Apprendre Java en 154 minutes

Tutorial Java 1.6 - HESB-TI

Stéphanie Calderara
José R. Beuret
Quentin Cosendey

Mai 2008

Table des matières

Introduction

Tour d'horizon

La programmation Java est un langage qui a été développé par l'entreprise Sun Micorsystem dans les années 80. Plusieurs versions apparaitront dans les années suivantes. Chacune de ces versions apportera à Java de nouvelles classes et de ce fait, le rendra l'un des langages les plus modernes.

Vous verrez par la suite qu'avec Java, vous pourrez programmer n'importe quelle application. Java peut aussi être utilisé pour programmer des applets pour Internet, mais nous n'en parlerons pas dans ce manuel.

Apprendre Java en une après-midi

Le but de ce manuel consiste à vous enseigner la programmation Java en une après-midi. Ce document s'adresse donc à tous ceux qui désirent avoir une première approche en programmation. Vous verrez qu'après avoir lu ce manuel, vous serez capable de programmer de petites choses déjà très intéressantes. vous serez également en mesure de mieux comprendre les autres cours en Java.

Il faut bien comprendre qu'il est impossible de tout savoir en si peu de temps. Il vous faudra essayer dès que vous le pourrez, de programmer. De préférence commencez par de petits projets sans grande importance, car il vous faut de l'expérience et comme vous le savez, l'experience se gagne en essayant. Ne vous découragez surtout pas.

Comment s'équiper

Il est aussi intéressant de mentionner que Java est un langage portable, c'est-à-dire que vous pouvez programmer sur un système Linux, Windows ou encore Mac OS. Lorsque vous programmez en Java, votre code compilé deviendra du byte code. Ce langage n'est compris que par un environement Java (JRE). Mais cette partie est contenu dans la plupart des IDE Java.

Il vous faut donc un IDE. Il en existe plusieurs mais les plus connus sont de loin NetBeans et Eclipse. Vous êtes tout-à-fait libre de choisir ce-

lui qui vous convient le mieux. Nous vous conseillons tout de même d'utiliser NetBeans. En effet, les créateurs de NetBeans sont également ceux qui ont crée Java. De plus Sun Microsystems fourni un effort considérable dans le développement des outils utilisant les technologies Java. Il vous faut également JDK 6. C'est un kit de développement proposé par Sun Microsystems. Dans ce pack, vous trouverez entre autres Java Runtime Environment (JRE).

Tous les logiciels qui vous sont nécessaires sont accessibles gratuitement par téléchargement à cette adresse

http://java.sun.com/javase/downloads/index.jsp

Comment bien comprendre ce manuel

Nous vous conseillons vivement de lire les chapitres de ce manuel dans l'ordre donné. Bien entendu, si vous avez déjà de bonnes connaissances en Java, vous pouvez sauter les chapitres déjà acquis.

Si vous êtes un débutant, prenez le temps de bien lire les chapitres dans l'ordre. N'hésitez pas à revenir sur ce que vous n'avez pas compris. Ce manuel sert d'introduction au monde de Java, vous ne pouvez pas forcément comprendre tout d'un coup.

Afin de mieux s'y retrouver, les mots-clés (mots réservé par le langage Java) seront mis en italique. Les extraits de l'API de Java seront dans une boîte jaune. Quant aux exemples de codes, vous les retrouverez dans des boîtes bleues. A la fin de ce manuel, vous trouverez toutes les références utilisées ainsi que des adresses de bons tutoriels pour approfondir vos connaissances en Java.

Chapitre 1

Objet, classe et modificateur

1.1 Le premier pas

Un proverbe chinois raconte qu'un voyage de 5'000 lieux commence toujours par un pas. Aussi nous allons introduire les notions essentielles pour une bonne compréhension de Java en termes simples et didacticiels.

Néanmoins avant de commencer à apprendre les diverses notions, nous allons analyser le programme suivant en observant les différents mots utilisés. Leur significations seront introduites progressivement tout au long de ce chapitre.

```
public class HelloWorld {
  public static void main(String[] args) {
    System.out.println("This is a friendly HelloWorld!");
  }
}
```

Ce premier programme nous produira la sortie suivante :

```
This is a friendly HelloWorld!
```

Nous allons à présent apprendre les différents concepts de la programmation objet.

1.2 Le concept de l'objet

L'objet en Java est une donnée structurée, c'est à dire qu'elle peut elle-même contenir des données plus petite. Un langage orienté objet ou à objet

consiste en la possibilité de décrire tout objet de la vie courante en formalisme informatique. Ainsi il est possible de construire une structure ou une donnée qui représentera l'objet de la vie courante associé.

La plus grande difficulté d'un langage à objet consiste à « réfléchir en terme d'objet », c'est-à-dire que tout se relie à un moins un objet. De bons exercices pour s'entrainer à cet philosophie consistent à choisir un objet dans votre entourage, puis de le décrire au niveau de sa structure ou son contenu, mais également au niveau de ses actions ou événements possibles. Par exemple, il s'agit de décrire une porte, une fenêtre, un chat ou encore un chien, en se posant les questions : « Comment est-il ? » et « Que fait-il ? ».

Si vous avez fait l'exercice avec une autre personne, et que celle-ci a décrit l'objet d'une autre manière, c'est qu'elle a construit un autre formalisme pour l'objet en question. Il se peut même que ni l'un, ni l'autre ait tord.

De plus, lorsqu'il s'agit de formaliser un objet de la vie réelle, il faut veiller que ce qu'on formalise, correspond effectivement à l'ensemble des objets du même genre que celui choisit. Si ce n'est pas le cas, nous avons créer un formalisme trop spécifique. A l'opposé, il est aussi parfois utile d'« oublier » certaines contraintes de la vie réelle pour avoir un modèle utilisable, sans quoi le modèle serait trop difficile à programmer, voire même impossible. Personne souhaiterait créer un objet représentant un chat qui contiendrait toutes ses cellules, lesquels auraient des mitochondries ou autres éléments biologiques. Afin de créer un tel objet, nous pourrions nous contenter des informations concernant son âge, la couleur de sa fourrure ou encore la couleur de ses yeux.

1.3 Le concept de la classe

Une classe est une sorte de descriptif qui concerne directement ou indirectement un type d'objet. C'est une sorte de regroupement de toutes les fonctionnalités disponibles pour un type d'objet. Il est souvent utilisé, dans la littérature, l'expression « la classe de ». En effet pour chaque type d'objet, il y a une seule classe. Et par classe, il n'y a qu'un seul type d'objet qui y est décrit, cela n'empêche pas qu'il peut y avoir des interactions avec d'autres types d'objet. Par exemple, un chat et une souris auront une interaction, néanmoins dans la classe Chat, seul le type Chat y sera décrit et dans la classe Souris, seul le type Souris y sera décrit.

1.4 La classe et l'objet

A présent que nous sommes introduits aux concepts d'objet et de classe, nous pouvons le décrire selon le langage Java. En Java, le nom de la classe est aussi le même nom du type d'objet qui y est décrit. De plus, la classe doit être placée dans un fichier comportant le nom et ayant comme extension de

fichier « .java ». En d'autres termes si l'on construit un type d'objet nommé Chat, nous allons le faire dans la classe Chat qui se trouve dans le fichier Chat.java sur votre ordinateur.

```
public class Chat {
  // ...
}
```

La description de la classe Chat se fait à l'aide de divers moyens comme par exemple l'attribut ou la méthode.

1.4.1 L'attribut ou variable

L'attribut, appelé également variable, consiste en une donnée nécessaire pour une description, elle nous permet de mémoriser une information, qui pourra être reprise par la suite. En d'autres termes, elle permet de contenir une information que nous avons jugée nécessaire dans la modélisation.

Dans notre exemple pour décrire un chat, nous souhaitons garder des informations telles que son âge, la couleur de sa fourrure ou encore la couleur de ses yeux, comme nous le montre l'exemple suivant :

```
public class Chat {
  public int age;
  public Color yeux;
  public Color fourrure;
}
```

Nous pouvons observer que int (type nombre) et Color (type Couleur) sont eux-mêmes des types de données.

1.4.2 La méthode

La méthode consiste en une action ou une réaction à un événement, elle utilise ce qu'on appelle des paramètres, et renvoie un résultat. Les paramètres sont des indications pour définir le comportement de la méthode. Le résultat renvoyé s'appellent également la valeur de retour. Chose particulière, il y a toujours un résultat renvoyé, mais ce dernier peut être « rien » avec le type *void*. En effet *void* est un type qui ne fait rien du tout.

Dans notre exemple, nous allons introduire une méthode où le chat peut manger une souris, ainsi qu'une autre méthode mentionnant s'il a assez mangé. Nous ne décrirons pas le contenu de ces deux méthodes.

```
public class Chat {
  public int age;
  public Color yeux;
  public Color fourrure;

  public void mange(Souris s) {
    // ...
  }

  public boolean assezMange() {
    // ...
  }
}
```

A noter que le type boolean représente une réponse oui ou non, ou plutôt vrai ou faux avec les valeurs true ou false.

1.4.3 Le constructeur

Nous avons jusqu'à présent décrit ce qu'est un chat, en d'autres termes ce qu'on juge intéressant de mémoriser lorsqu'on parle de chat. Nous avons également décrit ce qu'un chat peut faire. Il nous manque encore une notion essentielle dans la description d'un chat, qui est la manière dont le chat est mémorisé tout au début, en d'autres termes c'est « l'acte de naissance » de l'objet dans la mémoire de l'ordinateur.

Imaginons deux chats Cannelle et Wifi. Cannelle a une année, possède les yeux bruns, ainsi qu'une couleur de fourrure châtain. Tandis que Wifi a trois ans, possède les yeux bleus, ainsi qu'une couleur de fourrure noir. Partant du point de vue que nous allons utiliser notre description du chat, nous aimerions construire nos deux chats. Nous allons donc devoir développer un outil permettant de mémoriser l'état initial de notre objet. Ce mécanisme s'appelle le constructeur. Contrairement aux méthodes, il ne renvoie rien, même pas void. La raison étant qu'il « renvoie » ou enregistre dans la mémoire les différentes valeurs avec celles que nous lui donnons par défaut.

Nous allons donc élaborer un constructeur pour le type Chat dans notre exemple. Nous devons aussi remarquer que le nom utilisé pour le constructeur doit être identique au nom de la classe.

```
public class Chat {
  public int age;
  public Color yeux;
  public Color fourrure;

  public Chat(int un_age, Color des_yeux,
```

```
    Color une_fourrure) {

    age = un_age;
    yeux = des_yeux;
    fourrure = une_fourrure;
  }

  public void mange(Souris s) {
    // ...
  }

  public boolean assezMange() {
    // ...
  }
}
```

1.5 Le point-virgule

Vous l'avez sans doute déjà remarqué ; à la fin de chaque ligne il y a un point-virgule (;). Le point-virgule indique une fin d'instruction. Cela permet à l'ordinateur de reconnaître une fin d'instruction est de la traiter de manière indépendante avec la suivante.

1.6 Le point

Le point est l'opérateur permettant d'accéder à un élément à l'intérieur d'un autre, par exemple un attribut ou une méthode d'un objet.

1.7 L'organisation des classes

Nous sommes en train de construire progressivement une classe Chat, nous savons également que le fichier Chat.java existera. En Java, il existe le concept de paquet, on parle aussi de librairie ou de bibliothèque. C'est un ensemble de classe que l'on rassemble sous un même nom appelé *package*.

Dans notre exemple, nous avons la classe Chat, mais nous pouvons également imaginer avoir les classes Chien et Souris. Nous pourrions donc créer un package appelé animaux.

1.7.1 Mot clé package

Afin de faire appartenir une classe à un package, il faut placer en première ligne, en début de fichier, *package* suivit du nom du package et placer le fichier dans le répertoire correspondant au nom du package.

Ainsi si nous souhaitons faire ajouter la classe Chat au paquet animaux, nous devrons enregistrer le document Chat.java dans un répertoire animaux.

```
package animaux;

public class Chat {
  // ...
}
```

Si au contraire nous souhaitons ajouter la classe Chat au paquet animaux.domestique, nous devrons enregistrer le document Chat.java dans un sous répertoire d'animaux et le nommer domestique.

```
package animaux.domestique;

public class Chat {
  // ...
}
```

Il existe un package particulier qui est le package courant, le package provenant du répertoire courant ou actuel. Dans ce cas particulier, il n'est pas nécessaire de mentionner le nom du package. Implicitement, les autres classes du même répertoire seront considérées comme du même package.

1.7.2 Mot clé import

Il est parfois nécessaire de reprendre ce qui a déjà été programmé dans un autre package, il nous faut donc pouvoir y accéder. Nous disposons pour cela du mot clé *import*.

Dans notre exemple, nous utilisons la classe Color qui se trouve dans le package java.awt, notre exemple deviendrait donc

```
package animaux.domestique;
import java.awt.*;

public class Chat {
  // ...
}
```

Le symbole .* signifie que nous souhaitons accéder à toutes les classes figurant dans le paquet java.awt. Il est aussi important de remarquer qu'on n'importe pas les classes figurant dans les sous répertoires. Par exemple, en

important les classes de java.awt, on n'importe pas les classes figurant dans java.awt.event.

Il se peut aussi qu'on souhaite uniquement accéder à la classe Color, dans ce cas il est possible d'importer que la classe voulue en la mentionnant directement, comme nous le montre l'exemple.

```
package animaux.domestique;
import java.awt.Color;

public class Chat {
  // ...
}
```

1.8 Les modificateurs

Jusqu'à présent, dans les codes précédents, apparaissent les mots *public* et *static*, ils font partis de ce que l'on nomme les modificateurs. En d'autres termes, ils indiquent un contexte d'utilisation. Tout au long de ce document, vous serez introduit progressivement à la plupart de ces modificateurs.

