Java : de l'esprit à la méthode

Distribution d'applications sur Internet

Deuxième édition

Michel BONJOUR, Gilles FALQUET,
Jacques GUYOT et André LE GRAND

Java : de l'esprit à la méthode

Distribution d'applications sur Internet

Deuxième édition

Vuibert Informatique

Java : de l'esprit à la méthode – Distribution d'applications sur Internet – 2e édition
Michel BONJOUR, Gilles FALQUET, Jacques GUYOT et André LE GRAND

Conception de la couverture : Jean Widmer

Contact : informatique@vuibert.fr

Les programmes et exemples figurant dans ce livre ont pour but d'illustrer les sujets traités. Il n'est donné aucune garantie quant à leur utilisation dans le cadre d'une activité professionnelle ou commerciale.

ISBN 2-7117-8647-1

à Anne-Murielle, Florence, Nicky et Sandra

Table des matières

Remerciements

Nous tenons à remercier ici :

Ian Prince, à l'origine de notre intérêt pour les hypertextes et les systèmes distribués.

Nos collègues de travail pour les échanges sur Java autour d'un café.

Les participants à notre premier cours Java pour leur patience, leur intérêt et leurs remarques constructives.

Les étudiants qui ont travaillé avec nous dans nos projets autour de Java et des bases de données.

Nos familles et nos amis qui ont supporté notre Javamania.

À propos des auteurs

Michel Bonjour, licencié en informatique de gestion, est désormais employé à la société Unicible (Lausanne) après avoir été assistant du groupe Bases de Données de l'Université de Genève.

fdfdfdfdfdf

Gilles Falquet, docteur ès sciences mention informatique, est actuellement maître d'enseignement et de recherche à l'Université de Genève. Il mène des travaux de recherche sur les modèles et technologies des bases de données et hypertextes.

Jacques Guyot, docteur ès sciences mention informatique, est chargé de cours à l'Université de Genève et responsable de l'informatique du département des finances de Montres Rolex S.A. Il s'intéresse à l'intégration des traitements et des données dans le cadre du Web.

André Le Grand, docteur en informatique de l'Université de Montpellier II, est actuellement maître-assistant à l'Université de Genève et à l'Institut de Hautes Études en Administration Publique (Lausanne). Ses travaux portent sur les méthodes de conception orientées objet et l'ingénierie des systèmes d'information.

Carnet de route cybernautique des auteurs

nov 93	Notre premier butineur sur Mac (NCSA Mosaic)
hiver 93	Premier support de cours sur le WEB
sept 94	Transparents complets d'un cours BD + SQL sur le Web pour les étudiants de sciences économiques et sociales
déc 94	Création sur le WEB du BNF Club avec la syntaxe de ADA, puis SQL, PL/SQL d'Oracle, SQL2, Modula2, etc.
mars 95	Enseignement post-grade sur Internet et création d'une visite guidée du Web Ouverture de WebTerrasse, site expérimental (WWW-BD)
avril 95	Travaux sur la publication des BD sur Internet :
	- génération d'hypertexte pour une BD (Langage LAZY)
	- création de passerelles WWW-SGBD (Oracle, Farandole2)

17 sep 95	Naissance de Noé
déc 95	Présentation de LAZY à la 4ème conférence WWW à Boston
jan 96	Test de JavaScript
fév 96	Ajout de la syntaxe de Java au BNF Club
	Démarrage du JavaCorner cuiwww.unige.ch/java
mars 96	Participation au cours du 3ème cycle romand : le Web et ses outils
	Portage de LAZY sur le WebServer d'Oracle
avril 96	Outils hypermédia sur le dictionnaire Oracle
mai 96	Début du projet FNRS (Fond national de la recherche suisse) pour l'étude de la génération d'hypertextes à partir des bases de données
	Organisation d'un cours sur Java
juin 96	Etude des spécifications de JDBC 1.0
	Compilateur de LAZY en Java
	Générateur et génération d'une documentation hypertextuelle pour Java dans le BNF Club
1997	Cours et Evangélisation Java !
1998	Implantation d'une œuvre d'art en Java avec l'artiste Hervé Grauman, pour l'Office fédéral de statistique en Suisse (www.collectivepainting.ch)
	Utilisation de Java et JDBC dans l'enseignement et la recherche (structures de données, interfaces de systèmes d'information)
mai 98	Ecriture d'une applet pour dessiner les diagrammes syntaxiques pour une spécification BNF
oct 98	Ecriture d'un générateur d'outils syntaxiques en pur Java

La suite est à découvrir sur http://cuiwww.unige.ch/java.

Introduction

Internet, FTP, TCP/IP, E-Mail, autoroutes de l'information et les derniers en date WWW, HTML, HTTP, XML, IIOP, CORBA, Java sont autant d'acronymes remplissant les discussions et les revues d'informatique. Utiles ou simples gadgets ? Technologies sûres ou de transition ? Quels nouveaux domaines émergent ? Comment l'informatique est-elle modifiée ? Autant d'interrogations qui sont soulevées.

Répondre à l'ensemble d'entre elles nécessitera certainement plusieurs années, demandera à ce que la société s'habille de ces nouveaux atours, s'émerveille, s'effraie et finalement intègre ces nouvelles technologies productrices de nouvelles valeurs. À notre avis, les informaticiens doivent prendre conscience de ces enjeux sociaux et développer une éthique de l'utilisation de ces nouvelles technologies[1]. Technologie sans conscience...

Notre propos restera plus prosaïque et moins polémique : comment utiliser efficacement Java, ce langage de programmation ?

Notre première partie, consacrée à **l'esprit** Java, situe le langage dans son contexte et décrit la transition « avant Java - après Java ». Nous y introduisons également les concepts essentiels pour la programmation distribuée sur Internet ou pour l'animation, par exemple.

La seconde partie décrit **la forme** de Java, il s'agit ici de décrire la syntaxe de Java et la sémantique associée. Le langage Java étant relativement simple, les règles de grammaire sont peu nombreuses.

La troisième partie s'attache à examiner **le fond** de Java. Nous étudierons un ensemble de problématiques telles que : comment dessiner, animer ou communiquer, et proposerons des solutions programmées en Java. Le lecteur trouvera à travers celles-ci de nombreux exemples de la syntaxe Java et pourra ainsi la maîtriser par la pratique. De plus, il abordera par ces problématiques l'utilisation

1. Le lecteur trouvera à la page 491 des références que nous trouvons intéressantes.

des classes standard de Java et l'application des différents concepts propres à la programmation orientée objet.

La quatrième partie du livre est dédiée à **la méthode**. Maîtrisant la syntaxe, la sémantique et les classes standard de Java, le programmeur doit alors faire face à la complexité de ses propres créations. Il doit effectuer des choix de conception pour ses objets qui ne seront pas sans importance quant à leur réutilisation, leur généricité, leur testabilité, etc. Cette partie comporte également une description des méthodes de sécurité de Java et de l'interface de programmation d'applications bases de données JDBC (*Java DataBase Connectivity*) proposée par Sun, Cette partie s'achève sur l'élaboration d'applications distribuées avec RMI et CORBA.

En annexe on trouvera : la grammaire Java en BNF, une comparaison Java C++, une description du langage HTML et une présentation de JavaScript.

Trois ans plus tard...

Ecrire une deuxième version d'un livre en informatique peut relever du cauchemar... Trois ans après avoir écrit la première version de l'ouvrage que vous tenez dans les mains, nous avons été heureux de constater que l'objet de notre enthousiasme des premiers jours (en janvier 96) pour ce qui était alors une sorte de révolution est devenu plus mature, plus solide. Les qualités techniques propres à Java ont été reconnues par des centaines de milliers de développeurs, la plupart des universités et autres institutions enseignant la programmation l'ont adoptés comme langage de support aux cours d'informatique. Mais surtout, ce sont les packages définissant les services de base offerts aux programmeurs qui se sont étoffés. De sept, ils sont passés à plus de cinquante dans la version 1.2 de Java, qui a été renommée officiellement version 2 par Sun. Cela ne tient pas compte des packages disponibles pour traiter du commerce électronique, de la manipulation des images en trois dimensions, etc., qui fournissent un cadre solide pour produire rapidement des applications complexes.

Mais au delà des qualités conceptuelles du langage et de la richesse de ses packages, Java a su émuler des efforts conséquents autour de lui dans les domaines annexes suivants :

- les **machines virtuelles** Java ont été considérablement améliorées et les différences enregistrées entre une exécution en code natif et l'utilisation d'une machine virtuelle exécutant une conversion à la volée (*Just In Time*) deviennent de plus en plus minces. Cela a largement contribué à faire fondre les résistances existant chez certains développeurs par rapport à l'utilisation d'un code interprété;
- les éditeurs de logiciel ont produit des **environnements de programmation** autour de Java qui n'ont rien à envier à d'autres langages tels que

Smalltalk, C++, etc. Ces environnements ont pour premier objectif de faciliter la recherche d'objets et de méthodes aidant à la résolution d'un problème. Comme nous le constaterons dans la suite de ce livre, la syntaxe et les concepts de Java sont relativement simples. Par contre, la richesse de ses packages demande un investissement certain. On ne peut devenir un bon développeur Java sans les connaître, ils sont incontournables;

- Java, par un effet de rebond, après avoir été « Applet » est devenu « Servlet »: la problématique initiale était dans le contexte statique du Web, de pouvoir exécuter du code sur un poste client. La mobilité du code Java soutenue par le slogan « *Write once, run everywhere* » (Écris-le une fois, exécute-le partout) était indispensable pour gommer la variété des postes clients connectés sur le Web. Mais il existe une variété identique du coté des serveurs: les mêmes arguments d'économie de réécriture ont séduit les développeurs d'**applications sur serveur**. Du coup, Java a quitté son image de gadget pour animer les pages Web;
- Java s'est enfoui **au cœur du silicium**, dans des processeurs spécialisés, dans des cartes à puces, dans une bague (JavaRing). Cette « cristallisation » de la machine virtuelle Java s'est accompagnée de l'écriture d'un système d'exploitation, le Java OS, et de la publication d'une norme d'interconnexion de composants (JINI). Cette incrustation de la technique au cœur d'objet quotidien est à nos yeux le réel enjeu de Java et des techniques de la programmation distribuée utilisant du code mobile. En effet, même si vous possédez un ordinateur, vous possédez également une quantité d'autres objets attendant un ajout « d'intelligence » et une connexion au reste du monde. L'incorporation dans le quotidien des processeurs Java et à l'image de celle des moteurs électriques si bien intégrés dans les objets que l'on a oublié leur existence;
- Java, à travers l'*Entreprise Java Beans* (EJB), devient le cadre conceptuel pour la production d'applications à partir de composants. Ici, se dévoile l'ultime « Graal » de l'informatique. Non seulement, il est possible d'utiliser le code Java universellement, mais en plus il serait possible de le **réutiliser**. Avec les Beans, il est possible de développer des petits composants comme des boutons ou des champs de saisie qui sont assemblés dans une interface graphique; les messages émis sont connectés aux actions propres à l'application. Cela constitue le degré zéro de la réutilisation. L'intérêt de la réutilisation croît avec la taille des composants. Avec EJB, il s'agit de créer des « gros » composants. Si pour créer un magasin virtuel, on trouve les composants: gestion des

clients, gestion des fournisseurs, gestion des commandes, gestion des expéditions, gestion des paiements par carte, gestion des catalogues alors on peux effectivement assembler et paramétrer les règles de gestion propres à son entreprise. Le projet San Fransisco d'IBM est un prototype de ce que peut devenir cette orientation. EJB sera sans doute le modèle adopté dans l'architecture CORBA par les sociétés actives dans le domaine des connecticiels.

Mais écrire une nouvelle version d'un livre n'est pas seulement constater que l'on a choisi un bon sujet. C'est pour Java 2 :

- réviser la syntaxe du langage (classe interne, nouveaux usages des modificateurs, etc.);
- adapter les exemples aux nouvelles API;
- utiliser la notation UML (*Unified Modeling Language*) pour les graphes de classes;
- décrire le nouveau modèle d'événements « par délégation »;
- décrire l'usage des classes collections (nouvelles classes de java.util) pour créer des structures de données;
- décrire l'utilisation des fichiers archives (jars);
- introduire les nouveaux composants d'interface SWING;
- décrire les composants logiciels JavaBeans;
- décrire les mécanismes d'introspection, de sérialisation, etc.;
- ajouter un exemple complet d'interaction avec une base de données en utilisant JDBC;
- ajouter la création d'application répartie avec RMI *(Remote Method Invocation*);
- introduire l'utilisation de Java dans une architecture CORBA.

Nous avons conservé pour ces ajouts le style concis et conceptuel de la première version. Nous avons éliminé les annexes sur l'API. En effet, l'explosion du nombre de packages, de classes et de méthodes de Java 2 rendait impossible l'impression papier de tels index. La documentation comprimée de Java 2 tient dans seize millions de caractères, notre livre dans à peine un: il vous faut donc considérer ce livre comme une synthèse des connaissances nécessaires et utiles et vous référer constamment à la documentation pour les détails.

Conventions utilisées dans cet ouvrage

Le texte est en police Sabon.

Un mot important est écrit **ainsi**.

Dans le texte, un mot-clé, une classe ou une méthode de Java est écrit *ainsi*.

Le code Java est en police Courrier :

```
// ceci est un commentaire dans une ligne de code.
```

Partie I

L'esprit Java

L'accès à ces différents concepts, le sujet, la finalité et l'infini, se fait manifestement à travers celui de conscience. Voilà donc, au regard de toute philosophie qui se développe une méfiance vis-à-vis de l'image informatique de notre esprit, la clef de l'irréductible écart!

Gérard Chazal, *Le miroir automate*.

Chapitre 1
Avant Java

La partie la plus saillante d'Internet est le World Wide Web (WWW ou plus simplement Web). Celui-ci matérialise le métissage de deux concepts : celui des réseaux de communication interconnectés (Internet) et celui d'hypertexte. Chacun de ces concepts a une origine et des objectifs qui lui sont propres, tandis que leur union ouvre des perspectives absolument novatrices et impensables voilà quelques années encore (le premier « outil Web » en mode texte a vu le jour en 1991 au CERN).

Nous allons maintenant définir quelques concepts afin de préparer la transition vers Java... et de clarifier notre discours.

1.1 Interconnexion de réseaux

L'interconnexion de réseaux a pour origine l'Arpanet (*Defense Advanced Research Project Agency Network*), un réseau mis en place à partir de 1966 aux États-Unis. La notion d'interconnexion de réseaux permet de connecter des réseaux informatiques basés sur des protocoles différents. Pour y parvenir, il est nécessaire de définir un protocole d'interconnexion commun au-dessus du protocole de gestion de chaque réseau (voir figure 1.1). L'*Internet Protocol* (IP) fournit ce service, définissant des adresses uniques pour un réseau et une machine hôte. Le protocole IP assume deux fonctions principales :

- le routage d'un paquet à travers des réseaux successifs, depuis la machine source jusqu'à celle du destinataire, identifiée par son adresse IP. L'adresse peut ainsi être celle de la prochaine passerelle ou celle du destinataire final ;

- le découpage des flux d'information initiaux en paquets de taille standardisée et leur réassemblage ultérieur.

Les réseaux empruntés lors d'une communication assurent le transfert des paquets IP. Ceux-ci sont encapsulés (au départ et en quittant une passerelle IP) par le protocole du réseau qui les transmet; ils sont ensuite décapsulés lorsqu'ils quittent ce réseau (en entrant dans une passerelle et en arrivant à destination).

Le succès de ce protocole est principalement dû à la relative simplicité de réalisation des passerelles IP. La disponibilité de telles passerelles sur de nombreux types de machines a favorisé l'utilisation de IP dans des environnements hétérogènes.

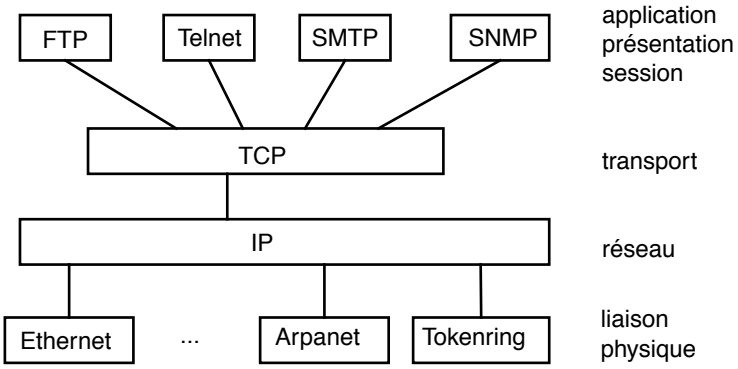


Figure 1.1 Couches logiques selon le modèle OSI

Principaux protocoles et services Internet

TCP/IP (*Transmission Control Protocol*) est la couche de transport au-dessus de IP assurant la connexion de bout à bout entre deux hôtes. TCP/IP fournit un service de circuit virtuel et assure l'acquittement des paquets ainsi que la détection des erreurs. En utilisant TCP/IP, deux hôtes peuvent échanger des paquets d'information en faisant abstraction des réseaux qui les interconnectent.

Au-dessus de TCP/IP, différents services ont été développés, afin d'offrir une homogénéité entre applications de même nature (par exemple entre applications de courrier électronique). Parmi ces services on peut citer :

FTP (*File Transfer Protocol*) permet d'échanger des fichiers entre deux machines connectées à Internet. De vastes collections de fichiers, de documents, de programmes et d'images sont disponibles dans des archives FTP. Pour éviter des connexions trop lointaines, des « sites miroirs » ont été mis en place au niveau des principaux acteurs d'Internet (nations, universités, etc.), faisant office d'archives locales. Ainsi, les utilisateurs suisses peuvent télécharger de nombreux programmes en provenance des États-Unis ou du Japon depuis un site miroir situé à Zürich.

