Support de cours Java

Structures de données
Notions en Génie Logiciel
et Programmation Orientée Objet

H. Mounier

Université Paris Sud

Notations

Les sigles suivants seront fréquemment utilisés

☛	Point notable, auquel il faut prêter attention
➶	Point positif, agréable, du langage
➴	Point négatif, désagréable, du langage
$\Rightarrow$	Implication logique

- Tout code source java sera écrit dans une police particulière, `type courier`.
- Une notion définie, expliquée ou précisée apparaîtra *comme ceci*.
- Des termes jugés importants apparaîtront **comme ceci**.
- Des termes jugés cruciaux apparaîtront **comme ceci**.

Table des matières

Préface

Ces notes de cours rassemblent des éléments de base du langage Java. Les chapitres I, II et III sont introductifs ; les chapitres V et VI introduisent des concepts généraux essentiels en programmation, indépendamment de tout langage ; enfin les chapitres IV, VII, VIII, IX et X fournissent de manière concrète les bases du langage.

Après les trois premiers chapitres qui donnent des aperçus généraux sous divers angles, le chapitre IV expose les bases purement procédurales (c.à.d. non orientée objet) du langage. Au chapitre V des notions de génie logiciel génériques sont exposées. Le chapitre suivant contient des définitions précises de ce que sont une classe, un objet, ainsi que les relations (notamment l'héritage) qui les relient. La substance concrète de ce qui est décrit au chapitre VI fait l'objet du chapitre VII. Le mécanisme d'exceptions de Java est ensuite exposé au chapitre VIII. Diverses classes utilitaires simples sont données au chapitre IX. Enfin, le chapitre X concerne les classes de Java implantant diverses structures de données (telles les tableaux dynamiques, les listes, les tables de hachage, les arbres) et algorithmes (tels le tri) associés.

I – Entrée en matière

Références bibliographiques

Java examples in a Nutshell, D. Flanagan, 2ième édition [Flaa]

I.1 Qu'est-ce que Java, en trois lignes

Le début de l'ouvrage de référence, *The Java Language Specification* par J. Gosling, B. Joy et G. Steele [GJS96] résume fort bien l'esprit dans lequel le langage a été conçu ainsi que le but poursuivi : "JAVA is a general purpose, concurrent, class-based, object-oriented language. It is designed to be simple enough that many programmers can achieve fluency in the language. Java is related to C and C++ but is organized rather differently, with a number of aspects of C and C++ ommited and a few ideas from other languages included. Java is intended to be a production language, not a research language, and so, as C.A.R. Hoare suggested in his classic paper on language design, the design of Java has avoided including new and untested features."

I.2 Exemples de "Hello World" en différents langages

2.1 Avec OSF/Motif widgets

```
#include <X11/Intrinsic.h>
#include <X11/StringDefs.h>
#include <Xm/Xm.h>
#include <Xm/Form.h>
#include <Xm/Label.h>
#include <Xm/PushB.h>
```

```
typedef struct APP_DATA {
    char    *mtext;
    char    *etext;
} APP_DATA, *P_APP_DATA;

static XrmOptionDescRec options[] = {  /* options de la ligne de commande */
    {"-mtext", "*mtext", XrmoptionSepArg, NULL},
    {"-etext", "*etext", XrmoptionSepArg, NULL}
};
static XtResource resources[] = {      /* ressources */
    {"mtext", "Mtext", XtRString, sizeof(String),
     XtOffset(P_APP_DATA, mtext), XtRString, "Maison pauvre, voie riche"},
    {"etext", "Etext", XtRString, sizeof(String),
     XtOffset(P_APP_DATA, etext), XtRString, "Quitter"}
};

static Arg args[10];                   /* arguments passes aux widgets */
static void quit_action(Widget w, caddr_t client_data,
                        XmAnyCallbackStruct *call_data);

void main(int argc, char *argv[]) {
    APP_DATA data;
    Widget   main_widget, form_widget, hello_message, exit_button;

    main_widget = XtInitialize(argv[0], "Xmhello", options,
                               XtNumber(options), &argc, argv);
    XtGetApplicationResourceds(main_widget, &data, resources,
                               XtNumber(resources), NULL, 0);
    form_widget = XtCreateManagedWidget("Form",
                    xmFormWidgetClass, main_widget, NULL, 0);
    XtSetArg(args[0], XmNtopAttachment, XmATTACH_FORM);
    XtSetArg(args[1], XmNleftAttachment, XmATTACH_FORM);
    XtSetArg(args[2], XmNLabelString,
             XmStringCreateLtoR(data.etext,
                                XmSTRING_DEFAUKT_CHARSET));
    exit_button = XtCreateManagedWidget("Exit",
                    xmPushButtonWidgetClass, form_widget,
                    (ArgList) args, 3);

    XtAddCallback(exit_button, XmNactivateCallback,
                  quit_action, NULL);
    XtSetArg(args[0], XmNtopAttachment, XmATTACH_WIDGET);
    XtSetArg(args[1], XmNtopWidget, exit_button);
```

```
    XtSetArg(args[2], XmNleftAttachment, XmATTACH_FORM);
    XtSetArg(args[3], XmNrightAttachment, XmATTACH_FORM);
    XtSetArg(args[4], XmNbottomAttachment, XmATTACH_FORM);
    XtSetArg(args[5], XmNLabelString,
             XmStringCreateLtoR(data.mtext,
                                XmSTRING_DEFAUKT_CHARSET));
    hello_message = XtCreateManagedWidget("Hello",
                    xmLabelWidgetClass, form_widget,
                    (ArgList) args, 6);
    XtRealizeWidget(main_widget);
    XtMainLoop();
}

static void quit_action(Widget w, caddr_t client_data,
                        XmAnyCallbackStruct *call_data) {
    XtCloseDisplay(XtDisplay(w));
    exit(0);
}
```

- Avantage : **souple**.
- Désavantages : **code long**, **ne fonctionne que sous XWindows** (pas sur Macintosh ou Windows 98).
- Remarque : code **3 fois plus long en Xlib**.

2.2 Hello world en Tcl/Tk

- Tcl : langage de commandes interprété, peu structuré. Tk : bibliothèque graphique.
- Code d’une fenêtre simple

```
proc helloWorld {
    toplevel .helloworld
    label .helloworld.label -text "La haine seule fait des choix"
    button .helloworld.button -text "Quitter" -command exit
    pack .helloworld.label .helloworld.button
    }
```

- Avantage : **codage** de **mini** applications **simple**.
- Désavantage : Langage de **mauvaise qualité** (au sens du génie logiciel) et lent.

2.3 Hello world en Java

- Code d’une fenêtre simple en Java

```
import java.awt.*
import java.awt.event.*

class HelloWorld extends CloseableFrame {
    public void paint(Graphics g) {
        this.setLayout(new FlowLayout(FlowLayout.CENTER, 15, 15));
        button b = new Button("Quitter");
        this.add(b);
        b.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent a) {
                  System.exit(0); } });
        g.drawString("Jour apres jour, c'est un bon jour", 75, 100);
    }

    public static void main(String args[]) {
        Frame f = new HelloWorld();
        f.show();
    }
}
```

- Avantages :
  - Code très **compact** (3 fois plus court qu'en OSF/Motif, 6 fois plus court qu'en Xlib).
  - Langage de **bonne qualité** (en génie logiciel).
  - Nécessairement **orienté-objet**.
  - **Fonctionne sans modifications** sous UNIX, Windows 98/NT, MacOS.

I.3 Un autre exemple

3.1 Applet de gribouillage Java

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

public class Scribble extends Applet {
  int last_x;
  int last_y;

  public void init() {
    MouseListener ml = new MyMouseListener(this);
    MouseMotionListener mml = new MyMouseMotionListener(this);
    this.addMouseListener(ml);
```

```
    this.addMouseMotionListener(mml);
  }
}

class MyMouseListener extends MouseAdapter {
  private Scribble scribble;
  public MyMouseListener(Scribble s) { scribble = s; }
  public void mousePressed(MouseEvent e)  {
    scribble.last_x = e.getX();
    scribble.last_y = e.getY();
  }
}

class MyMouseMotionListener extends MouseMotionAdapter {
  private Scribble scribble;
  public MyMouseMotionListener(Scribble s) { scribble = s; }
  public void mouseDragged(MouseEvent e) {
    Graphics g = scribble.getGraphics();
    int x = e.getX(), y = e.getY();
    g.drawLine(scribble.last_x, scribble.last_y, x, y);
    scribble.last_x = x; scribble.last_y = y;
  }
}
```

II – Historique et propriétés de Java

Références bibliographiques

Understanding O-O Programming with Java,

T. Budd, [Bud98],

II.1 Propriétés du langage ; Technologies disponibles

1.1 Oak

- Originellement **Oak**, 1991, *James Gosling*, **Sun Microsystems**.
- But de Oak : *langage embarqué* pour des appareils de communication (téléphone, télévision, ordinateurs, ...).
- 2 caractéristiques cruciales de ce langage embarqué :
  - **taille réduite** (⇒ codage compact).
  - **fiabilité** (⇒ fonctionnement en mode dégradé, en réponse à des exceptions).

1.2 Propriétés embarquées

Plusieurs propriétés de java reflètent ceci :

- Langage **réduit et simple** (notamment en ce qui concerne lesinstructions).
- Peut être transformé en une **représentation interne compacte**.
- ✗ *Pointeurs* et `goto` eliminés.
- Traitement d'exceptions partie intégrante du langage ; le programmeur est souvent **FORCÉ de gérer les exceptions** (c'est une bonne chose !).

1.3 Naissance de Java

- Java hérite principalement sa syntaxe (procédurale) du C.
- Langage généraliste, aussi versatile que C++.
- Plusieurs **simplifications notables** par rapport au C++.
- Très vaste bibliothèque de classes standard (plus de 3000 classes dans plus de 160 paquetages pour le JDK 1.5)
- A partir de 1993, chez Sun, développement pour créer un **langage adapté à Internet**.
- En **1995**, annonce offcielle de Java (concu, entre autres, par James Gosling, Patick Naughton, Crhis Warth, Ed Frank, Mike Sheridan et Bill Joy).
- Milieu **1996**, sortie de Java 1.02, première version distribuée par JavaSoft (filiale de Sun).
- Début **1997**, sortie de Java 1.1. **Beaucoup d'améliorations significatives**. Première version à être jugée sérieuse du langage.
- Été **2004**, sortie de Java 1.5 ; diverses améliorations et ajouts intéressants.

1.4 Adaptation au web (I)

- Pourquoi le caractère "embarqué" initial de java est-il **bien adapté au Web** (transfert de pages HTML et exécution de programmes distante *via* Internet) ?
- Le schéma client/serveur classique est le suivant :[-2.5ex]
  - **envoi de requête** du client vers le serveur,
  - **traitement de la requête** par le serveur,
  - **envoi de la réponse** du serveur au client.

1.5 Adaptation au web (II)

Inconvénients de ce schéma :[-2.5ex]

- **temps de transmission** souvent lents.
- Les **serveurs** peuvent être **chargés** (beaucoup de clients à servir).
- les **clients** sont, par contraste, assez fréquemment **peu chargés**.

1.6 Adaptation au web (III)

⇒ *Calculs coté client via* des *applets* : plutôt que d'exécuter le programme et de transmettre la réponse, le serveur **transmet le programme**. Le programme s'exécute localement sur le client. Ainsi :

[-3ex]

- le programme s'exécute sur une machine **moins chargée**,

- le **seul retard** est le **temps de transmission initial** du programme. Lorsqu'il y a plusieurs requêtes, la 1^{ere} prend du temps, les suivantes ne souffrent pas du transfert *via* Internet.

1.7 Interpéteurs de bytecode

- Client et serveur peuvent être sur 2 ordinateurs (processeurs) de types différents, avec des OS différents. ⇒ Le source java doit être traduit en un *bytecode* **indépendant de la plate forme** logicielle et matérielle. Ce bytecode (code dont les instructions sont longues d'1 ou 2 octet) est un langage sur une machine imaginaire, une *machine virtuelle*. Ressemble à un assembleur générique.
- Transformation du bytecode en code machine *via* : [-3ex]
  - des interpréteurs.
  - des **compilateurs** "juste-à-temps" (*JIT* : Just In Time) de performances plus proches d'un exécutable C ou C++ classique.
- Apparition de **compilateurs natifs**, c.à.d. transformant du code source Java en code machine natif pour tel ou tel processeur (jove ; cygnus, au dessus de gcc, ...).

1.8 Sécurité

- Un programme s'exécutant sur un serveur ne peut faire beaucoup de dégâts sur la machine client. Un programme s'exécutant coté client peut, en théorie, avoir accès à beaucoup de ressources, d'où un danger.
- ⇒ *gestionnaire de sécurité*, sur le client, **limitant les actions possibles** du programme envoyé par le serveur. Par ex., interdiction d'accéder au système de fichiers ou de transmettre à d'autres que le client ou le processeur du serveur.

II.2 Manifeste en 11 points

2.1 "White paper description"

Java est un langage :
- **simple**,
- **orienté objet**,
- **réparti**,
- **interprété (ou compilé)**,
- **robuste**,

- **sûr**,
- **indépendant de l'architecture**,
- **portable**,
- **efficace**
- **multitâches ou multi-activités (multi-thread)** et
- **dynamique**.

2.2 Java est simple

- Plus simple que C++ :
  - Nombreux mots clés éliminés.
  - Pas de pré-processeur.
  - Bibliothèque très étendue et **directement intégrée au langage**.
  - Pas de surcharge d'opérateurs, de fonctions indépendantes, de `goto`, de structures, d'unions ni de pointeurs.
  - Pas de fichiers d'en-tête.
  - Pas d'héritage multiple ; à la place, notion *d'interface*, venant d'Objective C. Bien moins complexe.
- **Pas de pointeurs visibles** au niveau du programmeur. Bien sûr, en interne, les pointeurs sont largement utilisés ; mais ceci est caché pour l'utilisateur.

2.3 Java est orienté objet

Les langages C++ et Object Pascal ont construit des caractéristiques orientées objet au dessus d'un langage qui ne l'est pas.

En java, on est **forcé de faire de l'orienté objet** et des bénéfices comme l'encapsulation et la réutilisabilité sont faciles à obtenir.

2.4 Java est réparti

- Java a été construit avec Internet en tête. Riche bibiothèque pour
  - l'accès aux **URL** (Universal Resource Locators),
  - la programmation client/serveur *via* des **sockets** TCP et UDP,
  - l'exécution de **méthodes distantes** (*RMI* : Remote Method Invocation).
  - la conception d'applications réparties selon le modèle d'espaces (issus du langage Linda) avec *JavaSpaces*,
  - la **gestion de serveurs Web** *via* les *Servlets*,
  - la **communication d'objets distants inter-langages** avec des IDL (Interface Definition Language) *CORBA* (Common Request Broker Architecture),

– l'administration de réseaux *via* SNMP (Simple Network Management Protocol) avec JMAPI.

2.5 Java est interprété (ou compilé)

- Le source java est éventuellement transformé en un assembleur d'une machine imaginaire, une *machine virtuelle*. Cet assembleur, ou *bytecode*, peut être interprété. Désavantage : **lenteur d'exécution**.
- ⇒ Notion de *compilateur "à la volée"* ou "juste à temps". La Traduction du bytecode au langage machine est effectuée juste avant l'exécution.
- ⇒ Performances **avoisinant celles des langages compilés** classiques. Puis, apparition de *compilateurs natifs*, avec des performances égales à celles du C.

2.6 Java est robuste

- 🚀 Gestion des erreurs matérielles et logicielles, *via* un mécanisme d'*exceptions*. Exemples : ouverture d'un fichier inexistant, division par zéro, création d'un point de communication réseau (socket) vers une @IP inexistante, ... Le programmeur est **forcé de gérer** diverses exceptions.
- 🚀 Gestion automatique de la mémoire ; présence d'un *ramasse-miettes* (pas de possibilité de `new` sans `delete`).
- Verification à l'exécution des compatibilités de type lors d'un cast.

2.7 Java est sûr

- Écriture mémoire erronée : quasi-impossible en java, car **pas de pointeurs**.
- 🚀 **Indices** de tableau **testés** avant qu'ils soient référencés.
- Test qu'une variable a été assignée avant d'être utilisée.
- Bytecode également testé avant d'être exécuté :
  – test de bon accès aux classes,
  – tests de congestion et de famine de la pile des opérandes,
  – test de conversion illégale de données,
  – test d'accès aux ressources : fichiers,
  – ...

2.8 Java est indépendant de l'architecture

- Le bytecode est **indépendant de la plate-forme**.
- 🚀 Les bibliothèques sont **intégrées de manière standard au langage**, à l'encontre de C++.

2.9 Java est portable

- Un même programme peut être compilé sur une machine et exécuté sur une autre, quel que soit le processeur ou l'OS.
- La **taille** des types de données est **toujours la même** en java.

2.10 Java est efficace

- Initialement, les interpréteurs rendaient l'exécution de programmes java lente (environ 20 fois plus lente que du C).
- Les compilateurs à la volée (*JIT*) la rendent presque aussi rapide que des programmes compilés classiques.
- Des **compilateurs natifs**, fournissent du code machine natif pour tel ou tel processeur ; performances égales à celles du C (jove ; cygnus, au dessus de gcc, ...).

2.11 Java est multitâches

- L'un des premiers langages à posséder en interne des **tâches**, ou activités (*threads*) d'exécution.
- 🚀 La **coordination** des activités est aisée (*moniteurs de Hoare* et événements).

2.12 Java est dynamique

- Exécution coté client ⇒ dynamisme plus aisé à mettre en œuvre que dans d'autres langages.
- **Chargement** des classes **en cours d'exécution**, lorsque nécessaire, éventuellement à travers le réseau. Chargement dynamique des classes possible grâce à des informations de typage consultables en cours d'exécution.

La liste est donnée par thème, chaque élément étant suivi, entre parenthèses, du nom de la technologie Java correspondante. Son éventuelle disponibilité apparaît ensuite : au sein du JDK, paquetage optionnel ou extension en accès d'avant première.

2.13 Technologies Réseaux

1. Flux de données réseau TCP et UDP par **sockets** (`Socket`, ... ; JDK).
2. **Appel de méthodes distantes** (RMI ou Remote Method Invocation ; JDK).

3. Interopérabilité réseau inter-langage *via* **CORBA** (IDL ou Interface Definition Langage ; JDK).
4. Appel de méthodes distantes au dessus du protocole Internet d'interopérabilité réseau inter-langage (**RMI-IIOP** ou Remote Method Invocation over Internet Inter-Orb Protocol ; paquetage optionnel).
5. Fonctions de **serveurs HTTP** (Java Servlets ; paquetage optionnel).

2.14 Technologies Réseaux (suite)

1. **Communication** distribuée **par espaces** (JavaSpaces).
2. Applications **mutli-agent** réseau (JDMK, Java Dynamic Management Kit).
3. **Administration distribuée** (Java Management ; paquetage en accès d'avant première).
4. Gestion de **courier** (Java Mail ; paquetage optionnel).
5. Service de **nommage et** de **répertoires** (JNDI ou Java Naming Directory Interface ; paquetage optionnel).

2.15 Technologies graphiques & sonores

1. Gestion d'interfaces graphiques (AWT ou Abstract Window Toolkit et Swing, formant les JFC ou Java Foundation Classes ; JDK).
2. Composants réutilisables, éditables au sein d'un concepteur d'interfaces graphiques ou "GUI builder" (Java Beans ; JDK).
3. Dessin vectoriel 2D (Java2D ; JDK).
4. Traitement d'images de base (Java advanced imaging ; paquetage optionnel).
5. Synthèse d'images et VRML (Java3D ; paquetage optionnel)
6. Gestion multimédia (JMF, Java Media Framework ; extension standard).
7. Synthèse vocale (Java Sound).

2.17 Technologies de Sécurité

1. Liste de contrôle d'accès ou "ACLs" (JDK)
2. Authentification et autorisation (JAAS ou Java Authentication and Authorization Service ; paquetage en accès d'avant première)
3. Flux réseau sécurisé par des SSL ou Secure Socket Layer (JSSE ou Java Secure Socket Extension ; paquetage en accès d'avant première)
4. Cryptographie (JCE ou Java Cryptography Extension ; extension standard)

2.18 Technologies de Gestion de données

1. Structures de données de base (listes, arbres, tables de hachage) et tri (`Collections` ; JDK)
2. Accès à des bases de données par SQL (JDBC ou Java Database Connectivity ; JDK)

III – Paquetages de Java

Références bibliographiques

Voir `http ://www.java.sun.com`

III.1 Technologies et paquetages

1.1 Aperçu des technologies disponibles

La liste est donnée par thème, chaque élément étant suivi, entre parenthèses, du nom de la technologie Java correspondante. Sa éventuelle disponibilité apparaît ensuite : au sein du JDK extension standard ou paquetage optionnel.

- **Réseaux** :
  1. Flux de données réseau TCP et UDP par sockets (`Socket`, ... ; JDK)
  2. Appel de méthodes distantes (RMI ou Remote Method Invocation ; JDK)
  3. Interopérabilité réseau inter-langage *via* CORBA (IDL ou Interface Definition Langage ; JDK)
  4. Fonctions de serveurs HTTP (Java Servlets ; extension standard)
  5. Communication distribuée par espaces (JavaSpaces)
  6. Applications mutli-agent réseau (JDMK, Java Dynamic Management Kit)
  7. Administration distribuée (JMX ou Java Management eXtension ; extension standard)
  8. Gestion de courier (Java Mail ; extension standard)
  9. Service de nommage et de répertoires (JNDI ou Java Naming Directory Interface ; extension standard)
- **Graphique, images et sons** :

1. Gestion d'interfaces graphiques (AWT ou Abstract Window Toolkit et Swing, formant les JFC ou Java Foundation Classes ; JDK)
2. Composants réutilisables, éditables au sein d'un concepteur d'interfaces graphiques ou "GUI builder" (Java Beans ; JDK)
3. Dessin vectoriel 2D (Java2D ; JDK)
4. Traitement d'iamges de base (Java advanced imaging ; extension standard)
5. Synthèse d'images et VRML (Java3D ; extension standard)
6. Gestion multimédia (JMF, Java Media Framework ; extension standard)
7. Synthèse vocale (Java Sound)

- **Sécurité** :
  1. Liste de contrôle d'accès ou "ACLs" (JDK)
  2. Authentification et autorisation (JAAS ou Java Authentication and Authorization Service)
  3. Flux réseau sécurisé par des SSL ou Secure Socket Layer (JSSE ou Java Secure Socket Extension ; paquetage optionnel)
  4. Cryptographie (JCE ou Java Cryptography Extension ; extension standard)
- **Gestion de données** :
  1. Structures de données de base (listes, arbres, tables de hachage) et tri (`Collections` ; JDK)
  2. Accès à des bases de données par SQL (JDBC ou Java Database Connectivity ; JDK)

1.2 Paquetages du JDK 1.5

JDK : *Java Development Kit*, Versions de référence du langage, tel que produit par Sun.

`Paquetage`	But
`java.applet`	Fournit les classes nécessaires pour créer une applet et celles qu'une applet utilise pour communiquer avec son contexte.
`java.awt`	Contient toutes les classes pour créer des interfaces graphiques et pour dessiner des graphiques et des images (API de base ; voir `swing` plus loin).

`java.awt.color`	Classes pour les couleurs.
`java.awt.datatransfer`	Interfaces et classes pour le transferts de données entre applications.
`java.awt.dnd`	"Drag and Drop" ou "glisser-placer" un élément graphique à l'aide d'un pointeur (en général la souris).
`java.awt.event`	Interfaces et classes de gestion des différents événements associés aux composants AWT.
`java.awt.font`	Interfaces et classes reliées aux polices de caractères.
`java.awt.geom`	Classes Java 2D pour définir et réaliser des opérations sur des objets bi-dimensionnels.
`java.awt.im`	Classes et interfaces du cadre de méthodes d'entrée (input method framework). Ce cadre permet à des composants de recevoir des entrées clavier en Japonais, Chinois ou Coréen (pour lesquelles il faut plusieurs touches pour un caractère).
`java.awt.im.spi`	Interfaces permettant le déploiement d'entrées pour tout environnement d'exécution Java.
`java.awt.image`	Classes pour créer et modifier des images.
`java.awt.image.renderable`	Classes et interfaces pour produire des images indépendantes du rendu (du périhérique d'affichage ou d'impression).
`java.awt.print`	Classes et interfaces pour une API générale d'impression.
`java.beans`	Classes reliées au développement de composants "beans" (composants réutilisables pouvant être édités graphiquement).
`java.beans.beancontext`	Classes et interfaces reliés à un contexte de bean (conteneur qui définit un environnement d'exécution du ou des bean(s) qu'il contient).
`java.io`	Entrées/sorties systèmes au travers de flux de données, de la sérialisation et le système de fichiers.
`java.lang`	Classes fondamentales du langage Java.
`java.lang.annotation`	.
`java.lang.instrument`	Classes d'agents d'instrumentation de programmes tournant sur une JVM.
`java.lang.management`	Gestion de la machine virtuelle et du système d'exploitation hôte.

`java.lang.ref`	Fournit des classes de références à des objets, supportant un certain degré d'interaction avec le ramasse-miettes.
`java.lang.reflect`	Classes et interfaces pour obtenir des informations de reflexion sur les classes et les objets.
`java.math`	Classes pour de l'arithmétique entière et décimale en précision arbitraire.
`java.net`	Classes pour des applications réseaux *via* des sockets.
`java.nio`	Classes de tampons, conteneurs de données.
`java.nio.channels`	Classes de canaux connectant des flux d'entrée/sortie comme des fichiers ou des sockets.
`java.nio.channels.spi`	Fournisseur de service pour les canaux.
`java.nio.charset`	Classes de codage/décodage octet/Unicode.
`java.nio.charset.spi`	Fournisseur de service pour les codeurs/décodeurs.
`java.rmi`	Fournit le paquetage RMI (Remote Method Invocation) d'appel de procédure distante pour des applications réseaux.
`java.rmi.activation`	Support pour l'activation d'objets RMI.
`java.rmi.dgc`	Classes et interfaces pour le ramasse-miettes distribué utilisé par RMI.
`java.rmi.registry`	Une classe et deux interfaces pour le registre RMI.
`java.rmi.server`	Classes et interfaces de support pour les serveurs RMI.
`java.security`	Classes et interfaces pour le cadre de sécurité.
`java.security.acl`	Les classes et interfaces de ce paquetage ont été redues obsoltes par les classes de `java.security`.
`java.security.cert`	Classes et interfaces pour analyser et gérer les certificats.
`java.security.interfaces`	Interfaces pour générer des clés RSA (algorithme "AsymmetricCipher" de Rivest, Shamir et Adleman).
`java.security.spec`	Classes et interfaces pour des spécifications de clés et des spécifications de paramètres d'algorithmes.
`java.sql`	Fournit le paquetage JDBC.
`java.text`	Classes et interfaces pour gére du texte, des dates, des nombres et des messages d'une manière indépendante de la langue.

`java.util`	Contient le cadre des collections, le modèle d'événements, des utilitaires de gestion du temps et de la date ainsi que d'autres utilitaires divers (un analyseur lexical, un générateur de nombres aléatoires et un tableau de bits).
`java.util.concurrent`	Classes de programmation concurrente.
`java.util.concurrent.atomic`	Classes de programmation sans verrou adaptées à la multiplicité des threads ("lock-free and thread-safe").
`java.util.concurrent.locks`	Classes de verrou et d'attente (distinctes des synchronisation (`synchronized`) et moniteurs de Hoare (`wait()`)).
`java.util.jar`	Classes pour lire et écrire dans des fichiers JAR (Java ARchive), basé sur le standard de fichier ZIP avec un fichier optionnel de manifeste.
`java.util.logging`	Classes de sauvegarde en ligne ("logging").
`java.util.prefs`	Classes de sauvegarde et restauration des préférences utilisateur et système.
`java.util.regexp`	Classes de manipulation des expressions régulières.
`java.util.zip`	Classes pour lire et écrire dans des fichiers JAR (Java ARchive), basé sur les standards de fichier ZIP et GZIP.
`javax.accessibility`	Définit un contrat entre des composants d'interface utilisateur et des technologies d'assitance (par exemple aux personnes handicapées).
`javax.crypto`	Classes de cryptographie.
`javax.crypto.interfaces`	Classes d'interfaces pour les clés de Diffie-Hellman.
`javax.crypto.spec`	Classes de spécification de clés et de paramètres d'algorithme.
`javax.imageio`	Classes d'entrées/sorties d'images.
`javax.imageio.event`	Classes de gestion d'évenements synchrone durant la lecture et l'écriture d'images.
`javax.imageio.metadata`	Classes de lecture et d'écriture de métadonnées d'images.
`javax.imageio.plugins.bmp`	Classes de plugin BMP.
`javax.imageio.plugins.jpeg`	Classes de plugin JPEG.
`javax.imageio.spi`	Classes de interfaces pour les lecteurs, écrivains, transcodeurs et flux d'images.

`javax.imageio.stream`	Classes d'entrées/sorties image bas niveau à partir de fichiers et de flux.
`javax.management`	Classes JMX de base.
`javax.management.loading`	Classes de chargement dynamique avancé.
`javax.management.modelmbean`	Classes de définiton de ModelMBean.
`javax.management.monitor`	Classes de définition des moniteurs.
`javax.management.openmbean`	Classes de types ouverts et descripteurs mbean ouverts ("open").
`javax.management.relation`	Classes de définition du service de relation.
`javax.management.remote`	Interfaces pour l'accès distant aux serveurs MBean JMX.
`javax.management.remote.rmi`	Connecteur pour l'API JMX distante utilisant les RMI pour la transmission des requêtes à un serveur MBean.
`javax.management.timer`	Classes de définition d'un timer MBean.
`javax.naming`	Classes et interfaces d'accès aux services de nommage.
`javax.naming.directory`	Classes héritant de `javax.naming` pour l'accès aux services de répertoires.
`javax.naming.event`	Classes de gestion des événements lors de l'accès aux services de nommage.
`javax.naming.ldap`	Classes de gestion de LDPAv3.
`javax.naming.spi`	Classes de chargement dynamique des services de nommage.
`javax.net`	Classes pour les applications réseaux.
`javax.net.ssl`	Classes du paquetage de sockets sécurisées.
`javax.print`	Classes de gestion d'impression.
`javax.print.attribute`	Classes de types d'attributs du service d'impression.
`javax.print.attribute.standard`	Classes d'attributs d'impression spécifiques.
`javax.print.event`	Classes d'événements et d'auditeurs pour impression.
`javax.rmi`	Classes de gestion des RMI-IIOP.
`javax.rmi.CORBA`	Classes portables RMI-IIOP.
`javax.security.auth`	Classes d'autentification et d'autorisation.
`javax.security.auth.callback`	Classes de collecte d'information (nom, mot de passe, etc.) et d'affichage (erreurs, avertissements, etc.).
`javax.security.auth.kerberos`	Classes utilitaires reliées au protocole Kerberos.
`javax.security.auth.login`	Classes d'autentification dynamiquement chargeable.

`javax.security.auth.spi`	Classes d'interface pour implanter des modules d'autentification.
`javax.security.auth.x500`	Classes de gestion X500 (Principal and Private Credentials).
`javax.security.cert`	Classes de certificats à clé publique.
`javax.security.sasl`	Classes de gestion de SASL.
`javax.sound.midi`	Classes et interfaces d'entrée/sortie, de séquençage et de synthèse de données MIDI.
`javax.sound.midi.spi`	Classes de gestion d'implantation de périphériques MIDI.
`javax.sound.sampled`	Classes d'acquisition, de traitement et de rejeu de données audio échantillonnées.
`javax.sound.sampled.spi`	Classes de gestion d'implantation de périphériques audio.
`javax.sql`	Classes d'accès de données coté serveur.
`javax.sql.rowset`	Classes de gestion de JDBC `RowSet`.
`javax.sql.rowset.serial`	Classes de sérialisation entre types SQL et types Java .
`javax.sql.rowset.spi`	Classes de gestion d'implantation de fournisseur de synchronisation.
`javax.swing`	Ensemble de composants "légers" (entièrement Java) qui se comportent de manière quasi-identique sur toutes les plates-formes.
`javax.swing.border`	Classes et interfaces pour afficher des bodures autour d'un composant Swing.
`javax.swing.colorchooser`	Classes et interfaces utilisées par le composant `JColorChooser`.
`javax.swing.event`	Événements créés par des composants Swing.
`javax.swing.filechooser`	Classes et interfaces utilisées par le composant `JFileChooser`.
`javax.swing.plaf`	Une interface et plusieurs classes abstraites utilisées par Swing pour fournir des possibilités de rendu adaptables (pluggable look-and-feel capabilities).
`javax.swing.plaf.basic`	Objets d'interface utilisateur construits conformément au rendu standard (the Basic look-and-feel).
`javax.swing.plaf.metal`	Objets d'interface utilisateur construits conformément au rendu "métallique" ("metal" look-and-feel).
`javax.swing.plaf.multi`	le rendu (look and feel) multiplexé permet à un utilisateur de combiner un rendu auxiliaire avec le rendu par défaut.

`javax.swing.plaf.synth`	Rendu (look and feel) dans lequel l'affichage (paint) est délégué.
`javax.swing.table`	Classes et interfaces pour gérer les `java.awt.swing.JTable`.
`javax.swing.text`	Classes et interfaces pour gérer des composants textes éditables ou non.
`javax.swing.text.html`	Classe `HTMLEditorKit` et ses classes de support pour créer des éditeurs de texte HTML.
`javax.swing.text.html.parser`	Classes et interfaces pour analyseur de documents HTML.
`javax.swing.text.rtf`	Classe `RTFEditorKit` pour créer des éditeurs de texte au format RTF (Rich-Text-Format).
`javax.swing.tree`	Classes et interfaces pour gérer les `java.awt.swing.JTree`.
`javax.swing.undo`	Support pour les possibilités de "undo/redo" dans une application (comme un éditeur de texte).
`javax.transaction`	Trois exceptions levées par l'ORB pendant la désérialisation.
`javax.transaction.xa`	Définit le contrat entre le gestionnaire de transactions et le gestionnaire de ressources pour l'enregistrement et le désenregistrement au sein de transactions JTA.
`javax.xml`	Constantes des spécifications XML.
`javax.xml.datatype`	Types de données XML.
`javax.xml.namespace`	Espace de nommage XML.
`javax.xml.parsers`	Traitement de données XML.
`javax.xml.transform`	Transformations de données.
`javax.xml.transform.dom`	Transformations de type DOM.
`javax.xml.transform.sax`	Transformations de type SAX2.
`javax.xml.transform.stream`	Transformations de type flot et URI.
`javax.xml.validation`	Validation de langage.
`javax.xml.xpath`	API pour XPath 1.0.
`org.ietf.jgss`	Utilisation portable d'autentification, de données intègres et de données confidentielles.
`org.omg.CORBA`	Correspondance entre les API de l'OMG CORBA et Java, incluant la classe ORB (Object Request Broker).
`org.omg.XXXX`	Paquetages de l'OMG (27 paquetages).
`org.w3c.dom`	Interfaces pour DOM au sein d'XML.
`org.w3c.dom.bootstrap`	.
`org.w3c.dom.ls`	.
`org.xml.sax`	API pour SAX.

`org.xml.sax.ext`	Gestionnaires SAX2 optionnels.
`org.xml.sax.helpers`	Classes d'aide SAX.

III.2 Sources de documentation externes

2.1 Sites importants

- Le site Sun
  `http ://www.javasoft.com`
- Les API Java XX (par ex. 1.5) de Sun
  `http ://www.javasoft.com/products/jdk/XX/docs/api/index.html`
- L'almanach Java, une liste de mini exemples pour chaque classe du langage. Très utile
  `http ://javaalmanac.com/`
- Divers cours sur Java, en français
  `http ://java.developpez.com/cours/`

2.2 Sites utiles

- La documentation générale indiquée par Sun
  `http ://java.sun.com/docs/`
- Les tutoriels
  `http ://java.sun.com/docs/books/tutorial`
- Documentations ENST, dont le JDK et le Java Tutorial de Sun
  `http ://www-inf.enst.fr/softs/`
- Compilateur GNU pour Java, GCJ (compilation en code machine)
  `http ://gcc.gnu.org/java/`
- Plusieurs cours sur les réseaux de l'UREC
  `http ://www.urec.fr/cours/`
- Java-Linux
  `http ://www.blackdown.org`
- Liens sur Java
  `http ://www.webreference.com/programming/java.html`
- Cours en ligne sur Java
  `http ://www.eteks.com`
- Beaucoup de liens sur Java
  `www.teamjava.com/links`
- Les RFC en HTML à l'université d'Ohio
  `http ://www.cis.ohio-state.edu/hypertext/information/rfc.html`

- Plus de 16 000 liens sur les objets et composants
  `http ://www.sente.ch/cetus/software.html`

IV – Bases procédurales de Java

Références bibliographiques

- *The Java Language Specification*, J. Gosling, B. Joy et G. Steele [GJS96],
- *Java in a Nutshell*, D. Flanagan [Flab].

IV.1 Variables et types de données

1.1 Identificateurs

- *Identificateur* : suite de
  - **lettres**
  - **minuscules ou majuscules**,
  - **chiffres**,
  - **underscore** (_) **et dollar** ($).

  Un identificateur **ne doit pas** commencer par un chiffre.
- Java distingue minuscules et majuscules (`Valeur` diffère de `VALEUR`).
- Conventions
  - Toute **méthode** *publique* et *variable d'instance* commence par une minuscule. Tout changement de mot descriptif se fait *via* une majuscule. Exs. : `nextItem, getTimeOfDay`.
  - **Variables locales** et *privées* : lettres minuscules avec des underscores. Exs. : `next_val, temp_val`.
  - **Variables dites** `final` représentant des constantes : lettres majuscules avec underscores. Exs. : `DAY_FRIDAY, GREEN`.
  - Tout **nom de classe** ou d'**interface** commence par une majuscule. Tout changement de mot descriptif se fait *via* une majuscule. Exs. : `StringTokenizer`, `FileInputStream`.

1.2 Représentation littérale

- *Entiers* :
  - ☛ les valeurs octales commencent avec un `0`.
  - Ainsi `09` génère une erreur : `9` en dehors de la gamme octale `0` à `7`.
  - Ajouter un `l` ou `L` pour avoir un entier `long`.
- *Nombres à virgules* : par défaut des `double`. Ajouter un `f` ou `F` pour avoir un `float`.
- *booléens* : 2 valeurs possibles, `true` et `false`. ☛ `true` (resp. `false`) n'est pas égal à `1` (resp. `0`).
- *Chaînes de caractères* : ☛ doivent commencer et se terminer **sur la même ligne** ...
- *Caractères* : unicode (sur 16 bits), manipulables comme des entiers, par `'a'`, `'@'`, ...

Séquence escape	Description
\ddd	Caractère octal ddd
\uxxxx	Caractère hexadécimal unicode xxxx
\'	Apostrophe
\"	Guillemets
\\	Backslash (barre oblique inversée)
\r	Carriage return
\n	New line
\f	Form feed
\t	Tabulation
\b	Backspace

1.3 Variables

- exemple (Pythagore) :