1.8.1 Modificateurs de portée

Les modificateurs de portée servent à définir les endroits depuis lesquels un attribut ou une méthode est accessible. Pour cela chaque modificateur possède une définition bien précise. Par élément, il ne peut y avoir qu'un seul modificateur de portée. Il existe quatre modificateurs, nous n'en présenterons que trois, le quatrième sera introduit lors d'un autre concept en programmation objet.

Modificateur public

Le modificateur *public* signifie que l'élément est accessible depuis n'importe où.

Modificateur private

Le modificateur *private* signifie que l'élément est accessible uniquement depuis l'intérieur de sa classe.

Modificateur package

Le modificateur *package* signifie que l'élément est accessible uniquement depuis le même package de sa classe. Contrairement aux autres modificateurs de portée, lorsqu'on souhaite utiliser ce modificateur, il n'y a pas de mot clé

qui y est associé. En d'autres termes ne pas ajouter un modificateur revient à déclarer l'élement avec le modificateur package.

Imaginons dans la classe Chat de notre exemple une méthode estSurveille() qui ne renvoie rien et accessible uniquement à l'ensemble des classes du même package.

```
package animaux;

public class Chat {
  public int age;
  public Color yeux;
  public Color fourrure;

  public Chat(int un_age, Color des_yeux,
    Color une_fourrure) {

    age = un_age;
    yeux = des_yeux;
    fourrure = une_fourrure;
  }

  public void mange(Souris s) {
    // ...
  }

  public boolean assezMange() {
    // ...
  }

  void estSurveille() {
    // Le chat ne peut plus faire n'importe quoi.
  }
}
```

Cette méthode pourra être utilisée par exemple dans la classe Chien ou toutes autres classes dans le package animaux.

```
package animaux;

public class Chien {
  public Chien() {
  }

  public void surveilleUnChat(Chat c) {
    c.estSurveille(); // accès permis
  }
```

```
}
```

1.8.2 Modificateur static

Le modificateur *static* consiste à signaler que l'élément ne dépend que de la classe et non d'un objet en particulier. Les accès se font donc par le nom de la classe elle-même. Nous appelons aussi les attributs statiques les attributs de classe. Les méthodes statiques sont, elles, appelées les méthodes de classe. Il existe aussi un abus de notation consistant à utiliser un objet pour y faire accès, nous vous le déconseillons fortement d'une part car il s'agit d'une mauvaise interprétation du concept en soi, d'autre part avec cet abus vous aurez probablement des difficultés par la suite à différencier ce qui concerne l'objet en soi, de ce qui concerne uniquement la classe.

Pour revenir à notre exemple, nous pouvons imaginer une méthode de classe getPlanete() qui renverrait la planète sur laquelle vie les chats. En effet, nous supposons que tous les chat vivent sur la même planète. De plus nous observons que la méthode ne dépend pas des objets qui seront créés à partir de cette classe. Néanmoins c'est une information qui concerne les chats.

```
package animaux;

public class Chat {
  public int age;
  public Color yeux;
  public Color fourrure;

  public Chat(int un_age, Color des_yeux,
    Color une_fourrure) {

    age = un_age;
    yeux = des_yeux;
    fourrure = une_fourrure;
  }

  public void mange(Souris s) {
    // ...
  }

  public boolean assezMange() {
    // ...
  }

  public static Planete getPlanete() {
    // retourne la planète Terre
  }
```

```
  void estSurveille() {
    // Le chat ne peut plus faire n'importe quoi.
  }
}
```

1.8.3 Modificateur final

Le modificateur *final* est un modificateur qui interdit à une variable de prendre une nouvelle valeur. En d'autres termes, une fois qu'une valeur est placée dans la variable, la variable ne peut plus la changer.

1.9 Le concept de l'encapsulement

Jusqu'à présent nous avons systématiquement utilisé le modificateur public, ainsi tout le monde peut modifier les valeurs de nos attributs. Bien que simple, cela convient que très peu à une programmation rigoureuse et sécurisée. Le concept d'encapsulement consiste à rendre privé les attributs, mais de pouvoir les modifier, si l'on souhaite, par des méthodes.

Pour cela on peut, sans obligation et selon notre volonté, élaborer deux méthodes, l'une pour lire, l'autre pour modifier. On les reconnait souvent facilement car elles commencent par convention par get ou set. Notre exemple deviendrait donc, en rajoutant la notion de poids :

```
package animaux;

public class Chat {
  private int age;
  private Color yeux;
  private Color fourrure;
  private int poids;

  public Chat(int un_age, Color des_yeux,
    Color une_fourrure) {

    age = un_age;
    yeux = des_yeux;
    fourrure = une_fourrure;
  }

  // accéder à une donnée privée
  public int getPoids() {
    return poids;
  }
```

```
  // modifier une donnée privée
  public void setPoids(int nouveau_poids) {
    poids = nouveau_poids;
  }

  public void mange(Souris s) {
    // ...
  }

  public boolean assezMange() {
    // ...
  }

  public static Planete getPlanete() {
    // retourne la planète Terre
  }

  void estSurveille() {
    // Le chat ne peut plus faire n'importe quoi.
  }
}
```

Dans la littérature, on parle aussi d'attributs *mutable* si ils sont modifiables et *unmutable* s'ils ne sont pas modifiables.

1.10 Le concept de la référence

Une référence ne contient pas la donnée directement, mais uniquement l'endroit dans la mémoire où se trouve la donnée. Néanmoins toutes les manières d'accéder à l'objet ont été rendues intuitives.

1.10.1 Déclaration

Qu'est-ce que signifie réellement « déclarer une variable » ? C'est tout simplement le fait de la créer. Essayez toujours de choisir des noms de variables le plus explicite possible. Cela simplifie considérablement la relecture du code. Il faut tout de même respecter les quelques règles suivantes en matière de nomination de variable :

Le nom doit toujours commencer par une lettre, un blanc souligné (_) ou encore un dollar ($). En aucun cas une variable ne peut commencer par un chiffre. Ceci s'applique uniquement pour le premier caractère. Vous pouvez choisir n'importe quel nom sauf les mots réservés déjà pris par le langage.

Dans l'exemple ci-dessous, nous avons créé une variable de type Chat avec pour nom « monChat ». Notons que pour le moment, la variable monChat ne contient pas de valeur.

```
Chat monChat;
```

1.10.2 Instanciation

L'instanciation consiste en la création d'un objet. Dans la littérature, on parle de créer une instance pour créer un objet. On utilise pour cela le mot clé *new* qui s'utilise suivit d'un constructeur.

Par exemple, si nous souhaitons créer un chat agé d'une année, ayant les yeux bleues et la fourrure brune, nous écririons :

```
new Chat(1, Color.BLUE, Color.BLACK);
```

Nous pouvons créer ce qu'on appelle un objet sans référence. C'est ce que nous avons fait avant. Nous pouvons aussi imaginer créer un objet directement dans un des paramètres d'une méthode. Pour reprendre l'exemple du chien et du chat :

```
Chien monChien = new Chien();
monChien.surveilleUnChat(
  new Chat(1, Color.BLUE, Color.BLACK));
```

1.10.3 Initialisation

L'initialisation consiste en l'application de la première valeur dans la variable. Ayant des références, ce sera donc l'adresse qui sera considérée dans l'initialisation.

Dans l'exemple précédent nous avons fait une initialisation de la variable monChien. Voici le code, si l'on devait faire également une initialisation de la variable monChat.

```
Chat monChat = new Chat(1, Color.BLUE, Color.BLACK);
Chien monChien = new Chien();

monChien.surveilleUnChat(monChat);
```

Il est aussi important de remarquer que le type d'objet construit doit être identique au type d'objet de la référence. Nous verrons plus tard que ce principe sera d'une certaine manière un peu plus souple.

1.10.4 Affectation

L'affectation consiste en l'application d'une nouvelle valeur qui remplacera la précédente. Dans notre exemple, nous construisons deux chats bien distincts qui seront chacun à leur tour mis dans la variable monChat.

```
Chat monChat = new Chat(1, Color.BLUE, Color.BLACK);
monChat = new Chat(0, Color.BROWN, Color.WHITE);
```

Il est important de ne pas rajouter Chat en début de la seconde ligne dans l'exemple précédent, car sinon le compilateur interprétera qu'il y a deux variables de même nom, ce qui ne conviendra pas, d'où une erreur. Tout comme pour l'initialisation le type d'objet construit doit être identique au type d'objet de la référence.

1.11 La référence null

Nous savons que dans les exemples précédents, nous construisons un Chat et le plaçons dans la référence monChat. Néanmoins, que ce passe t-il si l'on tente de créer une référence sans y affecter une valeur, puis que l'on y accède ? Nous avons en Java la référence null, c'est-à-dire la référence qui ne pointe sur rien. Comme elle ne pointe sur rien, il est impossible d'y appliquer une méthode ou un attribut. Cette référence est utile si l'on souhaite effacer une valeur d'une référence.

L'exemple suivant produira une erreur, car la méthode surveilleUnChat(Chat c) appelle c.estSurveille(), mais comme la variable monChat est null, alors c est null, d'où l'erreur.

```
Chat monChat = new Chat(1, Color.BLUE, Color.BLACK);
monChat = null;

Chien monChien = new Chien();
monChien.surveilleUnChat(monChat); // produira une erreur
```

La même situation se passerait si, on n'initialise pas monChat

```
Chat monChat; // variable pas initialisé
// équivalent à Chat monChat = null;

Chien monChien = new Chien();
monChien.surveilleUnChat(monChat); // produira une erreur
```

1.12 La référence this

Nous avons vu que notre constructeur utilise des noms de variable différents, mais nous aimerions néanmoins pouvoir réutiliser les mêmes dans le constructeur et dans les variables d'instance, comme nous le rappelle l'exemple suivant.

```
public class Chat {
  private int age;
  private Color yeux;
  private Color fourrure;

  public Chat(int un_age, Color des_yeux,
    Color une_fourrure) {

    age = un_age;
    yeux = des_yeux;
    fourrure = une_fourrure;
  }

  // etc...
}
```

Si nous le faisons directement nous aurons un problème, car les variables existantes déjà localement, seront privilégiées à celle de l'instance.

```
public class Chat {
  private int age;
  private Color yeux;
  private Color fourrure;

  public Chat(int age, Color yeux,
    Color fourrure) {

    // les variables d'instances ne sont pas mises à jour
    age = age;
    yeux = yeux;
    fourrure = fourrure;
  }

  // etc...
}
```

Java dispose pour cela d'une référence particulière qui nous indique l'adresse de l'objet courant. Cette référence s'appelle *this* et elle pointe sur l'objet qui

est en train d'être utilisé.

```
public class Chat {
  private int age;
  private Color yeux;
  private Color fourrure;

  public Chat(int age, Color yeux,
    Color fourrure) {

    // les variables d'instances sont mises à jour
    this.age = age;
    this.yeux = yeux;
    this.fourrure = fourrure;
  }

  // etc...
}
```

De plus, vu que *this* pointe sur l'objet courant, il n'est pas permis d'utiliser this dans une méthode statique, vu que cette méthode ne dépend pas d'une instance. En d'autres termes, dans une méthode statique il n'y a pas d'objet courant. Il est important de ne pas confondre cela avec créer ou utiliser des objets dans une méthode statique.

Voici encore un exemple, de ce qui est permis

```
public class Chat {
  private int age;
  private Color yeux;
  private Color fourrure;

  public Chat(int age, Color yeux,
    Color fourrure) {

    // les variables d'instances sont mises à jour
    this.age = age;
    this.yeux = yeux;
    this.fourrure = fourrure;
  }

  public static int ageMoyen(Chat c1, Chat c2) {
    return (c1.age+c2.age)/2;
  }
}
```

et de ce qui ne l'est pas.

```
public class Chat {
  private int age;
  private Color yeux;
  private Color fourrure;

  public Chat(int age, Color yeux,
    Color fourrure) {

    // les variables d'instances sont mises à jour
    this.age = age;
    this.yeux = yeux;
    this.fourrure = fourrure;
  }

  public static int ageMoyen(Chat c) {
    return (this.age+c.age)/2; // this représente quoi?
    // cette méthode est statique
  }
}
```

1.13 Combinaison de modificateur

Les modificateurs peuvent se combiner normalement, il existe certaines exceptions dues au concept qui ne sont pas compatibles. La combinaison des modificateurs est importante car elle détermine le contexte d'utilisation : accessible depuis où ? de classe ou d'instance ? modifiable ? Ce qui signifie respectivement à choix public ou private, puis static ou non, et enfin en dernier final ou non.

1.13.1 La constante

La constante est un cas particulier d'utilisation de modificateur. Par le principe d'une constante, elle n'est pas différente d'une instance à une autre, donc elle est static. De plus une fois définie, elle ne doit plus être modifiable, donc elle doit être final. La dernière question que nous devons nous poser consiste à choisir entre une constante accessible uniquement à l'intérieur de la classe, au même package ou au contraire accessible de partout. Le choix nous est propre en fonction de ce que nous souhaitons faire.