De plus, des serveurs spécialisés dans le catalogage des documents situés sur différents sites FTP ont été développés. De tels serveurs (comme ARCHIE, par exemple) comportent une base de données permettant l'indexation d'un nombre considérable de documents.

Telnet (*Terminal Protocol*) est un protocole d'accès à un hôte en mode « terminal ». Il permet à un utilisateur de se connecter à distance à des serveurs de calcul, des banques de données ou d'autres services hébergés par des ordinateurs centraux.

NNTP (*Network News Transfer Protocol*) permet la constitution de groupes de communication (*newsgroups*) organisés autour de thèmes (loisirs, politique, culture, etc.). Une architecture hiérarchique de sites miroirs a également été mise en place afin de limiter le volume d'informations à distribuer.

SMTP (*Simple Mail Transfer Protocol*) définit quant à lui un service de base en matière de courrier électronique.

SNMP (*Simple Network Management Protocol*) est un protocole chargé de la gestion du réseau. Il est généralement utilisé pour la communication avec les systèmes électroniques tels que passerelles, routeurs, multiplexeurs, etc.

Format des adresses IP

Revenons aux adresses IP : une adresse est constituée de quatre octets dont les valeurs sont séparées par des points (exemple : 136.102.233.49). Les R premiers octets (avec R = 1, 2 ou 3) sont consacrés à l'adresse du réseau et les H suivants (avec H = 3, 2 ou 1) à celle de la machine hôte. Le rapport R/H définit trois classes d'adresses : A, B et C (voir figure 1.2), chacune permettant d'identifier un certain nombre de réseaux et de machines. La classe d'une adresse est codée dans les deux premiers bits de celle-ci.

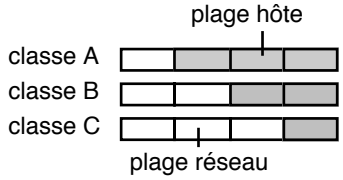


Figure 1.2 Format d'une adresse IP

Attribution des adresses IP

L'*InterNIC Register* est un organisme chargé de distribuer les plages d'adresse des différents réseaux au niveau international. La hiérarchie est généralement la suivante (voir figure 1.3) :

- zone géographique (pays) ;

- organisation, entreprise;
- département;
- hôte.

Des serveurs de noms établissent un niveau logique au-dessus du niveau physique des adresses, afin de supporter des réorganisations au niveau des réseaux et des machines. Ainsi, *cui.unige.ch* et *cuiwww.unige.ch* sont deux adresses logiques d'un même hôte du Centre universitaire informatique de l'université de Genève, dont l'adresse IP est 129.194.70.1.

Dans le cas du courrier électronique, le format généralement utilisé est la concaténation d'un nom d'utilisateur et de l'adresse IP d'un hôte hébergeant un serveur SMTP : *johnny@rock.musique.fr* pourrait correspondre à l'adresse de qui vous savez.

Il n'est pas nécessaire de démontrer l'importance prise en quelques années par Internet et le Web.

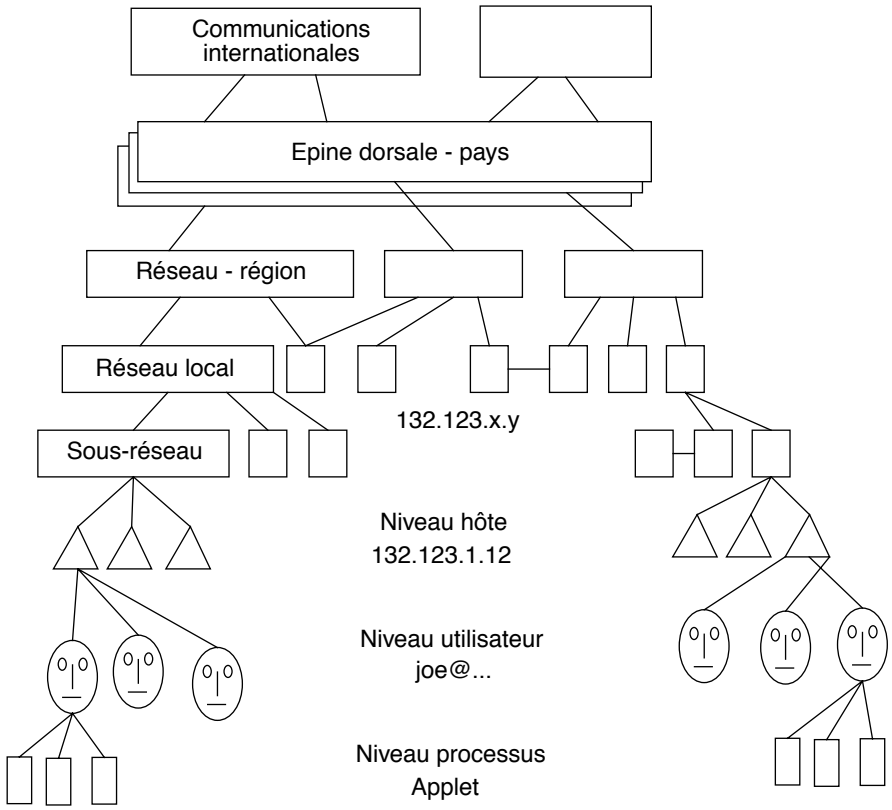


Figure 1.3 Interconnectivité sur le réseau Internet

1.2 Hypertextes

L'idée initiale de « liens associatifs » entre des données et des documents d'origines diverses est attribuée à Douglas Engelbart, dans les années 50. Alors directeur de l'ARC (*Augmentation Research Center of Stanford Research Institute*), D. Engelbart avait pour but de créer des logiciels capables de supporter le travail coopératif et de faciliter la communication entre les utilisateurs. On notera au passage que ce même programme de recherche étudiait également les écrans multifenêtres ou encore l'utilisation de la souris pour manipuler des complexes informationnels.

Le lien associatif est constitué d'une référence (la cible) et d'une ancre (la source). La référence et l'ancre résident normalement dans des nœuds informationnels (textes, images, voire sons) différents. Il est ainsi possible de tisser une véritable toile représentant les liens sémantiques entre différentes informations. L'utilisateur peut alors choisir son chemin dans ce réseau selon ses intérêts et ses objectifs, « naviguant » entre les nœuds en suivant les liens (voir figure 1.1).

Des outils tels que HyperCard sur Macintosh ou les systèmes de documentation et d'aide en ligne des logiciels Excel, Oracle ou FrameMaker nous ont familiarisés avec cette approche navigationnelle où l'on doit « cliquer » pour passer d'une explication à une autre, suivant les besoins.

Plus récemment, de nombreux CD-ROM ont été réalisés selon ce paradigme, utilisant des liens hypertextuels pour présenter un sujet en proposant de nombreux itinéraires de parcours. Le contenu des nœuds est désormais étendu à d'autres types de données tels que : image, graphique, vidéo, son, etc., faisant évoluer le principe de l'hypertexte vers celui de l'hypermédia.

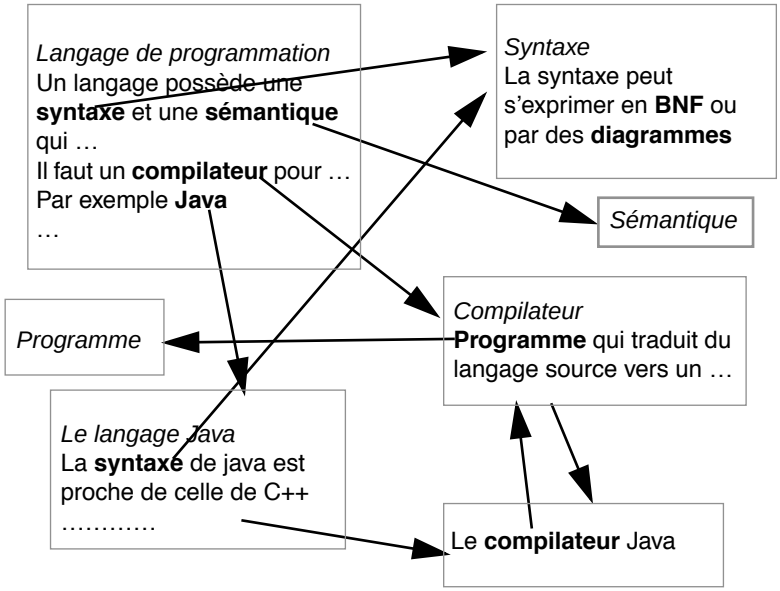
1.3 Web

Imaginez un instant que vous ayez à disposition un hypertexte et que les nœuds de celui-ci soient répartis sur les différents hôtes d'Internet, des liens hypertextuels permettant la navigation d'un nœud à l'autre, d'un hôte à l'autre. Le contenu des nœuds serait varié, constitué aussi bien d'images, de textes, d'animations ou de sons.

Ne rêvez plus, vous ne serez pas l'inventeur de ce système! En effet, le Web (World-Wide Web [3]) correspond trait pour trait à la description ci-dessus. Restent encore à définir quelques notions de manière plus précise.

Le Web peut être vu comme le **métissage** de deux toiles (*Web* en anglais) :

- une toile **physique**, constituée des réseaux interconnectés formant Internet;
- une toile **sémantique**, constituée des réseaux d'hyperliens.



Les ancres sont indiquées en **gras**

Figure 1.1 Structure d'hypertexte

Ce métissage entre liens physiques et liens logiques n'aurait certainement pas suffit à lui seul à déclencher le véritable ouragan Web que l'on connaît aujourd'hui. C'est sans aucun doute le développement de logiciels de navigation (browsers, butineurs) très simples à utiliser et disponibles (souvent gratuitement) sur les plateformes les plus répandues (PC, Macintosh, puis Unix) qui a véritablement « mis le feu » et attiré le grand public.

Les butineurs les plus répandus (Mosaic, Navigator, MacWeb, WinWeb, Explorer, HotJava, etc.) se basent tous sur des principes d'interface assez simples que l'on peut résumer ainsi :

- chaque fenêtre affiche le contenu d'un nœud d'information (texte formaté, images, etc.);
- des « zones sensibles » sont placées dans le contenu du nœud, signalées de manière visuelle (exemple : texte souligné en rouge ou image clignotante);
- ces zones sensibles sont les ancres de liens hypertextuels (voir point 1.2). Lorsque l'utilisateur active une ancre (par un clic de la souris), le nœud référencé par le lien est téléchargé depuis la machine qui l'héberge vers la machine de l'utilisateur, cela à travers le réseau;
- le contenu de ce nœud est ensuite affiché dans la fenêtre, matérialisant ainsi le nouvel « emplacement » de l'utilisateur dans l'espace Web.

Il faut encore préciser que les butineurs ont un rôle « passif » en ce qui concerne la présentation des informations : les indications de formatage, de définition des ancres ou d'autres facilités d'interaction sont entièrement déterminées dans le nœud. Cette particularité a permis le développement de butineurs compatibles entre eux, ne différant que par les fonctionnalités supplémentaires qu'ils offrent aux utilisateurs. Certaines de ces fonctionnalités sont destinées à assister le cybervoyageur dans son périple à travers le Web :

- édition de *bookmarks*, listes de nœuds regroupés par l'utilisateur selon ses propres thèmes d'intérêt;
- gestion de l'historique des nœuds visités;
- gestion d'un cache (stockage local d'éléments fréquemment rencontrés : icônes, logos, afin d'éviter leur téléchargement systématique).

Les points suivants présentent trois éléments importants du Web, à savoir :

- HTML : le langage de description du contenu des nœuds d'information;
- URL : le format de nommage des nœuds;
- HTTP : le protocole d'interaction entre un client (le butineur) et un serveur (l'hôte qui héberge le nœud désiré).

HTML

HTML (*HyperText Markup Language*) est un langage de description du contenu et de la structure d'un nœud d'information. HTML est lui-même dérivé d'un ensemble de styles et de descriptions issu d'une norme appelée SGML (*Standard Generalized Markup Language*).

Pour simplifier notre discours, admettons que le nœud d'information sur lequel nous allons travailler soit un **document**. Nous pouvons alors décrire :

- des éléments de structure du document : son titre, ses paragraphes, sous-paragraphes, listes numérotées, images;
- des éléments de formatage du contenu : texte en italique, tabulations.

HTML permet d'exprimer ces différentes règles de structuration et de formatage d'une manière indépendante de la machine et du logiciel utilisés pour afficher le document. À cet effet, HTML définit des délimiteurs (ou balises) que l'on introduit dans le contenu d'un document et qui seront détectés et interprétés par le logiciel d'affichage.

L'exemple suivant montre les possibilités de base de HTML :

```
<TITLE>Jump in the Web</TITLE>
<H1>Once Upon a Time ...</H1>
Welcome to the world of HTML
this is the first paragraph.<P>
this is the <B>second</B> and <I>some URL</I><P>
<A HREF="http://www.oracle.com"> at Oracle </A><P>
```

```
<A HREF="http://cuiwww.unige.ch"> at SQL7 </A><P>
<A HREF="http://mistral.culture.fr/louvre/"> Le Louvre </A><P>
<IMG SRC="smile.gif">
```

La figure 1.1 ci-dessous reproduit notre document tel qu'il apparaît sur un Macintosh, en utilisant le butineur Mosaic. La tâche du butineur consiste à charger le code HTML du document à travers le réseau et à en exécuter les commandes afin d'afficher le document formaté.

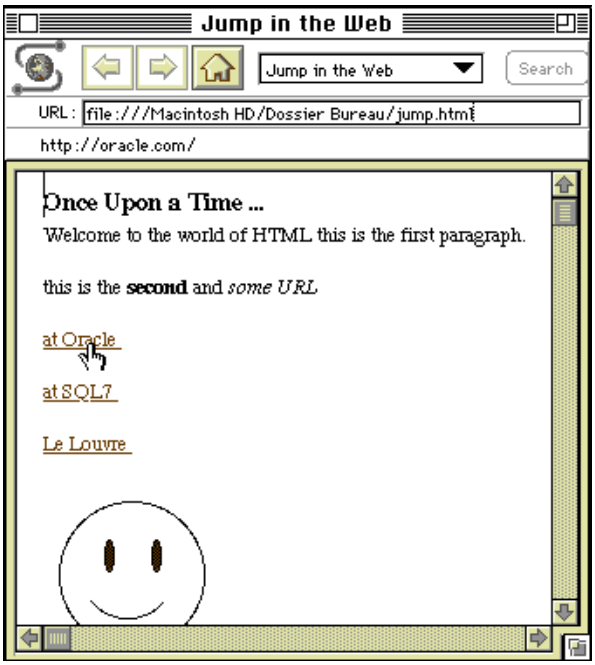


Figure 1.1 Affichage d'un document HTML

Les délimiteurs (*tags* en anglais) sont entourés de crochets <> et vont en général par paire : <*XXX*> et </*XXX*>. Découvrons certains d'entre eux en étudiant le code HTML correspondant au document ci-dessus :

```
<TITLE>Jump in the Web</TITLE>
```

donne un titre au document.

```
<H1>Once Upon a Time ...</H1>
```

donne un en-tête à une partie du document (niveaux 1 à 6).

```
Welcome to the world of HTML this is the first paragraph.<P>
```

indique la fin d'un paragraphe.

```
this is the <B>second</B> and <I>some URL</I><P>
```

définit une style (dans ce cas : gras et italique).

```
<A HREF="http://www.oracle.com"> at Oracle </A><P>
```

définit une référence et son ancre. C'est ainsi que l'on déclare des liens hypertextes.

```
<IMG SRC="smile.gif">
```
inclut une image.

Nous consacrons l'annexe D à la définition des principaux délimiteurs de HTML. L'intérêt de HTML pour un programmeur Java réside dans le fait qu'un programme Java peut être invoqué depuis le contenu d'un document HTML. On parle alors d'*applet* Java (voir chapitre 10, page 123).

URL

Un URL (*Uniform Resource Locator*) permet de localiser de manière unique une ressource (un nœud d'information) sur Internet. Il identifie à la fois :

- le protocole à utiliser pour charger la ressource (*file, ftp, http*, etc.);
- la machine hôte (adresse IP, éventuellement avec un numéro de port);
- le chemin d'accès à la ressource (arborescence du système de fichiers);
- le nom de la ressource (nom de fichier).

Le format complet est le suivant :

```
protocole://hôte.organisation.domaine[:port]/chemin/nom
```

Ce format respecte les conventions de nommage mentionnées au point 1.1.

L'URL peut encore être étendu afin d'indiquer le nom d'une sous-référence interne au nœud, à l'aide du suffixe (*#sous-référence*), comme dans l'exemple suivant :

```
http://cuicui.unige.ch:9876/db/java/livre/preface.html#merci
```

Cela permet une navigation à l'intérieur d'un même nœud, comme par exemple des renvois entre paragraphes d'un même document. Nous verrons qu'un URL peut également servir à démarrer une application et à lui fournir des paramètres, permettant ainsi d'interroger une base de données ou d'interagir avec un système complexe à distance.

HTTP

HTTP (*HyperText Transfer Protocol*) est un protocole destiné aux échanges entre un client Web (butineur) et un serveur (appelé « serveur HTTP » ou « serveur Web ») hébergeant des nœuds d'information. Le serveur HTTP et l'ensemble des nœuds forment ce que l'on appelle communément un « site Web ».

