$ `)`
ArgumentList:
Expression
ArgumentList `,` *Expression*
ArrayCreationExpression:
`new` *PrimitiveType DimExprs Dims*$_{\text{opt}}$
`new` *ClassOrInterfaceType DimExprs Dims*$_{\text{opt}}$
DimExprs:
DimExpr
DimExprs DimExpr
DimExpr:
`[` *Expression* `]`
Dims:
`[ ]`
Dims `[ ]`
FieldAccess:
Primary `.` *Identifier*
`super .` *Identifier*

MethodInvocation:
Name (*ArgumentList*$_{\mathsf{opt}}$)
Primary . *Identifier* (*ArgumentList*$_{\mathsf{opt}}$)
`super` . *Identifier* (*ArgumentList*$_{\mathsf{opt}}$)
ArrayAccess:
Name [*Expression*]
PrimaryNoNewArray [*Expression*]
PostfixExpression:
Primary
Name
PostIncrementExpression
PostDecrementExpression
PostIncrementExpression:
PostfixExpression ++
PostDecrementExpression:
PostfixExpression −−
UnaryExpression:
PreIncrementExpression
PreDecrementExpression
+ *UnaryExpression*
− *UnaryExpression*
UnaryExpressionNotPlusMinus
PreIncrementExpression:
++ *UnaryExpression*
PreDecrementExpression:
−− *UnaryExpression*
UnaryExpressionNotPlusMinus:
PostfixExpression
~ *UnaryExpression*
! *UnaryExpression*
CastExpression
CastExpression:
(*PrimitiveType Dims*$_{\mathsf{opt}}$) *UnaryExpression*
(*Expression*) *UnaryExpressionNotPlusMinus*
(*Name Dims*) *UnaryExpressionNotPlusMinus*
MultiplicativeExpression:
UnaryExpression
MultiplicativeExpression * *UnaryExpression*
MultiplicativeExpression / *UnaryExpression*
MultiplicativeExpression % *UnaryExpression*
AdditiveExpression:
MultiplicativeExpression
AdditiveExpression + *MultiplicativeExpression*
AdditiveExpression − *MultiplicativeExpression*
ShiftExpression:
AdditiveExpression
ShiftExpression << *AdditiveExpression*
ShiftExpression >> *AdditiveExpression*
ShiftExpression >>> *AdditiveExpression*
RelationalExpression:
ShiftExpression
RelationalExpression < *ShiftExpression*
RelationalExpression > *ShiftExpression*
RelationalExpression <= *ShiftExpression*

RelationalExpression >= *ShiftExpression*
RelationalExpression `instanceof` *ReferenceType*
EqualityExpression:
RelationalExpression
EqualityExpression == *RelationalExpression*
EqualityExpression != *RelationalExpression*
AndExpression:
EqualityExpression
AndExpression & *EqualityExpression*
ExclusiveOrExpression:
AndExpression
ExclusiveOrExpression ^ *AndExpression*
InclusiveOrExpression:
ExclusiveOrExpression
InclusiveOrExpression | *ExclusiveOrExpression*
ConditionalAndExpression:
InclusiveOrExpression
ConditionalAndExpression && *InclusiveOrExpression*
ConditionalOrExpression:
ConditionalAndExpression
ConditionalOrExpression || *ConditionalAndExpression*
ConditionalExpression:
ConditionalOrExpression
ConditionalOrExpression ? *Expression* : *ConditionalExpression*
AssignmentExpression:
ConditionalExpression
Assignment
Assignment:
LeftHandSide AssignmentOperator AssignmentExpression
LeftHandSide:
Name
FieldAccess
ArrayAccess
AssignmentOperator: one of
`= *= /= %= += -= <<= >>= >>>= &= ^= |=`
Expression:
AssignmentExpression
ConstantExpression:
Expression

Table des figures

Index