Une utilisation particulière de la constante consiste à prédéfinir des choix, puis de les utiliser dans le contexte voulu. Imaginons dans notre exemple que nous souhaitons introduire le mental du chat, nous allons pour cela introduire les constantes TROP_HEUREUX, HEUREUX, PARTIELLEMENT_HEUREUX, et MALHEUREUX. Les constantes s'écrivant

de préférence que en majuscule pour mieux les reconnaître.

```
public class Chat {
  private int age;
  private Color yeux;
  private Color fourrure;
  private int mental;

  public static final int TROP_HEUREUX=0;
  public static final int HEUREUX=1;
  public static final int PARTIELLEMENT_HEUREUX=2;
  public static final int MALHEUREUX=3;

  public Chat(int age, Color yeux,
    Color fourrure) {
    this(age, yeux, fourrure, HEUREUX);
    // this utilisé de cette manière appelle
    // un constructeur de la même classe
  }

  public Chat(int age, Color yeux,
    Color fourrure, int mental) {

    // les variables d'instances sont mises à jour
    this.age = age;
    this.yeux = yeux;
    this.fourrure = fourrure;
    this.mental = mental;
  }

  // ...

  public int getMental() {
    return mental;
  }

  public void setMental(int mental) {
    this.mental = mental;
  }
}
```

```
Chat monChat
  = new Chat(1, Color.BLUE, Color.BROWN, Chat.HEUREUX);
monChat.setMental(Chat.TROP_HEUREUX);
```

Cette utilisation excessive de constantes est utilisée pour faciliter la lecture de code, cela permet également de savoir ce qu'a prévu exactement le

programmeur de la classe.

1.14 La méthode main

Pour pouvoir exécuter un programme, il faut pouvoir désigner un point d'entrée. En d'autres termes, il faut pouvoir signaler la première instruction à faire. En Java, comme il n'y a que des classes, la méthode main doit donc être dans une classe. N'ayant rien de particulier avec une instance, elle doit donc être static. De plus, devant être accessible depuis l'extérieur de la classe, elle doit être public.

L'argument args est en faite un tableau contenant tous les paramètres que Java transmet à l'application. Nous pouvons à présent comprendre parfaitement ce que fait l'exemple suivant.

```
public class HelloWorld {
  public static void main(String[] args) {
    // Le contenu du programme
  }
}
```

1.15 Afficher du texte dans la console du système opérationnel

Le système opérationnel ou système d'exploitation tel que Mac OS X, Linux ou Unix fournissent plusieurs points d'entrées et sorties. Le point de sortie standard pour pouvoir afficher du texte est en fait l'objet *out* se trouvant dans la classe *System*. Nous pouvons y accéder directement. De plus, la méthode pour afficher du texte est la méthode *println*.

Nous pouvons, à présent comprendre pleinement notre premier programme, présenté en début de chapitre.

```
public class HelloWorld {
  public static void main(String[] args) {
    System.out.println("This is a friendly HelloWorld!");
  }
}
```

Ce premier programme nous produira la sortie suivante :

```
This is a friendly HelloWorld!
```

Chapitre 2

Type primitif et classe String

2.1 Motivation

Jusqu'à présent nous n'avons vu que le comportement des objets. Il y a cependant des données qui ne sont pas des objets, elles s'appellent les types primitifs. Ce nom provient du fait que historiquement ce sont des types de nombre pouvant être traités en tant que tel dans le processeur de l'ordinateur. Java propose huit types primitifs. Nous allons voir dans ce chapitre en quoi les types primitifs sont différents des objets. Nous allons également aborder la classe String en profondeur et l'utiliser afin de mieux montrer la différence qu'il y a entre un type primitif et un objet.

2.2 Les types primitifs

Normalement les tailles de type primitif sont propres à chaque système opérationnel, toutefois, Java devant standardiser son comportement sur chaque système, à choisi d'affecter des tailles fixes à chacun de ces types.

2.2.1 Les nombres entiers

Les types des nombres entiers sont au nombre de quatre et ayant chacun une capacité différente. Un bit représente deux valeurs 0 et 1. Ainsi avec n bits, nous pouvons représenter 2^n valeurs.

Type	Dimension	Représentation
byte	8 bits	-128 à 127
short	16 bits	-32'768 à 32'767
int	32 bits	-2'147'483'648 à 2'147'483'647
long	64 bits	immense

Le type *int* est également appelé *integer*.

2.2.2 Les nombres à décimales

Les types de nombre à décimales sont au nombre de deux et ayant chacun une capacité différente. Tous les nombres à décimales ne peuvent pas être enregistrés tel quel, ils sont arrondis pour être dans le format des nombres à virgule dans l'informatique. Les nombres à virgule sont généralement décomposés en trois parties : l'une pour le signe, une autre pour un nombre entier et une troisième pour un exposant. Nous avons donc un nombre d de la forme :

$$d = s * n * 10^e$$

où s représente le signe -1 ou $+1$, n est le nombre entier, et e est une puissance (positive ou négative).

Afin de pouvoir remarquer qu'il y a une perte, prenez votre calculatrice est faite $\sqrt{2} * \sqrt{2} - 2$, il est fort probable que vous n'obteniez pas 0, mais un nombre extrêmement petit. En effet $\sqrt{2}$ est arrondie selon le système ci-dessus, est donc sa mise au carré ne correspond plus à 2. Afin d'augmenter la précision, on utilise généralement le type double, et très peu le type float.

Type	Dimension
float	32 bits
double	64 bits

2.2.3 Le type logique ou booléen

Le type logique, ou plus communément appelé *boolean*, est un type qui ne possède que deux valeurs qui sont vrai ou faux. Les valeurs sont *true* pour vrai et *false* pour faux.

2.2.4 Les caractères

Les caractères sont en faite des nombres. Pour chacun des nombres on décide conceptuellement à quel symbole il correspond. C'est pourquoi il existe plusieurs encodages tels que latin-1 ou utf-8. Car chaque encodage attribut son propre symbole à un nombre.

Type	Dimension
char	16 bits

2.2.5 Conversion implicite

Dans les types primitifs il y a des règles de conversion implicite. Nous allons uniquement traiter deux de ces conversions.

int vers long

Lorsque nous écrivons le code suivant, nous pourrions avoir la fausse impression que nous créons un nombre de type long pour le placer dans la variable de type long, il n'en n'est rien. En effet, le nombre créé est un int, la variable étant de type long, le nombre sera converti en type long.

```
long l = 12;
```

Afin d'éviter ce problème, il faut rajouter un suffixe L pour indiquer que c'est bien un nombre de type long.

```
long l = 12L;
```

char vers int

Nous avons vu qu'un caractère n'est qu'en fait un nombre, c'est pourquoi il est permis d'enregistrer « une lettre » dans un int.

```
long l = 'A'; // enregistre le nombre correspondant à 'A'
```

2.3 Opérateurs numériques

Il existe différents opérateurs numériques, qui représentent les opérations arithmétiques standards. Nous connaissons déjà les opérateurs + et * qui reprennent l'addition et la multiplication mathématique. Il existe aussi l'opérateur incrément ++ qui peut à la fois se mettre avant ou après la variable. Observons pour cela le code source suivant.

```
int i=3;
i++; // i devient 4
++i; // i devient 5
```

Néanmoins leur signification ne sont pas les mêmes, comme nous le montre l'exemple suivant. Car l'opérateur se plaçant avant à une priorité élevé.

```
int i = 3;
i = i++ * 3 // i * 3 + 1

int j = 3;
j = ++j * 3; // (j+1) * 3
```

2.4 Valeur contre référence

Une variable mémorise une référence lorsqu'il s'agit d'un objet, et mémorise une valeur lorsqu'il s'agit d'un type primitif, la différence est de taille. En effet, lorsqu'on passe une valeur à une méthode en paramètre, la valeur est copiée. Alors que lorsqu'il s'agit d'une référence, c'est la référence qui est copiée. Ainsi l'accès à l'objet est resté intacte, et donc l'objet pourra être modifié depuis l'intérieur de la méthode, ce que l'utilisateur n'avait pas forcément prévu. Afin de mieux illustrer ce principe, nous allons observer deux programmes qui font la même chose, une fois avec un int, une autre fois avec un chat.

```
public class Destroyer {
  public static void destroyer(Chat c) {
    c = new Chat(
      2063, Color.YELLOW, Color.GREEN, Chat.TROP_HEUREUX);
  }

  public static void destroyer(int i) {
    i = 0;
  }

  public static void main(String[] args) {
    Chat monChat
      = new Chat(1, Color.BLUE, Color.BRWON, Chat.HEUREUX);
    int i = 007;

    destroyer(monChat);
    destroyer(i);

    System.out.println(monChat.getMental());
    System.out.println(i);
  }
}
```

```
0
7
```

où 0 est la valeur de la constante Chat.TROP_HEUREUX.

On observe bien que l'objet contenu dans monChat est modifié, et que l'integer n'est pas modifié.

2.4.1 Empêcher la modification de la valeur

Nous avons vu que si une variable détient le modificateur final, cela empêche la valeur d'être modifiée. Ainsi si le modificateur est présent dans

une variable d'une méthode, cela signifie que la modification de la variable est interdite.

```
public class Destroyer {
  public static void destroyer(final Chat c) {
    // c = new Chat(
    //  2063, Color.YELLOW, Color.GREEN, Chat.TROP_HEUREUX);
    // la ligne précédente serait interdite
  }

  public static void destroyer(int i) {
    i = 0;
  }

  public static void main(String[] args) {
    Chat monChat
      = new Chat(1, Color.BLUE, Color.BRWON, Chat.HEUREUX);
    int i = 007;

    destroyer(monChat);
    destroyer(i);

    System.out.println(monChat.getMental());
    System.out.println(i);
  }
}
```

```
1
7
```

où 1 est la valeur de la constante Chat.HEUREUX.

2.4.2 Avantage des références

Parfois, il est bien de pouvoir modifier le contenu d'un objet directement, cela évite de devoir retourner une valeur qui correspond à la valeur modifiée, puis de récupérer cette valeur et de la placer dans la bonne référence.

```
public class Destroyer {
  public static void destroyer(Chat c) {
    Chien pluto = new Chien();
    pluto.surveilleUnChat(c);
    c.setMental(Chat.TROP_HEUREUX);
    // le chat est surveille par pluto
    // il n'y a donc plus aucun risque
    // pour le chat
```

```
  }

  public static void destroyer(int i) {
    i = 0;
  }

  public static void main(String[] args) {
    Chat monChat
      = new Chat(1, Color.BLUE, Color.BRWON, Chat.HEUREUX);
    int i = 007;

    destroyer(monChat);
    destroyer(i);

    System.out.println(monChat.getMental());
    System.out.println(i);
  }
}
```

```
0
7
```

où 0 est la valeur de la constante Chat.TROP_HEUREUX.

2.5 Classes Wrapper

Les types primitifs ne sont pas des objets, ainsi il nous est impossible de bénéficier des avantages de l'objet. C'est pourquoi il existe des classes wrapper, qui enveloppent le type primitif.

Type primitif associé	Type
byte	Byte
short	Short
int	Integer
long	Long
boolean	Boolean
char	Char
float	Float
double	Double

Toutes les classes wrapper possèdent un constructeur ayant comme unique argument le type primitif associé. Il est ainsi possible d'utiliser toute la puissance des concepts objets.

```
Integer i = new Integer(25);
Char c = new Char ('E');
```

2.6 La classe String

Une string est une chaine de caractère, par exemple :

```
String chaine = "Bonjour";
```

On peut également l'écrire d'une autre manière :

```
String chaine = new String("Bonjour");
```

Une string vide (sans caractère) peut être créé ainsi :

```
String str1 = "";
// ou bien
String str2 = new String();
```

Les strings sont très utile pour afficher du texte, c'est pourquoi si un objet doit être affiché, on fait appelle à la fonction toString() qui existe dans toutes les classes. Cette méthode renvoie alors une String. Un autre élément important est que Java nous fourni beaucoup de méthode dans la classe String. Il est donc possible de faire beaucoup de manipulation sur les strings. C'est en particulier utile pour du traitement de texte.

Nous allons explorer à présent la classe String.

2.6.1 Méthode trim

La méthode *trim* supprime les espaces en début et en fait de string. C'est par exemple utile si une autre personne à rajouté par erreur des espaces, et que votre programme n'était pas conçu pour. La méthode renvoie une nouvelle String sans les espaces.

```
String trim()
```

```
public class StringTestator {
  public static void main(String[] arg) {
    String str = " Un espace devant et entre les mots.";
    System.out.println(str);
    System.out.println(str.trim());
  }
}
```

```
 Un espace devant et entre les mots.
Un espace devant et entre les mots.
```

2.6.2 Méthode equal

La méthode equal permet de comparer deux string pour savoir si elles sont identiques.

```
boolean equal(Object string)
```

```
public class StringTestator {
  public static void main(String[] arg) {
    String str1 = " Un espace devant et entre les mots.";
    String str2 = " Un espace devant et entre les mots.";
    System.out.println(str1.equal(str2));
  }
}
```

```
true
```

2.6.3 Méthode equalIgnoreCase

La méthode equalIgnoreCase permet de comparer deux string sans tenir compte des majuscules et minuscules.

```
boolean equalIgnoreCase(String string)
```

```
public class StringTestator {
  public static void main(String[] arg) {
    String str1 = " Un eSpace dEvant et eNtre les mots.";
    String str2 = " uN esPace devAnt ET entre lEs mots.";
    System.out.println(str1.equal(str2));
  }
}
```

```
true
```

2.6.4 Méthode length

La méthode *length* renvoie la longueur de la String.

```
int length()
```

2.6.5 Méthode charAt

La méthode *charAt* renvoie le caractère de la position choisie, les positions commencent à 0 jusqu'à length()-1.

```
char charAt(int position)
```

2.6.6 Méthode substring

La méthode substring extrait une String de la String originale.

```
String substring(int begin_position, int end_position)
```

2.6.7 Méthode concat

La méthode *concat* permet d'ajouter une String à la fin de la String courante, puis renvoie ce résultat.

```
String concat(String string)
```

```
public class StringTestator {
  public static void main(String[] arg) {
    String str1 = "Un ";
    String str2 = str1.concat("must!");
    System.out.println(str2);
  }
}
```

```
Un must!
```

Opérateur de concaténation

En Java, il est aussi possible d'utiliser l'opérateur de concaténation +. Voici comment l'exemple précédent est modifié.

```
public class StringTestator {
  public static void main(String[] arg) {
    String str1 = "Un ";
    String str2 = str1 + "must!";
    System.out.println(str2);
  }
}
```

```
Un must!
```

Chapitre 3

Conditions, boucles et tableaux

3.1 Introduction

Au chapitre 2, nous avons découvert l'utilité des variables et les différents types existants. Nous savons donc maintenant calculer avec des nombres entiers ou réels, et nous connaissons quelques opérations élémentaires sur les chaînes de caractères grâce à la classe String. Mais nous avons encore un vide à combler : comment faire lorsque nous devons traiter une série de données identiques, de surcroît lorsqu'on ne sait pas à l'avance leur nombre ? Nous pouvons certes créer des variables avec des noms du type donnee1, donnee2, donnee3, ... mais ce n'est pas la bonne solution. Cela s'avère rapidement inutilement complexe et peu pratique à utiliser en plus d'être assez peu flexible. C'est alors que les tableaux simples viennent à la rescousse. Dans la troisième partie de ce chapitre, nous allons découvrir que grâce à eux, il est très aisé de gérer une suite de données identiques. Puis nous allons ensuite voir quelles en sont les limites et explorer les tableaux dynamiques, qui eux vont plus loin en permettant avec une simplicité extrême de gérer des séries de données dont la quantité varie au cours du temps. Mais avant de s'attaquer à ces fameux tableaux, nous devons d'abord apprendre ce qu'est une condition et une boucle, et comment s'en servir pour rendre nos programmes un peu moins monotones. Nous en aurons besoin dans la quatrième et dernière partie, qui mettra en pratique tout ce que nous avons appris en mélangeant tous les ingrédients.