Fonctionnement du serveur

Le serveur HTTP réceptionne les requêtes des clients (exprimées à l'aide d'URL) et leur retourne le contenu des nœuds référencés. Il sert également de passerelle (*gateway*) vers des applications invoquées par le client. De plus, il gère les droits d'accès à certains nœuds et récolte des statistiques sur l'utilisation des services (nombre de requêtes, provenance des clients ou nœuds les plus demandés).

L'offre en matière de serveurs HTTP est assez importante, des systèmes ayant été développés pour les principales plateformes : Unix, Windows, Macintosh. La mise en œuvre d'un serveur HTTP est relativement simple. Néanmoins, il faut distinguer les deux cas suivants, très différents en matière de sécurité, de charge (nombre d'accès simultanés)... et de coût de licence :

- installation intranet (locale), accessible uniquement depuis les machines du réseau interne à l'organisation ou à l'entreprise;
- installation Internet, accessible (potentiellement du moins) depuis n'importe quelle machine connectée à Internet.

Ainsi, le coût d'un serveur HTTP varie fortement selon la configuration, de quelques centaines de francs français pour un serveur élémentaire sur PC à plusieurs centaines de milliers de francs pour une installation Internet sur une machine UNIX multiprocesseurs.

1.4 Perdu dans le cyberespace

La plupart des entreprises ont mis en place un ou plusieurs sites Web (un investissement minimal pour une visibilité maximale était le slogan dans les premiers temps du Web!). Cette abondance peut rapidement décourager un cybernaute : « Tout est là, mais comment trouver ce dont j'ai vraiment besoin? » Deux attitudes non exclusives sont alors possibles :

- utiliser les bases de données d'indexation (Yahoo, AltaVista, etc.) qui, à partir de mots clés spécifiés par l'utilisateur, retournent les URL de nœuds répondant aux critères de recherche;
- surfer sur le Web en se promenant au gré des hyperliens. En profiter au passage pour noter les nœuds intéressants et pour organiser ainsi son propre catalogue d'URL (*bookmark*).

Comme vous pouvez le constater, ces démarches ne sont pas très éloignées de la consultation d'un dictionnaire ou d'une encyclopédie « papier ».

1.5 Création des nœuds

Les documents servant de nœuds à l'hypertexte peuvent être créés à l'aide de traitement de texte (prendre garde à sauvegarder les documents en format ASCII), les délimiteurs étant insérés à la main ou au moyen de macros (séquences programmées). Des logiciels spécialisés dans l'édition de documents HTML ont vu le jour et facilitent grandement l'édition de documents très « garnis ».

La génération de documents HTML à partir d'autres formats est très intéressante dans le cas où de gros volumes d'information (documentation, livre, catalogue) ont déjà été structurés et mis en forme dans un traitement de texte. Des

convertisseurs existent pour traduire en HTML des documents issus de Word (format RTF), FrameMaker (format MIF), LaTeX, etc.

Dans le cas des images, de nombreux utilitaires permettent de convertir les fichiers issus de logiciels de dessin (CorelDraw, Canvas, Painter, etc.) au format GIF, très répandu sur le Web.

1.6 Architecture client-serveur

Le client-serveur, le paradigme de ces dernières années, visait à l'élaboration d'applications dont la tâche se répartit au moment de l'exécution entre le client et le serveur. Le schéma (célèbre) du Gartner Group établissait une typologie en fonction des types d'activités exécutées et de leur répartition sur le client et le serveur (voir tableau 1.1).

Dans cette table, nous avons ajouté une étoile (*) lorsqu'il n'existe pas, à nos yeux, une grande classe représentative de ce type. Le développement en client-serveur s'est principalement bâti autour des types 2-3-4.

Type	Activité sur le serveur	Activité sur le client	Description
0	Données Traitements Présentation		Architecture centralisée avec des terminaux.
1	Données Traitements Présentation	Présentation	Le client n'est utilisé que pour la présentation (par exemple avec X11).
2	Données Traitements	Présentation	Le client est un serveur de présentation intégrant le contrôle de la logique de l'affichage.
3	Données Traitements	Traitements Présentation	Les traitements sont répartis entre le client et le serveur (par exemple les procédures PL/SQL d'Oracle s'exécutant sur le client, dans les *forms* et dans le SGBD serveur).
4	Données	Traitements Présentation	Le serveur est un pur serveur de données (une base de données sans procédures stockées).
5	Données	Données Traitements Présentation	Excel + ODBC* EIS (*Executive Information System*) + BD locale*.

Type	Activité sur le serveur	Activité sur le client	Description
6		Données Traitements Présentation	Un poste de travail (PC, Mac) isolé.

Tableau 1.1 Typologie client-serveur et répartition des activités

Les principaux problèmes rencontrés dans la mise en œuvre du client-serveur sont :

- **la difficulté de répartition de l'activité entre le serveur et le client;** cette répartition est difficile à effectuer *a priori*, l'idéal étant qu'elle soit vue comme une optimisation des performances. Cela n'est possible que si le langage de programmation est identique sur le serveur et sur le client (PL/SQL par exemple) ou si l'ensemble de l'application est obtenu par génération (avec NSDK de Nat system, par exemple). L'utilisation d'outils tels que PowerBuilder ou SQL-Windows introduisent une rupture de langage entre le serveur et le client qui peut rendre cette tâche ardue et/ou coûteuse en temps de reprogrammation;
- **le coût de distribution des logiciels (versions) et la complexité de la gestion des configurations**; généralement, seul le code exécutable du logiciel est à distribuer sur le client (*run-time*) ainsi que les gestionnaires de réseaux. Cette tâche peut devenir un cauchemar qui est fonction de nombreux paramètres :
  - N : nombre de postes clients;
  - A : nombre de versions actives du SGBD;
  - B : nombre de versions actives du gestionnaire d'écran;
  - C : nombre de versions actives du système d'exploitation client;
  - D : nombre de protocoles réseaux utilisés.

  Admettons que la complexité soit égale au produit de ces variables. En début de projet, on estime les valeurs A, B, C, D à 1. La complexité est donc celle du nombre de postes clients. Mais après quelques mois ou années, A, B, C, D ont généralement pris des valeurs entre 1 et 4... D'où une augmentation de la complexité que nous vous laissons calculer;
- **la faible portabilité**; nous n'avons rencontré aucun générateur d'applications ne nécessitant pas de retravailler les documents générés, bien qu'ils se déclarent tous multicibles. Il est donc difficile de faire cohabiter des communautés d'utilisateurs ne partageant pas les mêmes types de postes de travail.

Le client-serveur représente à nos yeux une étape transitoire entre l'architecture « terminaux en étoile autour d'un ordinateur central » et celle du « client universel et du serveur universel baignant dans l'intranet/Internet ». Cette étape aura essentiellement permis d'installer les réseaux dans les entreprises.

1.7 Langages de programmation orientés objet

L'évolution des langages de programmation depuis les années 50 a été marquée par quelques grandes étapes. Celles-ci correspondent à l'intégration successive de nouvelles idées issues du domaine du génie logiciel dans les langages. Elles ont permis à chaque fois d'étendre le champ d'application de l'informatique.

Langages évolués

L'introduction du langage FORTRAN dans le domaine scientifique et du COBOL pour les applications de gestion marque l'arrivée des langages évolués. Ces langages permettent au programmeur de s'exprimer dans des termes plus proches de son propre langage, et non plus à l'aide de codes correspondant aux diverses instructions exécutables par la machine utilisée. Le COBOL introduit également la structuration hiérarchique des données.

Langages structurés

L'élargissement du champ d'application de l'informatique conduit à développer des programmes de plus en plus grands et de plus en plus complexes à gérer. La programmation structurée introduit un ensemble d'instructions de contrôle (*si ... alors ... sinon*, *répéter ... jusqu'à*, etc.) permettant d'exprimer un algorithme en les combinant entre elles et surtout d'éliminer les instructions de saut « sauvages » (*goto*), jusqu'alors omniprésentes. Chaque bloc d'instructions possède désormais un seul point d'entrée et un seul point de sortie, ce qui facilite grandement la lecture et la compréhension des programmes.

Le typage strict des données est également introduit, permettant de détecter dès la compilation des erreurs graves pouvant conduire à la corruption de la mémoire.

Algol, puis Pascal et C ont grandement contribué à la diffusion de la programmation structurée.

La méthode de développement sous-jacente à la programmation structurée est la décomposition des problèmes en sous-problèmes par raffinements successifs, appelée aussi approche descendante (*top-down*).

Langages modulaires

Toujours dans le but de maîtriser des logiciels dont la taille devient considérable (plusieurs millions de lignes d'instructions), des langages comme Modula puis

Ada mettent en avant le concept de programmation modulaire. Un **module** regroupe des sous-programmes et des données qui ont un but commun, que ce soit la gestion d'un type de données, d'un périphérique, d'une fonction complexe du système, etc. L'idée de base de cette approche est de construire les programmes par assemblage de modules.

Langages à objets : de Simula à Java

Les langages à objets proposent de voir un système informatique comme un ensemble d'**objets** qui réalisent des activités et communiquent entre eux par envoi de **messages**. Les objets d'un même type sont regroupés dans des **classes** qui définissent leur structure et leur comportement. La réutilisation de classes existantes est facilitée par des possibilités d'adaptation, comme la notion de **sous-classe**.

L'ancêtre des langages à objets est Simula-67, créé en 1967, sans doute un peu trop tôt pour que le commun des programmeurs en saisisse tout l'intérêt. Le second membre éminent de la famille est Smalltalk-80 (1980!). Ce dernier, véritable « puriste », pousse à l'extrême la programmation objet puisque tout y est objet, même les blocs d'instructions d'un programme. Le langage est très dynamique, le compilateur ne fait pas de vérification de types; ce n'est qu'au moment où un objet reçoit un message, à l'exécution, qu'il décide si oui ou non il peut y répondre et comment. Smalltalk est indissociable de son environnement de développement qui comprend une riche bibliothèque de classes prédéfinies et des outils graphiques d'édition, de débogage et de butinage.

Parallèlement, mais démarrant un peu plus tard, plusieurs langages à objets ont été développés à partir de langages existants : Object Pascal, défini par Apple pour la programmation du Macintosh ou Objective C (une extension du C basée sur le principe de Smalltalk de typage dynamique), utilisé dans l'environnement NEXTSTEP de NeXT.

Parmi les langages à objet, c'est le langage C++ qui a incontestablement rencontré le plus large succès. Il s'est répandu chez les développeurs grâce, entre autres, à sa compatibilité avec le langage C. Cette contrainte de compatibilité avec un langage non-objet, fixée dès le départ, a été magistralement surmontée par l'architecte principal du langage (B. Stroustrup [11]). Il n'en reste pas moins que C++ est un langage complexe où le paradigme objet n'est pas central.

C'est dans ce contexte que Java apparaît en 1995 sur Internet. Si les éléments de base de la syntaxe Java sont proches de ceux de C++ (donc de C), il n'en va pas de même au niveau sémantique. De ce point de vue, Java prend une orientation plus radicalement objet qui le rapproche de Smalltalk ou d'Eiffel. Il se libère de l'usage explicite des pointeurs encore imposé par C++ et interdit certaines définitions à la sémantique très complexe, comme la redéfinition d'opérateurs ou

l'héritage multiple. La simplicité de Java et le large support dont il jouit déjà pourrait bien en faire le Pascal de la fin du siècle, c'est-à-dire le langage qui amène la programmation objet non seulement chez les développeurs professionnels mais aussi chez tous ceux qui veulent s'initier à la programmation sans nécessairement en faire leur profession.

Chapitre 2
Révolution an 4 : Java

Java n'est plus radicalement nouveau, il s'inscrit dans la continuité de l'évolution des langages. Par contre, ses capacités concernant la distribution d'applications sur Internet en font un outil révolutionnaire. En quatre ans, il aura aussi affirmé son ambition de remplacer C++.

2.1 Java, c'est quoi?

Développé par Sun Microsystems, Java est un langage de programmation orienté objet, adapté à la distribution d'applications sur Internet et s'intégrant au Web. Les caractéristiques du langage et leur adéquation à Internet et au Web forment des couples indissociables. Commençons par examiner les caractéristiques de ce nouveau langage.

2.2 Caractéristiques principales

Java est orienté objet

Il appartient logiquement à la mouvance des langages de programmation orientés objet (voir point 1.7). Le paradigme **objet** est *a priori* simple à expliquer :

- les objets (instances) sont issus de moules (classes);
- ils communiquent entre eux à l'aide de messages;
- les messages sont évalués par les méthodes de l'objet, induisant des modifications de son état ou de son comportement;
- les objets vivent en famille, ils héritent du comportement de leurs aînés (héritage entre classes) et spécialisent ce comportement.

Cette orientation objet n'est pas en soi la qualité principale de Java (il existe des centaines de langages à objets ayant ces propriétés!). Les qualités de Java résident dans l'assainissement effectué par ses concepteurs au niveau de l'implantation des concepts du langage. Ces choix d'implantation ont été guidés dans Java par des objectifs de sécurité, de sûreté de programmation, de portabilité, de distribution et de performance.

Java supprime l'utilisation des pointeurs

Lors de la création d'un objet (méthode *new*), un objet est désigné par une **référence**. Dans les langages tels que C ou C++, les objets sont plus naturellement désignés par des **pointeurs**. De plus, des opérations arithmétiques (addition, comparaison, etc.) sont autorisées sur ces pointeurs. Cette pratique, considérée comme indispensable par certains programmeurs, est une source très fréquente d'erreurs.

En Java, la seule opération possible sur une référence est sa copie. Java interdit donc toutes les manipulations arithmétiques sur les désignateurs d'objets, éliminant ainsi de nombreux risques d'erreurs... voire de virus.

Java supporte les tableaux

En Java, les tableaux ne sont pas représentés uniquement par une suite d'emplacements mémoire référencés par un pointeur. Ils prennent la forme d'une structure, munie d'une borne inférieure et d'une borne supérieure. Lors de l'accès à l'un des éléments du tableau à l'aide d'un indice, l'appartenance de cet indice à l'intervalle spécifié par les bornes est vérifiée. Cette vérification systématique peut sembler une évidence, puisqu'elle permet de détecter bien des erreurs de programmation. Néanmoins, de nombreux langages comme C ou C++ ne l'appliquent pas.

Java supporte et réclame un typage fort

Java supprime les conversions automatiques de types, aucune conversion n'étant effectuée implicitement. Dès lors, toutes les conversions doivent être déclarées par le programmeur de manière explicite.

L'absence de pointeurs autorise une vérification accrue des types d'objets par le compilateur. En effet, aucun pointeur ne pouvant être « forgé » sur un objet, toutes les opérations sur celui-ci (copie de référence, passage de paramètres) sont explicitées dans le code du programme, ce qui permet de vérifier la cohérence des types.

Java gère automatiquement la mémoire

L'absence de pointeurs permet également d'établir pendant l'exécution une liste des objets encore utilisés (ceux dont les références sont encore atteignables par

les processus en cours) et une liste de ceux qui ne le sont plus. Ces derniers, désormais inutiles, peuvent donc être éliminés de la mémoire par un processus spécialisé (le *garbage collector* ou plus poétiquement *ramasse-miettes*).

Afin d'éviter l'émiettement de la mémoire, on peut déplacer des objets dans la mémoire, puisque toutes les références à un objet sont connues. Java utilise à cet effet des algorithmes performants de nettoyage et de compactage de la mémoire, exécutés parallèlement au déroulement normal du programme.

Le prix à payer pour cette optimisation de la mémoire est de quelques pour cent sur la performance générale du programme (dans les applications critiques, le programmeur peut désactiver et réactiver le mécanisme à volonté). Néanmoins, les bénéfices sont énormes : le programmeur n'a plus besoin de désallouer des objets devenus inutilisables; mais surtout le risque d'erreurs portant sur des objets encore référencés mais déjà supprimés est éliminé. Ainsi, nous ne devrions plus entendre :

> « Je croyais ne plus avoir besoin de cet objet et je l'ai supprimé, mais j'avais oublié qu'il était référencé par cet autre objet, vous comprenez, c'est tellement complexe, une erreur est vite arrivée et le compilateur n'a rien dit. »

Notons enfin que de telles erreurs, responsables d'un grand nombre de « crash » de programmes, sont extrêmement difficiles à détecter car elles ne sont pas systématiquement reproductibles.

Java remplace l'héritage multiple par la notion d'interface

L'héritage multiple, du fait de sa complexité, n'a pas été retenu comme mécanisme d'implantation du polymorphisme en Java. Les concepteurs du langage lui ont préféré la notion d'*interface* (conceptuellement proche de celle de protocole dans Objective-C). Une interface permet de spécifier des protocoles, des comportements abstraits ou génériques. Ce mécanisme permet également d'attribuer des comportements multiples à un objet et de fédérer des objets selon leurs comportements.

Java est multiprocessus

La notion de processus (*thread*) est intégrée dans le langage Java. Un mécanisme de synchronisation est défini au niveau de la syntaxe, permettant de protéger aussi bien un bloc de code, une méthode ou une variable d'une exécution concurrente. Autoriser ainsi le programmeur à raisonner en termes d'actions indépendantes lui permet de simplifier la conception de la programmation. Les événements sont traités par les processus concernés, les processus communiquent et coopèrent pour accomplir des tâches plus globales.