```
class Variables {
    public static void main(String args[]) {
        double a = 3;
        double b = 3;
        double c;
        c = Math.sqrt(a*a + b*b);
        System.out.println("c = " + c);
    }
}
```

- Règles de visibilité usuelles pour une variable définie à l'intérieur d'un bloc entre `{}`.
- Une variable ne peut avoir le même nom qu'une déclarée dans un bloc englobant :

```
class Scope {
    public static void main(String args[]) {
        int var = 3;
        {
            int var = 2;    // Erreur de compilation
        }
    }
}
```

1.4 Mots clés du langage

Attention a ne pas utiliser comme nom de variable un mot clé réservé, dont voici la liste :

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

1.5 Nature des variables

On distingue 7 natures de variables :
- les variables d'instance
- les variables de classe
- les variables de type tableau
- les paramètres des méthodes
- les param‘etres des constructeurs
- les variables de type exception
- les variables locales

1.6 Types primitifs

- Dans certains langages `2+2` : appel de la méthode “plus” sur une instance d'objet représentant deux, passant une autre instance de deux ...
- ⇒ Pour des raisons de performance : *types primitifs* en java, strictement analogues aux types correspondants dans des langages non orientés objets.
- Huit types primitifs :
  - **Entiers** : `byte`, `short`, `int` et `long`, tous signés.
  - **Nombres à virgule flottante** : `float` et `double`.
  - **Caractères** : `char`.
  - **booléens** : `boolean`, pour les valeurs logiques.

1.7 Types entiers et flottants

- Toute **assignation**, explicite ou par passage de paramètres, fait l'objet d'une **vérification de type**.
- Pas de coercition ou de conversion systématique. Une différence de type est une erreur de compilation, pas un avertissement (warning).
- Types de données entiers :
  - `byte` : à n'utiliser que pour des manipulations de bits.
  - `short` : relativement peu utilisé car sur 16 bits.
  - `int` : dans toute expression avec des `byte`, `short`, `int`, tous sont *promus* à des `int` avant calcul.

1.8 Plages de variation

Nom	Taille	Plage de variation
`long`	64 bits	`-9 223 372 036 854 775 808...` `9 223 372 036 854 775 807`
`int`	32 bits	`-2 147 483 648 ...` `2 147 483 647`
`short`	16 bits	`-32 768 ...32 767`
`byte`	8 bits	`-128 ...127`
`double`	64 bits	`1.7e-308 ...1.7e308`
`float`	32 bits	`3.4e-38 ...3.4e+38`

1.9 Transtypage (ou conversions, “cast”)

- Conversions possibles en java. Conversion automatique **seulement possible** lorsque le compilateur sait que la variable destination est assez grande.
- Si des `bytes`, `short`, `int` et `long` font partie d'une expression, tout le monde est promu à `long`. Si une expression contient un `float` et pas de `double`, tout le monde est promu à `float`. S'il y a un `double`, tout le monde est promu à `double`. Tout littéral à virgule est considéré comme `double`.

1.10 Caractères

- Un caractère est codé par un entier allant de `0` à `65536` (selon le standard unicode).
- On peut se servir des caractères comme entiers :

```
int  trois  = 3;
char un     = '1';
char quatre = (char) (trois + un);
```

  Dans `quatre` : `'4'`. `un` a été promu à `int` dans l'expression, d'où la conversion en `char` avant l'assignation.

1.11 Booléens

- Type renvoyé par tous les opérateurs de comparaison, comme `(a < b)`.
- Type **requis** par tous les opérateurs de contrôle de flux, comme `if`, `while` et `do`.

1.12 Tableaux

- Création pouvant être faite en deux temps :
  - Déclaration de type, les `[]` désignant le type d'un tableau

```
int tableau_entiers[] ;
```

  - Allocation mémoire, *via* `new`

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

- Pour les tableaux, la **valeur spéciale** `null` représente un tableau sans aucune valeur.
- Initialisation

```
int tableau_initialise[] = { 12, 34, 786 };
```

- Vérification par le compilateur de stockage ou de référence **en dehors des bornes** du tableau.

1.13 Tableaux multidimensionnels (I)

- Tableaux multidimensionnels crées comme

```
double matrice[][] = new double[4][4];
```

  Ce qui revient à

```
double matrice[][] = new double[4][];
matrice[0] = new double[4];
matrice[1] = new double[4];
matrice[2] = new double[4];
matrice[3] = new double[4];
```

1.14 Tableaux multidimensionnels (II)

- **Initialisation par défaut** de tous les éléments **à zéro**.
- Des expressions sont permises dans les initialisations de tableaux

```
double m[][] = {
            { 0*0, 1*0, 2*0 },
            { 0*1, 1*1, 2*1 },
            { 0*2, 1*2, 2*2 }
};
```

IV.2 Opérateurs

2.1 Opérateurs arithmétiques

Op.	Résultat	Op.	Résultat
`+`	addition	`+=`	assignation additive
`-`	soustraction	`-=`	assignation soustractive
`*`	multiplication	`*=`	assignation multiplicative
`/`	division	`/=`	assignation divisionnelle
`%`	modulo	`%=`	assignation modulo
`++`	incrémentation	`-`	décrémentation

- Les opérateurs arithmétiques fonctionnent comme en C.
- Une différence : le modulo agit également sur les nombres à virgule.

Exemple d'incrémentation

```
class IncDec {
    public static void main(String args[]) {
        int a = 1;
        int b = 2;
        int c = ++b;
        int d = a++;
        c++;
        System.out.println("a = " + a);
        System.out.println("b = " + b);
        System.out.println("c = " + c);
        System.out.println("d = " + d);
        }
}
```

La sortie du programme est

```
Prompt > javac IncDec
Prompt > java IncDec
a = 2
b = 3
c = 4
d = 1
```

2.2 Opérateurs entiers sur les bits

- On peut manipuler les bits des types entiers `long`, `int`, `short`, `char` et `byte` à l'aide des opérateurs suivants

Op.	Résultat	Op.	Résultat
`-`	NON unaire bit-à-bit		
`&`	ET bit-à-bit	`&=`	assignation avec ET bit-à-bit
`|`	OU bit-à-bit	`|=`	assignation avec OU bit-à-bit
`^`	OU exclusif bit-à-bit	`^=`	assignation avec OU exclusif bit-à-bit
`>>`	décalage à droite	`>>=`	assignation avec décalage à droite
`>>>`	décalage à droite **avec remplissage de zéros**	`>>>=`	assignation avec décalage à droite et remplissage de zéros
`<<`	décalage à gauche	`<<=`	assignation avec décalage à gauche

- Visualisation de l'effet de `>>>`

```
int a = -1;
```

```
a = a >>> 24;
```

donne en forme binaire

```
11111111111111111111111111111111        -1
>>>24
00000000000000000000000011111111       255
```

2.3 Opérateurs relationnels

Op.	Résultat
`==`	égal à
`!=`	différent de
`>`	strictement supérieur à
`<`	strictement inférieur à
`>=`	supérieur ou égal à
`<=`	inférieur ou égal à

- Tout type java, y compris les types primitifs et les références à des instances d'objets peuvent être comparés avec `==` et `!=`
- **Seuls les types numériques** peuvent être comparés avec les opérateurs d'ordre. Les entiers, flottants et caractères peuvent être comparés avec les opérateurs d'ordre strict.
- Chaque opérateur renvoie un type `boolean`.

2.4 Opérateurs booléens logiques

Les opérateurs suivants agissent uniquement sur des opérandes `boolean`

Op.	Résultat
`&`	ET logique
`&=`	assignation avec ET
`|`	OU logique
`|=`	assignation avec OU
`^`	OU exclusif logique
`^=`	assignation avec OU exclusif
`||`	OU avec court cricuit

`==`	égal à
`&&`	ET avec court circuit
`!=`	différent de
`!`	NON unitaire logique
`? :`	if-then-else ternaire

- les courts circuits (`&&` et `||`) fonctionnent comme en C (la deuxième opérande est exécutée conditionnellement à la valeur booléenne de la première). Ansi le code suivant ne génère pas d'erreur à l'exécution.
  `if (denom != 0 && num / denom > 10)`
- ↘ Les versions non court circuitées donnent lieu à évaluation des deux opérandes. Le code suivant génère une erreur
  `if (denom != 0 & num / denom > 10)`
  `java.lang.ArithmeticException : / by zero`
- ⇒ **Toujours utiliser les versions court circuitées** (`&&` et `||`), n'utilisant les versions mono-caractère que dans le cas d'opérations bit-à-bit.
- L'opérateur if-then-else ternaire (`? :`) fonctionne comme en C. Par exemple
  `ratio = (denom == 0 ? 0 : num / denom);`

2.5 Priorité des opérateurs

Priorité haute							Sens de priorité
`[]`	`.`	`()`[1]					gauche à droite
`++`	`-`	`~`	`!`	- (unaire)	`()` (cast)	`new`	droite à gauche
`*`	`/`	%					gauche à droite
`+`	`-`						gauche à droite
`>>>>>`		`<<`					gauche à droite
`>`	`>=`	`<`	`<=`	`instantceof`			gauche à droite
`==`	`!=`						gauche à droite
`&`							gauche à droite
`^`							gauche à droite
`|`							gauche à droite
`&&`							gauche à droite
`||`							gauche à droite
`? :`							gauche à droite
`=`	`op=`						droite à gauche
Priorité basse							

[1] Appel de méthode.

IV.3 Contrôle de flux

3.1 Instruction `if-else`

- Forme strictement analogue à celle du C

```
  if ( expression-booleenne ) expression1;
[ else expression2; ]
```

- `expressioni` peut être une expression composée entourée de `{}`.
- `expression-booleenne` est toute expression renvoyant un `boolean`.
- ☛ Il est de BONNE PRATIQUE **d'entourer d'accolades une expression** même si elle n'est pas composée. Ce qui permet, lorsque l'on veut rajouter une expression, de ne rien oublier, comme c'est le cas ci-après

3.2 Instruction `if-else`

```
int octetsDisponibles;

if (octetsDisponibles > 0) {
    CalculDonnees();
    octetsDisponibles -= n;
} else
    attendreDautresDonnees();
    octetsDisponibles = n;
```

où la dernière ligne devrait, d'après l'indentation, faire partie du bloc `else`.

3.3 Instruction `break`

- Utilisation courante strictement analogue à celle du C : pour sortir d'un `case` à l'intérieur d'un `switch`.
- Autre utilisation : sortie d'un bloc marqué. Marquage par étiquette : un identificateur suivi de `' :'` placé devant le bloc

```
class Break {
    public static void main(String args[]) {
        boolean t = true;
a:      {
b:          {
c:              {
                    System.out.println("Avant le break");
                    if (t)
```

```
                                break b;
                            System.out.println("Jamais execute");
                        }
                        System.out.println("Jamais execute");
                    }
                    System.out.println("Apres b:");
                }
            }
        }
```

`break` suivi du nom de marquage du bloc permet une sortie du bloc marqué par cette étiquette. La sortie écran du code est

```
Avant le break
Apres b:
```

- ☛ On ne peut se brancher à une étiquette qui n'est pas définie devant un des blocs englobant, sinon `break` serait identique à `goto`.

3.4 Instruction `switch` (I)

- Forme strictement analogue à celle du C

```
switch ( expression ) {
    case valeur1 :
        break;

    case valeurN :
        break;
    default :
}
```

- `expression` peut être tout type primitif (les `valeuri` doivent être du même type qu'`expression`)
- ☛ C'est une erreur répandue que d'oublier un `break`. Il est donc de BONNE PRATIQUE d'**utiliser des commentaires** du type `// CONTINUER`. Exemple d'équivalent de `wc` (word count, comptage du nombre de lignes, mots et caractères)
- Exemple d'équivalent de `wc` (word count, comptage du nombre de lignes, mots et caractères)

```
class WordCount {
  static String texte =
    "Trente rayons convergent au moyeu               " +
    "mais c'est le vide median                       " +
```

```
    "qui fait marcher le char.                          " +"\n"+

    "On faconne l'argile pour en faire des vases,       " +
    "mais c'est du vide interne                         " +
    "que depend leur usage.                             " +"\n"+

    "Une maison est percee de portes et de fenetres,    " +
    "c'est encore le vide                               " +
    "qui permet l'habitat.                              " +"\n"+

    "L'Etre donne des possibilites,                     " +
    "c'est par le non-etre qu'on les utilise.           " +"\n"+

    "Tao-to king, Lao-tseu, XI                          \n";

    static int long = text.length();



  public static void main(String args[]) {
    booelan dansMot    = false;
    int     nbreCars   = 0, nbreMots   = 0, nbreLignes = 0;

    for (int i = 0; i < long; i++) {
        char c = texte.charAt(i);
        nbreCars++;
        switch (c) {
            case '\n' : nbreLignes++; // CONTINUER
            case '\t' :               // CONTINUER
            case ' '  : if (dansMot) {
                            nbreMots++;
                            dansMot = false;
                        }
                        break;
            default   : dansMot = true;
        }
    }
    System.out.println("\t" + nbreLignes + "\t" + nbreMots +
                                             "\t" + nbreCars);
  } // main()
} // class WordCount
```

3.5 Instruction `return`

- Même usage qu'en C.
- ☛ C'est une **erreur de compilation** que d'avoir du code inatteignable en java.

3.6 Instructions `while`/`do-while`/`for`

- Mêmes usages et syntaxe qu'en C. `while`

```
[ initialisation; ]
while ( terminaison ) {
    corps;
  [ iteration; ]
}
```

- `do-while`

```
[ initialisation; ]
do {
    corps;
  [ iteration; ]
} while ( terminaison );
```

- `for`

```
for (initialisation; terminaison; iteration)
    corps;
```

3.7 Instruction `continue`

Comme pour `break`, possibilité d'avoir un nom d'étiquette comme argument. Exemple de table de multiplication triangulaire.