3.2 If - un mot simple qui cache une réalité complexe

En Java, comme dans de nombreux autres langages de programmation, une condition se matérialise par une instruction « if » ainsi qu'une partie « else » (signification : sinon) optionnelle :

```
if (condition) {
  // instructions
}
else {
  // Instructions
}
```

Les instructions qui se trouvent entre les deux premières accolades ne seront réellement exécutées que si et seulement si la condition qui est entre les parenthèses du if est vraie. Le deuxième bloc (la deuxième paire) sera exécuté au contraire si la condition consignée par le if est fausse. Ce bloc est facultatif. Pour indiquer la condition, nous avons à notre disposition plusieurs opérateurs mathématiques de comparaison (liste ci-dessous). Nous pouvons bien sûr comparer des nombres entre eux et d'autres objets, mais la réelle utilité est de comparer la valeur d'une variable et un nombre ou la valeur de deux variables entre elles. Voici ci-dessous quelques exemples d'instructions de comparaison qui trouveraient leur place entre les parenthèses d'un if :

```
3 > 5 // 3 est plus grand que 5. Faux dans tous les cas.
```

```
nombre <= 7 // la valeur de la variable nombre est plus
// petite ou égale à 7
```

```
nombre1 == nombre2 // la valeur de la variable nombre1 est
// égale à la valeur de nombre2.
```

```
nombre != 8.5 // La valeur de nombre est différente de 8.5
```

```
true // true est vrai, false est faux, d'ailleurs on
// pourrait aussi utiliser une variable de type boolean
```

Il existe 6 opérateurs de base :

== pour vérifier l'égalité exacte (à ne pas confondre avec le simple = qui permet d'attribuer une valeur à une variable)

!= pour vérifier une inégalité
$>, <, >=, <=$ pour vérifier respectivement si le premier terme est supérieur ou inférieur au deuxième, strictement ou non.

Grâce à cela, nous pouvons comparer les types primitifs (les nombres) entre eux. Les variables de type boolean, qui contiennent déjà true ou false, peuvent être testées directement (Ex : if(mon_boolean)) Pour comparer des objets, même des chaînes (String), nous ne pouvons par contre pas utiliser ces opérateurs. Il faut avoir recours à des méthodes spécialisées. La plus connue est equals mais il existe aussi compareTo (que nous avons vu dans le chapitre sur les chaînes). Par exemple :

```
String test = "hello";
if (test.equals("Hello")) {
  // instructions
}
```

Ici, les deux chaînes "hello" et "Hello" seront comparées. Si elles sont équivalentes, le bloc if sera exécuté. Ce qui n'est ici pas le cas, car la correspondance minuscules/majuscules est prise en compte.

Nous en connaissons maintenant assez pour attaquer la partie suivante : les boucles

3.3 Les boucles

Dans la vie de tous les jours, il est parfois nécessaire de répéter une même série d'opérations plusieurs fois de suite sous certaines conditions : on se lève tous les matins de la semaine pour aller au travail sauf si on a les vacances par exemple, ou bien on mange trois fois dans la journée.

On appelle une série d'instructions exécutées plusieurs fois consécutives une *boucle*. En programmation, on fait la distinction entre deux sortes principales de boucles : celles qui sont répétées un certain nombre de fois connu d'avance, et celles qui sont répétées jusqu'à ce qu'une certaine condition soit ou ne soit plus remplie. Nous allons commencer par cette dernière, qui est plus facile.

3.3.1 la boucle while - quand on ne sait pas quand s'arrêter

Comme vous l'aurez compris, le mot-clé important ici est *while* (traduction : « pendant que » ou « tant que »). Voici succinctement comment il s'utilise :

```
while (condition) {
```

```
  // instructions
}
```

Les instructions qui se trouvent entre les accolades seront répétées jusqu'à ce que la condition spécifiée à la suite du mot-clé while *ne soit plus remplie*, ou autrement dit tant que la condition est vérifiée. La syntaxe de la condition attendue est la même que celle utilisée dans les blocs if/else.

Voici un exemple qui demande à l'utilisateur de taper le mot « exit » au clavier tant que cela n'a pas été fait.

```
String saisie = "";
Scanner in = new Scanner(System.in);
while (!saisie.equals("exit")) {
  System.out.println(
    "Tapez « exit » pour quitter le programme");
  saisie = in.next();
  saisie = saisie.trim();
}
```

Le point d'exclamation devant la condition saisie.equals(...) permet d'inverser le résultat de la méthode equals, ce qui revient à demander la non-équivalence de la valeur de la variable avec "exit". Le bloc entre accolades, soit le message d'invite et l'attente d'une saisie par l'utilisateur, sera exécuté tant que la condition inscrite dans le while, c'est-à-dire notre non-équivalence, est vérifiée. Ce qui est effectivement le cas jusqu'à ce que l'utilisateur tape exactement « exit ». La classe Scanner utilisée ici est assez importante et offre de nombreuses façons d'obtenir des données entrées au clavier de manière simple. Sa méthode next permet en particulier de récupérer un mot sans espace, ce qui est parfaitement adapté pour ce que nous faisons dans notre cas. Il serait tout aussi facile de demander un nombre... nous aurons l'occasion de réutiliser cette classe plusieurs fois d'ici la fin de ce chapitre. Si vous êtes intéressé à la connaître plus, référez-vous à la documentation officielle de Java sur le web.

Voici maintenant un deuxième exemple, celui qu'il *ne faut surtout pas suivre*, vous comprendrez pourquoi quand vous aurez essayé si vous ne tiltez pas immédiatement :

```
while (true) {
  // instructions
}
```

On appelle ceci une boucle infinie, ou, autrement dit, une boucle qui se répète jusqu'à ce que plantage magistral de votre pauvre ordinateur surchargé s'en suive. Le bloc est répété tant que la condition est vérifiée. Or, l'erreur ici est que le mot-clé true est, par définition, toujours vrai. Par conséquent, le programme restera indéfiniment dans le bloc sans jamais en sortir. Ce type de bug est courant en cours de développement, et, heureusement pour nous, les ordinateurs modernes nous offrent tous les moyens d'arrêter le monstre de manière totalement fiable sans avoir à tirer la prise...

Et voici un dernier exemple. Qu'en pensez-vous ?

```
while (false) {
  // Instructions
}
```

Si vous avez prédit sans tester que les instructions de cette boucle ne serait jamais exécutées, vous avez parfaitement raison. L'explication est la même que précédemment : false étant par définition toujours faux, la condition n'est jamais vérifiée et les instructions ne sont par conséquent jamais parcourues. Ce qui nous amène à une constatation importante : la condition est évaluée *avant* de parcourir les instructions associées. Ainsi, une boucle while peut très bien ne jamais être exécutée, comme celle ci-dessus.

Il existe une variante de while : la boucle do...while, dont voici la syntaxe :

```
do {
  // instruction
} while (condition);
```

Notez le point-virgule final. Avec cette variante, la condition est toujours évaluée *après* les instructions. Ce qui assure qu'elles sont exécutées *au moins une fois*. Le reste du processus est le même. L'utilisation de l'une plutôt que l'autre est souvent une histoire de goût, car ces deux constructions sont pratiquement toujours interchangeables.

3.3.2 la boucle for - quand on sait à l'avance combien de fois

Dans notre exemple précédent avec l'utilisateur qui était censé taper « exit », nous ne connaissions pas d'avance combien d'essais seraient nécessaires avant qu'il ne comprenne. Avec les boucles for que nous allons voir maintenant, il en va autrement : nous connaissons en principe toujours d'avance le nombre de fois qu'une boucle for va s'exécuter.

Voici rapidement la syntaxe de cette instruction :

```
for (initialisation; condition; incrément) {
  // instructions
}
```

Comme vous le voyez, elle est un peu plus difficile que la boucle while. On distingue trois parties entre les parenthèses du for :

- L'initialisation : dans cette première partie, on crée généralement une variable de type entière et on lui affecte une valeur de départ. On appelle cette variable un *compteur*, et la pratique courante est de l'appeler avec une seule lettre de l'alphabet, souvent i, j, k ou n. On n'est bien sûr pas obligé de créer une nouvelle variable à ce moment-là, ni qu'elle soit entière d'ailleurs et réutiliser une autre déjà existante. L'avantage de la créer à cet endroit est qu'on limite son champ d'action : la variable ainsi déclarée n'existe que dans le cadre de la boucle for. Une fois qu'elle est terminée, la variable n'existe plus. Ce qui permet d'éviter quelques erreurs difficiles à déboguer, et ce qui permet aussi de ne pas avoir de collisions de noms de variables lorsque plusieurs boucles se suivent ou s'imbriquent, ce qui est très régulièrement le cas.
- La condition : De même que pour la boucle while, la boucle for est répétée tant que la condition spécifiée est vérifiée.
- L'incrémentation : Dans cette partie, on incrémente généralement le compteur déclaré en première partie. Incrémenter signifie littéralement augmenter d'une unité, mais on est libre, si on le souhaite, de modifier les variables que l'on désire. Nous ne devons pas obligatoirement incrémenter d'une unité à la fois et pas obligatoirement non plus uniquement la variable compteur. On appelle parfois cette partie le *pas*, car elle représente souvent de combien la variable compteur varie à chaque étape successive de la boucle.

Des étapes, parlons-en, car c'est un peu particulier : lorsque l'ordinateur rencontre une boucle for, il comence par exécuter la partie initialisation, qui ne le sera qu'une seule et unique fois en début de boucle. Ensuite, il évalue une première fois la condition. Si elle n'est pas vérifiée, il ne va pas plus loin et quitte directement le bloc sans se poser d'autres questions. Si elle l'est, il parcourt les instructions dans l'ordre avant de passer à la partie incrémentation. Puis, il réévalue la condition et effectue un nouveau tour de boucle si elle est toujours vérifiée, ou la quitte sinon.

Comme c'est peut-être un peu compliqué, voici un exemple simple qui se contente d'afficher successivement les nombres de 1 à 100 :

```
for (int i = 1; i <= 100; i++) {
  System.out.println(i);
```

```
}
```

Reprenons ce que nous avons dit plus haut : nous commençons par la partie initialisation, qui commande de créer une variable int i contenant la valeur 1. Ensuite nous passons à la condition et nous remarquons que i est bien inférieur ou égal à 100, donc on continue. Nous affichons la valeur courante de la variable i, pour le moment 1. Nous passons à la partie incrémentation. L'instruction i++ est l'abréviation de i+=1 qui lui-même est l'abréviation de i=i+1. Nous affectons donc à i la valeur qu'il possède plus une unité, soit i=1+1=2 dans notre exemple. La variable i vaut maintenant 2 qui est toujours inférieur ou égal à 100, nous affichons une fois de plus la valeur de i avant de l'incrémenter à nouveau. Et ainsi de suite jusqu'à ce que i=100+1=101 ce qui fera échouer la condition et qui commandera la sortie de boucle.

Voici un deuxième exemple plus intéressant : il s'agira ici de calculer la factorielle d'un nombre entier. Pour rappel, formellement, la factorielle d'un nombre entier n se définit comme étant le produit 1 * 2 * 3 * ... * (n -1) * n. Par exemple la factorielle de 5 est 120 car 1*2*3*4*5=120.

```
public int factorielle (int n) {
  int resultat = 1;
  for (int i = n; i > 0; i--) {
    resultat *= i;
  }
  return resultat;
}
```

Cette méthode reçoit en paramètre le nombre n dont nous souhaitons calculer la factorielle. On initialise le compteur i de la boucle for à n. Cette fois-ci, l'instruction d'incrémentation est i–, ce qui équivaut à i-=1 ou i=i -1, qui correspond à diminuer i d'une unité à chaque tour de boucle. Nous faisons tourner la boucle jusqu'à ce que i arrive à la valeur 0. La variable résultat contiendra au final la valeur n * (n -1) * (n -2) * ... * 3 * 2 * 1, ce qui est précisément la définition de la fonction factorielle.