Java produit du code interprétable par une machine virtuelle

Le résultat de la compilation d'un programme Java est un fichier de *byte-code*. Ce *byte-code* spécifie des instructions (appelées aussi pseudo-code) exécutables par une machine virtuelle. Cette machine virtuelle est munie de contraintes qui vont garantir certaines propriétés lors de l'exécution d'un programme. Ce niveau intermédiaire d'interprétation du code établit une indépendance par rapport à la machine physique supportant l'exécution de la machine virtuelle. Ce principe de pseudo-code a déjà connu ses heures de gloire dans le passé, avec le Pascal UCSD qui avait été porté sur plusieurs architectures de microprocesseurs 8 bits de la fin des années 70 (dont l'Apple II).

Le chargement du code est dynamique

Le code Java est chargé uniquement s'il est utilisé. Le code peut ainsi résider sur des serveurs différents. Il peut être développé par des équipes différentes situées dans des lieux géographiques distants, et reliées par le réseau. La gestion des versions est simplifiée, le code chargé correspondant toujours à la version la plus récente.

Java utilise des standards

L'arithmétique flottante de Java est conforme à la norme IEEE 754, ce qui garantit que les calculs effectués sur n'importe quelle machine physique supportant Java donnent toujours les mêmes résultats. Il est ainsi possible de distribuer une tâche complexe de calcul sur un ensemble hétérogène de plateformes (Mips, Pentium, PowerPC, Sparc, etc.). L'encodage des caractères est effectué sur 16 bits selon la norme Unicode, ce qui permet de traiter les signes diacritiques de plusieurs langues autres que l'anglais (qui n'en possède pas!).

La syntaxe de Java est familière

Les concepteurs de Java ont eu la sagesse de conserver les formes syntaxiques de C et de C++ pour spécifier les expressions (*i++, <<, &,* etc.) et les structures de contrôle (*if, for, while,* etc.). Les programmeurs connaissant déjà ces langages peuvent ainsi garder certaines de leurs habitudes.

La syntaxe de Java est simple

Le corpus des règles est limité et les constructions sont simples, ce qui facilite l'apprentissage de Java. Cependant, nous ne pouvons pas en dire autant de Java lui-même. Les difficultés de programmation en Java ne résident en effet pas dans la syntaxe, mais dans l'articulation des concepts objet. Un bon programmeur C++ ou Smalltalk deviendra rapidement un très bon programmeur Java.

Java, une fois maîtrisé, peut améliorer la qualité des programmes objet, mais il ne peut en aucun cas améliorer la qualité du **choix** de la décomposition en ob-

jets. Dès lors, « comment traduire la réalité en objets ? » reste la question primordiale, à laquelle nous tenterons par ailleurs de répondre dans la partie IV.

2.3 Adéquation de Java à Internet et au Web

Le potentiel commun d'Internet et du Web en matière d'accès et de distribution d'informations n'est plus à démontrer. Les standards tels que HTTP ou HTML offrent un niveau élevé de compatibilité qui garantit la portabilité des informations (principalement des documents) dans de nombreux environnements. Malheureusement, cette portabilité est loin d'être aussi évidente au niveau des programmes : les langages, les environnements de développement, les systèmes d'exploitation et les plateformes matérielles définissant tous leurs propres « standards », l'incompatibilité entre systèmes demeure la seule norme.

Dès lors, le prochain défi à relever consiste à proposer du code pouvant être développé sur une machine, transmis sur le réseau et exécuté sur d'autres machines totalement différentes, quels que soient leur système d'exploitation et leur processeur. De plus, l'exécution de ce code doit fournir exactement les mêmes résultats (précision arithmétique, interface graphique, etc.). Mission impossible ? Peut-être pas, mais les qualités requises pour la distribution à travers le réseau sont nombreuses :

- le code doit être **robuste** : c'est-à-dire que tout doit être mis en œuvre pour minimiser les risques de « crash ». Cela est encore plus vrai si le programme est un assemblage de plusieurs classes provenant de sources différentes sur le réseau;
- le code doit être **indépendant** : le client doit pouvoir exécuter des programmes sans se soucier du type de matériel utilisé par le serveur. Le programmeur doit pouvoir développer son application sans connaître lui-même ou imposer aux utilisateurs un type de plateforme particulier;
- le code doit être **sûr** : le client doit avoir la certitude de pouvoir charger des applications depuis n'importe quel point du réseau sans craindre des virus, des manipulations ou des destructions de fichiers, ni des problèmes de communication avec des serveurs peu scrupuleux (qui s'informent, par exemple, du contenu de vos disques).

Java est robuste

En éliminant l'utilisation des pointeurs, en gérant l'allocation et la désallocation de la mémoire, en détectant les usages illicites des tableaux, en ne permettant pas la surcharge des opérateurs, en substituant à l'héritage multiple les interfaces, en utilisant les standards, en ayant une syntaxe allégée, Java contribue grandement à améliorer la qualité et la sécurité des programmes. Les erreurs qui demeurent sont généralement de type fonctionnel (dues à une mauvaise concep-

tion). Le programmeur étant déchargé des responsabilités de gestion de la mémoire, il peut mieux se concentrer sur l'activité de l'objet et sur la qualité de sa conception.

L'interprétation par une machine virtuelle et le chargement dynamique permettent d'éviter des phases de compilation et d'édition de liens qui peuvent devenir très lourdes dès que le projet atteint une certaine taille. Le compilateur, avec sa vérification stricte du typage, permet lui aussi de détecter de nombreuses erreurs de programmation (passage de paramètres, appel de méthodes, etc.).

Java possède également des mécanismes assez subtils en ce qui concerne le réglage de la visibilité et de l'héritage entre classes. Ces mécanismes peuvent être pleinement utilisés pour développer des logiciels complexes et de taille importante, en appliquant les apports méthodologiques du génie logiciel.

L'architecture Java est neutre et portable

L'utilisation de byte-code et celle des standards Unicode et IEEE754 rendent Java indépendant de la plateforme d'exécution.

Les seules parties dépendantes des plateformes sont encapsulées dans la machine virtuelle chargée d'interpréter les byte-codes et dans un package (*awt.peer*) qui détermine l'apparence des objets graphiques pour chaque système d'exploitation (MacOS, Solaris, Windows95/NT, etc.). Une même application saura donc s'adapter à la machine du client afin de respecter les conventions de son interface graphique utilisateur (GUI).

L'objectif de cette neutralité et de cette portabilité est qu'un programmeur puisse développer son application une seule fois et que tous les utilisateurs potentiels puissent l'exécuter, indépendamment des choix qu'ils ont effectués pour leur poste de travail.

Java est sûr

La sécurité, dans un environnement complètement distribué comme le réseau Internet, est une priorité absolue. Il est impensable de tester tous les programmes avant de les charger, de certifier toutes les sources et tous les serveurs du réseau. La seule solution est que le poste de travail se protège lui-même.

L'encapsulation de l'exécution du code dans une machine virtuelle permet la portabilité par rapport à la machine physique, mais elle assure aussi un découplage par rapport aux ressources (mémoire, système de fichiers, canaux de communication) de cette dernière.

L'application, une fois chargée, est enfermée dans la machine virtuelle et n'a donc pas **directement** accès aux instructions de la machine physique, ni à ses ressources. Chacune de ces ressources sera alors vue comme un service de la ma-

chine virtuelle. L'utilisateur peut ainsi régler en tout temps les capacités de sa machine virtuelle (donc des applications chargées à l'intérieur), en fonction du degré de confiance qu'il a envers le serveur ou la société qui a produit le programme :

- pour un jeu séduisant, il peut ne donner au programme que le droit de s'exécuter;
- pour gérer son compte d'épargne, il peut autoriser le programme mis à disposition par sa banque à créer un fichier local des transactions.

Atteindre ce degré de confiance demande une grande rigueur, la machine virtuelle devant se méfier de tous les programmes qui sont chargés. A cet effet, elle effectue sur chacun d'entre eux une série de tests (voir chapitre 27) afin de prouver un certain nombre de propriétés attestant que l'exécution des bytes-codes ne peut avoir un comportement caché (virus, etc.). De plus, l'espace des noms du code est isolé de celui de sa provenance; il est ainsi impossible qu'un objet se substitue à un autre. Enfin, le code est testé par rapport aux droits qui lui sont attribués (lire/écrire des fichiers locaux, communiquer localement/à l'extérieur, etc.).

Ces trois qualités – robustesse, portabilité et sûreté – sont celles qui vont assurer le succès futur de Java.

2.4 Code mobile

Un code (programme compilé) est dit mobile s'il est **transportable** (le lieu de son exécution est indépendant de son lieu de stockage) et **portable** (le support de ces exécutions ne se restreint pas à un type particulier de machine).

Il est toujours difficile d'établir l'aspect révolutionnaire d'événements pendant leur survenance. Cependant, on peut réutiliser les typologies, examiner les questionnements, établir quelques bilans.

Les remarques que nous faisons ici prennent leur sens dans le cadre d'Internet ou d'un intranet. Un **intranet** est l'utilisation des technologies d'Internet (TCP/IP, Java, Web, etc.) restreinte à une communauté d'utilisateurs et de services définis pour celle-ci. L'intranet typique est celui défini par la communauté des employés d'une entreprise. Les services et les applicatifs sont ceux leur permettant de réaliser leur mission. L'intranet peut être étendu à une ville, une région, un regroupement d'organisations (des universités par exemple). Un intranet n'est pas défini par la topologie de son réseau, mais par la spécificité de ses services et de ses utilisateurs. Il peut être rattaché à Internet, généralement à travers un mur pare-feu (*firewall*) qui définit les droits d'accès des flux entrants et sortants.

Programmation distribuée en Java

Nous reprenons ici la typologie du modèle client-serveur (voir p. 13) en considérant les apports de Java, d'Internet et des bases de données.

Type	Activité sur le serveur	Activité sur le client	Description
0	Données Traitements Présentation		Des terminaux connectés par l'ordinateur central sur Internet avec Lynx (butineur en mode caractères 24x80), traitements par programmes sur le serveur.
1	Données Traitements Présentation	Présentation	Utilisation du Web avec un terminal X (par exemple, HotJava depuis un Mac).
2	Données Traitements	Présentation	Le Web avec exécution du butineur sur le client (traitements par programmes sur le serveur), situation avant Java.
3	Données Traitements	Traitements Présentation	Les traitements sont répartis entre le client et le serveur (Java, JavaScript sur le client, application Java sur le serveur).
4	Données	Traitements Présentation	Le serveur est un pur serveur de données (utilisation de Java et d'un kit de connexion aux bases de données, JDBC par exemple).
5	Données	Données Traitements Présentation	Comme dans 4 + une base de données locale (Oracle Lite) + une connexion aux outils bureautiques locaux.
6		Données Traitements Présentation	Application Java sur le client.
7	Données Traitements Présentation	Données Traitements Présentation	Au fond, la typologie n'est plus classifiante et le mode de travail sera un mélange de tout, sur le client et sur le serveur.

Tableau 2.1 Typologie de la répartition des activités

Si nous réexaminons les difficultés rencontrées dans la mise en œuvre du client-serveur (voir point 1.6), nous constatons que :

- la répartition de l'activité ne pose pas de problème : il est possible de transférer le code Java du serveur vers le client. Le contenu des documents HTML détermine ce que le client doit exécuter;
- le coût de la distribution diminue fortement : le client charge l'application lorsqu'il charge le document, obtenant ainsi la dernière version de celle-ci. Le coût de distribution se résume au coût de transfert, celui-ci

pouvant être réduit grâce à l'utilisation d'un système de cache dans le butineur et de miroirs diminuant les distances et augmentant les capacités d'accès. Seul le butineur doit être installé sur le client;

- la portabilité est maximale, autant du côté des clients qui ne connaissent pas les détails d'implantation des serveurs que du côté des serveurs qui ne connaissent pas les configurations matérielles des clients ou leur système d'exploitation.

Le trio Java, BD et Web (y compris TCP/IP) remplit toutes les cases de la typologie. Les serveurs et les clients y acquièrent des statuts d'universalité qui les rendent indépendants. La logique du couple client-serveur est remplacée par celle d'un serveur accessible par tous les clients et d'un client connectable sur tous les serveurs.

Des documents statiques aux documents dynamiques animés

La vision classique d'un document est celle d'une composition d'éléments eux-mêmes composés d'autres éléments plus simples (chapitres composés d'un titre et d'un ensemble de sections, sections composées d'un titre et de paragraphes, etc.), jusqu'à des éléments primitifs comme les caractères ou les images.

Un document devient dynamique dès le moment où l'un de ses éléments n'est plus stocké tel quel sur un support mais construit (calculé) automatiquement au moment où le lecteur y accède. Dès le début du Web, ses concepteurs ont prévu cette possibilité en définissant un protocole et une forme d'URL qui permettent de demander à un serveur non pas le contenu d'un nœud d'information déjà stocké mais le calcul de ce contenu à partir de paramètres et de données propres au serveur. Cette possibilité est largement utilisée pour accéder à des bases de données : le serveur reçoit un URL décrivant les informations recherchées; il effectue la recherche correspondante puis transmet le document ainsi créé au client.

Les langages de scripting comme JavaScript (voir annexe E) rendent les documents actifs en y introduisant du code exécutable. Une partie du document possède alors un comportement propre et peut interagir avec l'utilisateur. Les formulaires avaient déjà introduit une possibilité d'interaction, limitée à la saisie de données dans des champs et à leur envoi à un serveur.

Avec Java, la dynamicité aborde une nouvelle dimension car le code qui anime certains éléments du document n'a pas besoin d'être inclus dans le document : une référence à un code stocké sur un serveur suffit. C'est au moment où le document est visualisé que le code mobile est amené du serveur sur la machine cliente où a lieu la visualisation. Ainsi, on désynchronise complètement la production du document et celle du code. Des experts en programmation peuvent développer des programmes génériques de haute qualité pour gérer différents ty-

pes de contenus, le producteur de document peut choisir parmi les codes à disposition celui qui convient en fonction du type de contenu qu'il veut générer ou animer.

Examiner les questionnements

Nous avons trouvé trois questions qui reçoivent une réponse technique définitive avec le trio Java, BD et Web. Ces questions semblent périphériques à la création de logiciels, mais pourtant leur apporter quotidiennement une solution accapare une grande partie des ressources informatiques.

- **Qui est mon public cible et comment est-il équipé matériellement?** Utilise-t-il une architecture PC (386, 486, Pentium, ou P6) ou Macintosh (68000 ou PowerPC) ou de station de travail? Avec quelles configurations mémoire et disque? Sous quel système d'exploitation : MS-DOS, Windows 3.11/95/NT, MacOS, Unix (Linux, AIX, Solaris, etc.)? Répondre clairement à cette question pour une entreprise ou pour un éditeur de logiciel est un casse-tête.
  La réponse du client universel donnée par les outils Internet semble si simple : le client n'a besoin que d'une connexion TCP/IP (de capacité adaptée à son activité) et d'un butineur supportant l'exécution de Java;
- **Comment distribuer et mettre à jour mes applications, mes documents?** On revient ici à la question des configurations augmentée de la problématique du support : disquette ou CD-ROM.
  La réponse du chargement du document et de son code associé et/ou du déclenchement d'une application sur le serveur semble aussi régler la question;
- **Comment migrer mes applications?**
  Ici, la question est rendue difficile par le nombre des clients (plusieurs milliers dans une entreprise) et par le fait que le passage de l'ancien système au nouveau doit être effectué le plus rapidement possible (la migration de mille postes clients demande une énorme préparation).
  La réponse apportée par les outils Internet est « changer un URL » ou « modifier une adresse IP dans un serveur de nom ».

Les questions concernant Internet restent nombreuses : la sécurité, le cryptage, la monnaie électronique (e-cash, digicash), etc.

Néanmoins, les solutions déjà apportées aux problèmes ci-dessus nous semblent suffisamment intéressantes pour écarter l'hypothèse qu'Internet va cesser de se développer ou qu'il ne s'agit que d'une mode passagère.

Les piliers de la programmation distribuée

D'un point de vue physique, l'infrastructure des réseaux, les protocoles, les nouvelles architectures de machines constituent les fondations essentielles du développement d'applications distribuées à l'échelle d'Internet.

Cependant, d'un point de vue conceptuel[1], les piliers les plus importants pour l'écriture des nouvelles applications sont :

- les bases de données ;
- les hypertextes (Web);
- le code mobile (Java).

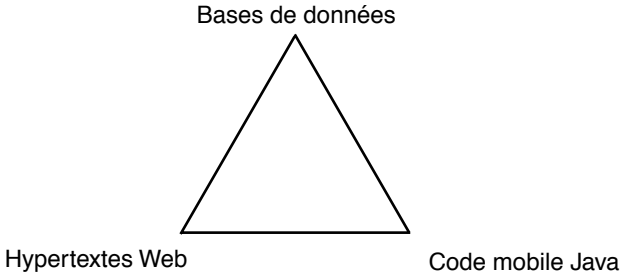


Figure 2.1 Piliers des applications intranet et Internet

Les **bases de données** sont la mémoire d'Internet; en effet, il est impensable de gérer un grand nombre de documents (plusieurs milliers de pages HTML sur un même serveur) sans recourir à une base de données. La gestion des liens (URL), des droits d'accès, de la mise à jour des données, la construction des index sont simplifiées grâce à elles. De plus, la base de données constitue également un moteur de recherche et de synthèse de documents.

Les **hypertextes** définissent l'interface de présentation et de navigation entre des documents qui contiennent des données, du code mobile et la logique de l'application. Ainsi, l'hypertexte permet d'inscrire dans les liens une logique d'application. Le parcours des liens peut se substituer à l'activation de boutons ou d'articles dans un menu.