```
class EtiquetteContinue {
    public static void main(String args[]) {
englobante:     for (int i = 0; i < 10; i++) {
                  for (int j = 0; j < 10; j++) {
                      if (j > i) {
                          System.out.println("");
                          continue englobante;
                      }
                      System.out.println(" " + (i * j));
                  }
                }
    }
}
```

V – Notions de génie logiciel

Références bibliographiques

- *Object-Oriented Analysis and Design with Applications*, G. Booch [Boo94],
- *Design Patterns*, E. Gamma, R. Helm, R. Johnson et J. Vlissides [GHJV95],
- *Data Structures*, M.A. Weiss [Wei98]

V.1 La légende des sept singes

1.1 Le petit singe flexible

En tant que le développeur vous aurez toujours cinq singes sur votre dos chacun cherchant à retenir votre attention :

- Le singe *ponctuel* aggripé à votre dos, les bras autour de votre cou, beuglant continuellement : "tu dois respecter les échéances !"
- Le singe *adéquat*, sur votre tête, tambourine sur sa poitrine et crie : "tu doit implanter correctement les spécifications !"
- Le singe *robuste*, sautant sur le dessus de votre moniteur, hurle : "robustesse, robustesse, robustesse !"
- Le singe *performant* essaie de grimper le long de votre jambe en vociférant : "n'oublie pas les performances !"
- Et de temps en temps, un petit singe, *flexible*, pointe timidement sont nez d'en-dessous le clavier. À ce moment là, les autres singes se taisent et ce calment. Le petit singe sort de sous le clavier, se met debout sur ses pattes, vous regarde droit dans les yeux et dit : "Tu dois rendre ton code facile lire et facile à modifier". Sa phrase à peine terminée, les autres singes se remettent à hurler en sautant sur le petit singe, le forçant à se terrer à nouveau sur le clavier. Les quatre premiers singes reviennent alors à leurs activités initiales.
  - Un bon développeur doit trouver un compromis pour rendre les 5 singes heureux.

- Deux autres singes font une apparition tapageuse de temps en temps :
- Le singe *réutilisable* dont le slogan est : "Ton code doit être réutilisable !"
- Le singe *sécuritaire* : "Ton Code doit être parfaitement sûr !"

1.2 Un vieux constat – l'histoire d'AdA

- Le respect des essais des échéances est souvent une pression commerciale puissante au détriment de la qualité d'un logiciel.
- L'insatisfaction des clients résultant de singes malheureux était déjà un constat fait au département de la défense américaine :
- En 1968-1970 au DoD ("Departement of Defense") : coûts matériels $\gg$ coûts logiciels
- De 1968 à 1973 : accroissement de 59 % des coûts informatiques et baisse très forte des coûts matériels (pour le logiciel, budget en 1970 : 3.10^9 \$, en 1990 30.10^9 \$)
- ceci pour un produit fini en général insatisfaisant :

Adéquation :	non correspondance aux besoins des utilisateurs
Fiabilité :	logiciel tombant souvent en panne Coût : coûts rarement prévisibles, perçu comme excessifs
Modifiabilité :	maintenance souvent complexe, coûteuse et non fiable
Ponctualité :	logiciel souvent en retard ; livré avec des capacités inférieures à celles promises
Transportabilité :	logiciel d'un système rarement utilisable sur un autre
Efficacité :	non utilisation optimale par le logiciel des ressources disponibles (temps de calcul, RAM)

  ce qui rend les différents singes tous malheureux.
- Coût de maintenance logicielle > coût du développement original. En 1975, un industriel dépense entre 40 et 75 % de son budget informatique pour la maintenance logicielle. En 1973, 450 langages généraux au DoD.
- Notamment pour des logiciels embarqués,
  - De 1975 à 1977, évaluation de 2800 langages, aucun réellement approprié, bien que Pascal, ALGOL et PL/1 comportent des éléments intéressants.
  - En 1977 appels d'offes international, 17 réponses, 4 sélectionnées, dont une de CII Honeywell Bull.
  - En 1979 sélection finale de la proposition de CII ; le langage, créé par Jean Ichbiah, prend pour nom Ada (Augusta Ada Byron, mathématicienne ayant travaillé avec Babbage sur ses machines différentielle et analytique)
  - En 83, manuel de référence Ada, approuvé comme norme ANSI, puis en 1987 adoptée par l'ISO

V.2 Buts du génie logiciel

2.1 Quatre buts généraux

Pour remédier à cet état de faits, on a cherché à identifier diverses qualités souhaitables d'un logiciel, de façon à contenter les sept singes.

Dans un article [RGI80], D.T. Ross, J.B. Goodenough et C.A. Irvine donnent 4 buts généraux :

- **modifiabilité**
- **efficacité**
- **fiabilité**
- **intelligibilité**

2.2 Modifiabilité

Deux raisons de vouloir modifier un logiciel :
- changement des spécifications (par exemple d'après une demande d'un client)
- correction d'une erreur

Dans un logiciel modifiable, on peut introduire des changements sans complexification (ni perte de lisibilité)

2.3 Efficacité

- Utilisation optimale des ressources en temps et en espace Pour des systèmes *temps rééel* : dont la ressource en temps est prédominante. Pour des systèmes *embarqués*, où il y a limitation de place (dans un satellite, une automobile) : ressource en espace prédominante
- On se préoccupe souvent **trop tôt** de l'efficacité, se polarisant de ce fait sur la micro-efficacité au détriment de la macro-efficacité ; dans [RGI80] : "Une bonne perspicacité reflétant une compréhension plus unifiée d'un problème a beaucoup plus impact sur l'efficacité n'importe quel tripotage de bits dans une structure déficiente"

2.4 Fiabilité

Peut-être critique (système de navigation d'un engin spatial) : "La fiabilité doit à la fois éviter les défauts de conception, d'étude et de construction, et permettre de récupérer les pannes et les défauts de performances" (cf. [RGI80]).

Pour toute panne, prévisible ou non, le logiciel doit entrer en mode dégradé en douceur, sans effet de bord dangereux (par exemple, une orbite dégradée)

2.5 Intelligibilité

- Sans doute le but le plus crucial pour bien gérer la complexité d'un système logiciel Pour qu'un système soit compréhensible, il doit être un modèle exact de notre vue du monde réel Pour améliorer la compréhensiblité :
  – Au bas niveau, lisibilité par un bon style de codage,
  – Au plus haut niveau, il être facile d'isoler les structures de données (**objets**) et les actions (**opérations**) correspondantes à celles du monde réel. Le langage choisi est important pour cela.

V.3 Principes de génie logiciel

3.1 Six principes de génie logiciel

De bons principes sont nécessaires pour réaliser les buts précédents (cf. [RGI80]), nommément :

- **abstraction**
- **dissimulation d'informations**
- **modularité**
- **localisation**
- **uniformité**
- **intégralité**

3.2 Abstraction et dissimulation d'information

- *Abstraction* : exemple de système de fichiers
  (i) fichiers d'enregistrements
  (ii) disques logiques (i-noeuds, blocs)
  (iii) carte contrôleur de disque
  On veut extraire les propriétés essentielles en omettant les détails qui ne le sont pas
  Au niveau (i) une structure (par exemple une adresse)
  Au niveau (ii) une suite de blocs
  Au niveau (iii) une suite de secteurs physiques
  Pour manipuler fichier on fera : Ouvrir, lire (écrire) un enregistrement, fermer. On ne s'occupe pas du niveau intérieur qui correspond à (ii) : Lecture, écriture de bloc, mise à jour d'i-noeud qui lui ne s'occupe pas du niveau (iii) : Déplacement de têtes de lecture, lecture ou écriture d'une suite de bits par tête magnétique.

  Pour manipuler fichier on fera : Ouvrir, lire (écrire) un enregistrement, fermer. On ne s'occupe pas du niveau intérieur qui correspond à (ii) : Lecture,

écriture de bloc, mise à jour d'i-noeud qui lui ne s'occupe pas du niveau (iii) : Déplacement de têtes de lecture, lecture ou écriture d'une suite de bits par tête magnétique On a donc également une abstraction des opérations ou abstraction algorithmique.

- On utilise la *dissimulation d'information* :

Abstraction	extraire les détails essentiels d'un niveau
Dissimulation d'informations	**rendre inaccessibles des détails qui ne doivent pas influencer d'autres parties du système**

- Exemple : on ne doit pas permettre à un utilisateur (niveau (i)) de pouvoir écrire directement sur un disque logique (niveau (ii)).
- On empêche ainsi les modules de haut niveau de reposer directement sur des détails de bas niveau
- autre exemple : une pile, codée en C supposons avoir un type pile dans `pile.h`

```
typedef struct __pile {
    int elements[TAILLE]; /* donnees, ici un tableau        */
    int *dessus;          /* bas de la pile; ne varie pas   */
    int *dessous;         /* haut de la pile; variable      */
} *pile;
```

et avoir défini les opérations

```
creer_pile(), empiler(), depiler()
```

Entre ces deux parties de code :

```
int     i, elt_pile;
pile   ma_pile;

ma_pile = creer_pile();
/* remplissage de la pile */
for(i = 0; i < 100; i++)
    elt_pile = depiler(ma_pile);
```

et

```
int    elt_pile;
pile   ma_pile;

ma_pile = creer_pile();
/* remplissage de la pile */
elt_pile = (ma_pile->elements)[99];
```

la deuxième solution doit être rejetée. Si l'on change la structure de données utilisée pour une pile (par exemple une liste chaînée au lieu d'un tableau), la dernière partie de code est désastreuse.

- L'abstraction aide à la **maintenabilité** et à l'**intelligibilité** en réduisant le nombre de détails à connaître à chaque niveau.
- La dissimulation d'informations aide à la **fiabilité** (empêcher toute opération non autorisée).

3.3 Exemple de pile basique en C

Voici un exemple de pile avec un tableau

```
#include <stdio.h>
#include <stdlib.h>

#define TAILLE  200

typedef struct __pile {
    int elements[TAILLE]; /* donnees, ici un tableau        */
    int *dessus;          /* bas de la pile; ne varie pas   */
    int *dessous;         /* haut de la pile; variable      */
} *pile;

pile creer_pile();
void empiler(pile ma_pile, int i);
int  depiler(pile ma_pile);
void detruire_pile(pile ma_pile);



/** initialiser une pile **/
pile creer_pile()
{
    pile nouvelle_pile = (pile) NULL;

    nouvelle_pile = (pile)calloc(1, sizeof(struct __pile));
    if (nouvelle_pile == (pile)NULL)
    {
        fprintf(stderr, "Creation de pile impossible\n");
        perror("Message systeme : ");
        exit(1);
    }
    /* bas de la pile   */
    nouvelle_pile->dessous = &(nouvelle_pile->elements)[0];
    /* au debut la pile est vide */
    nouvelle_pile->dessus  = &(nouvelle_pile->elements)[0];
```

```
    return(nouvelle_pile);

}/* creer_pile() */


/** empiler une valeur **/
void empiler(pile ma_pile, int i)
{
    (ma_pile->dessus)++;
    if( ma_pile->dessus == ((ma_pile->dessous) + TAILLE) )
    {
        printf("Debordement de pile\n");
        exit(1);
    }
    *(ma_pile->dessus) = i;

}/* empiler() */


/** depiler une valeur **/
int depiler(pile ma_pile)
{
    if(ma_pile->dessus == ma_pile->dessous)
    {
        printf("Assechement de pile\n");
        exit(1);
    }
    (ma_pile->dessus)--;
    return(*((ma_pile->dessus)+1));

}/* depiler() */


/** destruction d'une pile **/
void detruire_pile(pile ma_pile)
{

    if (ma_pile == (pile)NULL)
    {
        fprintf(stderr, "Destruction de pile impossible\n");
        return;
    }
```

```
    free((void *)ma_pile);

}/*  detruire_pile() */

/** main() **/
void main(void)
{
    int  valeur_entree;
    pile une_pile;

    une_pile = creer_pile();
    do
    {
        printf("\t\tEntrez une valeur a empiler \n \
                (-1 pour sortir, 0 pour depiler) : ");
        scanf("%d", &valeur_entree);
        if(valeur_entree != 0)
            empiler(une_pile, valeur_entree);
        else
            printf("Valeur du dessus de pile : %d\n",
                                    depiler(une_pile));
    } while(valeur_entree != -1);
    detruire_pile(une_pile);

}/* main() */
```

3.4 Uniformité, intégralité, validabilité

- L'*uniformité*, style de codage uniforme, soutient directement l'**intelligibilité**.
- Intégralité et validabilité soutiennent la **fiabilité**, l'**efficacité** et la modifiabilité.

  Abstraction : extrait les détails essentiels
  Intégralité garantit que les éléments importants sont présents

  Abstraction et intégralité : modules nécessaires et suffisants Efficacité améliorée (on peut ajouter les modules de bas niveau indépendamment de ceux de niveau supérieur)
- La *validabilité* implique un développement tel que le logiciel puisse être aisément testé, rendant le système aisément modifiable
- Validabilité et intégralité sont des propriétés peu aisées à mettre en oeuvre. Un fort typage aide à la validabilité

V.4 Stratégie de développement orientée objet

4.1 Modularité et localisation

- D'après Georges Miller, psychologue (1954) l'être humain ne peut gérer plus de 7 ± 2 entités à la fois au sein d'un même niveau.
- Un objet du monde réel devient un objet informatique ; **abstraction** et **dissimulation d'informations** sont la base de tout développement orienté objet
- Les spécifications et leur passage en objets est un cycle en quasi-perpétuelle évolution. Il faut donc un schéma cyclique où l'on peut modifier seulement une partie sans toucher au reste. On opérera donc comme suit :
  1. Identifier les **objets** et leurs **attributs**. Les objets découlent des groupes nominaux utilisés pour les décrire (exemple : une pile)
  2. Identifier les **opérations**. Ce sont les verbes que l'on utilise pour décrire les actions possibles sur l'objet (exemple : `creer_pile()`, `empiler()`, `depiler()`)
  3. Établir la **visibilité**. Par exemple un objet pourra avoir accès à toutes les actions d'une pile mais ne pourra pas voir les fonctions d'allocation dont la pile se sert
  4. Établir l'**interface**. Description de chaque opération avec ses arguments
  5. **Implanter** chaque objet.

VI – Notions de programmation orientée objet

Références bibliographiques

- *Object-Oriented Analysis and Design with Applications*, G. Booch, [Boo94],
- *Data Structures*, M.A. Weiss, [Wei98].

VI.1 POO, Objets, Classes

1.1 Les 5 attributs d'un système complexe

Il existe 5 attributs communs à tous les systèmes "complexes" :

- La complexité prend souvent la forme d'une **hiérarchie** dans laquelle un système complexe est composé de sous-systèmes reliés entre eux et ayant à leur tour leurs propres sous-systèmes, et ainsi de suite jusqu'à ce qu'on atteigne le niveau le plus bas des composants élémentaires.
- Le **choix des composants** primaires d'un système est relativement **arbitraire** et dépend largement de l'observateur du système.
- Les liaisons **intra-composants** sont généralement **plus fortes** que les liaisons **inter-composants**. Ceci a pour effet de séparer les dynamiques **haute fréquence** des composants (celles qui concernent la structure interne des composants) des dynamique **basse fréquence** (celles qui concernent l'interaction entre composants).
- Les systèmes hiérarchiques sont habituellement composés d'un **petit nombre** de genres différents **de sous-systèmes** qui forment des combinaisons et des arrangements variés.
- Un système complexe qui fonctionne a toujours évolué à partir d'**un système simple qui a fonctionné** ... Un système complexe conçu *ex-nihilo* ne fonctionne jamais et ne peut être rapiécé pour qu'il fonctionne. Il faut tout recommencer, à partir d'un système simple qui fonctionne.

1.2 Conception algorithmique

- Comme le suggère Dijkstra : "la technique à appliquer pour maîtriser la complexité est connue depuis très longtemps : *divide et impera* (diviser pour régner)" [Dij79]. Lorsque l'on conçoit un logiciel complexe. il est impératif de le **décomposer** en parties de plus en plus petites, chacune d'elles pouvant être ensuite **affinée indépendamment**.
- Dans la *décomposition algorithmique*, on réalise une analyse structurée descendante où chaque module du système est une étape majeure de quelque processus général.

1.3 Conception orientée objets

- Dans une *décomposition orientée objets*, on partitionne le système selon les entités fondamentales du domaine du problème. Le monde est alors vu comme une série d'agents autonomes qui collaborent pour réaliser un certain comportement de plus haut niveau. De ce point de vue, un objet est une entité tangible, qui révèle un comportement bien défini. Les objets effectuent des opérations, que nous leur demandons en leur envoyant des messages.
- Laquelle des deux décompositions est-elle la plus meilleure ? Les deux sont importantes, la vue algorithmique souligne l'ordre des événements, alors que la vue orientée objets met l'accent sur les agents responsables d'une action et sur les sujets d'une opération. Mais on ne peut construire un système logiciel complexe des 2 manières, car elles sont orthogonales. On utilise l'une, avec l'autre en filigrane pour exprimer l'autre point de vue.

1.4 Terminologie

De manière plus précise

- La *programmation orientée objets* est **une méthode mise en oeuvre dans laquelle les programmes sont organisés comme des ensembles d'objets coopérants. Chacun représente une instance d'une certaine classe, toutes les classes étant des membres d'une hiérarchie de classes unifiée par des relations d'héritage**.
- On désigne donc par langage orienté objets un langage répondant aux conditions suivantes : Un langage orienté objets est tel que :
  - Il supporte des objets qui sont des abstractions de données avec une interface d'**opérations** nommées et un **état** interne caché,
  - les objets ont un **type** associé (la classe),

  - les types (les classes) peuvent **hériter** d'attributs venant de super-types (les super-classes).
- La *conception orientée objets* est **une méthode de conception incorporant le processus de décomposition orienté objets et une notation permettant de dépeindre à la fois les modèles logiques et physiques, aussi bien que statiques et dynamiques du système à concevoir**.
- L'*analyse orientée objets* est **une méthode d'analyse qui examine les besoins d'après la perspective des classes et objets trouvés dans le vocabulaire du domaine du problème**.

1.5 Notion d'objet, de classe

- Nous avons précédemment parlé de manière informelle d'objet comme d'une entité tangible représentant quelque comportement bien défini.
  Nous verrons qu'un *objet* **a un état, un comportement et une identité** ; la structure et le comportement d'objets similaires sont définis dans leur classe commune.
  Les termes *instance* et objet sont **interchangeables**.
- Une *classe* est un **squelette** pour un ensemble d'objets qui partagent une **structure commune** et un **comportement commun**.

VI.2 Type ou classe ; objet

2.1 Notion de type ou de classe

- Un *type* (ou une *classe*) est constituée **d'attributs (ou champs), de déclarations d'opérations (ou signatures de méthodes) et de descriptions extensives d'opérations (ou corps de méthodes)**
- Ce que l'on peut résumer par la formule suivante

$$\texttt{TYPE} \equiv (\texttt{Champs}, \texttt{sig\_meth}_1,\ \texttt{corps\_meth}_1, \ldots, \texttt{sig\_meth}_n,\ \texttt{corps\_meth}_n)$$

  où $\texttt{sig\_meth}_i$ désigne la signature de méthode n° i et $\texttt{corps\_meth}_i$ désigne le corps de méthode n° i.
- Un type est par essence une entité **statique**, par opposition à un objet, de nature **dynamique**. D'une certaine manière, le type est un squelette, l'objet son incarnation.

2.2 Notion d'attribut, ou de champ

- Un *attribut* (ou champ) est une caractéristique d'un type
- Dans une type `Matrice`, le nombre de lignes et le nombre de colonnes sont des attributs. Dans un type `Point`, les coordonnées sont des attributs

2.3 Exemple d'attributs

Matrice 2×2 en Java

```
class Matrice2x2 {
    double a11, a12,
           a21, a22;
    ...
}
```

Matrice $n \times m$

```
class Matrice {
    int    nombreLignes, nombreColonnes;
    double valeurs[];
    ...
}
```

2.4 Notion de déclaration d'opération (ou de méthode)

- La *déclaration (ou signature) d'une opération (ou méthode)* est constituée du nom de la méthode suivi de ses paramètres et précédé de son type de retour. Par exemple, l'opération d'addition de matrices peut se déclarer comme suit :

  `Matrice ajouterMatrice(Matrice m1, Matrice m2)`

  L'opération `ajouterMatrices()` renvoie l'objet `m1 + m2` de type `Matrice`

2.5 Notion de description d'opération (ou de méthode)

- La *description extensive d'une opération (ou corps de méthode)* est la suite des opérations (primitives ou non) qu'elle réalise. Une *opération primitive* est une instruction du langage.

2.6 Exemple de description d'opération

Par exemple, l'opération `ajouterMatrices()` pourrait avoir comme corps simpliste (c.à.d. sans test d'aucune sorte) :