Et voilà ! Nous pouvons enfin attaquer nos fameux tableaux.

3.4 Les tableaux

Lorsque nous souhaitons manipuler une série de données identiques, il faut utiliser un tableau. Un tableau n'est rien d'autre qu'une variable unique dans laquelle sont en fait stockés une série de données du même type. Un tableau simple est défini par son *type* et sa *taille*. Son type est le même

pour tous les éléments du tableau, il n'est donc pas possible d'enregistrer conjointement des int et des double dans le même tableau. Sa taille est le nombre d'éléments qu'il peut stocker. Une fois définie, cette taille ne peut être modifiée, ce qui peut être un inconvénient assez lourd si le nombre d'éléments à traiter varie au cours du programme. C'est pour cela qu'il existe les tableaux dynamiques, que nous verrons dans la section suivante de ce chapitre. Mais pour le moment, occupons-nous des tableaux simples.

3.4.1 Déclarer un tableau

Déclarer une variable tableau se fait très simplement, presque comme une variable ordinaire :

```
int[] tableau;
tableau = new int[100];
```

La variable ci-dessus est un tableau de 100 valeurs int. On reconnaît qu'il s'agit d'un tableau grâce à la paire de crochets qui suit le mot-clé int définissant le type. Mais la première ligne ne suffit pas pour créer le tableau proprement dit, pour le moment il contient, comme pour un objet non initialisé, la valeur null. Sur la deuxième ligne, on rappelle qu'il s'agit bien d'un tableau de int, on précise en plus le nombre d'éléments qu'il contiendra entre crochets, ici 100. On a le choix entre indiquer un nombre fixe ou alors la valeur d'une variable de type int (uniquement une variable de type int). Nous aurions pu faire le tout en une seule ligne. Après ces instructions, notre tableau est disponible et nous pouvons commencer à l'utiliser.

Il existe une deuxième façon de déclarer un tableau en lui fournissant d'office les valeurs par défaut qu'il contiendra :

```
int[] tableau = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
```

Ce code créera un tableau de type int contenant 10 valeurs. Il est inutile de préciser la taille, le compilateur s'en chargera en comptant directement le nombre d'éléments dans les accolades. Après cette instruction, nous disposons donc d'un tableau de 10 éléments contenant les nombres de 1 à 10 dans l'ordre.

3.4.2 Accéder aux valeurs d'un tableau

C'est bien beau d'avoir déclaré un tableau, mais c'est un peu embêtant de ne pas pouvoir s'en servir. Afin d'accéder à un élément précis du tableau, nous devons faire suivre le nom de la variable de l'*indice* de l'élément auquel

nous souhaitons accéder. Cet indice s'écrit entre crochets juste après le nom de la variable et indique l'emplacement, le numéro d'ordre de l'élément à atteindre. Ensuite, on peut manipuler notre variable comme d'habitude. Le premier élément du tableau ne porte pas l'indice 1 mais *l'indice 0* et le dernier l'indice taille -1 au lieu de taille. Il est possible de connaître la taille du tableau en utilisant la propriété length, que tout tableau déclaré possède automatiquement.

Exemple rapide :

```
int[] tableau = new tableau[5]; // On déclare un nouveau
// tableau de type int contenant 5 éléments

tableau[0] = 123;
// On affecte la valeur 123 au premier élément

tableau[4] = 456;
// On affecte la valeur 456 au cinquième élément

tableau[tableau.length -1] = tableau[0] + tableau[4];
// On additionne la valeur du premier et du cinquième
// élément et on la stocke dans le dernier élément, qui
// est en fait le cinquième (la valeur 456 est donc perdue,
// mais on est pas censé le savoir, ce ne serait pas le cas
// si le tableau avait 6 éléments ou plus).

int n = 2;
tableau[n] = 4*n +7; // On peut bien sûr utiliser une
// variable de type int en tant qu'indice. Ici, ce sera
// donc le troisième élément qui sera modifié puisque n=2.

System.out.println(tableau[1]); // On affiche la valeur du
// deuxième élément... aucune valeur explicite ne lui a été
// affectée, sa valeur par défaut est 0.
```

3.4.3 Parcourir un tableau

Nous connaissons maintenant la base de la manipulation des tableaux. Il est temps d'apprendre à les parcourir de manière efficace en utilisant les boucles que nous avons vu dans la section précédente de ce chapitre. Pour parcourir un tableau, on choisit en général une boucle for, et on utilise la variable compteur de la boucle comme indice de tableau. Le compteur, comme l'indice de tableau, doivent varier entre 0 et tableau.length -1 et nous devons tout parcourir sans rien oublier. En principe, la boucle for démarre donc avec un compteur à 0, se termine avec une valeur de tableau.length -1 et est incrémentée de 1 à chaque tour. Nous connaissons donc parfaitement le nombre de tours à effectuer dans la boucle, d'où le choix d'une boucle for

plutôt qu'une boucle while (nous aurions pu aussi le faire avec une boucle while mais cela est moins pratique).

Voici un premier exemple qui affiche successivement les valeurs de notre tableau de tout à l'heure :

```
for (int i = 0; i < tableau.length; i++) {
  int elementCourant = tableau[i];
  System.out.println(elementCourant);
}
```

Notre boucle for continue tant que i est inférieur à tableau.length, donc jusqu'à la valeur limite tableau.length -1 (qui pour rappel contient le dernier élément du tableau). A chaque nouveau tour, nous affichons la valeur qui se trouve à l'indice i du tableau. Comme i démarre à 0 puis est incrémenté de 1 à chaque fois, nous affichons successivement tableau[0], tableau[1], tableau[2], etc. jusqu'à tableau[tableau.length -1].

La boucle for améliorée

La construction précédente de la boucle for étant abondamment utilisée et relativement rébarbative, Java nous propose un raccourci intéressant : la boucle for améliorée (en anglais : enhanced for loop) également appelée foreach, dont voici la syntaxe :

```
for (int elementCourant : tableau) {
  System.out.println(elementCourant);
}
```

En réalité, il n'y a aucune différence entre ce code et le précédent. En interne, Java utilise toujours un indice i variant de 0 à length -1. L'avantage de la construction améliorée est que vous n'avez pas à vous soucier de gérer une variable comme i, mais présente deux inconvénients majeurs : vous n'avez pas accès à la valeur de l'indice courant d'une part, et d'autre part vous êtes obligé de parcourir le tableau entièrement et dans l'ordre, ce qui n'est pas forcément ce que vous voulez et qui serait parfaitement possible avec la construction précédente (il suffit de changer la valeur de départ, la borne exprimée dans la condition ou l'incrément). En pratique cependant, nous avons généralement besoin ni de l'un ni de l'autre, ce qui fait que ce raccourci est souvent bien utile.

3.5 ArrayList, ou les tableaux dynamiques

Dans la section précédente, nous avons découvert les tableaux et leur utilisation. Nous avons notamment vu qu'une fois le tableau créé et sa taille définie, il n'est plus possible de la changer. Que faire alors si nous ne savons pas exactement à l'avance combien il y aura d'éléments dans notre tableau ? Il faudrait créer un second tableau plus grand, recopier les éléments du petit tableau dans le grand un par un, ajouter les nouveaux éléments à la fin du grand tableau, et enfin effacer définitivement le petit tableau pour n'utiliser plus que le grand par la suite. Et il faut refaire toutes ces manipulations à chaque fois que nous faisons dépasser la taille que nous pensions être la bonne. Bien sûr, c'est parfaitement envisageable, mais c'est inutilement lourd et peu pratique. La plupart du temps, lorsque nous manipulons des données complexes, il est relativement rare que nous connaissons la quantité à traiter. Bien souvent, nous ne le savons pas avant d'avoir récupéré toutes les données en question, et nous ne devons pourtant ne rien oublier. Une fois de plus, Java nous offre une classe qui fera tout cela pour nous et bien plus : voici ArrayList. Les règles de bonnes pratiques veulent qu'on préfère les tableaux dynamiques aux tableaux classiques qu'on a vu jusqu'ici, qui seraient, dans l'idéal, réservés à des cas où nous connaissons le nombre d'éléments à traiter et où nous savons qu'il ne changera plus ensuite, ou alors pour d'autres contraintes spécifiques comme une rapidité d'exécution accrue par exemple. Nous allons donc dans cette partie de chapitre découvrir les principales méthodes de cette classe. Nous ne verrons que les plus simples. Si vous voulez en savoir plus, vous pouvez vous référer à la documentation officielle sur le web.

3.5.1 Créer un tableau dynamique

Avant de pouvoir utiliser un tableau dynamique ArrayList, il faut créer un nouvel objet de cette classe. Un tableau dynamique ArrayList n'est en fait rien d'autre qu'un objet encapsulant un tableau classique. Cette encapsulation permet de le rendre intelligent et dynamique dans sa taille, sans que l'utilisateur final (nous !) ne se soucie du fonctionnement interne (copie de tableau, ajustement de la taille, etc.) Créons donc un tableau dynamique qui sera destiné à contenir des String par exemple :

```
ArrayList<String> tableau = new ArrayList<String>();
```

Comme nous pouvons le voir, il faut indiquer de quel type seront les données gérées par le tableau dynamique entre deux chevrons, ici nous avons dit que nous utiliserions un tableau de String, c'est ce qui est spécifié. Les traitements s'adapteront en fonction du type choisi, et nous pouvons choisir

le type d'objet que nous voulons à l'exception des types primitifs (comme int, float, double, etc.). Dans la littérature, on dit que le type String est un paramètre de type. Une classe acceptant un ou plusieurs paramètres de type est appelée classe générique. Elle utilise la capacité de généricité de Java, ce qui ne sera pas traité dans ce cours car bien trop complexe. Retenons que nous devons indiquer quel sera le type des données utilisées dans le tableau, nous en resterons là.

Maintenant, nous avons notre tableau dynamique, mais il est encore vide... il est temps de le remplir.

3.5.2 Ajouter des éléments : add

Commençons par le commencement : la méthode add permet d'ajouter de nouveaux éléments en fin de tableau :

```
public void add(E element)
```

Elle prend un seul paramètre, qui est, comme nous pouvons nous en douter, l'objet que nous voulons ajouter. Le type E est une notation généralisée, et est à remplacer par le type effectivement déclaré à la création de l'objet ArrayList. Nous avons précédemment créé un tableau de String, le type E correspond donc ici au type String.

Ajoutons quelques éléments à notre tableau de tout à l'heure :

```
tableau.add("One");
tableau.add("Two");
tableau.add("Three");
tableau.add("Four");
tableau.add("Five");
```

3.5.3 Obtenir la taille du tableau : size

```
public int size()
```

La méthode size s'utilise le plus simplement du monde : elle retourne un int qui contient la taille du tableau, ce qui équivaut exactement à la propriété length que nous avions vue pour les tableaux classiques. Notez qu'elle change en fonction des éléments que nous ajoutons ou enlevons. C'est le but des tableau dynamiques !

3.5.4 Récupérer un élément : get

Que contient le nième élément du tableau ? La méthode get nous donne la réponse :

```
public E get(int index)
```

Ici encore apparaît le paramètre générique E, qu'il faut toujours remplacer par String dans notre cas. Récupérons puis affichons donc le troisième élément :

```
String troisiemeElement = tableau.get(2);
System.out.println(troisiemeElement);
```

Si nous avez bien suivi, cela devrait afficher "Three". Rappelons-nous que le premier élément porte le numéro 0, cela n'a pas changé depuis ce que nous avons appris sur les tableaux classiques.

3.5.5 Modifier un élément : set

```
public E set(int index, E element)
```

La méthode set accepte deux paramètres : d'une part la position de l'élément à modifier, et le nouvel élément à placer dans le tableau d'autre part. Cette méthode retourne l'élément qui se trouvait à cet emplacement avant la modification, ce qui peut parfois être utile.

Modifions le dernier élément du tableau :

```
tableau.set(tableau.size() -1, "Seven");
```

3.5.6 Enlever des éléments : remove

Il existe deux façons différentes d'utiliser cette métode remove :

```
public E remove(int index)
public boolean remove(Object element)
```

La première permet d'enlever l'élément à la position index et retourne ce qui vient d'être supprimé. La deuxième permet d'enlever l'élément désigné par lui-même, et retourne true si un tel élément existe dans l'ArrayList. Fait intéressant, le « trou » qui fait suite à la suppression d'un élément est automatiquement comblé, car les éléments qui le suivent se décalent d'un cran.

Supprimons donc deux éléments de notre tableau pour l'exemple :

```
tableau.remove("Four");
tableau.remove(1);
```

3.5.7 Insérer un élément à une position arbitraire : une deuxième version de add

La méthode add vue précédemment pour ajouter un élément en fin de tableau possède une alternative :

```
public void add(int index, E element)
```

Cette version de la méthode add ajoute l'élément spécifié à la position voulue. tout est prévu pour que nous n'ayons vraiment rien à faire de plus : les éléments qui suivent la position d'insertion sont, comme pour la suppression, automatiquement décalés d'un cran. Il n'est de toute façon pas possible que des éléments partagent la même position, alors nous sommes obligés de les déplacer.