Le **code mobile** définit le comportement des objets. Il fait passer un document de l'état statique à l'état dynamique en spécifiant des fragments de comportement. On peut alors imaginer qu'à la description des données sera ajoutée une description des fragments de comportement, s'assemblant pour être sémantiquement pertinents. Par exemple, si l'on souhaite gérer ses emprunts à la banque, le fragment de code décrivant une calculatrice de gestion d'amortissement sera chargé et présenté en même temps que l'état de ses comptes.

---

1. Ces deux points de vue couvrent ce que l'on appelle le *network-computing*.

Les documents affichés seront de moins en moins stockés dans des fichiers. Ils seront le résultat de la synthèse :

- des données stockées dans la base de données;
- des fragments de code mobile associés aux types de données;
- des droits de l'utilisateur par rapport à son rôle;
- du contexte de l'utilisation;
- du chemin parcouru dans l'hypertexte.

... Restent à imaginer des méthodes de conception et de développement qui tiennent compte de cette nouvelle donne.

Chapitre 3

L'environnement de développement Java

Hormis l'infrastructure réseau qui est à acquérir, l'environnement complet de développement Java peut être téléchargé depuis Internet. On le trouve également sous forme de CD-ROM.

3.1 Outils

Nous allons rapidement passer en revue les différentes situations d'utilisation de Java, qu'il s'agisse de développement en local ou relié à Internet.

Développement d'applications en local

Pour le développeur d'applications personnelles, il est nécessaire de disposer d'un kit comprenant au minimum les outils suivants : un éditeur de texte, un compilateur Java, un interpréteur du code Java compilé.

Des environnements de développement sophistiqués intègrent également des éditeurs guidés par la syntaxe, des gestionnaires de projet, des débogueurs, des générateurs d'interface, etc.

Utilisation d'applets téléchargées depuis le réseau

Pour l'utilisateur d'applets (c'est-à-dire un client), les outils nécessaires sont :

- une connexion réseau (intranet/Internet);
- un gestionnaire de protocole TCP/IP;
- un butineur Web capable d'interpréter du code Java.

D'autres outils tels que courrier électronique, transfert de fichiers, compression/décompression, encryptage (PGP), etc., peuvent être utiles.

Pour distribuer des applets

Il existe des sites sur lesquels on peut déposer ses documents pour en faciliter la distribution, sans avoir à héberger un serveur sur sa propre machine. Cependant, si l'on souhaite réaliser un serveur HTTP, les outils nécessaires sont :

- une connexion réseau (intranet/Internet) supportant un débit en rapport avec le nombre d'utilisateurs;
- un serveur HTTP;
- une base de données; nous l'incluons ici car il est impensable de gérer une multitude de pages sans une base de données;
- un kit de développement Java;
- un éditeur HTML.

Si l'on héberge un serveur HTTP connecté à Internet, des outils supplémentaires sont à conseiller : gestionnaire de sécurité, gestionnaire des statistiques d'accès, etc.

Java sans Internet

Il n'est pas nécessaire de disposer d'une connexion Internet pour utiliser Java. De plus, le code développé étant portable, toutes vos applications peuvent être diffusées à d'autres personnes sans recompilation. Vous pouvez développer des applications et des applets sur un PC sans connexion. L'environnement de développement pour le MacOS, par exemple, inclut un *applet viewer* permettant d'exécuter des applets sans butineur.

3.2 Développer avec Java

Le cycle de développement d'un programme Java est très semblable à celui d'autres langages compilés puis interprétés par une machine virtuelle. Il se déroule de la manière suivante :

1. Le programme est stocké dans un fichier source de suffixe *.java*.
2. Le compilateur Java compile ce fichier et le transforme en un fichier de byte-code dont le suffixe est *.class*.
3. L'interpréteur Java charge le byte-code et le vérifie. Cette vérification consiste à garantir que le byte-code ne risque pas de perturber le fonctionnement de l'interpréteur ou de détruire des ressources sur votre machine. Elle est effectuée quelle que soit l'origine du byte-code, qu'il provienne du réseau ou de votre disque dur.
4. Une fois la vérification effectuée, l'interpréteur exécute le byte-code.

Notions d'applet et d'application

Un programme Java peut prendre deux formes distinctes, chacune étant adaptée à un contexte d'invocation et d'exécution différent.

La première (*applet*) est destinée à des programmes **invoqués depuis des documents HTML** et exécutés à l'intérieur d'un butineur ou d'un *applet viewer* équipé d'un interpréteur Java.

La seconde (*application*) permet de créer des applications au sens classique du terme, c'est-à-dire s'exécutant de manière autonome à l'aide de l'interpréteur.

Le cycle de développement est identique pour les deux formes, seul le contexte d'invocation et d'exécution varie :

- une applet réside en général sur un serveur. Un document HTML fait référence à l'applet à l'aide d'un URL. Au moment du chargement du document HTML, le butineur détecte un délimiteur d'invocation de l'applet, charge son code dans l'interpréteur intégré qui lance son exécution. L'applet s'affiche alors dans la fenêtre du butineur;
- une application quant à elle réside habituellement sur la machine où elle est exécutée. Elle s'exécute à l'aide d'un interpréteur.

La figure 3.1 ci-dessous illustre le cycle de développement en Java et les différences entre une applet (p) et une application (a). À noter la référence du document HTML vers l'applet, qui indiquera au butineur de charger son code.

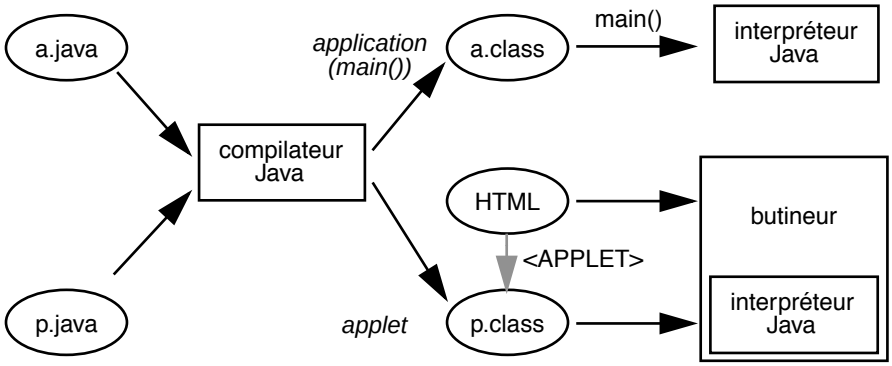


Figure 3.1 Développement et exécution d'un programme Java

Nous verrons au paragraphe 16.3 qu'il est également possible d'exécuter une applet comme une application.

3.3 Où trouver Java

Le kit de développement Java (JDK) fourni par Sun est actuellement disponible dans sa version 1.2 pour les plateformes suivantes :

- SPARC Solaris;

- Windows NT;
- Windows 95-98;
- MacOS.

Vous pouvez le charger avec sa documentation depuis le site suivant :

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

Contenu du JDK

Le kit de développement contient :

- un compilateur : le programme qui va traduire vos fichiers de texte source Java en byte-code;
- un *viewer* : le programme qui interprète vos fichiers byte-code;
- l'API Java : un ensemble de classes qui vous permettront de développer vos programmes;
- une documentation : un ensemble de fichiers HTML qui constitue une aide indispensable à la bonne compréhension de l'API Java.

D'autres outils de développement sont également disponibles :

- un générateur de documentation (*Javadoc*) : il permet de construire des documents à partir des commentaires mis dans les programmes;
- un débogueur : un programme pour la mise au point de vos programmes;
- un générateur de code natif : générant du code natif pour certaines architectures de machine;
- un désassembleur.

3.4 Mes premiers pas

Voici deux exemples très simples vous permettant de tester votre installation.

Java : première application

Un programme doit posséder une méthode *main()* qui sera le point de départ de l'exécution de ce dernier. Écrivez le programme suivant dans un fichier nommé *MaPremiereApplication.java :*

```
class MaPremiereApplication {
   public static void main (String args[]) {
      System.out.println("Ma Premiere Application");
   }
}
```

Ensuite, compilez le fichier avec la commande (sous Unix ou Windows/NT) :

```
javac MaPremiereApplication.java
```

Le résultat de cette compilation (pour autant qu'elle ne produise pas d'erreur) doit créer dans le même répertoire un fichier ayant une extension *.class*, soit : *MaPremiereApplication.class*.

Finalement, exécutez le programme avec la commande :

```
java MaPremiereApplication
```

Le programme doit alors afficher :

```
Ma Première Application
```

Sous d'autres systèmes d'exploitation (MacOs par exemple), l'affichage a lieu dans une fenêtre (voir figure 3.1) qui correspond au *stdout* (la sortie standard d'Unix).

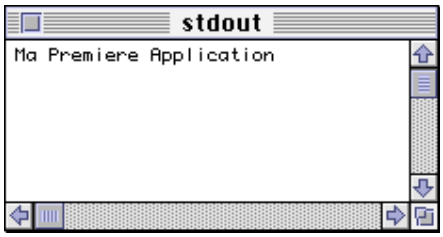


Figure 3.1 Ma première application (sous MacOs)

Une application doit toujours posséder une méthode *main()* déclarée comme suit :

```
public static void main (String args[]) {
   ... instructions à exécuter ...
}
```

Nous verrons plus loin la signification exacte des termes qui apparaissent dans cet énoncé.

Java : première applet

Écrivez le programme suivant dans un fichier nommé *MaPremiereApplet.java :*

```
import java.awt.Graphics;

public class MaPremiereApplet extends java.applet.Applet {

   public void paint(Graphics g) {
   g.drawString("Ma première applet !", 20, 20);
   }
}
```

Compilez ensuite le fichier avec la commande :

```
javac MaPremiereApplet.java
```

Le résultat de cette compilation (pour autant qu'elle ne produise pas d'erreur) doit créer dans le même répertoire un fichier ayant une extension *.class*, soit *MaPremiereApplet.class*.

Il nous faut maintenant invoquer cette applet depuis un document HTML qui devra se trouver pour le moment dans le même répertoire que le fichier *MaPremiereApplet.class*. Nous proposons d'écrire ceci dans un fichier *test.html* :

```
<title>Applet Une</title>
<h1>Vraiment la premi&egrave;re</h1>
<hr>
<applet code=MaPremiereApplet.class width=200 height=50>
</applet>
<hr>
<a href="MaPremiereApplet.java">Le source.</a>
<hr>
```

Ouvrez votre butineur et demandez à charger votre document *test.html*. Vous devriez obtenir le résultat suivant (figure 3.2) :

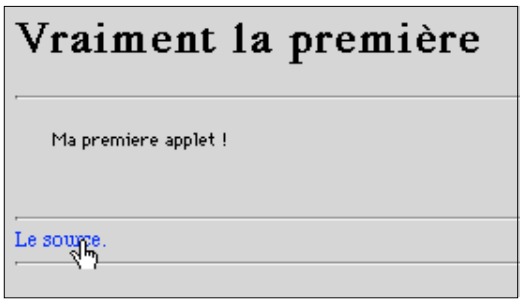


Figure 3.2 Ma première applet (vue dans un butineur)

3.5 Chercher de l'information sur Java

Nous avons essayé de ne pas indiquer d'URL car ils ne sont pas toujours très stables : les sites restent mais les documents sont souvent réorganisés. De plus, nous pensons que le plus important est de savoir **rechercher** l'information. Voici un exemple utilisant l'indexeur AltaVista (*http://www.altavista.com/*) de Digital Equipment Corporation (vous pouvez aussi essayer Yahoo, Lycos, Exite, etc.).

Quelques conseils pour débuter :

1. Ne pas interroger avec un mot trop vague ou trop commun : la recherche avec le mot clé *Java* retourne plus de 8 millions de liens (seulement 200000 liens en juin 96).
2. Cadrer la recherche avec un mot désignant un domaine : *java AND sécurité* ne retourne plus que 85 liens.
3. Utiliser les possibilités de tri (terme, liste des liens retournés).
4. Commencer à inspecter la liste des liens, tester les liens.

5. Si un nœud (ou une liste des liens retournés) vous intéresse, sauvez son URL dans une liste de référence (*bookmark*).
6. Recommencer avec des mots proches ou contraires (*Java AND virus*).
7. Si vous désirez avoir des liens en français, introduisez au moins un mot en français (*Java AND programmation*).
8. Recommencez régulièrement vos recherches, la fréquence dépendant de la vitesse d'évolution du domaine (semaine, mois, trimestre, etc.).

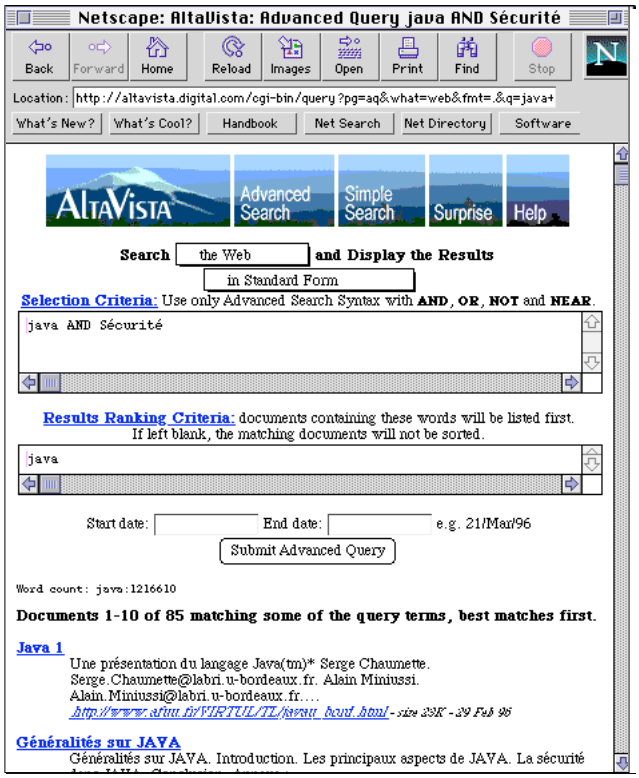


Figure 3.1 Recherche à l'aide de l'indexeur AltaVista, à l'époque où il était sans publicité!

Partie II

Le langage

> Telles des chèvres en détresse, sept Mercedes-Benz vertes, les fenêtres crêpées de reps grège, descendent lentement West End Street et prennent sénestrement Temple Street vers les vertes venelles semées de hêtres et de frênes près desquelles se dresse, svelte et empesé en même temps, l'évêché d'Exeter. Près de l'entrée des thermes des gens s'empressent. Quels secrets recèlent ces fenêtres scellées?
>
> G. Perec, *Les Revenantes*.

Chapitre 4

Les bases

Pour apprendre un langage, deux approches peuvent être suivies : la première, des concepts vers la syntaxe et la seconde, de la syntaxe vers les concepts. Nous avons choisi de suivre cette dernière car elle permet de partir de notions déjà rencontrées dans d'autres langages de programmation et de nous amener à un stade où il devient possible de construire des exemples intéressants à l'aide de notions déjà expliquées. De plus, en partant des bases, il est possible de faire rapidement état des différences existant avec d'autres langages tels que C ou C++ (voir à ce propos l'annexe A).

Littéraux, types, expressions, instructions

Nous allons examiner la syntaxe des littéraux et des types ainsi que les expressions et la structure des primitives indispensables à la définition des algorithmes.

Avant d'examiner comment programmer, il importe de savoir commenter le programme.

4.1 Commentaires

Trois styles de commentaires sont disponibles en Java. Le plus connu est celui qui provient du C, où le commentaire est encadré par /* et */.

```
/* ceci est un commentaire
   sur plusieurs lignes
... qui se termine ici */
```

Le second provient de C++ et permet de placer un commentaire sur la même ligne qu'une instruction. Dans ce cas, le commentaire commence par //.

```
int i // voici une variable entière
```

Enfin, le troisième style consiste à encadrer le commentaire avec /** et */. Cette forme est à utiliser uniquement dans les déclarations : elle est reconnue par un générateur automatique de documentation (*JavaDoc*). Le tableau 4.1 ci-dessous illustre ces trois styles de commentaires.

Style	Définition
`/* commentaire */`	Tous les caractères compris entre /* et */ sont ignorés.
`// commentaire`	Tous les caractères compris entre les // et le retour de la ligne sont ignorés.
`/** commentaire */`	Tous les caractères compris entre /** et */ sont ignorés par le compilateur mais peuvent être traités par des outils de documentation automatique.

Tableau 4.1 Styles de commentaires

Bonnes habitudes : il faut commenter les programmes

N'oubliez pas qu'un programme est écrit une seule fois, mais qu'il sera relu des dizaines de fois, par vous-même et par d'autres. Les commentaires doivent être une reformulation de la spécification du programme. Il faut donc éviter les commentaires qui se font simplement l'écho de l'instruction. Exemple :

```
int zoom=2 // zoom à 2
```

Ce commentaire ne donne aucun renseignement sur le rôle de *zoom* et sur le choix de la valeur *2*.

```
int zoom=2 // valeur par défaut du zoom au démarrage
```

Ce dernier lui est nettement préférable.

Nous allons maintenant examiner comment nommer précisément les objets de nos programmes.

4.2 Identificateurs

Les identificateurs permettent de nommer les différents éléments (variables, classes, méthodes, packages, etc.) des programmes Java.

identifier =
  "dans {a..z, A..Z, $, _}"
  < *" dans {a..z, A..Z,$,_,0..9,unicode character over 00C0}"* >

La règle de nommage ci-dessus indique que le premier caractère doit être une

lettre minuscule ou une lettre majuscule, ou encore le caractère souligné _ ou le caractère dollar *$*. Les caractères suivants (dès le second) peuvent être soit :

- des caractères *a..z* ou *A..Z;*
- le _ ou le *$;*
- des chiffres de *0* à *9;*
- les caractères Unicode (voir [13]) supérieurs à *0X00C0*, permettant d'introduire dans les identificateurs des caractères nationaux tels que Ç, ü, etc. (à l'exclusion cependant des caractères de contrôle).