```
Matrice ajouterMatrice(Matrice m1, Matrice m2)
{
  Matrice somme = new Matrice(m1.nombreLignes,
                              m1.nombreColonnes);

  for(int i = 0; i < m1.nombreLignes; i++)
    for(int j = 0; j < m1.nombreColonnes; j++)
    {
      (somme.valeurs)[i][j] = (m1.valeurs)[i][j] +
                              (m2.valeurs)[i][j];
    }

  return somme;
}
```

2.7 Exemple de classe

Classe (type) décrivant un cercle

```
class Cercle {
  // champs : rayon du cercle
  double r;
  // Constructeur : initialisation des champs
  Cercle(double nouvRayon) {
      r = nouvRayon;
  }
  // methode de calcul dŠune surface
  double calculeSurface() {
      return(3.1416*r*r);
  }
}// Fin de class Cercle
```

2.8 Notion d'état d'un objet

- L'*état* d'un objet **englobe toutes les propriétés (habituellement statiques) de l'objet plus les valeurs courantes (généralement dynamiques) de chacune de ces propriétés**.
- Une *propriété* est une caractéristique naturelle ou discrminante, un trait, une qualité ou une particularité qui contribue à rendre un objet unique. Par exemple, dans un distributeur, un numéro de série est une propriété statique et la quantité de pièces qu'il contient est une valeur dynamique.

2.9 Notion de comportement d'un objet

- Le *comportement* est **la façon dont un objet agit et réagit, en termes de changement d'état et de transmission de messages**.
- Généralement, un *message* est simplement une opération qu'un objet effectue sur un autre, bien que le mécanisme utilisé soit quelque peu différent. Dans la suite, les termes opération et message sont interchangeables.
- Dans la plupart des langages orientés objets et basés sur objets, les opérations que les clients peuvent effectuer sur un objet sont typiquement appelées des *méthodes*, qui font partie de la classe de l'objet.
- La transmission de messages est une partie de l'équation qui définit le comportement d'un objet. L'état d'un objet influence aussi son comportement. Par exemple, dans le cas d'un distributeur de boissons, nous pouvons déclencher une action (appuyer sur un bouton) pour réaliser notre sélection.
  - Si nous n'avons pas introduit suffisamment d'argent, il ne se passera probablement rien.
  - Si nous avons mis assez d'argent, la machine l'encaissera et nous servira une boisson (modifiant ainsi son état).

  Nous pouvons donc affiner la notion d'état :
- L'*état* d'un objet **représente les effets cumulés de son comportement**.
- La majorité des objets intéressants n'ont pas d'état entièrement statique. Ils contiennent des propriétés dont les valeurs sont lues et modifées en fonction des actions sur ceux-ci.

2.10 Comportement d'un objet : les opérations

- Une *opération* désigne un service qu'une classe offre à ses clients. En pratique, nous avons constaté qu'un client effectuait typiquement 5 sortes d'opérations sur un objet. Les 3 les plus courantes sont les suivantes :

Modificateur	une opération qui altère l'état d'un objet
Sélecteur	une opération qui accède à l'état d'un objet, mais qui n'altère pas celui-ci.
Itérateur	une opération qui permet d'accéder à toutes les parties d'un objet dans un ordre bien défini.

- Deux autres types d'opération sont courants :

Constructeur	une opération qui crée un objet et/ou initialise son état.
Destructeur	Une opération qui libère l'état d'un objet et/ou détruit l'objet lui-même.

- Avec des langages orientés objets, comme Java ou Smalltalk, **les opérations peuvent seulement être déclarées comme méthodes** (c.à.d. au sein d'une classe), le langage ne nous autorisant pas à déclarer des procédures

ou des fonctions séparées de toute classe. Ce n'est pas le cas en C++ et en AdA, qui autorisent le programmeur à écrire des opérations en dehors de toute classe.

2.12 Comportement d'un objet : rôle et responsabilités

- L'ensemble des méthodes associées à un objet constituent son *protocole*. Ce dernier définit la totalité des comportements autorisés pour l'objet. Il est utile de diviser ce protocole en groupes logiques de comportements.
- Ces groupes désignent les *rôles* que l'objet peut jouer ; un rôle s'apparente à un masque porté par l'objet et définit un contrat entre une abstraction et un client.
- Les *responsabilités* d'un objet sont constituées d'une part de la connaissance que l'objet maintient et d'autre part des actions qu'il peut réaliser. En d'autres termes, L'état et le comportement d'un objet définissent l'ensemble des rôles qu'il peut jouer, lesquels définissent les responsabilités de l'abstraction.
- La majorité des objets jouent plusieurs rôles au cours de leur existence, par exemple :
  - Un compte bancaire peut être créditeur ou débiteur, ce qui influe sur le comportement d'un retrait d'argent.
  - Durant une même journée, un individu peut jouer le rôle de mère, de médecin, de jardinier.

2.13 Comportement d'un objet : les objets en tant que machines

- L'existence d'un état dans un objet signifie que l'ordre dans lequel les opérations sont invoquées est important. Chaque objet peut donc être vu comme une **petite machine** ou un **automate à état finis** équivalent.
- Les objets peuvent être actifs ou passifs. Un objet *actif* contient sa propre tâche de contrôle, contrairement à un objet *passif*. Les objets actifs sont généralement **autonomes**, ce qui signifie qu'ils peuvent présenter un certain comportement sans qu'un autre objet agisse sur eux. Les objets passifs ne peuvent subir un changement d'état que lorsque l'on **agit explicitement sur eux**.

2.14 Notion d'identité d'un objet

- L'*identité* **est cette propriété d'un objet qui le distingue de tous les autres objets**.

- Deux objets peuvent être déclarés égaux en 2 sens différents. Ils peuvent être égaux au sens de leur **références** (les pointeurs internes qui référencent les données de l'objet en mémoire) ou au sens de leur **contenu** (égalité de leur état), bien qu'ils soient situés à des emplacements mémoire différents.

2.15 Nature d'un objet en Java

- Un objet Java peut être décrit par la formule suivante :

$$\texttt{OBJET} \equiv (\texttt{état}, \texttt{op}_1, \ldots, \texttt{op}_n, \texttt{ref})$$

  où `etat` ensemble des variables d'instance
  $\texttt{op}_i$ (pointeur sur) la méthode d'instance n° i
  `ref` (pointeur sur) un emplacement mémoire contenant l'état et des références internes vers les opérations (pointeurs sur les méthodes)
- Exemple d'objet, de type tasse à café. Des **attributs** d'une tasse à café pourront être :
  – sa couleur,
  – la quantité de café qu'elle contient,
  – sa position dans le café (la brasserie ou le bar)
  "Tasse à café" est un type et "la tasse à café rouge qui contient actuellement 38 millilitres de café et qui se trouve sur la dernière table du fond" est un objet. "Rouge", "38 millilitres" et "sur la dernière table du fond" constituent l'état de cet objet.
- Un type, ou une classe sert de modèle à partir duquel on peut *instancier* (créer) des objets contenant des variables d'instance et des méthodes définies dans la classe.

VI.3 Relations

3.1 Séparation de l'interface et de l'implantation

- Une idée clé est de **séparer l'interface externe d'un objet de son implantation**.
- L'*interface* d'un objet est constituée des messages qu'il peut accepter d'autres objets. Autrement dit, c'est la déclaration des opérations associées à l'objet.
- L'*implantation d'un objet* se traduit par la valeur de ses attributs et son comportement en réponse aux messages reçus.
- Dans un monde orienté-objets, un objet expose son interface aux autres objets, mais garde son implantation privée. L'implantation doit donc être séparée de l'interface. De l'extérieur, **le seul moyen pour interagir avec**

un objet est de lui envoyer un message (d'exécuter l'une de ses opérations).
- La séparation de l'interface et de l'implantation permet aux objets d'avoir la **responsabilité** de gérer leur propre état. Les autres objets ne peuvent manipuler cet état directement et doivent passer par des messages (ou opérations). L'objet qui reçoit un message peut décider de changer ou non son état. Par contre, il ne contrôle pas à quel instant il va recevoir des messages.
- Un aspect fondamental de la programmation orientée objet est que **chaque objet d'une classe particulière peut recevoir les mêmes messages**. L'interface externe d'un objet ne dépend donc que de sa classe.

3.4 Relations entre classes

- Il existe trois types fondamentaux de relations entre classes :
  - La **généralisation/spécialisation**, désignant une relation "est un". Par exemple, *une rose est une sorte de fleur* : une rose est une sous-classe plus spécialisée de la classe plus générale de fleur.
  - L'**ensemble/composant**, dénotant une relation "partie de". Par exemple, *un pétale est une partie d'une fleur*.
  - L'**association**, traduisant une dépendance sémantique entre des classes qui ne sont pas reliées autrement. Par exemple, *une fleur et une bougie* peuvent *ensemble* servir de décoration sur une table.
- La plupart des langages orientés objets comprennent des cominaisons des relations suivantes entre classes :
  - Association.
  - Héritage.
  - Agrégation.
  - Utilisation.
  - Instanciation.
  - Méta-classe.

3.5 Relations d'association entre classes

- *Relations d'association.* Une association dénote une dépendance sémantique. Par exemple, les objets de type `Client` et ceux de type `Facture` peuvent être associés dans le cas d'une commande d'un produit.
- On associe souvent à ce type de relation une *cardinalité*. L'exemple précédent exhibe une cardinalité de 1 pour n, un client pouvant avoir plusieurs factures qui lui sont associées. On distingue les cardinalités :
  - 1 pour 1,
  - 1 pour n,

– n pour n.
Une association 1 pour 1 est très étroite. Par exemple entre la classe `Facture` et la classe `TransactionCarteBancaire`.

3.6 Relations d'héritage entre classes

- L'*héritage* est une relation entre les classes dont l'une partage la **structure** ou le **comportement** défini dans une (*héritage simple*) ou plusieurs (*héritage multiple*) autres classes. On nomme *super-classe* la classe de laquelle une autre classe hérite. On appelle une classe qui hérite d'une ou plusieurs classes une *sous-classe*.
- Par exemple, prenons une classe `Surface2DSymetrique`. Considérons les classes `Pave2D` et `Disque` héritant de `Surface2DSymetrique`.
- L'héritage définit donc une **hiérarchie de la forme "est un"** entre classes. C'est le **test de vérité de l'héritage**.
- Dans une relation d'héritage, les sous-classes héritent de la structure de leur super-classe. Par exemple, la classe `Surface2DSymetrique` peut avoir comme champs :

l'abscisse de son centre de symétrie	`x`
l'ordonnée de son centre de symétrie	`y`
sa taille	`size`
sa couleur	`color`

  Et les classes `Pave2D` et `Disque` hériteront de ces champs. Une sous-classe peut définir d'autres champs qui viennent s'ajouter à ceux hérités des super-classes.
- De plus, toujours dans une relation d'héritage, les sous-classes héritent du comportement de leur super-classe. Par exemple, la classe `Surface2DSymetrique` peut avoir comme opérations :

`getSize()`	pour obtenir la taille de la surface
`getX()`	pour obtenir l'abscisse du centre de gravité
`getY()`	pour obtenir l'ordonnée du centre de gravité
`setXY()`	pour fixer la position de la surface
`setColor()`	pour fixer la couleur de la surface

  Et les classes `Pave2D` et `Disque` hériteront de ces champs. Une sous-classe peut définir d'autres opérations qui viennent s'ajouter à celles héritées des super-classes. En outre, une sous-classe peut redéfinir tout ou partie des opérations héritées des super-classes.
- Le *polymorphisme* est un mécanisme par lequel un nom peut désigner **des objets de nombreuses classes** différentes, tant qu'elles sont reliées par une super-classe commune. Tout objet désigné par ce nom est alors capable de répondre de différentes maniàres à un ensemble commun d'opérations.

3.12 Relations d'aggrégation entre classes

- L'aggrégation peut se faire par inclusion physique ou sémantique.
- Dans le cas d'inclusion physique, il peut y avoir :
  - *inclusion de valeur*, auquel cas l'objet inclus ne peut exister sans l'instance de l'objet englobant,
  - *inclusion de référence*, le lien étant plus indirect ; on peut alors créer et détruire indépendamment les instances de chaque classe.
- On peut avoir une représentation d'aggrégation plus indirecte, seulement *sémantique*. Par exemple, on peut déclarer une classe `Investisseur` contenant une clé dans une base de données qui permette de retrouver les actions que possède l'investisseur.
- Le test de relation d'aggrégation est le suivant : si (et seulement si) il existe une **relation ensemble/composant** entre deux objets, il doit y avoir une relation d'aggrégation entre leurs classes respectives.

3.13 Relations d'utilisation entre classes

Les relations d'utilisation entre classes sont similaires aux liens d'égal à égal entre les instances de ces classes. Une association indique un lien sémantique bidirectionnel ; une relation d'utilisation est une des évolutions possibles d'une association. On y précise l'abstraction cliente et l'abstraction fournisseur de certains services.

3.14 Relations d'instanciation entre classes

On veut utiliser des instances de classes distinctes qui ne sont pas reliées par une super-classe commune, en effectuant des opérations de manière générique. Soit le langage comprend directement les types génériques (comme par exemple C++), soit on peut (par exemple en Java) créer des classes conteneurs généralisées et utiliser du code de vérification de type (en utilisant en java la réflexion) pour imposer que tous les éléments contenus soient tous de la même classe.

3.15 Relations de méta-classes

On traite ici une classe comme un objet qui peut être manipulé. On obtient donc la classe d'une classe ou *méta-classe*. Cette notion n'est pas explicite dans le langage Java, mais la technique de réflexion s'en rapproche.

VII – Bases orientées objet de Java

Références bibliographiques

- *The Java Language Specification*, J. Gosling, B. Joy et G. Steele [GJS96],
- *Java in a Nutshell*, D. Flanagan [Flab].

VII.1 Classes et objets Java

1.1 Constitution d'une classe

- Rappel des notions de classe et d'objet, en deux mots (voir chapitre VI, p. 49) :
  - **Classe** : squelette ; *structure de données et code des méthodes* ; statique, sur disque
  - **Objet** : incarnation ; *état, comportement, identité* ; dynamique, en mémoire
- Une classe définit généralement deux choses :
  - les **structures de données** associées aux objets de la classe ; les variables désignant ses données sont appelés *champs*.
  - les **services** que peuvent rendre les objets de cette classe qui sont les *méthodes* définies dans la classe.

  Une **Classe** java est **déclarée** par le mot clé `class`, placé devant l'identificateur de la classe (son nom).

1.2 Champs et méthodes

- Un *champ* correspond à une **déclaration de variable**, le nom de la variable suivant la déclaration de son type :

```
class Point {
    int x;
    int y;
```

```
    ...
}
```

- Une *méthode* est constituée de :
  - un nom constitué par un identificateur
  - des paramètres formels : ceux-ci sont séparés par des ",". Lorsque la méthode n'a pas de paramètre, contrairement au langage C , il ne faut pas préciser void. Le nombre de paramètres est fixe : il n'est pas possible de définir des méthodes à arguments variables.
  - du type de retour est soit `void` (si la méthode ne retourne aucune valeur), soit un type primitif ou une référence vers un objet.
  - du corps de la méthode.
- Exemple de classe décrivant un cercle

```
class Cercle {
  // champs : rayon du cercle
  double r;
  // methode de calcul dŠune surface
  double calculeSurface() {
      return(3.1416*r*r);
  }
}// Fin de class Cercle
```

1.3 Déclaration de classe

- Un fichier source java doit **porter le même nom** que celui de la classe publique qui y est définie. Syntaxe générique

```
class NomClasse {
    type variableInstance1;
    type variableInstanceN;
    type nomMethode1(liste-parametres) {
        corps-methode;
    }
    type nomMethodeN(liste-parametres) {
        corps-methode;
    }
}
```

- Exemple

```
class Chat {
    String nom;              // nom du fauve
    int    age;              // en annees
    float  tauxRonronnement; // entre 0 et 1
```

```
    void viellir() {
        age += 1;
    }

    int retournerAge() {
        return(age);
    }
}
```

- Les **déclaration et implantation** d'une méthode sont **dans le même fichier**. Ceci donne parfois de gros fichier source (`.java`), mais il est plus facile (pour la maintenance) d'avoir les spécification, déclaration et implantation au même endroit.

1.4 Point d'entrée d'un programme (`main()`)

Un programme Java est constitué d'une ou de plusieurs classes. Parmi toutes ces classes, il doit exister au moins une classe qui contient la méthode statique et publique `main()` qui est le point d'entrée de l'exécution du programme.

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

Cette classe définit une classe Bonjour qui ne possède qu'une seule méthode. La méthode `main()` **doit être** déclarée `static` et `public` pour qu'elle puisse être invoquée par l'interpréteur Java. L'argument `args` est un tableau de `String` qui correspond aux *arguments de la ligne de commande* lors du lancement du programme. *`args[0]`* est le **1er argument**, `args[1]` est le 2ième argument, ...

1.5 Compilation

- Avant de pouvoir exécuter ce programme, il faut tout d'abord le compiler, par exemple avec la commande *`javac`* (sous le JDK standard, c.à.d. l'environnement de base).

  ```
  javac Bonjour.java
  ```

  La commande `javac` traduit le code source en code intermédiaire (*p-code*) java. Ce code (une forme d'assembleur générique) est évidemment **indépendant de la plate forme** sur laquelle il a été compilé.

1.6 Exécution

Autant de fichiers que **de classes** qui ont été définies dans le fichier source sont produits. Les fichiers *compilés* ont l'extension *.class*. Enfin, pour exécuter ce programme, il faut utiliser l'interpréteur de code Java et lui fournir le **nom de la classe** publique que l'on veut utiliser comme point de départ de notre programme (celle contenant la méthode `main(...)`), ans l'extension.

```
java Bonjour
```

1.7 Référence à un objet

- En Java, on ne peut accéder aux objets **qu'à travers une référence** vers celui-ci. Déclaration d'une variable `p` avec pour type un nom de classe :

  ```
  Point p;
  ```

  `p` : *référence à un objet* de la classe `Point`. Lorsque l'on déclare une classe comme type d'une variable, cette dernière a, par défaut, la valeur `null`. *`null`* est une référence à un `Object` (mère de toutes les classes Java), qui n'a pas de valeur (distinct de `0`) ; par ex. dans

  ```
  Point p;
  ```

  `p` a la valeur `null`.
- En fait, référence à un objet : pointeur. Mais l'**arithmétique** sur les **pointeurs est impossible** en java. **Seule chose permise** : **changer la valeur** de la référence pour pouvoir "faire référence" à un autre objet. Plus précisément, une référence pointe sur une structure où se trouve des informations sur le type ainsi que l'adresse réelle des données de l'instance d'objet.

1.8 Opérateur `new`

- `new` : **création** d'une instance **d'objet** d'une classe ; retourne une référence à cette instance d'objet.

  ```
  Point p  = new Point();    // ligne 1
  Point p2 = p;              // ligne 2
  p = null;                  // ligne 3
  ```

  Ligne 2 : tout changement à `p2` affecte l'objet référencé par `p`. `p2 = p` : aucune copie de l'objet ou allocation mémoire.

  Ligne 3 : décrochage de `p` de l'objet originel. `p2` permet toujours d'y accéder.
- Objet qui n'est plus référencé ⇒ le *ramasse-miettes* (*garbage collector*) récupère automatiquement la mémoire associée.

1.9 Instance d'objet

- *Instance* : copie individuelle de prototype de la classe, avec ses propres données : *variables d'instance*.
- Une fois la classe déclarée, pour pouvoir utiliser un objet de cette classe, il faut définir une **instance (d'objet)** de cette classe. Or les objets ne sont accessibles qu'à travers des références . Donc une définition qui spécifie un objet comme "une variable ayant le type de la classe choisie " ne fait que définir une référence vers un éventuel objet de cette classe.

  ```
  Date d;
  ```

  La variable `d` représente une référence vers un objet de type `Date`. En interne, cela réserve de la place pour le pointeur sous-jacent à la référence `d`. Mais cela **ne réserve pas de place mémoire pour une variable** de type `Date`.
- Si l'on veut une instance d'objet effective, il faut la créer explicitement avec le mot clé `new` et le constructeur de la classe `Date`.

  ```
  Date d;
  d =  new Date();
  ```

1.10 Méthode d'instance

- On peut voir une méthode comme un message envoyé à une instance d'objet. Pour afficher la date contenue dans l'objet d, on lui envoie le message imprimer :
  `d.imprimer()` ; De telles méthodes sont appelées *méthodes d'instance*.

1.11 Variables d'instance

- Les **Variables d'instance** sont déclarées en dehors de toute méthode

  ```
  class Point {
      int x, y;
  }
  ```

1.12 Op. point (.) – Déclaration de méthode

- Opérateur . : accéder à des variables d'instance et à des méthodes d'un(e instance d'un) objet.
- Ex. de déclaration de méthode

  ```
  class Point {
      int x, y;
      void init(int a, int b) {
          x = a;
  ```

```
            y = b;
        }
    }
```

En C, méthode sans paramètre : `nommethode(void)`. **illégal en java**. Les **objets sont passés par référence** (références d'instances à un objet passés par valeur). Les **types primitifs sont passés par valeur**. Les méthodes java sont donc similaires aux fonctions virtuelles du C++.

1.13 Instruction `this`

- *`this`* : référence à l'instance d'objet courante.
- ☛ Il est permis à une variable locale de **porter le même nom** qu'une variable d'instance ... Exemple d'utilisation de `this` évitant cela

```
void init(int x, int y) {
    this.x = x;
    this.y = x;
}
```

1.14 Constructeurs

- Même nom que celui de la classe. Pas de type de retour (pas même `void`).
- Classe décrivant un cercle

```
class Cercle {
  double r;  // champs : rayon du cercle
  // Constructeur : initialisation des champs
  Cercle(double nouvRayon) {
      r = nouvRayon;
  }
  double calculeSurface() {
      return(3.1416*r*r);  // methode de calcul
  }
}// Fin de class Cercle
```

- Exemple animalier

```
class Chat {
    String nom;              // nom du fauve
    int    age;              // en annees
    float  tauxRonronnement; // entre 0 et 1

    public Chat(String sonNom,
                int    sonAge,
                float  sonTauxRonron) {
```

```
        nom              = sonNom;
        age              = sonAge;
        tauxRonronnement = sonTauxRonron;
    }
}
```

- `this` peut-être également un appel à un *constructeur*

```
class Point {
    int x, y;
    // constructeur exhaustif
    Point(int x, int y) {
        this.x = x;   // var d'instance Point.x
        this.y = y;
    }
    // Appel du constructeur exhaustif
    Point() {
        this(-1, -1); // Point(int x, int y)
        }
}
```

1.15 Exemple de constructeurs

- Exemple animalier

```
class Chat {
    String  nom;              // nom du fauve
    int     age;              // en annees
    Color[] couleurPelage;    // ses differentes couleurs
    float   tauxRonronnement; // entre 0 et 1

    public Chat(String sonNom,
                int    sonAge,
                float sonTauxRonron,
                Color[] sesCouleurs) {
        nom              = sonNom;
        age              = sonAge;
        tauxRonronnement = sonTauxRonron;
        couleurPelage    = sesCouleurs;
    }

    public Chat() {
        this(new String("minou"), 1, 0.5,
             {Color.black, Color.white});
```

```
        }
    }
```

- Technique de *réutilisation* : créer un constructeur exhaustif (doté de tous les paramètres), puis créer d'autres constructeurs appelant systématiqement le constructeur exhaustif.

VII.2 Héritage

2.1 Héritage

- Les descendants par héritage sont nommés des *sous classes*. Le parent direct est une *super classe*. Une sous classe est une version **spécialisée** d'une classe qui **hérite** de toutes les variables d'instance et méthodes.

Mot-clé *extends*

```
class Point3D extends Point {
    int z;
    Point3D(int x, int y, int z) {
        this.x = x;
        this.y = y;
        this.z = z;
    }
    Point3D() {
        Point3D(-1, -1, -1);
    }
}
```

- Syntaxe générique

```
class NomClasse {
    type variableInstance1;
    type variableInstanceN;
    type nomMethode1(liste-parametres) {
        corps-methode;
    }
    type nomMethodeN(liste-parametres) {
        corps-methode;
    }
}
```

- **Pas d'héritage multiple**, pour des raisons de performances et de complexité (en maintenance). À la place, notion d'*interface*.
- Il existe une classe au sommet de la hiérarchie, *Object*. Sans mot-clé `extends`, le compilateur met automatiquement `extends Object`.

- De la même manière que l'on peut assigner à une variable `int` un `byte`, on peut **déclarer une variable de type** `Object` et y **stocker une référence** à une instance de toute **sous classe** d'`Object`.

VII.3 Surcharge, redéfinition

3.1 Instruction `super`

- *`super`* réfère aux variables d'instance et aux constructeurs de la *super classe*.

```
class Point3D extends Point {
    int z;
    Point3D(int x, int y, int z) {
        super(x, y);  // Appel de Point(x,y).
        this.z = z;
    }
}
```

- Cet appel au constructeur de la classe mère doit être **la 1ière ligne du constructeur**.
- `super` peut également se référer aux méthodes de la super classe : `super.distance(x, y)` appelle la méthode `distance()` de la super classe de l'instance `this`.
- Exemple animalier (voir l'**excellent** ouvrage "*le mystère des chats peintres*" de Heather Busch et Burton Silver, `http ://www.monpa.com/wcp/index.html`)

```
class ChatPeintre extends Chat {
  // Variables d'instances
  String    style;
  int       coteMoyenne;  // cote moyenne d'une oeuvre

  // Constructeurs
  public ChatArtiste(String sonNom, int sonAge,
                     float sonTauxRonron,
                     Color[] sonPelage,
                     String sonStyle, int saCote) {
      super(sonNom, sonAge, sonTauxRonron, sonPelage);
      style       = sonStyle;
      coteMoyenne = saCote;
  }

  // Methodes
  public peindre() {
      ...
```

```
    }
  }
```

3.2 Un artiste en pleine action

source : `http ://www.monpa.com/wcp/index.html`

3.3 Sous-typage, transtypage, `instanceof`

- Le typage d'une variable lui permet de référencer tout sous type (classe parente) ; la méthode `miauler()` est définie dans `Chat`. La méthode `peindre()` n'est définie que dans `ChatPeintre`.

```
Chat gouttiere = new Chat("zephir", 1, 0.9);
ChatPeintre moustacheDeDali =
  new ChatPeintre("dali",   // nom de l'artiste
                  2,        // son age
                  0.1,      // son taux rr
                 {Color.white, Color.black},
                 "aLaDali",// son style
                  20000);  // sa cote moyenne

moustacheDeDali.peindre();  // valide
gouttiere.peindre();        // illegal
```

- `instanceof` permet de savoir si un objet est d'un type donné ou non.

```
// true
System.out.print(gouttiere instanceof Chat);
// true
System.out.print(moustacheDeDali instanceof Chat);
// false
System.out.print(gouttiere instanceof ChatPeintre);
moustacheDeDali = null;
// false
System.out.print(moustacheDeDali instanceof ChatPeintre);
```

- Transtypage (ou "cast" en anglais) permet de changer le type, lorsque cela est permis.

```
Chat ch = new Chat("zephir", 1, 0.9);
ChatPeintre chP;
chP = ch;               // Erreur de compilation

if (ch instanceof ChatPeintre)// Bonnes manieres
    chP = (ChatPeintre)ch;    // transtypage
```

3.4 Surcharge de méthode

- Plusieurs méthodes peuvent porter le même nom : *surcharge de méthode*.
- Différentiation sur la *signature de type* : le **nombre et le type des paramètres**. Deux méthodes d'une même classe de mêmes nom et signature de type est illégal.
- Exemple de surcharge

```
class Point {
    int x, y;
    Point(int x, int y) {
        this.x = x;          this.y = y;
    }
    double distance(int x, int y) {
        int dx = this.x - x; int dy = this.y - y;
        return Math.sqrt(dx*dx + dy*dy);
    }
    double distance(Point p) {
        return distance(p.x, p.y);
    }
}
class PointDist {
    public static void main(String args[]) {
        Point p1 = new Point(0, 0);
        Point p2 = new Point(30, 40);
        System.out.println("p1.distance(p2) = " +
                                        p1.distance(p2));
        System.out.println("p1.distance(60, 80) = " +
                                        p1.distance(60, 80));
}
```

- Exemple animalier

```
class Chat {
      ....
void vieillir() {
    age += 1;
}

void vieillir(int n) {  // Surcharge de methode
    age += n;
}
```

3.5 Redéfinition de méthode

- Distance en perspective dans `Point3D` (distance 2D entre `x/z` et `y/z`) ⇒ *redéfinir* `distance(x, y)` de `Point2D`. Ex. de surcharge de distance 3D et de redéfinition de distance 2D

```
class Point {
    int x, y;
    Point(int x, int y) {
        this.x = x;
```

```
            this.y = y;
        }
        double distance(int x, int y) {
            int dx = this.x - x;
            int dy = this.y - y;
            return Math.sqrt(dx*dx + dy*dy);
        }
        double distance(Point p) {              // Surcharge
            return distance(p.x, p.y);
        }
    }// class Point

class Point3D extends Point {
  int z;
  Point3D(int x, int y, int z) {
    super(x, y);      // Appel de Point(x,y)
    this.z = z;
  }
  double distance (int x, int y, int z) {
    int dx = this.x - x;  int dy = this.y - y;
    int dz = this.z - z;
    return Math.sqrt(dx*dx + dy*dy + dz*dz);
  }
  double distance(Point3D other) {  // Surcharge
    return distance(other.x, other.y, other.z);
  }
  double distance(int x, int y) {   // Redefinition
    double dx = (this.x / z) - x;
    double dy = (this.y / z) - y;
    return Math.sqrt(dx*dx + dy*dy);
  }
}

class Point3DDist {
    public static void main(String args[]) {
        Point3D p1 = new Point3D(30, 40, 10);
        Point3D p2 = new Point3D(0, 0, 0);
        Point   p  = new Point(4, 6);
        System.out.println("p1.distance(p2) = " +
                                p1.distance(p2));
        System.out.println("p1.distance(4, 6) = " +
                                p1.distance(4, 6));
        System.out.println("p1.distance(p) = " +
                                p1.distance(p));
```

```
    }
}
```

L'affichage du programme est le suivant. **Pourquoi** ?