Ajoutons un élément supplémentaire pour essayer :

```
tableau.add(2, "Five");
```

3.5.8 Le retour de la boucle for améliorée

Nous savons maintenant consulter, modifier, ajouter et enlever des éléments comme bon nous semble. Il nous manque encore une opération indispensable : parcourir l'ensemble du tableau, pour l'afficher par exemple. C'est ici que la boucle for fait son retour en force. Nous avons deux alternatives : soit la boucle for normale, soit la version améliorée que nous avons vue rapidement à la fin de la deuxième partie. C'est ici que cette dernière démontre tout son intérêt. Jugeons plutôt la différence :

```
for (int i = 0; i < tableau.size(); i++) {
  String element = tableau.get(i);
```

```
  System.out.println(element);
}
```

Versus :

```
for (String element : tableau) {
  System.out.println(element);
}
```

Outre le fait d'être beaucoup plus lisible, la construction améliorée offre un autre avantage : si un jour les concepteurs de Java décident de renommer la méthode size en length, ou si, plus probable, nous décidons de remplacer votre ArrayList par un tableau classique ou utiliser une autre classe proposant encore plus de fonctionnalités, nous ne serons pas obligé de modifier le code.

Petit test pour voir si nous avons tout compris : essayons de dire ce qui sera affiché par les lignes ci-dessus. La solution figure à la fin de ce chapitre.

3.5.9 Les types primitifs et petites curiosités

Nous avons dit au début de cette section sur les ArrayList que le paramètre de type ne pouvait pas être un type primitif comme int, float, double, etc. En fait ce n'est pas tout à fait exact, il est toujours possible de profiter des avantages de l'objet par les classes Wrapper.

Déclarons pour l'exemple un ArrayList de Integer :

```
ArrayList<Intger> nombres = new ArrayList<Integer>();
```

... que nous pouvons ensuite utiliser par exemple comme ceci :

```
nombres.add(new Integer(-17));
nombres.add(new Integer(2));
nombres.add(new Integer(9));
nombres.add(new Integer(128));
nombres.add(new Integer(441));
```

puis supprimons l'integer ayant la valeur 128.

```
nombres.remove(new Integer(128));
```

Afin de bien comprendre quelle est la méthode utilisé, nous allons reprendre les deux alternatives proposées de remove appliquer à la classe Integer.

```
public E remove(int index)
public boolean remove(E element)
```

Ce qui dans notre cas concernant la classe Integer devient :

```
public Integer remove(int index)
public boolean remove(Integer element)
```

Voici encore la réponse de notre petit test :

```
One
Three
Five
Seven
```

Chapitre 4

Héritage

4.1 Motivation

L'héritage dans notre culture consiste à transmettre à une autre personne l'ensemble ou une partie de ce que nous possédons. Une autre approche de l'héritage consiste en notre héritage génétique, c'est-à-dire que nous transmettons ce que nous sommes. La notion d'héritage en informatique reprend les deux notions précédentes. Une classe héritant d'une autre reçoit non seulement l'ensemble des méthodes, mais également l'accès aux attributs.

4.2 Le modificateur protected

Nous connaissons déjà les mots-clés public et private. L'objectif en mettant public, est de rendre la méthode ou la donnée accessible en dehors de la classe, et private étant de ne pas rendre la méthode ou la données accessible en dehors de la classe. Un élément déclaré en private ne peut pas être hérité par une autre classe pour des questions de sécurité. Sinon, il suffirait de toujours créer une classe héritant de celle qu'on veut pirater. D'autre part, nous ne rendons pas les données publiques. Il nous faut donc trouver un compromis pour les données que nous souhaitons transmettre. Java introduit pour cela un autre modificateur nommé protected. Ce mot-clé, nous permet de rendre une donnée accessible par une sous-classe, mais pas aux autres.

4.3 Hériter d'une classe

La terminologie pour hériter d'une classe A vers une classe B est la suivante :

```
public class B extends A {
  //...
}
```

Le mot extends est le mot-clé permettant l'héritage. Nous devons aussi savoir que toute classe hérite par défaut de la classe Object. Reprenons l'exemple précédent :

```
public class A {
  // ...
}
```

```
public class B extends A {
  //...
}
```

est équivalent à

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

```
public class B extends A {
  //...
}
```

Dans la littérature, on parle souvent de superclasse, sousclasse, classe parent ou encore de classe enfant. Dans notre exemple A est la superclasse de B, ou B est la sous-classe de A. Ou analogiquement, A est la classe parent de B, ou B est la classe enfant de A. Une autre notation consiste à dire que B hérite ou dérive de A.

4.4 Le modificateur final

Il est aussi important de savoir que nous pouvons empêcher l'héritage en déclarant la classe finale avec le mot-clé final. C'est par exemple le cas de la classe Math en Java.

```
public final class Math extends Object {
  // ...
}
```

4.5 Overridden méthode

Nous avons jusqu'à présent observé de manière générale le processus d'héritage. Il nous manque encore la manière dont une méthode est réécrite, on parle aussi d'overridden, ou encore de méthode « overiddée ».

Imaginons l'exemple suivant :

```
public class BonbonCitron {
  private ArrayList composition;

  public BonbonCitron() {
    composition = new ArrayList();
    composition.add("Citron");
    composition.add("Sucre");
  }

  public ArrayList getComposition() {
    return composition;
  }

  public double getCalorie() {
    return 0.23;
  }
}
```

```
public class BonbonCitronLight extends BonbonCitron{
  public BonbonCitronLight() {
  }

  public double getCalorie() {
    // retourne 70% de la valeur du bonbon pas light.
    // Le problème est que si on appelle getCalorie
    // on va appeler cette méthode et cela
    // fera une boucle infinie.
  }
}
```

Le problème, que nous avons, est que nous souhaitons accéder à une méthode dont la signature coïncide exactement avec la méthode que nous sommes en train d'éditer. Nous devons pour cela apprendre un nouveau mot-clé nommé super, pour accéder à un élément de la super-classe. Notre code devient donc :

```
public class BonbonCitron {
  private ArrayList composition;
```

```
  public BonbonCitron() {
    composition = new ArrayList();
    composition.add("Citron");
    composition.add("Sucre");
  }

  public ArrayList getComposition() {
    return composition;
  }

  public double getCalorie() {
    return 0.23;
  }
}
```

```
public class BonbonCitronLight extends BonbonCitron {
  public BonbonCitronLight() {
  }

  // nous décidons de changer le comportement de
  // getCalorie() pour BonbonCitronLight
  public double getCalorie() {
    // retourne 70% de la valeur du bonbon non light.
    return 0.7*super.getCalorie();
  }
}
```

Nous devons encore remarquer que, comme le mot-clé this, le mot-clé super peut être utilisé de manière similaire dans le constructeur. Le constructeur de la superclasse est appelé avec super(), c'est une convention. Tout comme this, nous pouvons aussi rajouter des arguments. Si rien n'est explicité, Java appelle le constructeur par défaut de la superclasse.

4.6 Classe abstraite et modificateur abstract

Il arrive parfois que nous ne savons pas définir convenablement une classe car elle est trop abstraite. En Java, nous disposons du modificateur abstract pour déclarer une classe abstraite.

Une méthode peut être abstract, c'est-à-dire que nous souhaitons que la méthode existe plus tard mais que pour l'instant, le comportement de cette méthode n'est pas défini. Une donnée par contre ne peut pas être déclarée en abstract par convention.

Il est important de remarquer que dès qu'une méthode de la classe est abstraite, alors la classe elle-même est déclarée abstraite. Mais la réciproque

n'est pas vrai, en effet nous pouvons définir une classe abstraite sans qu'il y ait de méthode abstraite. Dans ce cas, on affirme que la classe construite n'est pas suffisamment concrète, malgré le fait qu'elle le pourrait normalement.

Nous allons à présent introduire une superclasse abstraite dans notre exemple.

```
public abstract class Bonbon {
  private ArrayList composition;

  public Bonbon() {
    composition = new ArrayList();
  }

  public ArrayList getComposition() {
    return composition;
  }

  // Actuellement nous ne pouvons pas dire combien
  // de calorie comporte le bonbon ne
  // connaissant pas sa composition.
  public abstract getCalorie();
}
```

Ayant cette superclasse, nous allons modifier nos deux classes vu auparavant.

```
public class BonbonCitron extends Bonbon {
  public BonbonCitron() {
    // le constructeur par défaut de la superclasse est
    // appelé ce qui revient à mettre:
    // super();
    composition.add("Citron");
    composition.add("Sucre");
  }

  // nous rendons concrète la méthode
  public double getCalorie() {
    return 0.23;
  }
}
```

```
public class BonbonCitronLight extends BonbonCitron {
  public BonbonCitronLight() {
  }

  public double getCalorie() {
```

```
    return 0.7*super.getCalorie();
  }
}
```

4.6.1 A quoi ça sert ?

Nous pouvons nous demander à quoi cela sert de définir une classe qui ne nous sert à rien explicitement. La raison est bien dans le mot « explicitement ».

Si un collègues trouve l'implémentation intéressante (dans notre cas ce sera l'exemple des bonbons), il sera aussi peut être intéressé à construire d'autres types d'objets différents mais similaires, qui reprennent en faite les mêmes concepts. Une classe abstraite est une manière de garantir que telle ou telle méthode existera, et donc à posteriori il y aura les mêmes concepts.

4.6.2 Ce qui n'est pas permis

Tout d'abord ce que nous n'osons pas faire, nous présentons une liste non exhaustive :

Instancier un objet de classe abstraite

Le code suivant produira une erreur à la compilation, car il est interdit de créer un objet abstrait. Car par le mot abstract, nous informons au compilateur que la classe n'est pas complètement utilisable ainsi.

```
Bonbon b = new Bonbon();
```

Héberger un objet d'une superclasse

Le code suivant produira une erreur, nous avons bien par la notion d'héritage qu'un bonbon light est un bonbon, néanmoins un bonbon (en général) n'est pas forcément light.

```
BonbonCitronLight b = new BonbonCitron();
```

4.6.3 Ce qui est permis

Nous pouvons aussi faire un certain nombre de chose :

Accéder à une méthode de la superclasse

Si la méthode est accessible en mode public ou protected, alors la machine virtuel de Java vérifiera si « bc » dans le code suivant possède une méthode getComposition(), n'étant pas le cas, elle vérifie dans la superclasse, puis dans la superclasse de la superclasse et ainsi de suite.

```
BombomCitron bc = new BonbonCitron();
bc.getComposition();
```

Héberger un objet d'une sous-classe

Comme vu précédemment, un bonbon light est un bonbon, c'est pourquoi le code suivant est correct.

```
BombomCitron bc = new BonbonCitronLight();
bc.getComposition();
```

4.7 Polymorphisme

Le principe du polymorphisme signifie qu'un élément peut avoir plusieurs « visages », pour cela observons le code suivant :

```
public static double chanceDeFaireUneCarrie(Bonbon b) {
  double p = b.getCalorie()/100000.0;
  if(p > 1.0) p = 1;

  return p;
}
```

Nous pouvons nous poser la question suivante : Qui est concrètement la méthode getCalorie() ? Malheureusement, personne ne pourra nous répondre, voici la raison par un simple exemple.

```
System.out.println(
  chanceDeFaireUneCarrie(new BonbonCitron()));
System.out.println(
  chanceDeFaireUneCarrie(new BonbonCitronLight()));
```

```
0.0000023
0.00000161
```

Nous observons que dans le premier cas, c'est la méthode getCalorie() de la classe BonbonCitron qui est choisie. Puis dans le deuxième cas, c'est la méthode getCalorie() de la classe BombomCitronLight().

Le polymorphisme n'est en faite qu'un mécanisme informatique pour accéder à la méthode de l'objet lui-même, et non pas à la méthode de la référence. Par contre la méthode doit exister dans la classe de la référence.

4.8 Le multi-héritages

Deux visions s'affrontent aujourd'hui concernant le multi-héritages. Dans la première vision, il est possible d'hériter de plusieurs classes en mêmes temps, ce qui peut parfois occasionner certains problèmes avec un ancêtre en commun dans les superclasses. La seconde vision est un processus analogue sans trop de contrainte et ne possèdant pas cette problématique des ancêtres en communs. Le langage Java est basé sur la deuxième vision de l'héritage.

Nous avons à disposition pour cela les interfaces, ce sont des « presque classes ». Elles n'ont pas de constructeur et elles peuvent faire du multi-héritages selon la première vision.

4.8.1 Créer une interface

Nous pouvons créer une interface en utilisant le mot-clé interface au lieu de classe. Nous pouvons aussi y déclarer des méthodes ou des constantes. Nous pouvons également voir une interface comme une sorte de classe abstraite.

Définissons pour notre exemple deux interfaces :

```
public interface CadeauDeNoel {
  public double getPrix();
  public String getDonnateur();
}
```

```
public interface CadeauDeHallowin {
  public double getPrix();
  public int getNoteDePrestationPourAcuisition();
}
```

4.8.2 L'héritage dans une interface

Nous souhaitons également introduire une super interface Cadeau comme suit :

```
public interface Cadeau {
  public double getPrix();
}
```

et reformuler nos deux interfaces comme suit :

```
public interface CadeauDeNoel extends Cadeau {
  public String getDonnateur();
}
```

```
public interface CadeauDeHallowin extends Cadeau {
  public int getNoteDePrestationPourAcuisition();
}
```

On souhaite également faire une interface représentant le plaisir en générale :

```
public interface Plaisir {
  public double getBonheur();
}
```

Nous aimerions aussi faire hériter nos deux interfaces de Plaisir, car cela nous plaît conceptuellement :

```
public interface CadeauDeNoel extends Cadeau, Plaisir {
  public String getDonnateur();
}
```

```
public interface CadeauDeHallowin extends Cadeau, Plaisir {
  public int getNoteDePrestationPourAcuisition();
}
```

Finalement, nous pouvons aisément construire du multi-héritages avec les interfaces sans se poser des problèmes d'implémentation, car il n'y a que des définitions de méthodes ou de constantes. On ne mentionne jamais comment se comporte une méthode, on ne signale finalement que son existance.