Exemples d'identificateurs valides en Java :

```
$valeur_system
dateDeNaissance
ISO9000
```

Exemples d'identificateurs non valides en Java :

```
ça          // Ç comme premier caractère
9neuf       // 9 comme premier caractère
note#       // # pas au dessus de 0X00C0
long        // OK mais c'est un mot réservé!
```

long est bien formé selon les règles énoncées ci-dessus, mais il appartient à la liste des mots réservés du langage Java (voir tableau 4.2). En effet, en Java comme dans les autres langages de programmation, un certain nombre de mots sont réservés à la construction des instructions. Dès lors, un identificateur ne doit pas être choisi dans cette liste.

abstract	boolean	break	byte	byvalue[1]
case	catch	char	class	const[2]
continue	default	do	double	else
extends	false	finally	float	for
goto[3]	if	implements	import	instanceof
int	interface	long	native	new
null	package	private	protected	public
return	short	static	super	switch
synchronized	this	throw	throws	transient
true	try	void	while	

Tableau 4.2 Liste des mots réservés en Java

1. Mot réservé mais pas utilisé actuellement.
2. Idem.
3. Idem.

Bonnes habitudes : choix des identificateurs

Nous suggérons de ne pas employer le *$* et le _ si vous devez utiliser des librairies en C, de ne pas utiliser le *$* en première position, de séparer les noms composés en mettant en capitale la première lettre des noms à partir du deuxième mot. Le tableau 4.3 illustre ces différentes suggestions à l'aide d'exemples.

Conseillé	Acceptable	Déconseillé
`nomDeMethode`	`nom_de _methode`	`$init`
`ceciEstUneVariable`	`ceci_est_une_variable`	`_type`

Tableau 4.3 Choix des identificateurs

4.3 Littéraux

Les littéraux définissent explicitement les valeurs sur lesquelles travaillent les programmes Java. Ils sont intimement liés aux types des variables qui les contiennent. Java supporte trois catégories de littéraux : les *booléens*, les *nombres* (entiers et flottants) et les *caractères*.

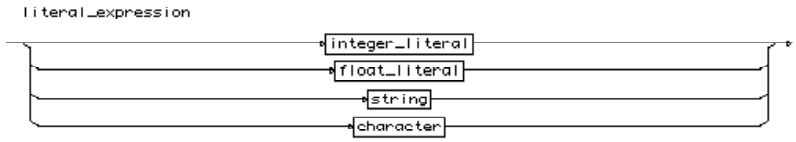


Diagramme 4.1 literal_expression[1]

Booléens

Les mots clés *false* et *true* définissent respectivement les valeurs de vérité faux et vrai. Ils permettent essentiellement d'initialiser des variables booléennes. À noter que la valeur d'une variable booléenne est **toujours** associée à l'un de ces deux littéraux et ne peut pas être assimilée à un *0* ou un *1* comme dans le cas des langages C ou C++ (voir p. 421).

```
boolean resultatAtteint = false ;
boolean continuer = true ;
```

Le code ci-dessus déclare deux variables booléennes et initialise leur valeur.

1. Les diagrammes syntaxiques que nous utilisons ont été établis à partir des spécifications du langage publiées par Sun. Nous avons préféré conserver les termes anglais originaux pour désigner les éléments syntaxiques.

Entiers

L'expression des nombres entiers peut se faire dans trois formats : en décimal, en octal et en hexadécimal.

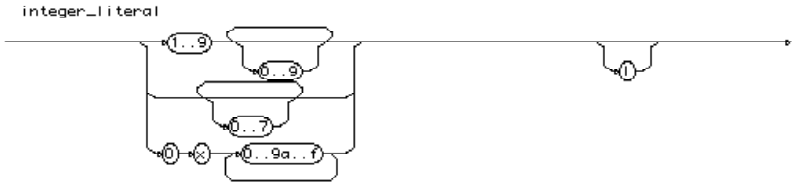


Diagramme 4.2 integer_literal

L'entier exprimé en décimal ne commence jamais par un zéro. Cela est extrêmement important pour éviter des erreurs.

```
int nbrDeMois = 12;
```

Pour exprimer un entier en octal, il suffit de le faire précéder d'un zéro.

```
int nbrDeDoigts = 012;    // = 10 en décimal
```

Enfin, pour exprimer un entier en hexadécimal, il suffit de le faire précéder d'un zéro suivi d'un X minuscule ou majuscule.

```
int dixHuit= 0x12;
```

Les nombres entiers déclarés littéralement correspondent au type entier *int* sur 32 bits. On peut forcer un entier à prendre le type entier *long* en le faisant suivre d'un L minuscule ou majuscule; il s'étend alors sur 64 bits.

Flottants

Le littéral flottant est utilisé pour définir des valeurs décimales. Il comporte une mantisse et éventuellement une partie exposant exprimée en puissances de 10. Un nombre flottant doit obligatoirement avoir un point décimal ou une partie exposant (sinon rien ne le distingue d'un nombre entier). Les diagrammes syntaxiques 4.3et 4.4expriment ces différentes possibilités.

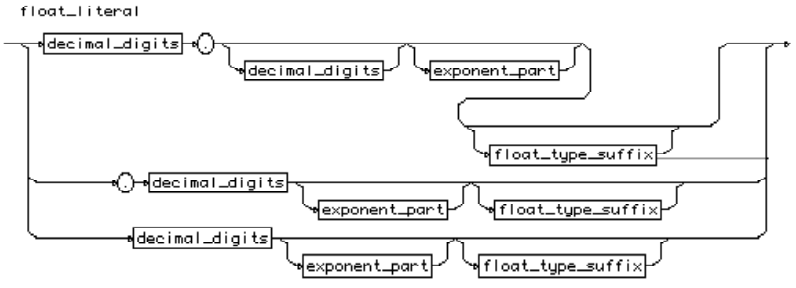


Diagramme 4.3 float_literal

Diagramme 4.4 decimal_digits

Exemples de flottants sans partie exposant :

```
2.    .5    2.5   2.0   .001
```

La partie exposant est préfixée par un E minuscule ou majuscule suivi de la valeur de l'exposant (voir diagramme 4.5).

Exemples de flottants avec partie exposant :

```
2E0   2E3   2E-3  .2E3  2.5E-3
```

Diagramme 4.5 exponent_part

Les nombres flottants déclarés littéralement correspondent au type flottant *double* sur 64 bits. On peut forcer un flottant à prendre le type *float* en lui faisant succéder un F minuscule ou majuscule; il s'étend alors sur 32 bits. On peut également forcer un flottant à prendre le type flottant *double* précision en lui faisant succéder un D minuscule ou majuscule; il s'étend alors sur 64 bits (diagramme 4.6) :

```
3.14F    3.14159D
```

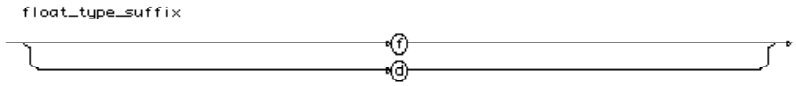


Diagramme 4.6 float_type_suffix

Caractères

Le littéral d'un caractère est un caractère entouré de deux apostrophes :

```
'x'     'a'     '4'
```

Il peut être choisi dans l'ensemble des caractères Unicode. Le tableau 4.4 ci-dessous montre les caractères de contrôle les plus courants ainsi que leur séquence de codification.

Caractère	Abréviation	Séquence
Continuation	<nouvelle ligne>	\

Tableau 4.4 Caractères de contrôle

Caractère	Abréviation	Séquence
Nouvelle ligne	N	`\n`
Tabulation	HT	`\t`
Retour arrière	BS	`\b`
Retour chariot	CR	`\r`
Saut de page	FF	`\f`
Backslash	\	`\\`
Apostrophe	'	`\'`
Guillemet	"	`\"`
Caractère octal	0377	`\377`
Caractère hexadécimal	0xFF	`\xFF`
Caractère Unicode	0xFFFF	`\uFFFF`

Tableau 4.4 Caractères de contrôle

Chaînes de caractères

Les chaînes de caractères sont formées d'une suite de caractères entourée de guillemets (voir diagramme 4.7); elles sont associées au type *String*. En Java, le type *String* constitue une classe à part entière et ne doit pas être confondu avec un vecteur de caractères. Le tableau 4.5 montre différents exemples de chaînes de caractères et le résultat de leur impression.

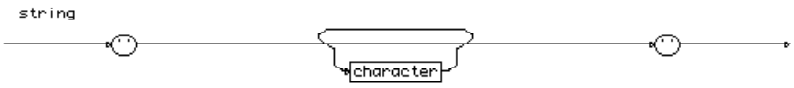


Diagramme 4.7 String

Chaîne de caractères	Résultat de l'impression
`""`	
`"\""`	"
`"texte sur une ligne"`	`texte sur une ligne`
`"texte sur \ndeux lignes"`	`texte sur` `deux lignes`
`"\ttabulé"`	`tabulé`

Tableau 4.5 Exemples de Strings

4.4 Déclaration des variables

L'utilisation d'une variable dans Java doit toujours être précédée par la déclaration de celle-ci. Le compilateur va utiliser les informations associées à la déclaration de la variable pour effectuer des vérifications : compatibilité de type dans les expressions, visibilité de la variable, etc. Cela peut sembler coûteux en temps, mais c'est une manière d'améliorer la qualité des programmes. En effet, le compilateur détectera des erreurs à la compilation qui, sinon, ne le seraient qu'au moment de l'exécution.

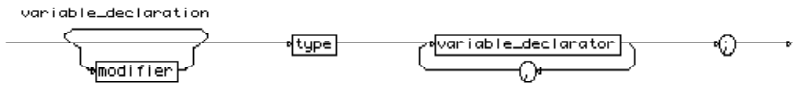


Diagramme 4.8 variable_declaration

La déclaration d'une variable (diagramme 4.8) s'effectue en préfixant le nom de la variable par son type (par exemple *int* pour un entier) :

```
int i;
```

La déclaration d'un ensemble de variables du même type s'effectue en préfixant le nom des variables par leur type :

```
int i,j,k;
```

Le diagramme 4.8montre également que l'on peut ajouter un modificateur *(modifier)* à la définition d'une variable. Par exemple, le modificateur *final* précise que la valeur de la variable ne pourra plus être changée après son affectation initiale : c'est ainsi que l'on définit une constante. Nous examinerons plus en détail ces différents modificateurs lorsque nous traiterons des classes (point 7.5, p. 100).

```
final int NbreDeRoues=4;
final float pi=3.14159;
```

Dans les exemples ci-dessus, on constate que l'on peut affecter une valeur à une variable lors de sa déclaration; le diagramme 4.9indique cette possibilité.

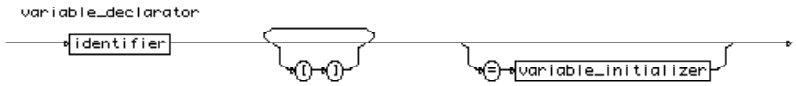


Diagramme 4.9 variable_declarator

Revenons sur la notion de type. Il existe deux catégories de types :

- les *types simples* comme les booléens, les entiers, etc., qui sont à considérer comme des types primitifs car ils ne sont pas construits à partir d'autres types;

- les *types composés* qui sont construits à partir d'autres types comme les vecteurs, matrices, classes ou interfaces.

Les diagrammes 4.10et 4.11illustrent la définition des types en Java.

Diagramme 4.10 type

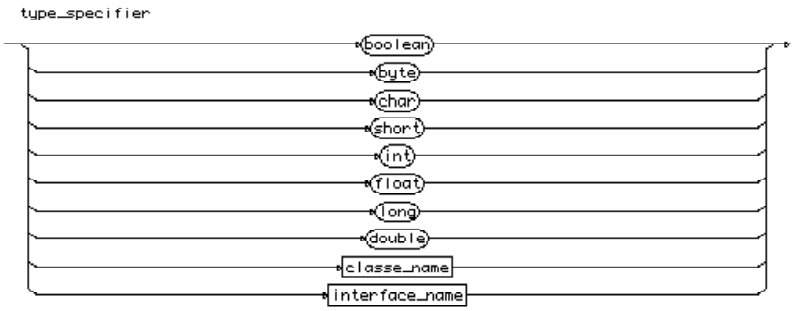


Diagramme 4.11 type_specifier

Nous allons maintenant examiner les types simples et le type vecteur.

Booléens

Une variable booléenne pourra contenir l'une des valeurs vrai ou faux. Cette variable pourra être assignée par les littéraux *true* ou *false* et elle pourra recevoir le résultat d'une expression logique :

```
boolean voitureArretee=true;
voitureArretee = (vitesse < 1);
```

Entiers

Il existe quatre types d'entiers : *byte, short, int* et *long*. Chacun diffère par la taille des entiers qu'il peut représenter, exprimée en nombre de bits (voir tableau 4.6).

Type	Nbre de bits	Exemple
byte	8	byte NbrEnfant;
short	16	short VolumeSon;
int	32	int i,j,k;
long	64	long detteSecu;

Tableau 4.6 Les types d'entiers

La déclaration d'une variable entière permet de représenter un nombre entier limité par le nombre de bits de sa représentation. Cette variable pourra être assignée par les littéraux entiers et recevoir le résultat d'une expression entière.

Flottants

Il existe deux types de flottants : *float* et *double*. Chacun diffère par la taille de sa représentation exprimée en nombre de bits (voir tableau 4.7).

Type	Nbre de bits	Exemple
`float`	32	`float ageMoyen;`
`double`	64	`double pi;`

Tableau 4.7 Les types de flottants

Une variable flottante pourra contenir un nombre flottant limité par le nombre de bits de sa représentation. Cette variable pourra être assignée par les littéraux flottants et recevoir le résultat d'une expression flottante.

Caractères

La déclaration d'une variable caractère permet de représenter une valeur définie sur l'ensemble des caractères Unicode. Cette variable pourra être assignée par des littéraux caractères et recevoir le résultat d'une expression caractère.

```
char separateur='\t';    // le séparateur est un tabulateur
```

Un caractère est représenté sur 16 bits en Java. Il sert à conserver la valeur d'un seul caractère. Les chaînes de caractères quant à elles sont représentées par la classe *String* (voir le point « Chaînes de caractères », p. 47).

Vecteurs et matrices

Les vecteurs, également appelés tableaux, sont déclarés en postfixant le type ou la variable par *[ ]* :

```
int i[ ];   // vecteur d'entiers
int[ ] j;   // autre notation pour le même type que i
char motsCroises[ ][ ];   // une matrice de caractères
```

On remarquera que les vecteurs ne sont pas contraints par des bornes au moment de leur déclaration. L'allocation de l'espace nécessaire au vecteur se fera explicitement au moyen d'une méthode *new* qui sera présentée dans le prochain chapitre (voir p. 61). On peut cependant allouer de l'espace au vecteur et l'initialiser lors de la déclaration :

```
int f[ ] = {1, 1, 2, 3*t, 5, 8, u-13, 21};
double[ ][ ] e1 = {{0.0, 1.0}, {-1.0, 0.0}};
```

4.5 Portée des variables

L'emplacement de la déclaration d'une variable définit l'espace de visibilité (la portée) de celle-ci. Ainsi, une variable est visible à l'intérieur du *bloc* où elle est déclarée.

Un *bloc* (voir p. 66) est défini comme un ensemble d'instructions comprises entre deux accolades *{ }*.

```
class Test { // debut de test
   public static void principal(String args[])
      { // debut de principal
        float x
        ...
       } // fin de principal
   public void methodeA ()
      { // debut de methodeA
        char c
        ...
       } // fin de methodeA
} // fin de principal
```

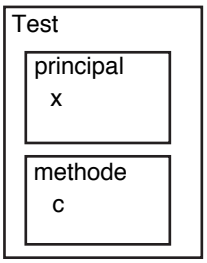


Figure 4.1 Visibilité d'une variable

Dans le programme *Test* ci-dessus, les variables *x* et *c* ne sont visibles que dans leur bloc respectif. Ainsi, dans le bloc de *methodeA*, il n'est pas possible d'accéder à la variable *x*. La figure 4.1 illustre l'espace de visibilité des variables *x* et *c*.

Examinons maintenant le cas des blocs imbriqués et celui de la redéfinition d'une variable. Lorsque des blocs sont imbriqués, les variables définies à un niveau supérieur sont visibles depuis les sous-blocs. Toutefois, si dans un sous-bloc on redéfinit une variable visible dans un bloc supérieur, cette nouvelle variable masque la variable supérieure à l'intérieur du sous-bloc (et de ses propres sous-blocs). Voyons cela à l'aide d'un exemple :

```
class Test { // debut de test
   public static void principal(String args[])
      { // debut de principal
        float echange; int a,b;
        ...
        if a < b then
          { int echange;
```

```
            echange=a;a=b;b=echange
          };
        } // fin de principal
    public void methodeA ()
} // fin de principal
```

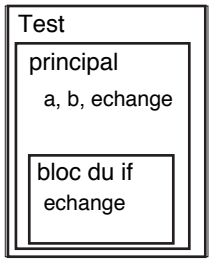


Figure 4.2 Visibilité d'une variable dans un sous-bloc

Dans le bloc du *if*, *a* et *b* référencent les variables du bloc *principal*, alors que *echange* référence la variable locale *echange* du bloc (voir figure 4.2).