```
Prompt > java Point3DDist
p1.distance(p2) = 50.9902
p1.distance(4,6) = 2.23607
p1.distance(p) = 2.23607
```

- Appel de `distance` sur un `Point3D` (`p1`) : exécution de `distance(Point p)` héritée de la super classe (méthode non redéfinie). Mais ensuite **appel de** `distance(int x, int y)` **de** `Point3D`, pas de `Point`.
- Sélection de méthode **selon le type de l'instance** et non selon la classe dans laquelle la méthode courante s'exécute : *répartition de méthode dynamique*.

3.6 Répartition de méthode dynamique

```
class Parent {
    void appel() {
        System.out.println("Dans Parent.appel()");
    }
class Enfant extends Parent {
    void appel() {
        System.out.println("Dans Enfant.appel()");
}
class Repartition {
    public static void main(String args[]) {
        Parent moi = new Enfant();
        moi.appel();
    }
}
```

- Lors de `moi.appel()`
  - Le compilateur vérifie que `Parent` a une méthode `appel()`,
  - l'environnement d'exécution remarque que la référence `moi` est en fait vers une instance d'`Enfant` ⇒ appel de `Enfant.appel()`
- Il s'agit d'une forme de *polymorphisme* à l'exécution.
- Cela permet à des bibliothèques existantes d'appeler des méthodes sur des instances de **nouvelles** classes **sans recompilation**.

3.7 Instruction `final`

- *Variable* d'instance ou *méthode non redéfinissable* : `final`. Pour des variables, convention de majuscules

  `final int FILE_QUIT = 1 ;` Les sous classes ne peuvent redéfinir les méthodes `final`. Petites méthodes `final` peuvent être optimisées (appels "en ligne" par recopie du code).
- ☛ `final` pour les variables est similaire au `const` du C++. Il n'y a pas d'équivalent de `final` pour les méthodes en C++.

3.8 Méthode `finalize()`

- Instance d'objet ayant une resource non java (descripteur de fichier) : moyen de la libérer.
- Ajout d'une *méthode* `finalize()` à la classe. **Appelée à chaque libération** d'une instance d'objet de cette classe.

3.9 Instruction `static`

- *méthode* `static` : utilisée **en dehors de tout contexte d'instance**.
- Méthode `static` ne peut appeler directement que des méthodes `static`. Ne peut utiliser `this` ou `super`. Ne peut utiliser une variable d'instance.
- ➘ *Variables* `static` : **visibles de toute autre portion de code**. Quasiment des variables globales. À utiliser avec parcimonie ...
- *Bloc* `static` : **exécuté une seule fois**, au premier chargement de la classe.

– Exemple

```
class Statique {
    static int a = 3;
    static int b;
    static void methode(int x) {
    System.out.println("x = " + x +
                       ", a = " + a +
                       ", b = " + b);
    }
     static {
          System.out.print("Initialisation" +
                           " du bloc statique");
           b = a * 4;
     }
     public static void main(String args[]) {
           methode(42);
```

```
    }
}
```

L'affichage est

```
Prompt > java Statique
Initialisation du bloc statique
x = 42, a = 3, b = 12
```

Initialisation de `a` et `b`. Exécution du bloc `static`. Appel de `main()`.

– Appel d'une variable ou méthode `static` par le `nom de la classe`

```
class ClasseStatique {
    static int a = 42;
    static int b = 99;
    static void appel() {
        System.out.println("a = " + a);
    }
}
class StatiqueParNom {
    public static void main(String args[]) {
        ClasseStatique.callme();
        System.out.println("b = " +
                           ClasseStatique.b);
    }
}
```

– Exemple animalier

```
class Chat {
  String     nom;              // nom
  int        age;              // annees
  Color[]    couleurPelage;    // couleurs
  float      tauxRonronnement; // de 0 a 1
  static int ageSevrage = 1;   // statique

  boolean estAdoptable() {
      if (age > ageSevrage) {
          return true;
      } else {
          return false;
      }
  }
}
```

3.10 Instruction `abstract`

- **Partie spécification, partie implantation** : *classes abstraites* .
- Certaines méthodes, sans corps, **doivent être redéfinies** par les sous classes : méthodes *abstraites*. C'est la *responsabilité de sous classe*.
- Toute classe contenant des méthodes abstraites (mot clé `abstract`) doit être déclarée abstraite. Les classes abstraites ne peuvent être instanciées par `new`. Pas de constructeurs ou de méthodes `static`. Une **sous classe** d'une classe statique **soit implante** toutes les méthodes abstraites, **soit est elle-même abstraite**.

Exemple

```
abstract class ParentAbstrait {
    abstract void appel();
    void moiaussi() {
        System.out.print("Dans ParentAbstrait.moiaussi()");
    }
}

class EnfantConcret extends ParentAbstrait {
    void appel() {
        System.out.print("Dans EnfantConcret.moiaussi()");
    }
}

class AbstractionMain {
    public static void main(String args[]) {
        ParentAbstrait etre = new EnfantConcret();
        etre.appel();
        etre.moiaussi();
    }
}
```

VII.4 Paquetages et interfaces

4.1 Paquetages

- À la fois un *mécanisme de nommage et un mécanisme de restriction de visibilité*.
- Forme générale d'un source java

```
une unique declaration de paquetage (optionnel)
declarations d'importations (optionnel)
```

```
une unique declaration de classe publique
declarations de classes privees (optionnel)
```

- Pas de déclaration de paquetage : les classes déclarées font partie du paquetage par défaut, sans nom. Une classe déclarée dans le paquetage `monPaquetage` ⇒ le source **doit être dans le répertoire** `monPaquetage` (il y a distinction minuscule-majuscule).

Syntaxe générique :

```
package pkg1[.pkg2[.pkg3]] ;
```

Par exemple `package java.awt.image ;` doit être stocké dans `java/awt/image` (sous UNIX), `java\awt\image` (sous Windows) ou `java :awt :image` (sous Macintosh).

La racine de **toute hiérarchie de paquetage** est une entrée de la variable d'environnement `CLASSPATH`.

↘ Ayant une classe `ClasseTest` dans un paquetage `test`, il faut

– soit **se mettre dans le répertoire père** de `test` et lancer `java test.ClasseTest`,

– soit **ajouter le répertoire** `test` **à la variable** `CLASSPATH` :

```
CLASSPATH=.;c:\code\test;c:\java\classes
```

– soit **lancer** :

```
java -dclasspath=.;c:\code\test;c:\java\classes ClasseTest
```

4.2 Instruction `import`

- Entrer les noms complets de classes et méthodes fort long ⇒ Tout ou partie d'un paquetage est amené en visibilité directe, avec `import`.
- Syntaxe générique `import pkg1[.pkg2].(nomclasse|*) ;`. Exemple

```
import java.util.Date;
import java.io.*;
```

  Chargement de gros paquetages ⇒ perte de performance en compilation. Pas d'effet à l'exécution.
- Toutes les classes livrées dans la distribution java sont dans le **paquetage** `java`. Les **Classes de base** du langage se trouvent dans `java.lang`. Il y a une importation implicite de `import java.lang.*`
- Deux classes de même nom dans 2 paquetages différents importés avec `*` : le compilateur ne dit rien jusqu'à l'utilisation d'une des classes, où c'est une erreur de compilation.
- Utilisation de noms complets. Au lieu de

```
import java.util.*;
class MaDate extends Date { ... }
```

on peut utiliser `class MaDate extends java.util.Date ...`

4.3 Protections d'accès

4 catégories de visibilité :
- Sous classe dans le même paquetage.
- Non sous classe dans le même paquetage.
- Sous classe dans des paquetages différents.
- Classes ni dans le même paquetage, ni sous classes.

Table des modificateurs de visibilité

	`private`	rien	`private protected`	`protected`	`public`
Même classe	oui	oui	oui	oui	oui
Même paquetage, sous classe	non	oui	oui	oui	oui
Même paquetage, non sous classe	non	oui	non	oui	oui
Paquetage différent, sous classe	non	non	oui	oui	oui
Paquetage différent, non sous classe	non	non	non	non	oui

- Déclaré `public` : peut être **vu de partout**.
- Déclaré `private` : **ne peut** être vu en **dehors d'une classe**.
- Pas de modificateur : visible des **sous classes et** des **autres classes du même paquetage**. *Situation par défaut.*
- Déclaré `protected` : peut être vu hors du paquetage, mais **seulement des sous classes**.
- Déclaré `private protected` : **ne peut** être vu **que des sous classes**.
- ☛ `protected` pas la même signification qu'en C++. Plutôt similaire au `friend` du C++. Le `protected` du C++ est émulé par `private protected` en java.

Exemple animalier

```
class Chat {
    // Les differents champs sont protected (et non private),
    //   de facon a etre visibles des sous-classes
    protected String     nom;              // nom du fauve
    protected int        age;              // en annees
    protected Color[]    couleurPelage;    // ses couleurs
```

```
    protected float      tauxRonronnement; // entre 0 et 1
    protected static int ageSevrage = 1;   // Champ statique


    // Les constructeurs doivent etre vus de partout
    public Chat(String sonNom, int sonAge, float sonTauxRonron,
                    Color[] sesCouleurs) {
        nom              = sonNom;
        age              = sonAge;
        tauxRonronnement = sonTauxRonron;
        couleurPelage    = sesCouleurs;
    }

    public Chat() {
        this(new String("minou"), 1, 0.5,
                        {Color.black, Color.white});
    }



    // Accesseurs
    public int retournerAge() {
        return(age);    }
    public String retournerNom() {
        return(nom);    }
    public Color[] retournerCouleurPelage() {
        return(couleurPelage);     }
    public float retournerTauxRonron() {
        return(tauxRonronnement);  }

    // Autres methodes
    public void vieillir() {
        age += 1;
    }
    public void vieillir(int n) {
        age += n;
    }
    public boolean estAdoptable() {
        if (age > ageSevrage) {
            return true;
        } else {
            return false;
        }
    }
```

```
    // Methode privee
    private void emettreSon(String adire) {
        // Emulation ultra pauvre du son
        System.out.println(" " + adire);
    }

    // Utilisation de la methode privee
    public void miauler(int nbMiaulements) {
        for(int i + 0; i < nbMiaulements; i++) {
            emettreSon("Miaou !");
        }
    }
}
```

4.4 Interfaces

- *Interfaces* : comme des classes, mais **sans variable d'instance** et des **méthodes** déclarées **sans corps**.
- Une classe peut implanter une nombre quelconque d'interfaces. Pour cela, la classe **doit fournir l'implantation de toutes les méthodes de l'interface**. La signature de type doit être respectée.
- Les interfaces vivent dans une hiérarchie différente de celles des classes ⇒ **deux classes sans aucun lien** hiérarchique **peuvent implanter la même interface**. Les interfaces sont aussi utiles que l'héritage multiple, mais donnent du **code plus facile à maintenir**. En effet, **ne repose pas sur des données, juste sur des méthodes**.
- Syntaxe générique

```
interface nom {
    type-retour nom-methode1(liste-parametres);
    type nomvariable-finale = valeur;
}
```

- Toutes les méthodes implantant une interface doivent être déclarées `public`.
- Variables déclarées à l'intérieur d'une interface implicitement `final`.

4.5 Exemple d'interface

- Syntaxe générique d'implantation d'interface

```
class nomclasse [extends superclasse]
                [implements interface0
                          [,interface1...]] {
```

```
    corps-de-classe
}
```

- Les crochets désignent des mots optionnels
- Exemple

```
interface Callback {
    void callback(int parametre) {
}

class Client implements Callback {
    void callback(int p) {
        System.out.println("Callback de " + p);
    }
}
```

4.6 Interface & résolution dynamique de méthode

- On peut déclarer des variables références à des objets utilisant une **interface comme type** au lieu d'une classe. Toute instance d'une classe implantant cette interface peut être stockée dans cette variable. Si l'on veut appeler une méthode *via* une telle variable, l'implantation correcte sera appellée selon l'instance courante. Les **classes** peuvent donc être **crées après le code qui les appelle**. Cette technique de *résolution dynamique de méthode* est coûteuse en temps.
- Aspect d'encapsulation

```
class TestInterface {
    public static void main(String args[]) {
        Callback c = new Client();
        c.callback(12);
    }
}
```

`c` ne peut être utilisé **que pour accéder à la méthode** `callback()` et non à un autre aspect de `Client`.

VIII – Exceptions

Références bibliographiques

– *The Java Language Specification*, J. Gosling, B. Joy et G. Steele [GJS96]

VIII.1 Fonctionnement général du système d'exceptions

1.1 Génération et gestion d'exceptions

- *Exception* : condition anormale survenant lors de l'exécution.
- Lorsqu'une exception survient :
  – un objet représentant cette exception est créé ;
  – cet objet est **jeté** (**thrown**) dans la méthode ayant provoqué l'erreur.
- Cette méthode peut choisir :
  – de gérer l'exception elle-même,
  – de la passer sans la gérer.
  De toutes façons l'exception est **captée** (**caught**) et traitée, en dernier recours par l'environnement d'exécution Java.
- Les exceptions peuvent être générées
  – par l'environnement d'exécution Java,
  – manuellement par du code.
- Les exceptions jetées (ou levées) par l'environnement d'exécution résultent de violations des règles du langage ou des contraintes de cet environnement d'exécution.

1.2 Les 5 mots clés

- Il y a 5 mots clés d'instructions dédiées à la gestion des exceptions : `try`, `catch`, `throw`, `throws` et `finally`.

- Des instructions où l'on veut surveiller la levée d'une exception sont mises dans un bloc précédé de l'instruction `try`.
- Le code peut capter cette exception en utilisant `catch` et la gérer.
- Les exceptions générées par le système sont automatiquement jetées par l'environnement d'exécution Java. Pour jeter une exception manuellement, utiliesr `throw`.
- Toute exception qui est jetée hors d'une méthode doit être spécifiée comme telle avec `throws`.
- Tout code qui doit absolument être exécuté avant qu'une méthode ne retourne est placé dans un bloc `finally`.

1.3 Schéma

Le schéma est donc

```
try {
    // bloc de code a surveiller
}
catch (EceptionType1 exceptObj) {
    // gestionnaire d'exception pour ExceptionType1
}
catch (EceptionType2 exceptObj) {
    // gestionnaire d'exception pour ExceptionType2
}
...
finally {
    // bloc de code a executer
    //   avant de sortir de la methode
}
```

1.4 Types d'exceptions

- Une classe est au sommet de la hiérarchie des exceptions : `Throwable`
- Deux sous-classes de `Throwable` :
  - `Exception` : conditions exceptionnelles que les programmes utilisateur devraient traiter.
  - `Error` : exceptions catastrophiques que normalement seul l'environnement d'exécution devrait gérer.
- Une sous-classe d'`Exception`, `RuntimeException`, pour les exceptions de l'environnement d'exécution.

1.5 Exceptions non gérées

- Considérons le code suivant où une division par zéro n'est pas gérée par la programme :

```
class ExcepDiv0 {
    public static void main(String args[]) {
        int d = 0;
        int a = 42 / d;
    }
}
```

- Lorsque l'environnement d'exécution essaie d'exécuter la division, il construit un nouvel objet exception afin d'arrêter le code et de gérer cette condition d'erreur.
- Le flux de code est alors interrompu et la pile d'appels (des différentes méthodes invoquées) est inspectée en quête d'un gestionnaire d'exceptions.
- N'ayant pas fourni de gestionnaire au sein du programme, le gestionnaire par défaut de l'environnement d'exécution se met en route.
- Il affiche la valeur en `String` de l'exception et la trace de la pile d'appels :

```
/home/mounier> java ExcepDiv0
java.lang.ArithmeticException: / by zero
        at ExcepDiv0.main(ExcepDiv0.java:4)
```

1.6 Instructions `try` et `catch`

- Un bloc `try` est destiné à être protégé, gardé contre toute exception susceptible de survenir.
- Juste derièrre un bloc `try`, il faut mettre un bloc `catch` qui sert de gestionnaire d'exception. Le paramètre de l'instruction `catch` indique le type et le nom de l'instance de l'exception gérée.

```
class ExcepDiv0 {
    public static void main(String args[]) {
        try {
            int d = 0;
            int a = 42 / d;
        } catch (ArithmeticException e) {
            System.out.println("Div par zero");
        }
    }
}
```

- La portée d'un bloc `catch` est restreinte aux instructions du bloc `try` immédiatement précédent.

1.7 Instructions `catch` multiples

- On peut gérer plusieurs exceptions à la suite l'une de l'autre.
- Lorsqu'une exception survient, l'environnement d'exécution inspecte les instructions `catch` les unes après les autres, dans l'ordre où elles ont été écrites.
- Il faut donc mettre les exceptions les plus spécifiques d'abord.

1.8 Instruction `throw`

- Elle permet de générer une exception, *via* un appel de la forme `trow TrhowableInstance ;` Cette instance peut être crée par un `new` ou être une instance d'une exception déja existante.
- Le flux d'exécution est alors stoppé et le bloc `try` immédiatement englobant est inspecté, afin de voir s'il possède une instruction `catch` correspondante à l'instance générée.
- Si ce n'est pas le cas, le $2^{\text{ième}}$ bloc `try` englobant est inspecté ; et ainsi de suite.
- Exemple

```
class ThrowDemo {
  static void demoproc() {
    try {
      throw new NullPointerException("demo");
    } catch (NullPointerException e2) {
      System.out.print("attrapee ds demoproc()");
      throw e2;
    }

  public static void main(String args[]) {
    try {
      demoproc();
    } catch(NullPointerException e1) {
      System.out.print("attrapee ds main()");
    }
  }
}
```

1.9 Instruction `throws`

- Si une méthode est susceptible de **générer une exception qu'elle ne gère pas**, **elle doit le spécifier**, de façon que ceux qui l'appellent puissent se prémunir contre l'exception.

- L'instruction `throws` est utilisée pour spécifier la liste des exceptions qu'une méthode est susceptible de générer.
- Pour la plupart des sous-classes d'`Exception`, le compilateur **forcera à déclarer** quels types d'exception peuvent être générées (sinon, le programme ne compile pas).
- Cette règle ne s'applique pas à `Error`, `RuntimeException` ou à leurs sous-classes.
- L'exemple suivant ne compilera pas :

```
class ThrowsDemo1 {
    static void proc() {
        System.out.println("dans proc()");
        throw new IllegalAccessException("demo");
    }

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

  Ce programme ne compilera pas :
  – parce que `proc()` doit déclarer qu'elle peut générer `IllegalAccessException` ;
  – parce que `main()` doit avoir un bloc `try`/`catch` pour gérer l'exception en question.

- L'exemple correct est :

```
class ThrowsDemo1 {
    static void proc()
            throws IllegalAccessException {
        System.out.println("dans proc()");
        throw new IllegalAccessException("demo");
    }

    public static void main(String args[]) {
        try {
            demoproc();
        } catch(IllegalAccessException e) {
            System.out.println(e + "attrapee");
        }
    }
}
```

1.10 Instruction `finally`

- Un bloc `finally` est toujours exécuté, qu'une exception ait été générée ou non. Il est exécuté avant l'instruction suivant le bloc `try` précédent.
- Si le bloc `try` précédent contient un `return`, le bloc `finally` est exécuté avant que la méthode ne retourne.
- Ceci peut être pratique pour fermer des fichiers ouverts et pour libérer diverses ressources.
- Le bloc `finally` est optionnel.

1.11 Classe `Throwable`

- Il est possible de générer ses propres exceptions en créant une sous classe d'`Exception`.
- On peut alors utiliser ou redéfinir l'une des méthodes, héritée de `Throwable` :
  - Outre le constructeur sans argument, un constructeur `Exception(String message)` avec un message d'erreur disponible *via* `getMessage()`.
  - la méthode `String getMessage()` qui renvoie le message fourni au constructeur précédent.
  - la méthode `String toString()`, qui fournit une chaîne formée du nom de la classe de l'objet courant, suivi d'un `:`, suivi du résultat de `getMessage()`
  - la méthode `fillInStackTrace()` qui enregistre dans l'objet courant des informations à propos de la pile d'appels en cours.
  - la méthode `void printStackTrace(PrintStream stream)` qui envoie sur le flux `stream` le résultat de `toString()`, suivi de la pile d'appels enregistrée par la méthode `fillInStackTrace()`. Si `stream` est absent, `System.err` est utilisé. Une autre forme, `void printStackTrace(PrintWriter stream)` est disponible.

1.12 Conclusion

- Le code suivant

```
FileInputStream fis;
try {
    fis = new FileInputStream("readme.txt");
} catch (FileNotFoundException e) {
    fis = new FileInputStream("default.txt");
}
```

- est plus propre que

```
#include <sys/errno.h>
```

```
int fd;
fd = open("readme.txt");
if (fd == -1 && errno == EEXIST)
    fd = open("default.txt");
```

IX – Classes utilitaires de base

Références bibliographiques

- *Java et Internet – Concepts et programmation*, G. Roussel, E. Duris, N. Bedon et R. Forax [RDBF02],
- *Java in a Nutshell*, D. Flanagan, [Flab],
- *The Java Language Specification*, J. Gosling, B. Joy, G. Steele [GJS96]

IX.1 Classes Object, System, PrintStream

1.1 Méthodes de la classe `Object`

- Racine de la hiérarchie des objets java.
- Méthodes :

`methode()`	But
`String toString()`	Renvoie une vue en chaîne de caractères de `this` ; par défaut, renvoie le nom de la classe suivi de son code de hachage.
`int hashCode()`	Renvoie le code de hachage associé à l'objet.
`boolean equals()`	Teste l'égalité, **la plus sématiquement significative possible**.
`protected Object clone()`	Renvoie une copie superficielle (champ à champ) de l'objet (`throws CloneNotSupportedException`).
`protected void finalize()`	Appelée en libération mémoire (`throws Throwable`).
`final void notify()`	Relâche le moniteur de l'objet et réveille une thread bloquée en attente de ce moniteur par un `wait()`.

`final void notifyAll()`	Relâche le moniteur de l'objet et réveille toutes les threads bloquées en attente de ce moniteur par un `wait()`.
`final void wait()`	Acquiert le moniteur de l'objet ou bloque la thread indéfiniment si le moniteur est déjà pris (`throws InterruptedException`).
`final void wait(long timeout)`	Acquiert le moniteur de l'objet ou bloque la thread pendant `timeout` millisecondes si le moniteur est déjà pris (`throws InterruptedException`).
`final Class getClass()`	Renvoie une représentation de la classe de l'objet.

1.2 Méthodes `toString()`, `hashCode()`

- `toString()` : *Forme affichable* de l'objet par `System.out.println()`. **La redéfinir** est de bon ton.
- `hashCode()` : *code de hachage* de l'objet ; utilisé dans `java.util.hashMap`.
- Contrat de la méthode `hashCode()` : Pour 2 `Object`, `c1` et `c2`, `c1.equals(c2)` impliquec`1.hashCode() == c2.hashCode()`
- Donc, si l'on redéfinit `equals()`, on doit redéfinir également `hashCode()`.

1.3 Méthode `equals()`

- Par défaut, teste l'égalité des références. Il est de bon ton de la **redéfinir** en test d'**égalité de contenu**.
- Erreur commune : surcharge au lieu de redéfinition ; le paramètre doit être **de type** `Object`.
- Exemple sur des classes de nombres complexes :

```
public class Complexe {
  protected double partieReelle, partieImaginaire;

  public Complexe(double r, double i) {
      partieReelle = r;
      partieImaginaire = i;
  }

  public boolean equals(Object obj) {
      if(!(obj instanceof Complexe)) {
          return false;
```

```
        }
        Complexe c = (Complexe)obj;
        return (partieReelle == c.partieReelle &&
            partieImaginaire == c.partieImaginaire);
    }
}
```

- Vérifier que la relation binaire induite est réflexive, symétrique et transitive. Vérifier également l'idempotence (plusieurs évaluations de `x.equals(y)` donne toujours le même résultat), et que `null` est "absorbant" : `x.equals(null)` est toujours `false`.

1.4 Champs et méthodes de la classe `System`

- Méthodes et champs utilitaires java.
- Champs :
  - `static InputStream in` entrée standard (par défaut le clavier)
  - `static PrintStream out` sortie standard (par défaut l'écran)
  - `static PrintStream err` sortie erreur standard (par défaut l'écran)

Méthodes :

`methode()`	But
`static long currentTimeMillis()`	renvoie le nombre de millisecondes depuis le 1er janvier 1970.
`static void exit(int status)`	arrête la machine virtuelle java en cours d'exécution.
`static void gc()`	demande au ramasse-miettes de récupérer la mémoire inutilisée.
`static void setIn(InputStream in)`	réassigne l'entrée standard.
`static void setOut(PrintStream out)`	réassigne la sortie standard.
`static void setErr(PrintStream err)`	réassigne la sortie erreur standard.