4.8.3 Lien existant entre une interface et une classe

Pour une classe, nous savons que nous pouvons hériter d'une seule classe. Nous avons également vu qu'une classe abstraite ou une interface oblige l'existence de méthode par la suite.

En Java le multi-héritages se note sous la forme :

```
public class A extends B implements C, D, E {
  // ...
}
```

où A est une classe, B est la superclasse, et C, D, E sont des interfaces à hériter. Nous observons aussi que pour hériter d'une interface, nous utilisons aussi le mot-clé implements qui signifie « implémente » ou « utilise ». Ainsi les promesses de méthodes doivent être respectées dans la classe, sans quoi elles doivent être mise abstraites.

4.8.4 Hébergement par des références

Il est possible de créer des références à partir d'interface ou de classe abstraite, par exemple nous pouvons écrire le code suivant :

```
public class BonbonCitron extends Bonbon
  implements CadeauDeNoel, CadeauDeHallowin {

  private String from;

  public BonbonCitron() {
    composition.add("Citron");
    composition.add("Sucre");
  }

  public BonbonCitron(String from) {
    this();
    this.from=from;
  }

  public double getBonheur() {
    return 100.25;
  }

  public double getPrix() {
    return 0.05;
  }

  public double getCalorie() {
    return 0.23;
  }

  public String getDonnateur() {
    return from;
```

```
  }

  public int getNoteDePrestationPourAcuisition() {
    return 5; // toujours bien si on donne un bonbon
  }
}
```

```
public class BonbonCitronLight extends BonbonCitron
  implements CadeauDeNoel, CadeauDeHallowin {
  // interfaces déjà présentes dans BonbonCitron
  // mais il n'y a aucun risque de confusion

  public BonbonCitronLight() {
  }

  public BonbonCitronLight(String from) {
    super(from);
  }

  public double getCalorie() {
    return 0.7*super.getCalorie();
  }

  public double getBonheur() {
    // on est déjà un peu moins heureux
    // si on reçoit un bonbon light
    return super.getBonheur()*0.8;
  }
}
```

```
public class Testator {
  public static void main(String[] arg) {
    Bonbon b = new BonbonCitron();
    b.getComposition();

    CadeauDeNoel cdn = new BonbonCitron();
    cdn.getPrix();

    CadeauDeHallowin cdh = new BonbonCitronLight();
    cdh.getNoteDePrestationPourAcuisition()

    Cadeau c = new BonbonCitron();
    c.getPrix();

    Plaisir p = new BonbonCitronLight();
    p.getBonheur();
  }
}
```

4.9 Le casting d'un type

Nous avons vu qu'un objet est toujours un objet de la superclasse, mais l'inverse n'est pas forcément vrai. En réalité « ce n'est pas forcément vrai », car on ne peut pas prédire si le type correspond. Néanmoins, si nous sommes certain que le type correspond, il existe un moyen de forcer la conversion comme nous le montre l'exemple suivant :

```
Bonbon b = new BonbonCitron();

BonbonCitron bc = (BonbonCitron)b;

Plaisir p = (Plaisir)b;
```

L'inconvénient de cette méthode est que si le type ne correspond pas, le programme ne sachant pas comment réagir devra s'arrêter. Java nous propose une solution moins radicale, en faisant un test de compatibilité en premier lieu. Nous avons pour cela le mot-clé *instanceof*, qui signifie littéralement « de même type que ». L'exemple suivant nous montre comment utiliser cet opérateur.

```
Bonbon b = new BonbonCitron();

if(b instanceof BonbonCitron) {
  BonbonCitron bc = (BonbonCitron)b;
}

if(b instanceof Plaisir) {
  Plaisir p = (Plaisir)b;
}
```

Nous connaissons à présent tous les concepts de l'héritage en Java, mais seul une longue pratique est efficace pour réellement maîtriser les concepts.

Chapitre 5

Introduction aux exceptions

5.1 Motivation

Nous savons tous que nous n'osons pas diviser par 0, car mathématiquement c'est une opération interdite. Dans n'importe quel langage de programmation une erreur sera levée, et le cours normal du programme sera interrompu. Aussi si nous devons fournir une méthode division(double i, double j) pour diviser un nombre par un autre, nous sommes bien embêté si le deuxième est zéro. Il nous faudrait donc un moyen d'avertir l'utilisateur qu'un problème est apparu sans pour autant arrêter le programme. Nous appellerons ce processus l'exception.

En Java, nous avons à disposition la classe Exception qui est la superclasse de toutes les types d'exceptions. En soi, une exception ne contient presque rien, si ce n'est un message pour indiquer plus précisément la cause de cette exception.

5.2 Lancer une exception

Une exception se « lance » juste avant que l'erreur n'apparaisse. En général, nous devons faire un petit test si un cas non désirable risque de se produire, comme nous le montre l'exemple suivant :

```
public static double diviseur(double a, double b)
  throws ArithmeticException {
  if(b==0) throw new ArithmeticException();

  return a / b;
}
```

Nous y remarquerons deux mots-clés *throw* et *throws*, le premier pour effectivement lancer une exception, le second pour avertir les utilisateurs

qu'une exception de type ArithmeticException peut être lancée. Nous verrons au prochain point comment attraper une tel exception.

5.3 Attraper une exception

Nous attrapons une exception lorsque chaque étape d'un bout de programme est surveillé, c'est pourquoi nous avons trois blocs à disposition : try - catch - finally.

Le bloc try : tente de faire ce qui est demandé, si une exception est lancée, le cours normal s'interrompt et le bloc catch est appelé.

Le bloc catch : intercepte une exception, il est alors temps de continuer avec un code alternatif. Il peut y avoir plusieurs blocs catch les uns à la suite pour traiter différentes sources de problème.

Le bloc finally : s'exécute dans tout les scénarios possibles, il est nécessaire de finir avec le code dans ce bloc. Ce bloc est facultatif.

Voici un mini-exemple pour schématiser :

```
try {
  // Plan A
}
catch(ArithmeticException e) {
  // Le plan A a échoué, et la cause est donnée par
  System.out.println(e.getMessage());

  // On peut aussi tenter de faire un plan B, au lieu de
  // simplement afficher la raison de l'échec du plan A
}
catch(Exception e) {
  // Hmmm, un autre problème est apparu. Si le problème
  // fut du type ArithmeticException, il aurait été attrapé
  // avant.
}
finally {
  // Dans les deux cas, nous avons fait du mieux que l'on
  // pouvait et l'on souhaite conclure avec le contenu
  // dans ce bloc.
}
```

5.4 Relancer une exception

Imaginons la situation suivante, nous avons implémenté une méthode solveLinearEquation(double a, double b) qui représente l'équation « $ax = b$ », nous savons aussi que la solution est de la forme $x = \frac{a}{b}$. Nous aimerions donc logiquement utiliser notre méthode diviseur(double a, double b). Nous

savons également que si b est zéro, une exception est lancée, nous aimerions donc l'attraper, puis lancer une autre exception expliquant à l'utilisateur que cette équation ne peut pas être résolue. Nous allons pour cela créer une classe EquationNotSolvableException qui hérite d'Exception et lancer un objet de ce type dans notre méthode.

```
public class EquationNotSolvableException
  extends Exception {
}
```

```
public class MyMathClass {
  public static double
    diviseur(double a, double b)
    throws ArithmeticException {

    if(b==0) throw new ArithmeticException();
  }

  public static double
    solveLinearEquation(double a, double b)
    throws EquationNotSolvableException {

    double x = 0.0;

    try {
      x = diviseur(a, b);
    }
    catch(ArithmeticException e) {
      throw new EquationNotSolvableException();
    }

    return x;
  }
}
```

5.5 Les exceptions - cause d'erreurs de conception

Observons le code suivant et imaginons ce qu'il fait :

```
public static double
  solvePositiveLinearEquation(double a, double b)
  throws EquationNotSolvableException {

  try {
    double x = solveLinearEquation(double a, double b);
```

```
    if(x < 0.0) {
      throw new EquationNotSolvableException();
      // cette exception sera elle aussi attrapée dans le
      // bloc catch ci dessous.
    }
    return x;

  }
  catch(EquationNotSolvableException e) {
    throw new EquationNotSolvableException();
  }
  finally {
    System.out.println("Youhou!");
    System.out.print("Je peux écrire n'importe quoi,");
    System.out.println(" ce texte ne sera jamais affiché!");
  }

  return 0.0;
}
```

Nous devons malheureusement accepter le fait que dans tous les cas, le texte dans le bloc finally sera affiché. Nous avons vu tout au début que le bloc finally est toujours exécuté. S'il y a un return avant que le bloc finally ait été appelé, alors le code dans le bloc finally est exécuté et ce n'est qu'à ce moment-là que le return est effectivement fait. Le même raisonnement est fait avec une exception lancée dans le bloc try ou catch.

Chapitre 6

Flux de fichier

6.1 Motivation

Jusqu'à présent nous avons uniquement travaillé avec la mémoire mise à disposition par l'ordinateur. Notre problème est que actuellement nous ne pouvons pas sauvegarder des données sur le disque dur. Ce chapitre nous proposent une introduction dans les manières d'écrire des fichiers qui pourront être lu par la suite.

Nous appellerons flux ou flot de fichier, la manière dont l'information est écrite. En informatique, nous distinguons deux grandes familles de flux de fichier, les flux binaires et textuels. Cependant nous devons bien comprendre qu'il n'y a que des 1 et des 0 qui sont écrits sur le disque dur, c'est pourquoi il est impératif de comprendre que seul l'interprétation de la donnée est différente.

Par exemple, nous avons vu dans les types primitifs qu'un integer s'étend sur 4 bytes et qu'un char s'étand sur 1 byte. Imaginons que nous souhaitons écrire le chiffre 1 dans un fichier. Si nous voulons écrire le caractère '1', nous allons écrire un nombre binaire sur 1 byte qui a comme représentation '1', alors que si nous voulons écrire l'integer 1, nous allons écrire le nombre binaire 1 sur 4 bytes ce qui fait 32 positions en base 2, c'est-à-dire 00000000000000000000000000000001.

Vu que nous avons une interprétation sur la manière d'avoir sauvegardé la donnée, il est logique d'avoir la même interprétation pour la récupérer. C'est pourquoi dans la littérature on nous explique qu'un flux en écriture doit être du même type en lecture.

6.2 La jungle des flux en Java

En Java, il y a près de 60 types de flux, pour écrire ou lire, pour un flux binaire ou textuel, pour un flux avec mémoire tampon intermédiaire (ce qu'on appelle buffer dans la littérature) ou non, et pour différents types de

destinations (fichier physique ou fichier simulé via une mémoire spécialement allouée). En bref, c'est la jungle !

Néanmoins, depuis Java 1.5, la classe Scanner est apparue. Elle complémente une autre classe déjà présente appelée PrintWriter. Avec ces deux classes nous pouvons faire de la gestion de flux très facilement. D'ailleurs la manière de gérer les flux a complètement changée depuis Java 1.5 avec l'apparition de la classe Scanner.

6.3 La gestion et création de flux - La nouvelle philosophie

Nous allons apprendre à manipuler les flux des classes Scanner et PrintWriter, puis comprendre le principe de la spécialisation des flux. En faisant la synthèse de ces deux points, nous aurons la possibilité de construire n'importe quel flux, selon nos besoins.

6.3.1 La classe File

Java dispose d'une classe File, cela nous permet d'avoir un point d'accès au disque dur. Néanmoins, le fichier hébergé dans l'objet de cette classe est logique, et donc pas forcément physique.

Il se peut qu'un fichier n'existe pas, comme nous le montre l'exemple suivant, car il est en effet peu probable que Windows intègre un répertoire Linux dans son répertoire système. Néanmoins, Java ne nous empêche pas de créer un objet avec comme destination de fichier C :/Windows/Linux/linux.bin .

```
File f = new File("C:\Windows\Linux\linux.bin");
```

Nous avons pour éviter ce problème, une méthode qui nous permet de vérifier si le fichier existe à l'endroit spécifié. La méthode s'appelle exists() et renvoie true si le fichier existe et false sinon.

```
File f = new File("C:\Windows\Linux\linux.bin");
if( ! f.exists()) {
  System.out.println("Le fichier n'existe pas.");
}
```

Afin de simplifier la structure en Java, un dossier est aussi un « fichier », c'est pourquoi Java propose deux autres méthodes isFile() et isDirectory(). Dans le cas où le fichier respectivement le répertoire n'existerait pas, la

méthode renvoie false comme nous le montre l'exemple suivant.

```
File f = new File("C:\Windows\Linux\linux.bin");
if(f.isFile()) {
  System.out.println("C'est un fichier.");
}

File d = new File("C:\Windows\Linux");
if(d.isDirectory()) {
  System.out.println("C'est un dossier.");
}
```

A présent que nous avons vu les spécificités de la classe File nous pouvons apprendre à placer un flux de fichier sur un fichier.

6.3.2 La classe PrintWriter

En premier, nous devons instancier un objet. Nous appelons donc un des constructeurs que Java nous proposent, parmi eux il y a les suivants :

```
public PrintWriter(String fileName)
public PrintWriter(File file)
public PrintWriter(OutputStream out)
public PrintWriter(Writer out)
```

Nous devons remarquer qu'appeler le premier constructeur, ne va en fait appeler que le deuxième. En effet,

```
PrintWriter monFlux = new PrintWriter("HelloWorld.txt");
```

est équivalent à

```
PrintWriter monFlux =
  new PrintWriter(new File("HelloWorld.txt"));
```

Les deux autres constructeurs seront utilisés lors de spécialisation, nous reviendrons un plus tard sur ce sujet. Nous allons à présent explorer les possibilités que la classe PrintWriter nous fournie. Nous allons apprendre comment écrire une donnée et comment la classe la gère, mais également apprendre à travailler avec un minimum de ressources, car elles sont souvent limitées par votre système opérationnel.