La redéfinition des variables est destinée à des variables temporaires utilisées à l'intérieur d'un bloc, comme par exemple l'indice d'une boucle.

4.6 Opérateurs

Les opérateurs sont regroupés par types d'opérations : numériques, comparaisons, logiques, manipulations de chaînes de caractères, manipulations binaires. Leur classification se base sur le nombre d'opérandes qu'ils nécessitent : 1 (unaire), 2 (binaire) et même 3 (ternaire).

Précédence des opérateurs (de la plus grande à la plus petite) Les opérateurs de même niveau sont évalués depuis la gauche				
.	[]	()		
++	--	!	~	instanceof
*	/	%		
+	-			
<<	>>	>>>		
<	>	<=	>=	
==	!=			
&				
^				

Tableau 4.8 Précédence des opérateurs

Précédence des opérateurs (de la plus grande à la plus petite) Les opérateurs de même niveau sont évalués depuis la gauche				
&&				
||				
?:				
=	op=			
,				

Tableau 4.8 Précédence des opérateurs

Les expressions composées sont évaluées de la gauche vers la droite, une table de précédence (voir tableau 4.8) indiquant la priorité entre les opérations.

Prenons comme exemple l'expression suivante :

```
x=z+w-y/(3*y^2)
```

Pour cette expression, l'ordre de **lecture** (de la gauche vers la droite) est :

```
= + - / ()
```

La table de précédence nous donne l'ordre d'**exécution** suivant :

```
() / + - =
```

À l'intérieur des parenthèses *()*, nous avons * ^ par ordre de lecture de même que par ordre d'exécution.

Assignation

L'opérateur d'assignation est le symbole *"="*. L'assignation affecte l'expression placée à droite de l'opérateur à l'expression (une variable) placée à gauche de celui-ci. Il est intéressant de constater que, dans Java, l'assignation est considérée comme un opérateur binaire qui modifie son opérande gauche.

```
j=2;          // expression d'assignation d'un litéral
i=j*3;        // expression d'assignation d'une expression
i=j*3(j=2);   // expression valide combinant les deux formes
```

À noter que cette dernière expression a pour résultat : *i=6*.

Trois autres opérateurs ont des effets de bord : ~, ++ et --. L'utilisation combinée de ces opérateurs dans une **même** expression risque d'en rendre la lecture confuse. On essayera donc de décomposer les expressions afin qu'elles ne contiennent chacune qu'un seul opérateur avec effet de bord.

Expressions numériques

Les expresssions numériques sont classiques. Cependant quelques raccourcis syntaxiques sont à connaître pour alléger le code.

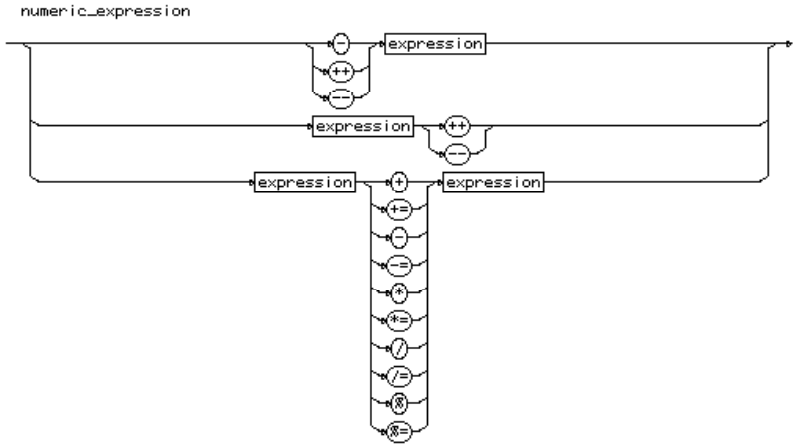


Diagramme 4.12 numeric_expression

Opérateur	Action	Exemple
-	Négation	`i=-j;`
++	Incrémentation de 1	`i++;`
--	Décrémentation de 1	`i--;`

Tableau 4.9 Opérateurs numériques unaires

Diagramme 4.13

On voit que ++ et – peuvent préfixer ou postfixer la variable. Exemple :

```
++i;  // est équivalent à i++;
```

Opérateur	Action	Exemple
+	Addition	`i=j+k`
+=		`i+=2; //i=i+2`
-	Soustraction	`i=j-k;`
-=		`i-=j;  //i=i-j`
*	Multiplication	`x=2*y`

Tableau 4.10 Opérateurs numériques binaires

Opérateur	Action	Exemple
*=		x*=x; //x=x*x
/	Division (tronquée si les arguments sont entiers)	i=j/k;
/=		x/=10; //x=x/10
%	Modulo	i=j%k;
%=		i%=2; //i=i%2

Tableau 4.10 Opérateurs numériques binaires

Pour les nombres entiers, les règles suivantes sont appliquées (voir diagramme 4.12, tableaux 4.9 et 4.10) :

- la division par *0* et le modulo par *0* génèrent une exception à l'exécution;
- les opérations dont la représentation dépasse la valeur maximum du type de l'entier débordent dans les entiers négatifs.

Pour les nombres flottants, les opérations ont la même sémantique :

- pour les incrémentations et décrémentations, on ajoute la valeur *1.0;*
- le % (modulo) appliqué à un flottant prend le sens de la division dans les entiers;
- la division par *0* et le modulo par *0* génèrent la valeur *inf;*
- les opérations dont la représentation dépasse la valeur maximum du type génèrent la valeur *inf* tandis que les débordements vers l'infiniment petit génèrent *0*.

Le programme *TestNumber* illustre le cas où l'on approche du maximum de la représentation pour les entiers et les flottants.

```
class TestNumber{
    public static void main (String args[]) {
        int i=1000, j=1000;
        float x=1, y=1;
        for (int k=0; k<100;k++) {
           i*=10;
           j/=10;
           x*=10000;
           y/=10000;
        System.out.println("\ni="+i+" j="+j+" x="+x+" y="+y);
        }
    }
}

i=10000          j=100    x=10000     y=0.0001
```

```
i=100000        j=10    x=1e+08    y=1e-08
i=1000000       j=1     x=1e+12    y=1e-12
i=10000000      j=0     x=1e+16    y=1e-16
i=100000000     j=0     x=1e+20    y=1e-20
i=1000000000    j=0     x=1e+24    y=1e-24
i=1410065408    j=0     x=1e+28    y=1e-28
i=1215752192    j=0     x=1e+32    y=1e-32
i=-727379968    j=0     x=1e+36    y=1e-36
i=1316134912    j=0     x=Inf      y=9.99995e-41
i=276447232     j=0     x=Inf      y=9.80909e-45
i=-1530494976   j=0     x=Inf      y=0
i=1874919424    j=0     x=Inf      y=0
```

4.7 Opérateurs relationnels

Les opérateurs relationnels permettent de tester tous les types de valeurs sur lesquels une relation d'ordre a été définie : les entiers, les réels, les caractères, etc. Le diagramme 4.14 et le tableau 4.11 décrivent la syntaxe et donnent un exemple de chacun des opérateurs relationnels.

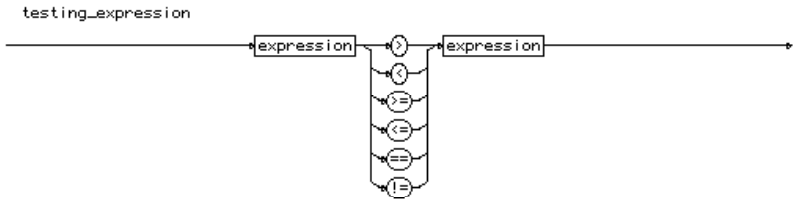


Diagramme 4.14 testing_expression

Opérateur	Action	Exemple
<	Plus petit que	`x<i;`
>	Plus grand que	`i>100;`
<=	Plus petit ou égal à	`j<=k;`
>=	Plus grand ou égal à	`c>='a';`
==	Egal à	`i==20;`
!=	Différent de	`c!='z';`

Tableau 4.11 Opérateurs relationnels

En ce qui concerne la relation d'ordre sur les nombres flottants, la prudence s'impose, car $x==y$ peut être vrai sans que $y<x \mid\mid y>x$ soit forcément vrai. Il faut également rester vigilant avec l'opérateur == qui peut être confondu dans un programme avec =. Mais, contrairement à ce qui se passe en langage C, cette erreur est détectée par le compilateur.

Opérateurs logiques

Les opérateurs logiques traitent des opérandes dont la valeur est booléenne. Les quatre opérateurs de base sont : la négation (*!*), le ET (*&*), le OU (*|*), le OU exclusif (*^*). Le tableau 4.12 présente ces opérateurs et le résultat de leur application sur des opérandes logiques (*p* et *q*). Le diagramme 4.15et le tableau 4.13 décrivent l'ensemble complet des opérateurs logiques.

p	q	!p	p & q	p | q	p ^ q
v	v	f	v	v	f
v	f	f	f	v	v
f	v	v	f	v	v
f	f	v	f	f	f

Tableau 4.12 Table de vérité des opérateurs logiques

Pour chaque opérateur logique (sauf pour la négation), on trouve une forme assignée où la variable affectée est aussi l'opérande de gauche de l'opérateur. Ainsi :

```
p&=(i=10); // est équivalent à p=p&(i=10)
```

On trouve aussi pour le ET et le OU une version avec évaluation raccourcie, dont les symboles sont respectivement *&&* et *||*. Dans ce cas, l'évaluation de l'expression logique est stoppée **dès** la détermination certaine de sa valeur de vérité. Ainsi, dans :

```
p1 && p2 && ... & pn
```

l'évaluation est stoppée dès qu'une expression *pi* s'évalue à *false*. De même, dans :

```
p1 || p2 || ... || pn
```

l'évaluation est stoppée dès qu'une expression *pi* retourne *true*.

Ce mécanisme permet aussi d'éviter d'évaluer des expressions qui pourraient générer une erreur, comme dans l'exemple suivant où une division par 0 est évitée :

```
(i!=0) && (x > ( y/i)); // y/i n'est pas évalué si i égale 0
```

Java possède également une expression ternaire de la forme :

```
p?e1:e2;
```

Dans ce cas, si *p* est vrai alors l'expression *e1* est évaluée, sinon *e2* est évaluée. Cette forme, du fait que l'assignation est considérée comme une expression, peut être utilisée comme une instruction *if*. Exemple :

```
p?e1:e2; // est équivalent à if (p) e1; else e2;
```

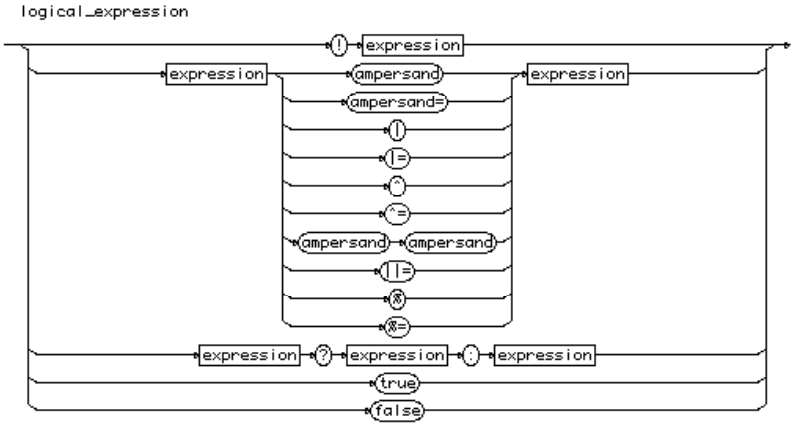


Diagramme 4.15 logical_expression

Opérateur	Action	Exemple
!	Négation	`!p;`
&	ET	`p & (i<10)`
|	OU	`p | q`
^	OU exclusif	`p ^ false`
&&	ET évalué	`p && q && r`
||	OU évalué	`p || q || r`
!=	Négation assignée	`p!=p;`
&=	ET assigné	`p&=q  // p= p & q`
|=	OU assigné	`p|=q  // p= p | q`
?:	Si alors sinon	`(i<10)?(j=2):(j=3)`

Tableau 4.13 Les opérateurs logiques

Enfin, ajoutons encore que les opérateurs *!, &, |,* et ^ sont étendus à des opérations de manipulation binaire sur les bits de la représentation des opérandes entiers. Ces opérateurs appliquent ainsi la table de vérité à chaque bit d'un entier, substituant *false* à 0 et *true* à 1.

Opérateurs sur les chaînes de caractères

La concaténation est l'opération que l'on effectue avec les chaînes de caractères. Cette opération ajoute la seconde chaîne à la fin de la première.

Les opérandes qui ne sont pas des chaînes de caractères sont automatiquement convertis.

La concaténation de chaînes de caractères est décrite par le diagramme 4.3 et le tableau 4.14. Nous avons déjà utilisé cet opérateur à plusieurs reprises dans des instructions d'impression telles que :

```
System.out.println("\ni="+i+" j="+j+" x="+x+" y="+y);
```

où les variables *i, j, x, y* sont automatiquement converties en chaînes de caractères.

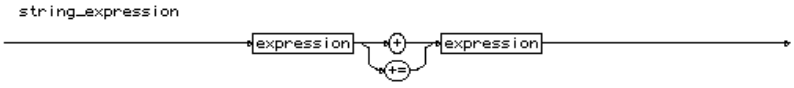


Diagramme 4.16 string_expression

Opérateur	Action	Exemple
+	Concaténation	`"conca"+"ténation";`
+=	Ajoute à la fin	`s+=" à la fin";`

Tableau 4.14 Opérateurs de concaténation

4.8 Opérateurs de manipulation binaire

En plus des manipulations directes réalisées par les opérateurs logiques (*!*, *&*, *|*, et *^*), il est possible d'effectuer des décalages (vers la droite ou la gauche) sur des représentations binaires (voir diagramme 4.17 et tableau 4.15). Il s'agit d'opérateurs à deux opérandes, le premier étant la valeur binaire sur lequel on effectue le décalage, le deuxième indiquant le nombre de décalages à effectuer. On a ainsi :

```
i>>k; // décaler i vers la droite de k bits avec son signe
i<<k; // décaler i vers la gauche de k bits
i>>>k; // décaler i vers la droite de k bits sans signe
```

Ces opérateurs ont une version assignée qui permet d'affecter directement le résultat d'un décalage effectué sur une variable à celle-ci.

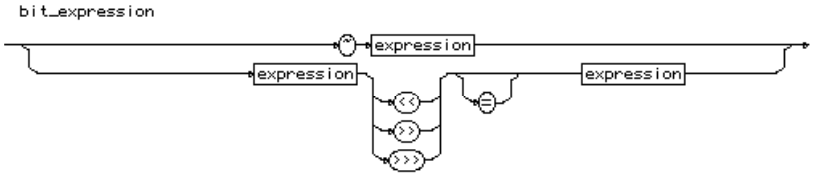


Diagramme 4.17 bit_expression

Opérateur	Action	Exemple
<<	Décalage à gauche	`i<<n;`
>>	Décalage à droite signé	`i>>4;`
>>>	Décalage à droite non signé	`i>>>2;`
<<=	Décalage à gauche assigné	`k<<=n;`
>>=	Décalage à droite signé assigné	`k>>=n;`
>>>=	Décalage à droite non signé assigné	`k>>>=n;`

Tableau 4.15 Opérateurs de manipulation binaire

Il est intéressant de noter les équivalences suivantes (tableau 4.16) entre les opérations de manipulation binaire et les opérations arithmétiques sur les entiers. Pour un nombre entier *i*, on a :

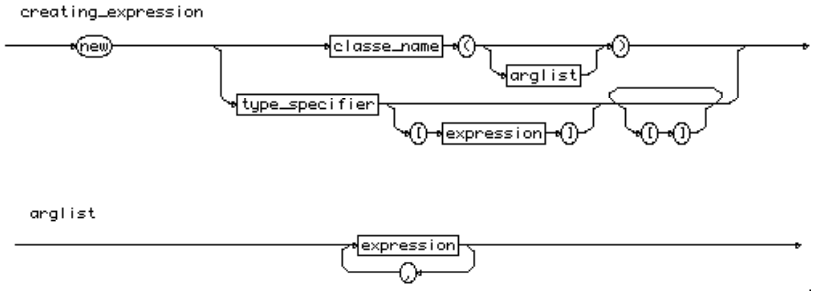
Opérateur binaire	Opérateur arithmétique	Condition
`~i`	`(-i)-1`	
`i>>k`	$i/2^k$	`si i>0`
`i<<k`	$i*2^k$	
`i>>>k`	$i/2^k$	`si i>0`
`i>>>k`	`(i<<k)+(2<<(L-k-1)` avec L=32 avec L=64	`si i<0` si i est int si i est long

Tableau 4.16 Correspondances entre opérateurs binaires et arithmétiques

4.9 Opérateur d'allocation

L'opérateur d'allocation *new* (voir diagramme 4.18) sert à la création dans deux cas distincts. Dans le premier, il s'agit de créer un objet, instance d'une classe spécifiée dans l'expression; nous traiterons ce cas lorsque nous approfondirons le concept de *classe* en Java (chapitre 6, page 77). Dans le second cas, il s'agit de créer et de fixer les dimensions de vecteurs ou de matrices.