1.5 Méthodes de `PrintStream`

- Méthodes :

`methode()`	But
`void close()`	Ferme le flux d'entrée/sortie
`void flush()`	Vide le tampon mémoire associé au flux (force l'écriture)

`void print(...)`	Affiche l'argument sur la sortie standard. Accepte des `boolean`, `char`, `int`, `long`, `float`, `double`, `Object` et `String`.
`void println(...)`	Même effet que `print()`, mais rajoute un saut de ligne
`void write(int b)`	Écriture binaire d'un octet sur le flux d'entrée/sortie

IX.2 Méthode `main()` et classes d'emballage des types primitifs

2.1 Méthode `main()` et ses arguments

- Syntaxe `public static void main(String args[]) ...`
- `public` : la méthode peut être appelée de partout
- `static` : pas besoin de créer d'objet pour l'appeler
- `void` : elle ne renvoie rien
- `String args[]` : `args` est un tableau de `String`
- 1$^{\text{ier}}$ argument `args[0]`, 2$^{\text{ième}}$ argument `args[1]`, ...
- Nombre d'arguments : `args.length`
- Attention ! Ne pas confondre
  - le champ `length` : nombre d'éléments d'un tableau
  - la méthode `length()` de la classe `String` : longueur de la chaîne de caractères
- Exemple d'affichage des arguments de la ligne de commande ainsi que de leur longueur :

```
class TestMain {
  public static void main(String args[]) {
      for(int i = 0; i < args.length; i++)
          System.out.println("arg no " + i+1 +
                             " : " + arg[i] +
                             " de longueur : " +
                             args[i].length());
  }
}
```

- Par un appel dans une fenêtre Dos (resp. une fenêtre terminal Unix/Linux) de la forme `java TestMain toto 4 gabuzomeu 7.8 +&)` affiche
  `arg no 1 : toto de longueur : 4`

```
arg no 2 : 4 de longueur : 1
arg no 3 : gabuzomeu de longueur : 9
arg no 4 : 7.8 de longueur : 3
arg no 4 : +&) de longueur : 3
```

2.2 Liste des classes d'emballage

- Permettent de disposer de méthodes utilitaires de manipulation des types primitifs.
- Héritent de la classe abstraite `Number`.
- Les classes d'emballage des types primitifs sont : `Boolean`, `Byte`, `Character`, `Short`, `Integer`, `Long`, `Float` et `Double`.
- Méthode `xxxValue()`, où `xxx` est l'un des noms de type primitf correspondant ; elle permet d'obtenir une variable du type primitif correspondant.

```
Integer un = new Integer(1);
int i      = un.intValue();
```

- Méthode `parseXXX(String)` où `XXX` est l'un des noms de classe précédent ;

elle permet d'obtenir un objet de type numérique ou booléen à partir d'une chaîne de caractères. Par ex. `parseDouble("2.5")` ; renvoie un `Double`. L'inverse est réalisé par `toString()`.

- Les constantes `MIN_VALUE` et `MAX_VALUE` contiennent les valeurs minimale et maximale.

IX.3 Scanner (java.util.Scanner)

3.1 Classe Scanner : aperçu

- La classe `Scanner` permet entre autres l'entrée facile de types primitifs et de `String` au clavier.
- Il suffit de créer un objet `Scanner` avec en argument le flux à lire, puis d'appeler une méthode `nextXXX()` selon le type primitif `XXX` à lire
- Exemple d'entrée d'un entier au clavier :

```
Scanner sc = new Scanner(System.in); // Creation d'un Scanner sur
                                     //   le flux System.in (le clavier)
int i = sc.nextInt();                // prise d'un entier sur ce flux
                                     //   (au clavier)
```

3.2 Classe `Scanner` : constructeurs

Différents constructeurs sont disponibles

`methode()`	But
`Scanner(File source)`	Construit un objet de type `Scanner` produisant des valeurs à partir du fichier spécifié.
`Scanner(InputStream source)`	Construit un objet de type `Scanner` produisant des valeurs à partir du flux d'entrée spécifié.
`Scanner(Readable source)`	construit un objet de type `Scanner` produisant des valeurs à partir de l'entrée spécifiée. Cette entrée doit implanter l'interface `Readable`, qui spécifie une source de caractères. À titre indicatif, l'ensemble des classes implantant cette interface est : `BufferedReader, CharArrayReader, CharBuffer, FileReader, FilterReader, InputStreamReader, LineNumberReader, PipedReader, PushbackReader, Reader, StringReader`.
`Scanner(String source)`	Construit un objet de type `Scanner` produisant des valeurs à partir de la chaîne spécifiée.

3.3 Classe `Scanner` : méthodes essentielles

- Rappel : un flux d'entrée est composé de lexèmes, ou atomes syntaxiques, qui sont séparés par des délimiteurs.
- Les méthodes `boolean hasNextXXX()` renvoient `true` si le prochain lexème correspond au type attendu. La chaîne `XXX` précédente est l'une des suivantes : `BigDecimal, BigInteger, Boolean, Byte, Double, Float, Int, Long, Short, Line` selon le type attendu, qui sera respectivement `BigDecimal, BigInteger, boolean, byte, double, float, int, long, short` pour les 9 premières, et une nouvelle ligne pour la dernière. Ainsi, `hasNextInt()` renvoie `true` si le prochain lexème est un `int`.

- Les méthodes `YYY nextXXX()` renvoient la valeur du prochain lexème selon le type correspondant à la chaîne XXX Ainsi, `int nextInt()` renvoie le prochain `int`, `String nextLine()` renvoie la prochaine ligne, `int nextDouble()` renvoie le prochain `double`, etc.
- La méthode `boolean hasNext()` renvoie `true` s'il y a un prochain lexème.
- La méthode `String next()` renvoie le prochain lexème disponible.

3.4 Classe `Scanner` : autres méthodes

Voici les autres méthodes de `Scanner` les plus importantes

`methode()`	But
`void close()`	Ferme ce `Scanner` (le flux associé).
`String findInLine(String pattern)`	Cherche la prochaine occurence du motif `pattern`, en ignorant les délimiteurs.
`boolean hasNext(String pattern)`	Renvoie `true` si le prochain lexème correspond au motif spécifié par `pattern`.
`String next(String pattern)`	Renvoie le prochain lexème s'il correspond au motif spécifié par `pattern`.
`Scanner skip(String pattern)`	saute les entrées qui correspondent au motif spécifié par `pattern`.
`Scanner useDelimiter(String pattern)`	Fixe le délimiteur au motif spécifié par `pattern`.

3.5 Classe `Scanner` : Exemples

- Exemple de lecture dans un fichier :

```
Scanner sc = new Scanner(new File("myNumbers"));
while (sc.hasNextLong()) {
    long aLong = sc.nextLong();
}
```

- Exemple de lecture à partir d'une chaîne de caractères, avec un délimiteur autre qu'un espace

```
String input = "1 fish 2 fish red fish blue fish";
Scanner s = new Scanner(input).useDelimiter("\\s*fish\\s*");
System.out.println(s.nextInt());
System.out.println(s.nextInt());
System.out.println(s.next());
```

```
System.out.println(s.next());
s.close();
```

La sortie produite par ce code est la suivante :

```
1
2
red
blue
```

Il est possible de récupérer les quatres lexèmes d'un coup :

```
String input = "1 fish 2 fish red fish blue fish";
Scanner s = new Scanner(input);
s.findInLine("(\\d+) fish (\\d+) fish (\\w+) fish (\\w+)");
MatchResult result = s.match();
for (int i=1; i<=result.groupCount(); i++)
    System.out.println(result.group(i);
s.close();
```

3.6 Exemple d'un cercle

Exemple d'une classe `Cercle` avec utilisation d'un `Scanner`

```
import java.util.Scanner;
/**
* Classe representant un cercle
**/
class Cercle {
  // champs : rayon du cercle
  double r;
  // Constructeur : initialisation des champs
  Cercle(double nouvRayon) {
      r = nouvRayon;
  }
  // methode de calcul dŠune surface
  double calculeSurface() {
      return(3.1416*r*r);
  }
}// fin de class Cercle

/**
* Ce programme affiche la surface dŠun cercle dont
* lŠutilisateur entre le rayon
**/
public class CercleMain {
```

```
  // methode main() : point dŠentree du programme
  public static void main(String[] args) {
      // pour les entrees de donnees au clavier
      Scanner entreeClavier = new Scanner(System.in);
      // capture dŠun double au clavier
      double rayon = entreeClavier.nextDouble();
      // creation dŠun objet de type Cercle
      Cercle monCercle = new Cercle(rayon);
      // calcul de sa surface
      surface = monCercle.calculeSurface();
      // affichage du resultat
      System.out.println("Voici la surface du cercle" +
                         "de rayon " + monCercle.r +
                         " : " + surface);
  }
}// fin de class CercleMain
```

IX.4 Classes `java.applet.Applet` et `java.lang.String`

4.1 Notion d'applet

- Applet : mini-application, dont le code est tĺéchargé à travers le réseau.
- Est visualisée par un navigateur ou par un visualiseur d'applets ("applet viewer").
- Diverses restrictions de sécurité.
- Une applet n'a pas de méthode `main()`.
- On étend la classe `java.Applet`, en redéfinissant diverses méthodes.
- Une applet n'est pas sous le contrôle de l'activité (thread) d'exécution : elle répond lorsque le navigateur le lui demande.
- Donc, pour des tâches longues, l'applet doit créer sa propre activité.

4.2 Méthodes à redéfinir

Méthodes de base d'`Applet` :

- `void init()` Appelée lors du premier chargement de l'applet. Utilisée pour des initialisations, de préférence à un constructeur.
- `void destroy()` Appelée lors du déchargement de l'applet. Utilisée pour libérer des ressources.
- `void start()` Applelée lorsque l'applet devient visible. Souvent utilisée avec des animations et des activités (threads).

- `void stop()` Applelée lorsque l'applet est masquée.

Une méthode héritée de `Container` : `public void paint(Graphics g)` que le navigateur appelle pour demander à l'applet sa mise a jour graphique.

Autres méthodes d'`Applet` :

- `String getAppletInfo()` Pour obtenir des informations à propos de l'applet
- `String[][] getParameterInfo()` Description des paramètres de l'applet.
- `AudioClip getAudioClip(URL url)` Renvoie une référence à une instance d'objet de type `AudioClip`.
- `void play(URL url)` joue l'`AudioClip` spécifié à l'adresse `url`.
- `Image getImage(URL url)` Renvoie une référence à une instance d'objet de type `Image`.

4.3 Exemple : un disque coloré

Classe `Disk` : surface circulaire colorée

```
import java.awt.*;

public class Disk {
  protected int       x, y;           // position du disque
  protected int       size;           // diametre du disque
  protected Color     color;          // couleur du disque

  public Disk(int Xpos, int Ypos, int radius) {
        x     = Xpos;     y  = Ypos;
        size  = radius;
        color = Color.red;                // Initialement rouge
  }

  // methodes fixant des attributs (modificateurs)
  public void setXY(int newX, int newY) { x = newX; y = newY;}
  public void setSize(int newSize)        { size = newSize; }
  public void setColor(Color newColor)    { color = newColor;}

  // methodes accedant aux attributs (accesseurs)
  public int getX()                       { return x; }
  public int getY()                       { return y; }
  public int getSize()                    { return size; }
  public Color getColor()                 { return color; }

  // Afficher le disque
  public void paint(Graphics g) {
    g.setColor(color);
```

```
    g.fillOval(x-(size/2), y-(size/2), size, size);
  }

}// public class Disk
```

Classe `DiskField`, qui affiche le disque précédent :

```
import java.applet.*;
import java.awt.*;

public class DiskField extends Applet {

  int    x = 150, y = 50, size = 100;   // position et diametre
  Disk   theDisk = null;

  // Initialisation de l'applet
  public void init() {
    theDisk = new Disk(x, y, size);  }

  //  Dessiner le disque
  public void paint(Graphics g) {
    // Demander au navigateur d'appeler la methode paint()
    //    pour afficher le disque
    theDisk.paint(g);
  }

  public void start() { ; }
  public void stop() { ; }

}// class DiskField
```

Pour afficher l'applet, on a besoin d'un fichier HTML qui la référence.

```
<APPLET code="DiskField.class" width=150 height=100>
</APPLET>
```

4.4 Construction de `String`

- Dans `java.lang` : `String` pour les chaînes à immuables et `StringBuffer` pour celles qui sont modifiables.
- `String` et `StringBuffer` sont déclarées `final`, de façon à réaliser certaines optimisations.
- Le constructeur générique de `String` se déclare comme suit :
  `String String(char tabChars[], int indiceDeb, int nbChars);` où `indiceDeb` débute à `0` pour le premier caractère de la chaîne.

- Exemples :

```
char desChars[] = {'a', 'b', 'c', 'd', 'e', 'f' };
String s1 = new String(desChars);
String s2 = new String(desChars, 2, 3);
```

  `s1` contient la chaîne `"abcde"` et `s2` contient `"cde"`. ☛ Le contenu du tableau est copié lorsque l'on crée une chaîne à partir d'un tableau. Si l'on modifie le tableau après avoir créé la chaîne, le contenu de l'instance de `String` restera inchangé.
- Constructeur de recopie `public String(String original)`.
- Il y a une syntaxe spéciale pour les chaînes qui permet une création-initialisation rapide :

```
String s = "abc";
System.out.println(s.length());
System.out.println("abcdef".length());
```

  Les 2 dernières lignes vont afficher respectivement 3 et 6.
- ☛ Ne pas confondre la méthode `length()` avec la variable d'instance `length` de références à des tableaux.

4.5 Concaténation de chaînes

- Java n'implante pas la surcharge d'opérateurs, cette technique donnant souvent lieu à des abus et rendant les gros programmes difficiles à lire.
- Il y a une exception à cette règle : l'opérateur `+`, qui existe également pour les chaînes. Le `+` agit alors comme un opérateur de concaténation, de façon à augmenter la lisibilité.
- Par exemple :

```
String s = "Impossible " + "d'eternuer " +
                           "les yeux ouverts.";
```

  est nettement plus lisible que

```
String s = new StringBuffer("Impossible ")
               .append("d'eternuer ")
               .append("les yeux ouverts.")
               .toString();
```

  qui est exactement ce qui se passe lorsque le code est exécuté.
- On ne peut insérer de caractères dans une `String`, ni changer ceux qui y sont déja. Il n'est pas possible non plus de faire de la concaténation. Les modifications sont alors réalisées sur une insantce de la classe soeur `StringBuffer`. Puis, il y a conversion en une `String` *via* la méthode `toString()`.

- La scission en 2 classes, dont une, `String`, est immuable, a été opérée afin d'avoir des performances plus grandes lors de manipulations (fréquentes) de `String`.
- ☛ La priorité des opérateurs peut jouer des tours. Par exemple, la ligne suivante : `String s = "quatre : " + 2 + 2;` ne mettra pas `"quatre : 4"` dans `s`, mais `"quatre : 22"` ; en effet, `"quatre : " + 2` est évalué d'abord, et le résultat de ceci (une chaîne) est ensuite concaténé à `2`. Il faut donc des parenthèses : `String s = "quatre : " + (2 + 2);`

4.6 Conversion de chaînes

- `StringBuffer` utilise une version surchargée de `append()` pour tous les types possibles.
- `append()` appelle en fait la méthode `valueOf()`. Pour des types primitifs, cette dernière renvoie une repésentation en chaîne.
- Pour des objets, elle appelle la méthode `toString()` de l'objet.
- `toString()` est une méthode de `Object`, donc tout objet en hérite.

C'est une BONNE PRATIQUE que de redéfinir `toString()` pour ses propres classes. Exemple

```
class Point {
    int x, y;
    Point(int x, int y) {
        this.x = x;
        this.y = y;
    }
    public String toString() {
        return "Point[" + x + "," + y + "]";
    }
    class toStringDemo {
        public static void main(String args[]) {
            Point p = new Point(20, 20);
            System.out.println("p = " + p);
        }
    }
```

4.7 Extraction

- `charAt()` permet d'extraire un caractère. Par ex. `"abc".charAt(1)` renvoie `'b'`.
- Pour extraire plus d'un caractère, utiliser `getChars()`. Prototype :
  `void getChars(int srcBegin, int srcEnd, char [] dst, int dstBegin);`

où `srcBegin` et `srcEnd` sont les indices de début et de fin d'extraction dans la `String` appelante. `dst` est le tableau de caractères contenant les caractères extraits. `dstBegin` est l'indice à partir duquel les caractères extraits vont être copiés dans `dst`.
- On peut ġalement convertir une `String` entière en tableau de caractères, à l'aide de la méthode `toCharArray()` : `char [] toCharArray()`
- Enfin, on peut convertir une `String` en tableau de `byte`, l'octet de poids fort étant jeté. ceci est utile pour exporter une chaîne en environnement ASCII (fichier texte de protocoles internet par ex.) : `byte [] getBytes()`

4.8 Comparaison

- `equals(String s)` teste si la `String` appelante est formée des mêmes caractères que `s`.
- `equalsIngnoreCase()` réalise la même chose sans faire de distinction minuscule-majuscule.
- `regionMatches()` compare une région de la `String` appelante à une région d'une autre. Son prototype est :
  ```
  boolean regionMatches(int toffset, String other, int oofset, int eln);
  ```
  où `toffset` est l'indice du début de comparaison de la chaîne appelante, `other` est l'autre `String`, `oofset` est l'indice de début de comparaison de l'autre chaîne et `len` est la longueur de compraraison. Il existe une autre forme de `regionMatches()` qui peut ignorer la distinction majuscule-minuscule :
  ```
  boolean regionMatches(int toffset, String other, int oofset, int eln);
  ```

4.9 Extraction/Comparaison

- `startsWith()` (resp. `endsWith()` teste si la chaîne appelante commence (resp. finit) par la chaîne fournie en paramètre. `"Nabuchodonosor".endsWith("nosor")` et `"Nabuchodonosor".startsWith("Nabu")` renvoient tous deux `true`. On peut également spécifier l'indice de début de comparaison. Par exemple l'expression `"HoueiNeng".startsWith("Neng", 5)` renvoie `true`.

4.10 Égalité

- La méthode `equals()` et l'opérateur `==` réalisent 2 opérations distinctes.
- `equals()` teste l'égalité caractère à caractère.
- `==` teste l'égalité des références (des adresses mémoires, ou pointeurs) pour voir si elles se réfèrent à la même instance.
- Exemple :

```
class EgalOuPasEgal {
    public static void main(String args[]) {
        String s1 = "Bonjour";
        String s2 = new String(s1);
        System.out.println("s1 + "equals()" + s2 " -> " + s1.equals(s2));
        System.out.println("s1 + "==" + s2 " -> " + (s1 == s2));
    }
}
```

4.11 Relation d'ordre

- `compareTo()` compare 2 `String` selon un ordre alphabétique.
- `int compareTo(String s)` renvoie un résultat négatif si la chaîne appelante est inférieure à `s` (le paramètre), `0` si elles sont égales et un résultat positif sinon.
- Moyen **mnémotechnique** : `caller.compareTo(parameter)` renvoie un entier du même signe que `caller - parameter`.

4.12 Recherche de sous-chaîne

- Recherche de l'indice d'occurrence d'un caractère ou d'une sous-chaîne dans une chaîne.
- 2 méthodes : `indexOf()` et `lastIndexOf()` sous plusieurs formes. Renvoient `-1` en cas d'échec.
- ```
  int indexOf(int car);
  int lastindexOf(int car);
  ```
  renvoient l'indice de la première (resp. la deernière) occurence (c.à.d. apparition) du caractère `car`.
- ```
  int indexOf(String str);
  int lastindexOf(String str);
  ```
  renvoient l'indice du premier caractère de la première (resp. la deernière) occurrence de la sous-chaîne `str`.
- ```
  int indexOf(int car, int fromIndex);
  int lastindexOf(int car, int fromIndex);
  ```
  renvoient l'indice de la première (resp. la dernière) occurrence du caractère `car` après (resp. avant) `formIndex`.
- ```
  int indexOf(String str, int fromIndex);
  int lastindexOf(String str, int fromIndex);
  ```
  renvoient l'indice du premier caractère de la première (resp. la dernière) occurrence de la sous-chaîne `str` après (resp. avant) `formIndex`.

4.13 Modifications sur une copie de `String`

- Puisque les `String` sont immuables, pour modifier une chaîne, on peut soit utiliser un `StringBuffer` ou utiliser l'une des méthodes suivantes, qui fournissent une copie modifiée d'une `String`.
- `substring()` extrait une `String` d'une autre. Par exemple :

  ```
  "Bonjour a tous".substring(8) -> "a tous"
  "Bonjour a tous".substring(6, 5) -> "r a t"
  ```

- `concat()` créée un nouvel objet, la concaténatée de la chaîne appelante et du paramètre :

  ```
  "Bonjour".concat(" a tous") -> "Bonjour a tous"
  ```

- `replace(char carSrc, char carDst)` remplace toutes les occurrences de `carSrc` par `carDst` :

  ```
  "Bonjour".replace('o', 'a') -> "Bajaur"
  ```

- `toLowerCase()` et `toUpperCase()` : conversion en majuscules (resp. minuscules)

  ```
  "Grenouille".toUpperCase() -> "GRENOUILLE"
  "BOEuf".toLowerCase()      -> "bouef"
  ```

- `trim()` enlève les espaces avant et après :

  ```
  "    J'ai besoin d'air        ".trim() ->
  "J'ai besoin d'air"
  ```

4.14 Autres méthodes de `String`

`methode()`	But
`String concat(String str)`	Concaténation de `this` à celle fournie en argument.
`boolean contains(String s)`	renvoie `true` si `this` contient la `String` argument.
`boolean contentEquals(StringBuffer sb)`	renvoie `true` si `this` est égale (au sens du contenu) à la `StringBuffer` argument.
`static String copyValueOf(char[] data)`	Conversion d'un tableau de caractères en `String`.
`static String format(String format, Object... args)`	Renvoie une `String` formattée (voir la documentation des API pour les chaînes format).
`boolean matches(String regex)`	renvoie `true` si `this` correspond à l'expression régulière `regex`.

`String[] split(String regex)`	Découpe `this` selon les délimiteurs fournis en tant qu'expression régulière.

4.15 StringBuffer

- C'est une chaîne modifiable et susceptible de croître et de décroître.
- Elle peuvent être construites avec un constructeur :
  - sans paramètre ; ce qui réserve de la place pour 16 caractères ;
  - avec un paramètre `int`, spécifiant la taille initiale du tampon ;
  - avec un paramètre `String`, spécifiant le contenu initial et réservant de la place pour 16 caractères supplémentaires.
- La longueur s'obtient par `length()` et la taille (en nombre de caractères) de la zone mémoire allouée (nommée la capacité du tampon) par `capacity()`.
- On peut pré-allouer de la place mémoire pour le tampon après qu'un `StringBuffer` ait été créé *via* `ensureCapacity()`.
- `setLength()` fixe la taille du tampon mémoire. Si la chaîne était plus longue que la nouvelle taille, elle est tronquée. Si la nouvelle taille est plus longue, il y a remplissage par le caractère nul (unicode 0).
- `charAt()` renvoie un caractère ùn indice donné ; `setCharAt()` remplace un caractère à un indice donné ; exemple :

```
class setCharAtDemo {
    public static void main(String args[]) {
        StringBuffer sb = new StringBuffer("Bonjour");
        System.out.println("tampon avant : " + sb);
        System.out.println("charAt(1) avant : " + sb.charAt(1));
        sb.setCharAt(1, 'a');
        sb.setLength(2);
        System.out.println("tampon apres : " + sb);
        System.out.println("charAt(1) apres : " + sb.charAt(1));
    }
}
```

qui affiche

```
tampon avant : Bonjour
charAt(1) avant : o
tampon apres : Ba
charAt(1) apres : a
```

- `getChars()` fonctionne de la même manière que son homologue de `String`. Prototype identique :

```
void getChars(int srcBegin, int srcEnd,
              char [] dst, int dstBegin);
```

- `append()` concatène le paramètre à la chaîne appelante. En général appelé *via* `+`.
- `insert()` insère une sous-chaîne à ùn indice spécifié :

  ```
  "L'envie d'etre roi".insert(8, "de tout sauf ")
  ```

  résulte en

  ```
  "L'envie de tout sauf d'etre roi"
  ```

4.16 `StringBuilder`

- C'est une chaîne modifiable ayant les même fonctionnalités que `StringBuffer` mais sans synchronisation multi-threads. Il est conseillé de l'utiliser pour les applications mono-thread.

X – java.util : Conteneurs et autres utilitaires

Références bibliographiques

- *A Course in Number Theory and Cryptography*, N. Koblitz [Kob87]
- *Java in a Nutshell*, D. Flanagan, [Flab],
- *The Java Language Specification*, J. Gosling, B. Joy et G. Steele [GJS96],
- *Java et Internet – Concepts et programmation*, G. Roussel, E. Duris, N. Bedon et R. Forax [RDBF02],
- *Data Structures & Problem Solving Using Java*, M.A. Weiss [Wei98],
- *Algorithms*, R. Sedgewick [Seg91]

X.1 Classes de java.util ; Classes et interfaces de comparaison

1.1 Classes et interfaces de `java.util`

On trouve les groupes de classes suivants :
- Comparaison sur des objets (interfaces `Comparable` et `Comparator`).
- Structures de données conteneurs (listes chaînées, arbres, tables de hachage).
- Expressions régulières (paquetage `java.util.regexp`).
- Classe `Date`, gestion de la date.
- Classe `EventObject`
- Classes `Timer` et `TimerTask`
- Classe `Observable`, super classe des objets observables.
- Classe `Random`, générateur de nombres pseudo-aléatoires.
- Classe `Stack`, pile d'objets.
- Classe `StringTokenizer`, lorsqu'instanciée avec un objet `String`, casse la chaîne en unités lexicales séparées par n'importe quel caractère.