1er principe

Un flux est une ressource limitée du système. Il est donc nécessaire de lui rendre aussi vite que possible. En d'autres termes à partir du moment que nous n'en avons plus besoin nous redonnons la ressource au système. Ainsi, un flux après n'avoir plus aucune utilité doit être fermé, pour cela nous disposons de la méthode close().

```
PrintWriter monFlux =
  new PrintWriter(new File("HelloWorld.txt"));
// J'écris plein de données sur HelloWorld.txt
// Je n'ai à présent plus besoin de mon flux
monFlux.close();
```

2ème principe

Un flux de type PrintWriter possède une mémoire tampon, ainsi ce qu'on croit écrire dans le fichier ne l'est pas automatiquement. En effet, les accès au disque sont « coûteux ». Il est donc préférable d'écrire plusieurs données à la suite en une seule fois. C'est pourquoi dans le cas où nous jugeons qu'il est nécessaire ou impératif d'écrire les données dans le fichier, il nous faut vider la mémoire tampon. Cette méthode s'appelle flush(). La méthode close() appelle aussi flush() avant de clôturer définitivement le flux.

```
PrintWriter monFlux =
  new PrintWriter(new File("HelloWorld.txt"));
// J'écris plein de données sur HelloWorld.txt

monFlux.flush();
// J'ai à présent la garantie que tous ce qui fût
// écrit l'est effectivement sur le fichier.

// J'écris encore quelques données.

// Je n'ai à présent plus besoin de mon flux, monFlux
// vide sa mémoire tampon avant de se clôturer.
monFlux.close();
```

Les méthodes pour écrire

Pour pouvoir écrire une donnée nous disposons de deux méthodes print et println, la deuxième étant comme la première ajoutant en plus une ligne supplémentaire dans le fichier.

Voici un éventail des méthodes à disposition :

```
public void print(boolean b)
public void print(char c)
public void print(double d)
public void print(float f)
public void print(int i)
public void print(long l)
public void print(String s)
```

ainsi que les méthodes ajoutant en plus un saut de ligne :

```
public void println(boolean b)
public void println(char c)
public void println(double d)
public void println(float f)
public void println(int i)
public void println(long l)
public void println(String s)
```

Nous avons aussi à disposition une méthode println() qui ne fait qu'un saut de ligne, elle est très utile pour ajouter un saut de ligne après avoir écrit plusieurs données sur une même ligne.

```
PrintWriter monFlux =
  new PrintWriter(new File("HelloWorld.txt"));
monFlux.print(12);
monFlux.print(true);
monFlux.print('c');
monFlux.println();

monFlux.flush();

monFlux.print("Toto");
monFlux.println(14);
monFlux.close();
```

Le code source suivant produira le contenu de fichier suivant :

```
12truec
Toto14
```

Nous avons acquis à présent une première approche pour l'écriture d'un fichier, qui consiste en les deux principes ainsi que l'écriture proprement dite. Actuellement, nous n'avons pas encore en notre connaissance tous les concepts pour pouvoir relire notre fichier et ainsi récupérer nos données. Ce sera notre principal objectif lors des trois points suivants.

6.3.3 La classe Scanner

Nous savons déjà que cette classe est d'une part récemment apparue, mais aussi qu'elle nous permettra de lire des données préalablement écrites.

En comparaison avec la classe PrintWriter, nous dirons que le premier principe est toujours valable, mais que le deuxième se fait de manière camoufler. En effet, l'objet Scanner lit en avance une partie des données, et y fait accès lors de nos requêtes. Dans la même philosophie, il n'y a pas de sens à avoir une méthode flush().

A nouveau, Java nous propose plusieurs constructeurs :

```
Scanner(String source)
Scanner(File source)
Scanner(InputStream source)
Scanner(Readable source)
```

Il est nécessaire de remarquer que :

```
Scanner monFlux = new Scanner("HelloWorld.txt");
```

n'est pas du tout équivalent à

```
Scanner monFlux = new Scanner(new File("HelloWorld.txt"));
```

Nous allons encore observer un peu plus en profondeur comment, le constructeur Scanner(String str) est utilisé.

```
Scanner monFlux = new Scanner("13ctrue45");
int i = monFlux.nextInt(); // i=13
char c = monFlux.nextChar(); // c='c'
boolean b = monFlux.nextBoolean(); // b=true
long l = monFlux.nextLong(); // l=45
```

Les méthodes pour lire

Nous allons à présent voir plus en profondeur comment l'on récupère les données, mais il est aussi important de vérifier que la donnée que nous allons lire est effectivement du type que nous attendons.

```
public boolean hasNextBoolean()
public boolean hasNextByte()
public boolean hasNextDouble()
public boolean hasNextFloat()
public boolean hasNextInt()
public boolean hasNextLine()
public boolean hasNextLong()
public boolean hasNextShort()
```

Dans le cas où la réponse à cette question est true, nous devons extraire la donnée avec la méthode correspondante :

```
public boolean nextBoolean()
public byte nextByte()
public double nextDouble()
public float nextFloat()
public int nextInt()
public String nextLine()
public long nextLong()
public short nextShort()
```

Ainsi par exemple, nous pouvons lire un integer puis un boolean :

```
Scanner monFlux = new Scanner(new File("HelloWorld.txt"));
int i = 0;
boolean b = false;
if( monFlux.hasNextInt() ) {
  i = monFlux.nextInt();
}

if( monFlux.hasNextBoolean() ) {
  b = monFlux.nextBoolean();
}
monFlux.close();
```

Cependant, nous souhaitions lire les deux données en même temps, ce qui n'est malheureusement pas le cas si par exemple ce n'est pas un integer qui

est lu en premier. Nous devons être plus vigilant, c'est pour cela que dès qu'une attente n'est pas satisfaite, nous devons quitter le cheminement normal. Voici une première approche :

```
Scanner monFlux = new Scanner(new File("HelloWorld.txt"));
int i = 0;
boolean b = false;
if( monFlux.hasNextInt() ) {
  i = monFlux.nextInt();
  if( monFlux.hasNextBoolean() ) {
    b = monFlux.nextBoolean();
  }
}
monFlux.close();
```

La deuxième approche consiste à lire aveuglement les données sans se préoccuper de la justesse du type, si une méthode rencontre un problème une exception est alors lancée.

```
try {
  Scanner monFlux = new Scanner(new File("HelloWorld.txt"));

  int i = monFlux.nextInt();
  boolean b = monFlux.nextBoolean();

  monFlux.close();
}
catch(Exception e) {
  // Un problème est apparu.
}
```

6.3.4 La spécialisation d'un flux

La spécialisation d'un flux consiste à rajouter des étapes intermédiaires dans le traitement de la donnée entre le moment de lecture/écriture dans le fichier et le moment de mise en mémoire. Nous connaissons déjà deux types de traitement : binaire ou à caractère, mais il y a également avec ou sans mémoire tampon supplémentaire, ainsi que sur fichier ou autre (par exemple sur une mémoire émulée).

Voici un exemple de spécialisation farfelu mais possible :

```
PrintWriter pw =
  new PrintWriter (
    new DataOutputStream (
```

```
        new BufferedOutputStream (
          new FileOutputStream(
            new File("dummystream.txt")))));
```

Nous avons ici, des flux imbriqués les uns dans les autres, lorsque nous appelons une méthode, le flux appelle la/les méthodes correspondante(s) et ainsi de suite. Ce qui permet à chacun des flux de faire le travail qui lui est propre puis de transmettre l'information plus loin.

6.3.5 La sérialisation d'une donnée

Nous allons à présent apprendre le principe d'une donnée sérialisée. Auparavant, nous avions le souhait qu'une donnée écrite puisse être à son tour être lue sans perte, ni superflux, ce principe s'appelle la sérialisation. Nous allons apprendre donc à éviter les pièges qui nous font perdre la sérialisation ainsi qu'un temps précieux lorsque nous ne reconnaissons pas immédiatement la source du problème.

Le type de flux

Nous devons absolument avoir des types de flux entrant et sortant de même sorte. Dans le cas contraire, il n'y aucune garantie que la donnée sera interprétée de la même façon à l'écriture qu'à la lecture.

Le retour de ligne non traité

Nous avons tendance à écrire une donnée avec une méthode println(... data) pour un type voulu. Et nous souhaitons relire le type avec la méthode next...(). Or, nous avons oublié de récupérer le caractère de fin ligne dans le fichier ! Il faut pour cela encore appeler nextLine() pour lire ce dernier caractère.

```
PrintWriter out =
  new PrintWriter(new File("HelloWorld.txt"));
out.println(16);
out.println(true);
out.print(132);
out.println("MyEndString");
out.close();

Scanner in = new Scanner(new File("HelloWorld.txt"));
int i = in.nextInt();
in.nextLine();
boolean b = in.nextBoolean();
in.nextLine();
```

```
int j = in.nextInt();
String str = in.nextLine();
in.close();
```

Le problème avec les types numériques

Le problème avec les types numériques n'apparait qu'avec les flux à caractère, cependant vu que nous pouvons changer la spécialisation en une ligne, nous devons nous en occuper pour tout les flux en prévention d'une manipulation ultérieure du code.

```
PrintWriter out =
  new PrintWriter(new File("HelloWorld.txt"));
out.print(34);
out.print(21);
out.close();
```

produira le contenu suivant dans le fichier :

```
3421
```

Notre problème est que si nous lisons notre sortie de fichier comme nombre, il n'en verra logiquement qu'un seule. C'est la raison pour laquelle nous utilisons la méthode ajoutant en plus une fin de ligne. En d'autres termes :

```
PrintWriter out =
  new PrintWriter(new File("HelloWorld.txt"));
out.println(34);
out.println(21);
out.close();
```

qui nous produit la sortie suivante, laquelle permet de lire les deux nombres distinctement :

```
34
21
```

Conclusion

Pour aller plus loin

Parce que l'univers de Java est vaste et complexe, parce qu'il est difficile de s'y retrouver seul, et parce qu'on apprend par la pratique et jamais mieux qu'en autodidacte, voici une série de liens vers d'autres tutoriels ou documentations qui vous permettront d'approfondir vos connaissances :

- http://pagesperso-orange.fr/emmanuel.remy/ (fr) L'excellent site personnelle d'un ingénieur qui prépare des tutoriels pour Java, mais aussi pour C++.
- Au coeur de Java 2 - Volume 1- Notions fondamentales, *Cay S. Horstmann , Gary Cornell* (fr)
- Java - tête la première, *Bert Bates, Kathy Sierra* (fr)
- http://java.developpez.com/livres/javaEnfants/ (fr) Java pour les enfants, les parents et les grands-parents. Un cours similaire à celui-ci qui reprend les bases et va plus loin avec la programmation d'interfaces graphiques.
- http://java.sun.com/docs/books/tutorial/ (en) The Java Tutorial. Le tutoriel officiel de Sun, qui couvre pratiquement tous les aspects du langage, depuis les bases de la programmation orientée objet jusqu'à la programmation d'interfaces graphiques en passant par la gestion du multimédia et du réseau.
- http://java.developpez.com/faq/java/ (fr) FAQ Java contenant des questions-réponses divers sur les éléments du langage.
- http://java.sun.com/javase/6/docs/api/ (en) L'indispensable documentation officielle de Java 6. Même si la documentation est une vraie jungle, elle permet pratiquement toujours de trouver la classe ou la méthode qui nous manque, et permet parfois de découvrir des fonctionnalités méconnues. Elle est donc bien plus qu'un simple aide-mémoire qui renseigne sur les paramètres des méthodes et l'utilisation des classes.

Le mot de la fin

Nous arrivons au terme de nos 154 minutes que nous nous étions initialement fixées. Peut-être avez-vous mis plus, ou moins de temps. Peut-être ce manuel a tenu ses promesses et vous avez effectivement appris les bases de Java en une seule après-midi, ou bien il ne les a pas tenues et vous avez au contraire pris beaucoup plus de temps car certains points n'étaient pas clairs. Dans tous les cas, n'hésitez pas à relire les passages que vous n'avez pas bien compris, car il est vraiment très important que vous maîtrisiez les fondations, afin que rien ne s'écroule lorsque vous déciderez d'ajouter des étages supplémentaires. Ce n'est pas grave de ne pas tout réussir du premier coup. L'expérience ne s'acquiert de toute façon pas en un jour et la seule manière de progresser est de pratiquer. Comme son nom l'indique, ce manuel d'introduction n'est hélas qu'une introduction et ne propose de loin pas suffisamment de pratique pour pouvoir prétendre connaître Java dans les détails. Nous l'avions dit dès le départ, ce n'était pas notre but, loin s'en faudrait pour y parvenir. Quoi qu'il en soit, nous sommes cependant persuadés qu'une fois arrivé ici, vous ne vous arrêterez pas en si bon chemin et que vous chercherez à en savoir plus sur ce fabuleux langage qu'est Java. Ce n'est plus le moment de vous décourager : il est maintenant temps de passer à la vitesse supérieure et de commencer à faire des programmes véritablement intéressants.

Tel Cristophe Colomb, vous venez tout juste de débarquer sur un nouveau continent. Un nouveau monde plein de satisfactions et de plaisir s'offre dès à présent à vous. Bien sûr, il y aura aussi des surprises et des épreuves à surmonter, mais rien n'est impossible, car vous connaissez maintenant les indispensables pour bien vous orienter dans cet environnement aux dimensions presque infinies. Bonne conquête !