- Journalisation (paquetage `java.util.logging`).
- Stockage de paramètres (paquetage `java.util.prefs`).
- Classe `BitSet`, ensemble de bits arbitrairement grand.
- Des classes de gestion de zone géographique, de gestion des fuseaux horaires, de gestion du calendrier.
- Deux classes, dépréciées, qui sont encore présentes pour des raisons de compatibilité :
- Classe `Vector`, tableau d'objets de grandeur variable.
- Classe `Properties`, extension de `Hashtable` permettant de lire et d'écrire des paires clé/valeur dans un flot.
- On trouve diverses interfaces :
  - 10 interfaces associées aux conteneurs.
  - `EventListener`, interface marqueur pour tous les gestionnaires d'événements.
  - `Comparator`, pour les objets définissant une relation d'ordre (via `compare(Object o1, Object o2)` et `equals(Object o)`).
  - `Observer`, définit la méthode `update()` nécessaire pour qu'un objet "observe" des sous-classes de `Observable`.

1.2 Interface `java.lang.Comparable`

- Deux eĺéments sont comparables (implanter `Comparable`) si l'on peut leur appliquer `public int compareTo(Object other)`
- Cette méthode renvoie la distance entre `this` et `other`, au sens de la relation d'ordre induite.

1.3 Interface `java.util.Comparator`

- Objets comparateurs : spécialisés dans la définition de relations d'ordre.
- Deux méthodes à implanter :
  - `int compare(Object o1, Object o2)`, offrant le même service que `compareTo()` de `java.lang.Comparable`
  - `boolean equals(Object o)` testant l'égalité de contenu.
- Les méthodes de comparaison doivent en général être compatibles avec le test dégalité.

X.2 Classes et interfaces conteneurs

2.1 Cadre de collections

Un cadre logiciel de collections est formé de
- Interfaces, ou types de données abstraits.
- Implantations, classes concrètes (structures de données réutilisables).
- Algorithmes, méthodes utilitaires comme tri ou recherche, polymorphes (fonctionnalités réutilisable).

2.2 Catégories de conteneurs

- Deux grandes catégories :
  - Type (interface) `Collection`, ou groupe d'objets.
  - Type (interface) `Map`, table d'association de couples clé-valeur.
- Dans `Collection`, deux sous-catégories :
  - Type `Set`, ne pouvant contenir 2 fois le même élément.
  - Type `List`, éléments indicés par des entiers positifs.
- Dans `Map`, l'objet clé permet d'accéder à l'objet valeur.
- Dans `Map`, couple clé-valeur : entrée, de type `Map.entry`.

2.3 Types de conteneurs

Différents types de conteneurs, selon l'interface et la structure de donnée.

		Implantations			
		Table de hachage	Tableau à taille variable	Arbre équilibré	Liste chaînée
Interfaces	`Set`	`HashSet`		`TreeSet`	
	`List`		`ArrayList`		`LinkedList`
	`Map`	`HashMap`		`TreeMap`	

2.4 Transitions entre conteneurs

- Dans `Map` :
  - `values()` renvoie une `Collection` des valeurs de la table
  - `keySet()` renvoie un `Set` des clés de la table
  - `entrySet()` renvoie un `Map.entry` des entrées (paires clés/valeur) de la table
  - Ce sont des **vues** de la table.
  - Une modification d'une vue est faite sur la table et vice versa.
- Dans `Collection` :

  - `toArray()` renvoie un tableau contenant tous les objets de la collection.
  - Ce n'est pas une vue qui est renvoyée.
- Dans la classe utilitaire `Arrays` :
  - `toArray()` renvoie un tableau contenant tous les objets de la collection.
  - Ce n'est pas une vue qui est renvoyée.

2.5 Interface `Collection`

Résumé des méthodes :

- `boolean add(Object o)` ajoute l'élément spécifié à la collection. renvoie `true` si la collection a été modifiée par l'opération (un `Set` ne peut contenir 2 fois le même élément).
- `boolean addAll(Collection c)` ajoute les éléments de `c` à la collection.
- `void clear()` vide la collection.
- `boolean contains(Object o)` teste si la collection contient `o`
- `boolean containsAll(Collection c)` teste si la collection contient la collection `c`
- `boolean equals(Object o)` teste l'égalité de contenu de la collection avec `o`.
- `int hashCode()` renvoie le code de hachage de la collection.
- `boolean isEmpty()` teste si la collection est vide.
- `Iterator iterator()` renvoie un itérateur sur les éléments de la collection.
- `boolean remove(Object o)` enlève une instance de `o` de la collection.
- `boolean removeAll(Collection c)` enlève de la collection tous les éléments de `c`.
- `boolean retainAll(Collection c)` enlève de la collection tous les éléments qui ne sont pas dans `c` (ne retient que ceux qui sont dans `c`).
- `int size()` renvoie le nombre d'éléments de la collection.
- `Object[] toArray()` renvoie un tableau contenant tous les éléments de la collection.
- `Object[] toArray(Object[] a)` renvoie un tableau contenant tous les éléments de la collection qui, à l'exécution, sont du type de `a`.

2.6 Interface `Map`

Résumé des méthodes :

- `void clear()` vide la collection.
- `boolean containsKey(Object key)` teste si la table contient une entrée avec la clé spécifiée.
- `boolean containsValue(Object value)` teste si la table contient une entrée avec la valeur spécifiée.
- `Set entrySet()` renvoie une vue ensembliste de la table.

- `boolean equals(Object o)` teste l'égalité de contenu de la table avec `o`.
- `Object get(Object key)` renvoie la valeur de la table correspondant à la clé `key`.
- `int hashCode()` renvoie le code de hachage de la table.
- `boolean isEmpty()` teste si la table est vide.
- `Set keySet()` renvoie une valeur ensembliste des clés de la table.
- `Object put(Object key, Object value)` associe la valeur `value` à la clé `key` dans la table. Si une valeur était déja associée, la nouvelle remplace l'ancienne et une référence vers la nouvelle est renvoyée, sinon `null` est renvoyé.
- `void putAll(Map t)` copie toutes les entrées de `t` dans la table.
- `Object remove(Object key)` enlève l'entrée associée à `key` de la table. Renvoie une référence sur la valeur retirée ou `null` si elle n'est pas présente.
- `int size()` renvoie le nombre d'entrées (paires clé-valeur) de la table.
- `Collection values()` renvoie une vue de type `Collection` des valeurs de la table.

2.7 Classe Arrays

- Classe de manipulation de tableaux.
- Méthode `static List` *`asList`*`(Object[] a)` renvoie une vue de type `List` de `a`.
- Sinon, 4 groupes de méthodes principales (en tout 54 méthodes) :
  - Dans ce qui suit, `Type` désigne soit un type primitif, soit `Object`. Voir la documentation Java des API pour les signatures précises des méthodes.
  - `static int` *`binarySearch`*`(Type[] a, Type key)` effectuant une recherche de `key` dans `a`.
  - `static int` *`equals`*`(Type[] a, Type[] b)` teste l'égalité élt. à élt. de `a` et `b`.
  - `static int` *`fill`*`(Type[] a, Type val)` affecte tous les éléments de `a` à `val`.
  - `static int` *`sort`*`(Type[] a)` trie `a` selon un alogrithme quicksort modifié.

2.8 Conteneurs immuables

- Toutes les méthodes de modification de `Collection` et `Map` sont documentées comme optionnelles.
- On doit les redéfinir, mais le code peut juste lever une `UnsupportedException`.
- Si toutes ces méthodes lèvent une telle exception, le conteneur est dit immuable.

- Dans **la classe** `Collections` (et non dans l'interface `Collection`), constantes (`public static final`) représentant des collections vides : `EMPTY_MAP`, de type `Map`, `EMPTY_SET` de type `Set` et `EMPTY_LIST` de type `List`.
- Singletons immuables :
  - `static Set singleton(Object o)`, singleton de type `Set`,
  - `static List singletonList(Object o)`, singleton de type `List`,
  - `static Map singletonMap(Object key, Object value)`, singleton de type `Map`.
- Vues immuables d'un conteneur :
  - `static Collection unmodifiableCollection(Collection c)`,
  - `static ListunmodifiableList(List list)`,
  - `static MapunmodifiableMap(Map m)`,
  - `static SetunmodifiableSet(Set s)`,
- Vues immuables triées d'un conteneur :
  - `static SortedMapunmodifiableSortedMap(SortedMap m)`,
  - `static SortedSetunmodifiableSortedSet(SortedSet s)`

2.9 Concurrence et synchronisation

- Aux exceptions de `Vector` et `Hashtable` près (existant depuis la version 1.0 du langage et conservées pour des raisons de compatibilité), les implantations de `Map` et de `Collection` sont non synchronisées.
- La classe `Collections` contient des méthodes renvoyant des vues synchronisées :
  - `static Collection synchronizedCollection(Collection c)`,
  - `static List synchronizedList(List list)`,
  - `static Map synchronizedMap(Map m)`,
  - `static Set synchronizedSet(Set s)`.
  - Et, pour les collections triées :
  - `static SortedMap synchronizedSortedMap(SortedMap m)`,
  - `static SortedSet synchronizedSortedSet(SortedSet s)`.

2.10 Itération de conteneurs

- Par le biais de l'interface `Iterator`.
- Elle définit des méthodes par lesquelles ont peut énumérer (un à la fois) des éléments d'une collection.
- Elle spécifie 3 méthodes :
  - `boolean hasNext()` renvoie `true` s'il y a encore des éléments dans la collection,

  - `Object next()` renvoie une référence sur l'instance suivante de la collection,
  - `remove()` enlève l'élément renvoyé dernièrement par l'itérateur.
- Ces itérateurs sont à échec rapide ("fail-fast")

```
Collection col = new ArrayList();
for (int i = 0; i < 10; i++)
    col.add(new Integer(i));
Iterator i1 = col.iterator();
System.out.println(i1.next());// Affiche 0
Iterator i2 = col.iterator();
System.out.println(i2.next());// Affiche 0
i1.remove();                  // modification via i1
  // leve une ConcurrentModificationException
System.out.println(i2.next());
```

- Le code suivant, par contre, s'exécute normalement

```
Collection col = new ArrayList();
for (int i = 0; i < 10; i++)
    col.add(new Integer(i));
Iterator i1 = col.iterator();
System.out.println(i1.next());    // Affiche 0
i1.remove();                      // modification via i1 (enleve 0)
System.out.println(i1.next());    // Affiche 1
Iterator i2 = col.iterator();
System.out.println(i2.next());    // Affiche 1
```

  la même instance de l'itérateur, déjà créé, doit être utilisée ensuite.
- Un exemple typique d'itération est la boucle `for` suivante

```
static void filter(Collection c) {
  for (Iterator i = c.iterator(); i.hasNext();)
      if (!cond(i.next()))
          i.remove();
}
```

  Noter que ce code est polymorphe (il fonctionne pour toute instance de `Collection`)

2.11 Squelettes d'implantation

- Des classes abstraites squelettes facilitent l'implantation.
- Les opérations (méthodes) de modification ne font rien sauf générer une exception de type `UnsupportedOperationException`.
- Par ex., pour créer une classe immuable de type `Collection`, il suffit d'hériter de `AbstractCollection` et d'implanter `Iterator iterator()` et `int size()`
- Pour définir des conteneurs modifiables, il faut implanter `boolean add(Object o)` et la méthode `boolean remove(Object o)` de l'itérateur renvoyé par `Iterator iterator()`.

- De la même manière, on dispose des classes `AbstractMap`, `AbstractSet`, `AbstractList` et `AbstractSequentialList`.

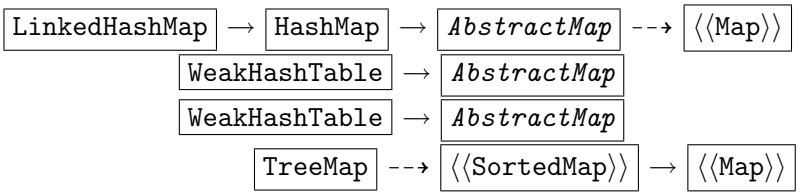
X.3 Conteneurs de type `Map`

3.1 Conteneurs de type `Map`

- `Map` est une interface qui représente un mécanisme de stockage clé/valeur.
- Une *clé* est un nom que l'on utilise pour accéder à une *valeur*.
- Il s'agit d'une représentation abstraite d'un *tableau associatif*.
- Les couples (clé, valeur) sont des instances de classes implantant l'interface `Map.entry`.

3.2 Conteneurs de type `Map`

- La hiérarchie des classes est la suivante (→ : hérite de, ⇢ : implante) :



- les classes `WeakHaskTable`, `HashMap`, `LinkedHashMap` et`IdentityHashMap` utilisent des tables de hachage.
- `TreeMap` utilise des arbres rouges-noirs.

3.3 Classe `HashMap`

- `HashMap` est la plus utilisée des `Map` en pratique.
- *Table de hachage* : une représentation d'une clé est utilisée pour déterminer une valeur autant que possible unique, nommée *code de hachage* (voir ce qui suit pour une brève description du hachage).
- Le code hachage est alors utilisé comme indice auquel les données associées à la clé sont stockées.

3.4 Classe `HashMap`

- Pour utiliser une table de hachage :
  - On fournit un objet utilisé comme clé et des données que l'on souhaite voir liées à cette clé.

  - La clé est hachée.
  - Le code de hachage résultant est utilisé comme indice auquel les données sont stockées dans la table.
- Les valeurs de codes de hachage sont cachées (encapsulées).
- Une table de hachage ne peut stocker que des clés qui redéfinissent les méthodes `hashCode()` et `equals()` de `Object`.

3.5 Classe HashMap

- La méthode `hashCode()` doit calculer le code de hachage de l'objet et le renvoyer.
- `equals()` compare 2 objets.
- Beaucoup de classes courantes de Java implantent la méthode `hashCode()`. C'est le cas de `String`, souvent utilisée comme clé.

3.6 Classe HashMap

- Les constructeurs de `HashMap` sont :
  - `HashMap()`, constructeur par défaut, construit une table de hachage vide.
  - `HashMap(int capaciteInitiale)`, construit une table de hachage de taille initiale `capaciteInitiale`.
  - `HashMap(int capaciteInitiale, float tauxCharge)`, construit une table de hachage de taille initiale `capaciteInitiale` et de taux de remplissage `tauxCharge` ; ce taux, nécessairement compris entre `0.0` et `1.0`, détermine à quel pourcentage de remplissage la table sera re-hachée en une plus grande.

  Si `tauxCharge` n'est pas spécifié, `0.75` est utilisé.

3.7 Classe HashMap

- Si l'on veut utiliser ses propres classes comme clé de hachage, il faut redéfinir `hashCode()` et `equals()` de `Object`.
- La valeur (`int`) renvoyée par `hashCode()` est ensuite automatiquement réduite par une opération modulo la taille de la table de hachage.
- Il faut s'assurer que la fonction de hachage utilisée répartit aussi uniformément que possible les valeurs renvoyées entre `0` et `capaciteInitiale`, la taille initiale de la table.

3.8 Classe HashMap

- Méthodes de `HashMap` :

`methode()`	But
`void clear()`	Réinitialise et vide la table de hachage.
`Object clone()`	Renvoie un duplicata de l'objet appelant. Toute la structure de la table de hachage est copiée, mais ni les clés, ni les valeurs ne le sont. Il s'agit d'une copie superficielle (shallow copy).
`boolean containsKey(Object key)`	Renvoie `true` s'il existe une clé égale à `key` (comme déterminé par la méthode `equals()`) dans la table de hachage. Renvoie `false` sinon.
`boolean containsValue(Object value)`	Renvoie `true` s'il existe une valeur égale à `value` (comme déterminé par la méthode `equals()`) dans la table de hachage. Renvoie `false` sinon.
`Set entrySet()`	Renvoie une vue de type `Collection` des entrées contenues dans la table.
`Object get(Object key)`	Renvoie une référence sur l'objet contenant la valeur associée à la clé `key`. Si `key` n'est pas dans la table de hachage, une référence `null` est renvoyée.
`boolean isEmpty()`	Renvoie `true` si la table de hachage est vide et `false` sinon.
`Set keySet()`	Renvoie une vue de type `Set` des clés contenues dans la table.
`Object put(Object key, Object value)`	Insère une clé et sa valeur dans la table de hachage. Renvoie `null` si la clé `key` n'est pas déja dans la table de hachage, ou la valeur précédente associée à `key` si elle est déja dans la table de hachage.
`void putAll(Map t)`	Copie toutes les entrées de `t` dans la table. Dans le cas où une entrée de même clé était déjà prśente dans la table, elle est écrasée par celle de `t`.
`Object remove(Object key)`	Enlève la clé `key` et sa valeur. Renvoie la valeur associée à `key`. Si `key` n'est pas dans la table de hachage, une référence `null` est renvoyée.
`int size()`	Renvoie le nombre d'entrées la table de hachage.

`String toString()`	Renvoie une forme affichable d'une table de hachage. Il s'agit de la liste des entrées de la table, chaque entrée étant entourée d'accolades et séparée de la suivante par une virgule. Chaque entrée est constituée de la forme affichable (*via* `toString()`) de la clé, suivi du signe `=`, suivi de la forme affichable (*via* `toString()`) de la valeur associée.
`Collection values()`	Renvoie une vue de type `Collection` des valeurs de la table.

3.9 Hachage : principe

- Une *fonction de hachage* est une fonction $f : x \to h$ aisément calculable, qui transforme une très longue entrée x en une sortie h nettement plus courte, (typiquement de 10^6 bits à 200 bits) et qui a la propriété suivante :
- **(Phach)** : **Il n'est pas "calculatoirement faisable" de trouver deux entrées différentes x et x' telles que $f(x) = f(x')$.**
- L'expression "l'opération $\mathcal{O}$ n'est pas calculatoirement faisable" signifie simplement tous les algorithmes actuellement connus pour réaliser $\mathcal{O}$ sont de complexité exponentielle.

3.10 Hachage : autentification

Application à l'autentification de messages :

- Supposons que Alice veuille envoyer un message à Bob, en signant son message.
- Les données qu'Alice veut transmettre sont constituées d'un message en clair suivi de ses prénom et nom, en clair, à la fin du message. Nommons x cet ensemble de données.
- Alice transmet alors x, en clair, suivi de $h = f(x)$ où f est une fonction de hachage.
- À la réception, Bob applique la fonction de hachage f au texte en clair x et le compare à h.
- Ainsi, Bob peut vérifier non seulement que le message provient bien d'Alice (que sa signature n'a pas été falsifiée), mais également que son message, en clair, n'a pas été altéré.
- Par supposition, aucun pirate n'aurait été capable de modifier x sans changer la valeur de $h = f(x)$.

3.11 Hachage : recherche

- Application à la recherche. Supposons avoir une clé de recherche relativement longue (un entier ou une chaîne de caractères).
- La sortie de la table de hachage sera un indice d'une table dans laquelle sont rangées les valeurs associées aux différentes clés.
- Prenons le cas où la clé est une chaîne de caractères x et où la fonction de hachage f choisie la transforme en $h = f(x)$ un indice entre 1 et p (il y a p indices différents dans la table de hachage).

3.12 Hachage : recherche

- La propriété (Phach) assure que les sorties de f sont quasi-unfiromément distribuées, en un sens probabiliste, dans $[1, p]$.
- Prenons comme exemple de fonction de hachage simple la fonction modulo un nombre premier.
- Prenons alors pour p un nombre premier (par exemple 101) et considérons la clé suivante : `VERYLONGKEY`

3.13 Hachage : recherche

- On décompose la clé selon la base de son alphabet (ici, il y a 32 signes dans l'alphabet considéré) :

$$22.32^{10} + 5.32^9 + 18.32^8 + 25.32^7 + 12.32^6 + 15.32^5 + 14.32^4 + 7.32^3 + 11.32^2 + 5.32 + 25$$

- La fonction de hachage considérée ne prend pas directement ce nombre pour en faire l'opération modulo 101, sa repésentation machine étant lourde à manier ; il s'écrit en effet en binaire par

$$10110001011001011001011000111101110001110101100101110 01$$

- Il est bien plus efficace de se servir de la représentation d'un polynôme par l'algorithme de Hörner, où `VERYLONGKEY` s'écrit, en base 32, de la façon suivante :

$$(((((((((22.32 + 5)32 + 18)32 + 25)32 + 12)32 + 15)32 + 14)32 + 7)32 + 11)32 + 5)32 + 25$$

3.14 Hachage : recherche

- L'algorithme de calcul de la fonction de hachage est alors

```
public final int hache(String cle, int tailleTable)
{
  int valHach = 0;
```

```
    h = cle.charAt(0);
    for(int i = 1; i < cle.length(); i++)
      valHach = ((valHach*32)+cle.charAt(i)) % tailleTable;

    return valHach;
  }
```

où `cle` est une `String` dans lequel on a stocké la clé.

3.15 Hachage : recherche

- Pour `p == 101` et `cle[]` valant `"VERYLONGKEY"`, cette fonction de hachage fournit 97.
- ☛ Le calcul d'un indice à partir d'une clé est rapide, mais **rien ne garantit que 2 clés distinctes donneront des indices distincts**.
- On nomme *collision* d'indice le fait que 2 clés distinctes donnent le même indice.
- Il faut alors une stratégie de *résolution de collision*.

3.16 Hachage : recherche

- Une stratégie simple et efficace est le *chaînage séparé*. À chaque fois qu'il y a une collision pour l'indice `i`, les clés sont rangées dans une liste chaînée n° `i`, associée à la case d'indice `i` de la table. Les différents éléments de la liste chaînée peuvent être rangés en ordre alphabétique croissant des clés, pour un accès plus rapide.
- Cette stratégie est bien adaptée au cas où l'on ne connaît pas, *a priori*, le nombre d'enregistrements (de paires clés/valeurs) à traiter, ce qui est le cas de la classe `HashMap` de Java.

3.17 Hachage : recherche

- En Java, un code de hachage est généré (*via* la méthode `hashCode()` définie dans la classe `Object`. Elle renvoie alors en général une conversion de l'adresse de l'objet en `int`, bien que ceci ne soit pas une obligation d'implantation du langage.
- La méthode `hashCode()` est redéfinie par les types suivants : `BitSet`, `Boolean`, `Character`, `Date`, `Double`, `File`, `Float`, `Integer`, `Long`, `Object` et `String`,

3.18 Hachage : recherche

- Pour `String`, le code est obtenu de l'une des 2 manières suivantes, selon sa longueur. Soit n la longueur de la suite de caractères et c_i le caractère d'indice i.
  - Si $n \leqslant 15$, le code de hachage est calculé par

$$\sum_{i=0}^{n-1} c_i.37^i$$

    en utilisant l'arithmétique des `int`

3.19 Hachage : recherche

  - Si $n > 15$, le code de hachage est calculé par

$$\sum_{i=0}^{m} c_{i.k}.39^i$$

    en utilisant l'arithmétique des `int`, où $k = \lfloor \frac{n}{8} \rfloor$ et $m = \lceil \frac{n}{k} \rceil$, ne prenant (dans la décomposition) que 8 ou 9 caractères de la chaîne.

Pour les implantations des autres types, voir [GJS96].

3.20 Itération d'une `HashMap`

- L'opération d'itération sur une `HashMap` est possible (via `values()`), mais présente 2 inconvénients :
  - (1) L'ordre d'itération est indéterminé.
  - (2) La complexité de l'itération est linéaire en la **capacité** de la table.
  - Pour un conteneur adapté à l'itération, c'est une fonction linéaire de la **taille** du conteneur.

3.21 Classe `LinkedHashMap`

- Pour corriger les insuffisances en itération de `HashMap`, le conteneur contient une table de hachage ainsi qu'une liste doublement chaînée de ses éléments.

- L'itération est ainsi de complexité linéaire en la taille de la table.
- L'ordre d'itération est celui d'insertion des clés.
- Il existe un constructeur supplémentaire par rapport à `HashMap`, `public LinkedHashMap(int initialCapacity, float loadFactor, boolean accessOrder)` dont le dernier paramètre spécifie le type d'ordre d'accés. Si `accessOrder` est

égal à `false`, l'ordre d'accès est celui des clés (valeur par défaut prise dans les autres constructeurs) ; s'il est égal à `true`, l'ordre d'itération est l'ordre d'accès des entrées du plus ancien au plus récent.
- La méthode protégée `removeEldestEntry(Map.Entry eldest)` renvoie un booléen représentant une condition impliquant la destruction de l'élélement transmis en paramètre.

```
import java.util.*;
public class CacheMap extends LinkedHashMap {
  int maxSize;
  // Construit un cache de taille maxSize de taille initiale
  //   vide avec une capacite de 16 et un facteur de charge de 75%
  public CacheMap(int maxSize) {
      super(16, 0.75f, true); // true pour choisir l'ordre d'acces
      this.maxSize = maxSize;
      }

  // Determine si "le plus ancien elt" doit etre jete
  protected boolean removeEldestEntry(Map.Entry eldest) {
      return (size() > maxSize);
  }

  public static void main(String args[]) {
      ChacheMap map = new CacheMap(3); // cache de taille 3
      map.put("1", "un");
      map.get("1");
      map.put("2", "deux");
      map.put("3", "trois");
      map.put("4", "quatre");
      Set entrySet = map.entrySet();
      // Iteration sur le conteneur
      for (Iterator it = entrySet.iterator(); it.hasNext(); )
          System.out.println(it.next() + " ");
}
```

- Les clés ne sont pas comparées avec `equals()` mais par égalité des références (==).
- on utilise `System.identityHashCode()` qui utilise les références et non `hashCode()`.
- On donne au constructeur la taille maximale de la table (et non la capacité et le facteur de charge). La taille est augmentée si besoin est.
- Utilisée en sérialisation, où l'environnement d'exécution associe un identificateur à chaque référence. Lorsqu'on rencontre une référence déjà prise en compte, on utilise son identificateur pour la représenter.

3.22 Classe `WeakHashMap`

- Se comporte comme `HashMap`, mais les clés sont des références faibles.
- Les clés qui ne sont référencées que par la table sont susceptibles d'être détruites par le ramasse-miettes pour libérer de la mémoire.

3.23 Interface `SortedMap`

- Implantation de `Map` dans laquelle les entrées peuvent être ordonnées suivant les clés.
- Il faut fournir 2 constructeurs suplémentaires :
  - l'un avec un paramètre de type `SortedMap`, réalisant une copie de la table fournie, avec le même ordre.
  - l'autre avec un paramètre de type `Comprarator` fixant l'ordre.

3.24 Interface `SortedMap`

- Méthodes supplémentaires de `SortedMap`
  - `Comparator comparator()` renvoie le comparateur associé à la table triée, ou `null` s'il utilise l'ordre naturel des clés.
  - `Object firstKey()` renvoie la plus petite clé de la table triée.
  - `SortedMap headMap(Object toKey)` renvoie une vue de la partie de la table triée dont les clés sont strictement plus petites que `toKey`.

3.25 Interface `SortedMap`

  - `Object lastKey()` renvoie la plus grande clé de la table triée.
  - `SortedMap subMap(Object fromKey, Object toKey)` renvoie une vue de la partie de la table triée dont les clés sont comprises strictement entre `fromKey` et `toKey`.
  - `SortedMap tailMap(Object fromKey)` renvoie une vue de la partie de la table triée dont les clés sont strictement plus grandes que `fromKey`.

3.26 Classe `TreeMap`

- Implante l'interface `SortedMap`.
- structure de données sous-jacente : arbres rouges-noirs.
- Les clés sont constamment ordonnées en ordre croissant, selon l'ordre naturel des clés, ou selon un objet de comparaison fourni à la création, selon le constructeur utilisé.

- Temps d'accès en insertion, recherche, suppression (`containsKey()`, `get(...)`, `put(...)` et `remove(...)`) en $\mathcal{O}(\log n)$ où n est la taille du conteneur.

X.4 Conteneurs de type `Collection` et Listes

4.1 Conteneurs de type `Collection`

Conventions :
- indentation : héritage,
- `«interface»`,
- `[[classe abstraite]]`

```
<<Collection>>
    <<List>>
    [[AbstractCollection]] (implements Collection)
        [[AbstractList]] (implements List)
            [[AbstractSequentialList]]
                LinkedList (implements List)
            ArrayList (implements List, RandomAccess)
        [[AbstractSet]] (implements Set)
            HashSet (implements Set)
                LinkedHashSet (implements Set)
            TreeSet (implements SortedSet)
```

4.2 Interface `List`

- Interface pour les classes de type listes et ensemble.
- `void add(int index, Object element)` insère `element` à la position spécifiée dans la liste.
- `boolean addAll(Collection c)` ajoute tous les élts. de `c` à la fin de la liste.
- `boolean addAll(int index, Collection c)` ajoute tous les élts. de `c` à la position spécifiée dans la liste.
- `void clear()` vide la liste.
- `boolean containsAll(Collection c)` teste si la liste contient tous les élts. de `c`.
- `Object get(int index)` renvoie l'élt. situé à la position spécifiée dans la liste.
- `int hashCode()` renvoie le code de hachage de cette liste.
- `int indexOf(Object o)` renvoie l'indice de la 1ère occurrence de `o` dans la liste.
- `boolean isEmpty()` test si la liste est vide.

- `int lastIndexOf(Object o)` renvoie l'indice de la dernière occurrence de `o` dans la liste.
- `ListIterator listIterator()` renvoie un `listIterator` des élts. de cette liste. Returns a list iterator of the elements in this list (in proper sequence).
- `ListIterator listIterator(int index)` renvoie un `listIterator` des élts. de cette liste à partir de `index`.
- `Object remove(int index)` enlève l'élt. spécifié de la liste.
- `boolean removeAll(Collection c)` enlève de la liste tous les élts. de `c`.
- `boolean retainAll(Collection c)` ne garde dans la liste que les élts. de `c`.
- `Object set(int index, Object element)` remplace l'élt. de la liste à la position `index` par `element`.
- `int size()` renvoie le nombre d'élts. de la liste.
- `List subList(int fromIndex, int toIndex)` renvoie une vue de la portion de liste comprise entre `fromIndex` inclus et `toIndex`, exclus.
- `Object[] toArray()` renvoie un tableau des élts. de la liste.
- `Object[] toArray(Object[] a)` renvoie un tableau des élts. de la liste. Le type du tableau renvoyé est le type de `a` à l'exécution.

4.3 Itération des listes

- Itérateur spécialisé des listes : interface `ListIterator`; Méthodes supplémentaires par rapport à `Iterator`.

`methode()`	But
`void add(Object o)`	insère l'élément spécifé dans la liste.
`boolean hasPrevious()`	teste si la liste contient un prédécesseur de l'élément courant.
`int nextIndex()`	renvoie l'indice de l'élément renvoyé par `next()`.
`Object previous()`	renvoie l'élément précédent de la liste.
`int previousIndex()`	renvoie l'indice de l'élément renvoyé par `previous()`.
`void set(Object o)`	remplace le dernier élément renvoyé par `next()` ou `previous()` par l'élément cpécifié.

4.4 Ensembles

- Sous-type de `Collection` ne pouvant contenir 2 élts. identiques.

- Ensembles implantent `Set`, ensembles ordonnés implantent `SortedSet`.
- Mêmes méthodes que `Collection`, sémantique modifiée : pas 2 élts identiques au sens d'`equals()`.
- Classe abstraite `AbstractSet` admet 3 sous-classes concrètes : `HashSet`, `LinkedHashSet`, `TreeSet`.

4.5 Classe `HashSet`

- Contient une table de hachage.
- Pas de gestion de relation d'ordre sur les éléments.
- `iterator()` renvoie les éléments dans un ordre quelconque.
- Opérations d'ajout, retrait et recherche prennent un temps constant.
- Constructeur avec capacité at taux de charge.

4.6 Classe `LinkedHashSet`

- Hérite de `HashSet`.
- Contient une `LinkedHashMap` pour stocker les eĺéments.
- Permet de maintenir un ordre entre les eĺéments.
- Cet ordre est nécessairement celui d'insertion dans l'ensemble.

4.7 Classe `TreeSet`

- Contient un `TreeMap` (arbre rouge-noir) pour stocker les éléments.
- Implante l'interface `SortedSet`.
- Les clés sont constamment ordonnées en ordre croissant.
- Temps d'accès en insertion, recherche, suppression (`containsKey()`, `get(...)`, `put(...)` et `remove(...)`) en $\mathcal{O}(\log n)$ où n est la taille du conteneur.
- Méthodes supplémentaires de l'interface `SortedSet` :

`methode()`	But
`Object first()`	renvoie le 1er elt. de l'ensemble trié.
`SortedSet headSet(Object toElement)` renvoie une vue des élts strictement plus petits que `toElement. Object last()`	renvoie le dernier elt. de l'ensemble trié.
`SortedSet subSet(Object fromElement, Object toElement)`	renvoie une vue des élts strictement compris entre `toElement` et `fromElement`.
`SortedSet tailSet(Object fromElement)`	renvoie une vue des élts strictement plus grands que `fromElement`.

4.8 Suites

- Suites à accès direct (dans n'importe quel ordre) : implantent `RandomAccess`.
- Suites à accès séquentiel (pour accéder à $i+1$, accéder à i d'abord) : héritent de `AbstractSequentialList`.
- Suites à accès direct : `ArrayList`.
- Suites à accès séquentiel : `LinkedList`.

4.9 Suites à accès direct

- Un telle suite possède une capacité initiale.
- S'il ne reste plus de place, il faut augmenter la taille de la liste, opération en $\mathcal{O}(n)$.
- L'insertion a donc une complexité au pire de $\mathcal{O}(n)$.
- En augmentant la taille astucieusement, on assure que la complexité de i insertions est en $\mathcal{O}(i)$.
- Le calcul de la taille, l'accès à un élément et l'affectation à une position donnée a une complexité en $\mathcal{O}(1)$
- La suppression est en $\mathcal{O}(n)$.

4.10 Classe `ArrayList`

- `ArrayList` est, grossièrement parlant, un tableau à longueur variable de références à des objets.
- `ArrayList` n'est pas synchronisée par défaut. Si l'on désire avoir un tableau à longueur variable synchronisé, utiliser
  `static Collection synchronizedCollection(Collection c)`
  de la classe `Collections`.
- On dispose de 3 constructeurs :
  - `ArrayList()` créé une liste de taille initiale 10 références.
  - `ArrayList(int size)` créé une liste de taille initiale `size` références.
  - `ArrayList(Collection c)` créé une liste avec les élts de `c`. La capacité initiale de la liste est de 110% celle de `c`.

Les différentes méthodes sont :

`methode()`	But
`void add(int index, Object element)`	L'objet spécifié par `element` est ajouté à l'endroit spécifié de la liste.

`boolean add(Object o)`	L'objet spécifié par `element` est ajouté à la fin de la liste.
`boolean addAll(Collection c)`	Les élts. de `c` sont ajoutés à la fin de la liste.
`boolean addAll(int index, Collection c)`	Les élts. de `c` sont ajoutés à l'endroit spécifié de la liste.
`void clear()`	vide la liste.
`Object clone()`	Renvoie un duplicata (copie superficielle) de la liste appelante.
`boolean contains(Object element)`	Renvoie `true` si `element` est contenu dans la liste et `false` sinon.
`final void ensureCapacity(int size)`	Fixe la capacité minimale de la liste à `size`.
`Object get(int index)`	renvoie l'élélement situé à la position spécifiée de la liste.
`final int indexOf(Object element)`	Renvoie l'indice de la 1ière occurrence de `element`. Si l'objet n'est pas dans la liste, `-1` est renvoyé.
`boolean isEmpty()`	Renvoie `true` si la liste ne contient aucun élément et `false` sinon.
`int lastIndexOf(Object element)`	Renvoie l'indice de la dernière occurrence de `element`. Si l'objet n'est pas dans la liste, `-1` est renvoyé.
`Object remove(Object element)`	Enlève la première occurrence de `element` trouvée dans la liste. Renvoie une référence sur l'élément enlevé.
`protected void removeRange(int fromIndex, int toIndex)`	Enlève les éléments situé entre les indice `fromIndex` (inclus) et `toIndex` (exclus).
`Object set(int index, Object element)`	Remplace l'élément à la position spécifiée par `element`.
`int size()`	Renvoie la taille de la liste.
`Object[] toArray()`	renvoie un tableau des élts. de la liste.
`Object[] toArray(Object[] a)`	renvoie un tableau des élts. de la liste. Le type du tableau renvoyé est le type de `a` à l'exécution.
`void trimToSize()`	Fixe la capacité de la liste au nombre d'éléments qu'elle contient actuellement.

4.11 Classe `ArrayList`

Exemple :

```
import java.util.ArrayList;
import java.util.Iterator;
```

```
class DemoArrayList {
    public static void main(String args[]) {
        // Taille initiale de 3
        ArrayList l = new ArrayList(3);

        System.out.println("Taille initiale : " + l.size());

        l.add(new Integer(1));
        l.add(new Integer(2));
        l.add(new Integer(3));
        l.add(new Integer(4));
        l.add(new Double(18.23));
        l.add(new Integer(5));
        System.out.println("1er element : " +
                                (Integer)l.get(0));
        System.out.println("Dernier element : " +
                                (Integer)l.get(l.size()-1));

        if (l.contains(new Integer(3)))
            System.out.println("l contient l'entier 3");

        // Listons les elements de la liste
        ListIterator it = l.listIterator();

        System.out.println("\n Elements dans la liste : ");
        while (it.hasNext())
            System.out.print(it.next() + " ");
        System.out.println();
  }
}
```

La sortie écran du programme est :

```
Taille initiale : 3
1er element : 1
Dernier element : 5
l contient l'entier 3
Elements dans la liste :
1 2 3 4 18.23 5
```

4.12 Suites à accès séquentiel

– Classe `LinkedList`, de structure sous-jacente une liste doublement chaînée.

- Ajout/suppression en début de liste en temps constant ($\mathcal{O}(1)$).
- Insertion/suppression d'un élt. juste après un élt. donné (par ex. par un itérateur) en temps constant.
- Accès à l'élt. i en $\mathcal{O}(i)$.

4.13 Classe `LinkedList`

Méthodes supplémentaires de `LinkedList`

`methode()`	But
`void addFirst(Object o)`	insère l'élt. spécifié au début de la liste.
`void addLast(Object o)`	ajoute l'élt. spécifié à la fin de la liste.
`Object getFirst()`	renvoie le 1er élt. de la liste.
`Object getLast()`	renvoie le dernier élt. de la liste.
`Object removeFirst()`	enlève et renvoie le 1er élt. de la liste.
`Object removeLast()`	enlève et renvoie le dernier élt. de la liste.

4.14 Classe `Stack`

- `Stack` implante une pile (file LIFO, Last In/First Out) standard.
- `Stack` est une sous classe de `Vector`. Elle hérite donc de toutes les méthodes de `Vector`, et en définit cetraines qui lui sont propres.
- $\Rightarrow$ `Stack` n'est pas une pile au sens puriste du terme ...
- Il est toutefois utile d'avoir accès aux méthodes de `Vector`.
- Un seul constructeur, sans arguments, qui créé une pile vide.

4.15 Classe `Stack`

- Méthodes propres de `Stack` :

`methode()`	But
`boolean empty()`	Renvoie `true` si la pile est vide et `false` sinon.
`Object peek()`	Renvoie l'élément du dessus de la pile, mais ne l'enlève pas.
`Object pop()`	Renvoie l'élément du dessus de la pile, en l'enlèvant.
`Object push(Object element)`	Pousse `element` sur la pile. `element` est également renvoyé.

`int search(Object element)`	Cherche `element` dans la pile. S'il est trouvé, son offset par rapport au dessus de la pile est renvoyé. Sinon, `-1` est renvoyé.

- Une `EmptyStackException` est jetée si l'on appelle `pop()` lorsque la pile est vide.

4.16 ((*Interface Enumeration*))

- Elle définit des méthodes par lesquelles ont peut énumérer (un à la fois) des éléments d'un `Vector` ou d'une `Hashtable`.
- **L'utilisation de cette interface est dépréciée**. Il est recommandé d'utiliser `Iterator` à la place.
- Elle spécifie 2 méthodes :
  - `boolean hasMoreElements()`
  - `Object nextElement()`

  la 1ière doit renvoyer `true` tant qu'il y a encore des éléments dans la collection, et la 2ième renvoie une référence sur l'instance suivante de la collection.

4.17 ((*Classe Vector*))

- `Vector` est, grossièrement parlant, un tableau à longueur variable de références à des objets.
- **L'utilisation de cette classe est dépréciée**. Il est recommandé d'utiliser `ArrayList` à la place.
- La classe `ArrayList` n'est pas synchronisée par défaut. Si l'on désire avoir un tableau à longueur variable synchronisé, utiliser la méthode `static Collection synchronizedCollection(Collection c)` de la classe `Collections`.

4.18 ((*Classe Dictionary*))

- `Dictionary` est une classe abstraite qui représente un mécanisme de stockage clé/valeur.
- **L'utilisation de cette classe est dépréciée**. Il est recommandé d'utiliser la classe `AbstractMap` à la place.

4.19 Classe `Hashtable`

- `Hashtable` est une implantation concrète de `Dictionary`, qui peut être utilisée pour stocké des `Objects` référencés par d'autres `Objects`.

- **L'utilisation de cette classe est dépréciée**. Il est recommandé d'utiliser `HashMap` à la place.

4.20 ((*Classe `Properties`*))

- Une liste de propriétés est une sous-classe de `Hashtable` (**classe dépréciée** ...) où la clé est une `String`.
- Pratique lorsque l'on veut une micro base de données.
- Utilisé par d'autres classes Java (par ex. type renvoyé par `System.getProperties()` qui donne les variables d'environnement).
- Un champ est défini `protected Properties defaults ;` qui contient une liste de propriétés par défaut.
- Constructeurs :
  - `Properties()`, créé un objet sans liste par défaut.
  - `Properties(Properties defaultProp)`, créé un objet avec `defaultProp` comme liste par défaut.

  Dans les 2 cas, la liste des propriétés est vide.
- Liste des méthodes propres de `Properties`

`methode()`	But
`String getProperty(String key)`	Renvoie la valeur associée à la clé `key` ; la recherche s'effectue d'abord dans la liste, puis, si rien n'est trouvé, dans la liste par défaut. Une référence `null` est renvoyée si `key` ne se trouve ni dans la liste, ni dans la liste par défaut.
`String getProperty(String key, String defaultVal)`	Renvoie la valeur associée à la clé `key` si elle est trouvée et renvoie `defaultVal` sinon. Cette dernière peut être une valeur par défaut ou un message d'erreur.
`void list(PrintStream streamOut)`	Envoie la liste des propriétés au flux de sortie associé à `streamOut`. Utilisé pour l'affichage.
`void load(InputStream streamIn) thows IOException`	la liste des paires clé, valeur est lue à partir du flux `streamIn` et est ajoutée à la table.
`Enumeration propertyNames()`	Renvoie une énumération des clés. Ceci inclut les clés trouvées dans la liste des propriétés par défaut. S'il y a des clés en double, une seule est retenue.
`void save(OutputStream streamOut, String description)`	Écrit la chaîne `description`, puis la liste dans le flux associé à `streamOut`.

- Dans la lecture par `load()` :
  - Les lignes commençant par un `#` ou un `!` sont traitées comme des commentaires (elles ne sont pas lues).
  - Les séparateurs clé-valeur sont : `=`, `:` ou un espace.
  - Chaque paire clé-valeur doit tenir sur une ligne, sauf si l'on place un \ en fin de ligne, auquel cas l'entrée peut se poursuivre à la ligne suivante.
  - Tous les caractères blancs en début de ligne sont éliminés.
- Dans l'écriture par `save()` :
  - Aucun élément de la table par défaut n'est écrit.
  - Si l'argulent `description` n'est pas nul, il est écrit, précédé en début de ligne par un `#`. Il sert donc de commentaire identificateur.
  - Ensuite, un commentaire est toujours écrit, constitué d'un `#`, suivi de la date et de l'heure et d'un passage à la ligne.
  - Puis, chaque paire de la table est écrite, une par ligne. La clé est d'abord écrite, suivi d'un `=`, suivi de la valeur.

Exemple d'utilisation de la liste par défaut :

```
import java.util.Dictionary;
import java.util.Hashtable;
import java.util.Properties;
import java.util.Enumeration;

class PropDemoDef {
  public static void main(String args[]) {
    Properties defList = new Properties();
    defList.put("Florida", "Tallahassee");
    defList.put("Wisconsin", "Madison");

    Properties capitals = new Properties(defList);
    Enumeration states;
    String str;

    capitals.put("Illinois", "Springfield");
    capitals.put("Missouri", "Jefferson City");
    capitals.put("Washington", "Olympia");
    capitals.put("California", "Sacramento");
    capitals.put("Indiana", "Indianapolis");

    // Montrer tous les etats et capitales de la table.
    states = capitals.keys();
    while(states.hasMoreElements()) {
      str = (String) states.nextElement();
```

```
      System.out.println("The capital of " +
                         str + " is " +
                         capitals.getProperty(str)
                         + ".");
    }

    System.out.println();

    // La Floride va etre trouvee dans la liste par defaut.
    str = capitals.getProperty("Florida");
    System.out.println("The capital of Florida is "
                       + str + ".");
  }
}
```

Exemple d'une base de données simpliste de numéros de téléphone.

```
import java.io.*;
import java.util.*;

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

    // Base de donnees : name et number
    Properties      ht = new Properties();
    String          name, number;

    // Autres variables
    BufferedReader  br =
      new BufferedReader(new InputStreamReader(System.in));
    String          finName;
    FileInputStream fin = null;
    boolean         changed = false;

    if (args.length > 1)
        finName  = new String(args[1]);
    else
        finName  = new String("agendatel.prp");

    // Essai d'ouverture de phonebook.dat
    try {
      fin = new FileInputStream(finName);
```

```
} catch(FileNotFoundException e) {
  // ignore missing file
}

// Si le fichier des nos de telephone existe,
//   charger les nos. existants.
try {
  if(fin != null) {
    ht.load(fin);
    fin.close();
  }
} catch(IOException e) {
  System.out.println("Erreur de lecture de " + finName);
}

// L'utilisateur entre les nouveaux noms et numeros.
do {
  System.out.println("Entrez un nouveau nom " +
                     "('quit' pour sortir) : ");
  name = br.readLine();
  if(name.equals("quit")) continue;

  System.out.println("Entrez le numero : ");
  number = br.readLine();

  ht.put(name, number);
  changed = true;
} while(!name.equals("quit"));

// Si l'agenda telephoneique a change, le sauvegarder.
if(changed) {
  FileOutputStream fout = new FileOutputStream(finName);
  ht.save(fout, "Agenda telephonique");
  fout.close();
}

// Charger des numeros en donnant un nom.
do {
  System.out.println("Entrez le nom a trouver " +
                     "('quit' pour sortir) : ");
  name = br.readLine();
  if(name.equals("quit")) continue;
```

```
      number = (String) ht.get(name);
      System.out.println(number);
    } while(!name.equals("quit"));
  }
}
```

4.21 Exemples : formes d'itération

```
// For a set or list
for(Iterator it=collection.iterator(); it.hasNext(); ) {
    Object element = it.next(); }

// For keys of a map
for(Iterator it=map.keySet().iterator(); it.hasNext(); ) {
    Object key = it.next();  }

// For values of a map
for(Iterator it=map.values().iterator(); it.hasNext(); ) {
    Object value = it.next();  }

// For both the keys and values of a map
for(Iterator it=map.entrySet().iterator(); it.hasNext(); ) {
    Map.Entry entry = (Map.Entry)it.next();
    Object key = entry.getKey();
    Object value = entry.getValue();  }
```

4.22 Exemple : utilisation d'une pile

```
LinkedList stack = new LinkedList();

// Push on top of stack
stack.addFirst(object);

// Pop off top of stack
Object o = stack.getFirst();

// If the queue is to be used by multiple threads,
// the queue must be wrapped with code to synchronize the methods
stack = (LinkedList)Collections.synchronizedList(stack);
```

4.23 Exemple : création d'une table de hachage

```
// Create a hash table
Map map = new HashMap();    // hash table
map = new TreeMap();        // sorted map

// Add key/value pairs to the map
map.put("a", new Integer(1));
map.put("b", new Integer(2));
map.put("c", new Integer(3));

// Get number of entries in map
int size = map.size();        // 2

// Adding an entry whose key exists in the map causes
// the new value to replace the old value
Object oldValue = map.put("a", new Integer(9));  // 1

// Remove an entry from the map and
//   return the value of the removed entry
oldValue = map.remove("c");  // 3
```

4.24 Exemple : itération de la table de hachage

```
// Iterate over the keys in the map
Iterator it = map.keySet().iterator();
while (it.hasNext()) {
    // Get key
    Object key = it.next();
}

// Iterate over the values in the map
it = map.values().iterator();
while (it.hasNext()) {
    // Get value
    Object value = it.next();
}
```

Bibliographie

[Boo94] G. Booch. *Object-Oriented Analysis and Design with Applications*. Benjamin/Cummings, Redwood City, CA, 1994. éd. française : *Analyse & conception orientées objets*, Addison–Wesley France, Paris, 1994. Un classique de l'ingéniérie du logiciel.

[Bud98] T. Budd. *Understanding Object-Oriented Programming with JAVA*. Addison-Wesley, Reading, MA, 1998. `http ://www.awl.com/cseng`. Excellent livre. Explique le "pourquoi" (vue compréhensive du langage) de la programmation java et pas seulement le "comment" (simple vue descriptive).

[Dij79] E. Dijkstra. *Programming Considered as a Human Activity*. Classics in Software Engineering. Yourdon Press, New York, 1979. Un classique.

[Flaa] D. Flanagan. *Java Examples in a Nutshell. A Tutorial Companion to Java in a Nutshell*. O'Reilly, Cambridge. Très bon livre d'exemples. Complète parfaitement le précédent.

[Flab] D. Flanagan. *Java in a Nutshell. A Desktop Quick Reference*. O'Reilly, Cambridge. Excellent livre. Notamment une description synthétique des classes et paquetages très bien faite.

[GHJV95] E. Gamma, R. Helm, R. Johnson, et J. Vlissides. *Design Patterns. Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, MA, 1995. Livre fondateur d'une technique maintenant classique en génie logiciel.

[GJS96] J. Gosling, B. Joy, et G. Steele. *The Java Language Specification*. Addison-Wesley, Reading, MA, 1996. Description très détaillée du langage par leurs fondateurs ; assez aride.

[Kob87] N. Koblitz. *A course in Number Theory and Cryptography*, volume 114 de *GTM*. Springer, New York, 1987. Deuxième édition. Un livre abordable sur un sujet difficile à traiter rigoureusement et simplement.

[RDBF02] G. Roussel, E. Duris, N. Bedon, et R. Forax. *Java et Internet – Concepts et programmation*. Vuibert, Paris, 2002. Tome 1 – Côté client. Un excellent livre de programmation réseau. Très bonne introduction à diverse aspects du langage hors réseau.

[RGI80] D.T. Ross, J.B. Goodenough, et C.A. Irvine. Software engineering : Process, principles and goals. Dans P. Freeman et A. Wasserman, éditeurs, *Tutorial on Software Design Techniques*. Computer Society Press of the IEEE, New York, 3e éd., 1980. Un article classique en génie logiciel.

[Seg91] R. Segdewick. *Algorithmes en langage C*. InterEditions, Paris, 1991. Un classique en algorithmique appliquée. Excellent compromis entre théorie et pratique.

[Wei98] M. A. Weiss. *Data Stuctures & Problem Solving Using Java*. Addison-Wesley, Reading, MA, 1998. `http ://www.awl.com/cseng/titles/0-201-54991-3`. Bon livre sur un sujet ultra-classique.

Index