Nous avons vu précédemment comment déclarer un vecteur (voir « Vecteurs et matrices », p. 50). Pour fixer la taille d'un vecteur, on utilise la syntaxe suivante :




```
int i[] = new int [100];       // i un vecteur de 100 entiers
int c[][] = new char[10][10];  // c une matrice de 10x10 caractères
```

Les indices des vecteurs et des matrices commencent à 0, ce qui signifie que la valeur maximum d'un indice sera *dimension du vecteur – 1* :

```
i[0] = 1;     // la première position de i est initialisée à 1
i[9] = 10;    // la dernière position de i est initialisée à 10
c[0][0]='a';  // première case de c
c[9][9]='z';  // dernière case de c
```

Pour un type ayant plusieurs dimensions, seule la première d'entre elles doit obligatoirement être fixée au moment de l'allocation :

```
int c[][] = new char[10][];  // c une matrice de 10 par ?
```

Les autres dimensions peuvent être déclarées ultérieurement :

```
c[0] = new char [24]; // première ligne de c = 24 caractères
```

Des dimensions différentes peuvent très bien exister dans la même variable :

```
c[1] = new char [12]; // deuxième ligne de c = 12 caractères
```

D'une manière générale, l'allocation d'une variable de dimension *n* peut être vue comme une série de *n-1* boucles imbriquées effectuant chacune une allocation :

```
int c[][] = new char[10][10];

// est équivalent à:

int c[][] = new char[10][]; // allocation de la matrice c (10 lignes)
for (int i=0; i< c.length; i++)
    c[i] = new char [10];  //allocation de chaque ligne de c
```

Diagramme 4.18 creating_expression

Nous reviendrons plus précisément sur les mécanismes d'allocation et sur la gestion de la mémoire en Java au point 6.2.

4.10 Conversions de types

Il arrive fréquemment que l'on ait besoin de la valeur d'un élément exprimée dans un type différent. On aimerait par exemple connaître la valeur entière d'un

caractère ou utiliser un entier en tant que flottant. Pour effectuer ce genre d'opérations, il faut faire appel à la conversion de types. Il y a deux catégories de conversions : les élargissements, qui conservent les valeurs, avec une éventuelle perte de précision et les rétrécissements, qui peuvent complètement modifier les valeurs.

Le rétrécissement se présente lorsqu'on convertit vers un type moins étendu que le type de départ. Par exemple, il est évident que les 64 bits d'un *long* ne pourront pas tenir dans les 8 bits d'un *byte*. Dans ce cas il y a troncature, seuls les 8 bits les moins significatifs sont conservés. Une conversion de rétrécissement doit être indiquée explicitement à l'aide de l'opérateur de « casting » *(<type>)*.

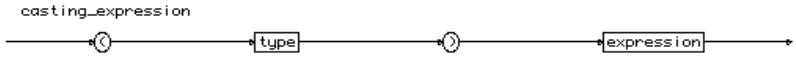


Diagramme 4.19 casting_expression

L'élargissement ne nécessite aucune notation particulière. L'élargissement d'un entier à un flottant peut faire perdre de la précision lorsque la mantisse du type flottant est plus courte que le nombre de bits de l'entier.

```
int i;
i == (int) ((float) i); // n'est pas toujours vrai
```

Voici quelques exemples de conversions :

```
// Rétrécissement avec troncation nécessaire
int a = 911000;
byte ba = (byte)a; // résultat: ba =-104
short sa = (short)a; // résultat: sa = -6504

// Rétrécissement sans troncation
a = 66;
char ca = (char)a; // résultat: ca = 'B' (66 = code Unicode de 'B')
ba = (byte)a; // résultat: ba = 66

// Rétrécissement d'un flottant
double d = 7.7e+204;
int a = (int)d; // résultat: a = 2147483647 (le plus grand entier)
d = 6789.6789;
a = (int)d; // résultat: a = 6789 (arrondi)

// Elargissement d'un entier
a = 2111222333;
float f = a; // résultat: f = 2.11122227e+9 (perte de précision)
double d = a; // résultat: d = 2.111222333e+9 (pas de perte de
précision)
```

Le tableau 4.17 ci-dessous énumère les conversions autorisées (oui), autorisées mais pouvant causer une perte de précision (perte) et celles nécessitant une conversion explicite ().

De \ à	byte	short	int	long	float	double	char
byte	oui	oui	oui	oui	oui	oui	oui
short	()	oui	ou	oui	oui	oui	()
int	()	()	oui	oui	perte	oui	()
long	()	()	()	oui	perte	perte	()
float	()	()	()	()	ou	oui	()
double	()	()	()	()	()	oui	()
char	()	()	oui	oui	oui	oui	oui

Tableau 4.17 Conversions entre types

Chapitre 5

Les structures de contrôle

Nous avons examiné les types, les variables et les expressions. Nous sommes maintenant capables d'énoncer une liste d'actions séquentielles à effectuer sur des variables. Un programme ne pouvant malheureusement pas se réduire à une séquence, il nous faut maintenant ajouter les différentes structures de contrôle : le *si_alors*, le *faire_tantque*, le *tantque_faire*, le *dans_le_cas*. Avant de poursuivre, le diagramme 5.1résume la syntaxe de toutes les instructions *(statements)*.

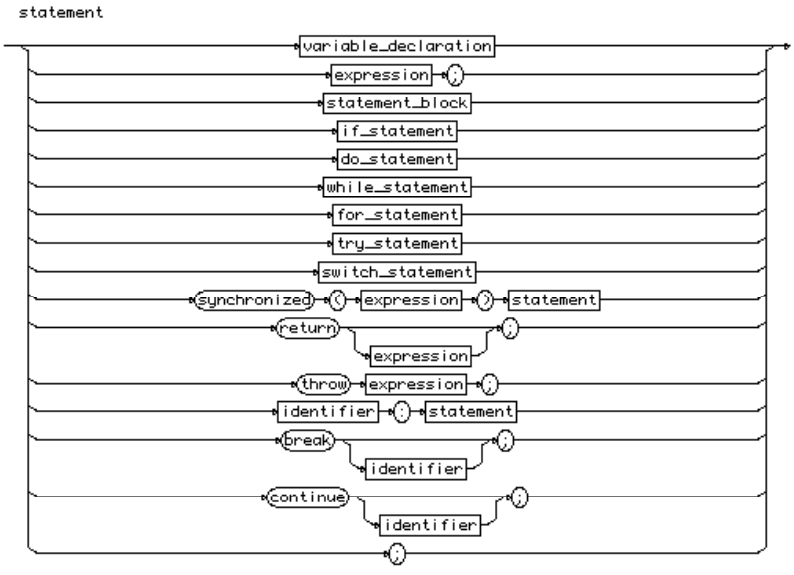


Diagramme 5.1 statement

5.1 Bloc d'instructions

Un bloc d'instructions est contenu entre accolades *{}* (voir diagramme 5.2). Il définit la visibilité des variables qui y sont déclarées. À chaque fois que dans une règle du langage Java on parle d'*instruction*, on sous-entend *bloc d'instructions* si l'on doit en décrire plusieurs.

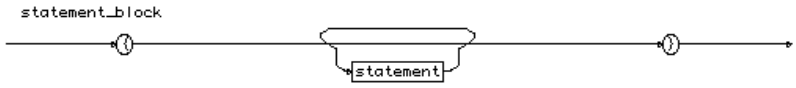


Diagramme 5.2 statement_block

5.2 Exécution conditionnelle

L'exécution conditionnelle d'instructions est la structure de base de la programmation. Il existe deux types d'exécution conditionnelle en Java :

- si_alors_sinon (*if then else*);
- dans_le_cas (*switch*).

Nous allons nous attarder quelque peu sur les différentes formes prises par ces expressions et les illustrer à l'aide d'exemples.

Si_alors_sinon (*if then else*)

Ce type d'exécution conditionnelle (voir diagramme 5.3) est le plus utilisé, permettant d'exprimer de nombreux cas d'alternatives. L'expression de test placée entre *()* doit toujours être une expression booléenne (s'évaluant à *true* ou *false*).

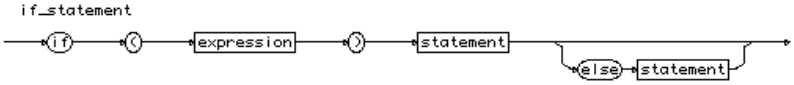


Diagramme 5.3 if_statement

Les deux formes suivantes peuvent être employées :

if (e) S1; l'instruction *S1* est exécutée si *e* est vraie :

```
if (i==1) s=" i est égal à 1 ";
```

if (e) S1 else S2; l'instruction *S1* est exécutée si *e* est vraie. L'instruction *S2* est exécutée si *e* est fausse:

```
if (i==1) s=" i est égal à 1 ";
else s=" i est différent de 1 ";
```

De ces deux formes, nous pouvons dériver :

if (e) {B1}; le bloc *B1* est exécuté si *e* est vraie :

```
if (i==1) {
   s=" i est égal à 1 ";
```

```
    i=j;
}
```

if (e) {B1} else {B2}; le bloc *B1* est exécuté si *e* est vraie. Le bloc *B2* est exécuté si *e* est fausse :

```
if (i==1) {
   s=" i est égal à 1 ";
   i=j;
}
else {
   s=" i est différent de 1 ";
   i=1515;
}
```

if (e1) S1 else if (e2) S2 ... else Sn; c'est une cascade de *si_alors*. L'instruction *Si* est exécutée si *ei* est vraie et que toutes les expressions testées auparavant sont fausses :

```
if (i==1)
    s=" i est égal à 1 ";
else if (i==2)
    s=" i est égal à 2 ";
else if (i==3)
    s=" i est égal à 3 ";
else s=" i est différent de 1,2 et 3 ";
```

Cette dernière forme nous amène à recommander une certaine prudence dans l'utilisation de *si_alors* en cascade. Ainsi, dans l'exemple ci-dessus, la dernière clause *else* se rapporte au dernier *si_alors*. Dans de tels cas, l'utilisation de *{}* est nécessaire pour bien marquer le début et la fin des instructions, comme le montre cet exemple :

```
i=0; j=3;
if (i==0)
   if (j==2)  s=" i est égal à 2 ";
else i=j;
System.out.println(i);
```

Que va imprimer le programme ci-dessus : 0 ou 3?

Il va imprimer 3, car *i==0* et *j!=2*, donc on exécute *i=j*. Le *else* se rapporte en effet au *if* le plus imbriqué.

Au cas où (*switch*)

L'instruction *switch* se comporte comme un aiguillage, exécutant des instructions différentes (clauses *case*) suivant le résultat d'une expression. Pour cela, elle commence par évaluer l'expression qui lui est liée puis elle compare la valeur retournée avec celle de chaque expression liée aux *case*. Dans le cas où il y a égalité, elle commence à exécuter le code lié à ce *case*, puis exécute toutes les instructions des cas suivants.

Si l'on désire isoler chaque clause *case*, il faut terminer son code avec un *break*. Si aucun cas d'égalité n'est trouvé et qu'une clause *default* est spécifiée, alors on exécute cette clause. Du point de vue syntaxique, les symboles *case* et *default* sont considérés comme des étiquettes (voir diagramme 5.4).

Examinons la forme générale de *switch* sans *break* :

```
switch (e) {
case c1: S1;
case c2: S2;
...
case cn: Sn;
default: Sd
};
```

Si l'expression *e* est évaluée à la valeur *ci*, alors les instructions *Si, Si+1, ... Sn, Sd* sont exécutées. Si l'expression *e* est différente de tous les *ci*, alors on exécute l'expression *Sd*.

Examinons la forme générale de *switch* avec *break* :

```
switch (e) {
case c1: S1 break;
case c2: S2 break;
...
case cn: Sn break;
default: Sd
};
```

Si l'expression *e* est évaluée à la valeur *ci,* alors l'instruction *Si* est exécutée. Si l'expression *e* est différente de tous les *ci,* alors on exécute l'expression *Sd*.

```
class TestSwitch{
    public static void main (String args[]) {
        int i=3;
        switch (i) {
          case 1: System.out.println("I"); break;
          case 2: System.out.println("II"); break;
          case 3: System.out.println("III"); break;
          case 4: System.out.println("IV"); break;
          case 5: System.out.println("V"); break;
          default: System.out.println("pas de traduction");
          }
    }}
```

Il faut encore préciser que le type de l'expression d'une clause *case* doit être *char, byte, short* ou *int*.

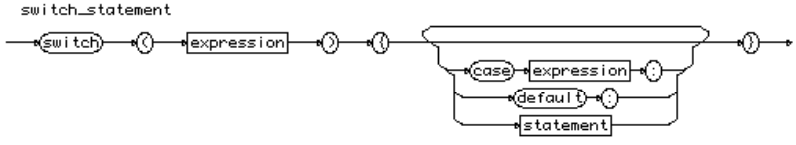


Diagramme 5.4 switch_statement

5.3 Exécution itérative

Les structures de contrôle itératives permettent d'exécuter des instructions de manière répétitive (boucle), l'exécution se terminant lorsqu'une condition de test n'est plus remplie. Java définit les structures itératives *while*, *do...while* et *for*, présentées ci-après.

Tant que_faire_ (*while*)

Dans ce cas (voir diagramme 5.5), on ne commence la boucle que si la condition est vraie; on exécute alors l'instruction liée au *while*. Si la condition est toujours vraie, on recommence la boucle, le processus s'arrêtant dès que la condition devient fausse. L'instruction liée au *while* peut donc ne jamais être exécutée, si la condition est fausse au départ. À noter encore que la condition doit être de type booléen. Deux formes de *while* peuvent être utilisées :

- *while (c) S1;* tant que *c* est vrai on exécute *S1;*
- *while (c) {B1};* tant que *c* est vrai on exécute le bloc *B1*.

```
class TestWhile{
    public static void main (String args[]) {
        int i=100, somme_i=0, j=0;
        // boucle 1: expression sans effet de bord
        while (j<=i) {
           somme_i+=j;
           ++j;
          }
        System.out.println("boucle  1:"+somme_i);
        // boucle 2: expression avec  effet de bord
        somme_i=0; j=0;
        while (++j<=i ) somme_i+=j;
        System.out.println("boucle  2:"+somme_i);
     }
}
```

Diagramme 5.5 while_statement

Faire_tant que_ (*do...while*)

Dans ce type de boucle, on commence par exécuter l'instruction liée au *do*. Ensuite, si la condition est vraie, on recommence la boucle. L'exécution est interrompue dès que la condition devient fausse. L'instruction liée au *do* est donc exécutée au minimum une fois, même si la condition est fausse au départ. La condition doit par ailleurs être du type booléen. Les deux formes de *do...while* sont :

- *do S1 while(c);* exécuter *S1* et répéter *S1* tant que *c* est vrai;

- *do {B1} while(c);* exécuter *B1* et répéter *B1* tant que *c* est vrai.

```
class TestDo{
    /* le résultat n'est pas erroné si l'on exécute
       une addition de trop si i= 0 ! */
    public static void main (String args[]) {
        int i=100, somme_i=0, j=0;
        // boucle 1: expression sans effet de bord
        do {
           somme_i+=j;
           ++j;
         } while (j<=i);
        System.out.println("boucle  1:"+somme_i);
        // boucle 2: expression avec  effet de bord
        somme_i=0; j=0;
        do somme_i+=j; while (++j<=i );
          System.out.println("boucle  2:"+somme_i);
     }
}
```

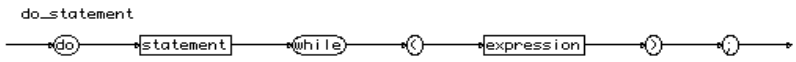


Diagramme 5.6 do_statement

Pour_faire_ (*for*)

Une boucle *for* est déclarée à l'aide d'une ou plusieurs instructions (*S1, B1*) et de trois expressions (*e1, e2, e3*) :

- la première (*e1*) initialise l'itérateur (après l'avoir éventuellement déclaré); cette expression n'est exécutée qu'une seule fois;
- la deuxième (*e2*) exprime la condition de test (tant qu'elle est vraie on continue la boucle); elle est vérifiée à chaque itération;
- la troisième (*e3*) modifie l'itérateur; elle est exécutée à chaque itération, à la suite de l'instruction (*S1*).

Le diagramme 5.7précise la syntaxe, dont les deux formes sont :

- *for (e1;e2;e3) S1;* exécuter *e1*. Tant que *e2* est vrai, exécuter l'instruction *S1* **et** l'expression *e3;*
- *for (e1;e2;e3) B1;* exécuter *e1*. Tant que *e2* est vrai, exécuter le bloc *B1* **et** l'expression *e3*.

Les expressions *e2* et *e3* sont optionnelles. Si elles ne sont pas précisées, le bloc *B1* doit alors contenir des instructions pour modifier l'itérateur et une instruction *break* pour quitter la boucle.

D'une manière générale, une boucle *for* est équivalente à la boucle *while* suivante : *e1; while(e2) {S1; e3}*.

Exemples de boucles *for* :

```
class TestFor{
   public static void main (String args[]) {
        int n[] = new int[100];
        // boucle 1: calculer la somme des entiers pour
        // indice du vecteur
        for (int i=1; i<100; i++) n[i]=n[i-1]+i;
        // boucle 2: imprimer le résultat  par
        // ordre décroissant
        for (int i=99; i>=0;) {
            // modification de i dans le bloc d'instructions
            System.out.println("somme("+i+")="+n[i]);
            i--;
        }
    }
}
```

Dans l'exemple *TestFor*, nous avons deux types de boucles : l'une croissante, l'autre décroissante. Dans la seconde, on remarque que l'itérateur est décrémenté en dehors de l'expression de modification (*e3*), qui est vide. Ce mécanisme permet de modifier l'itérateur de manière complexe depuis l'intérieur du bloc d'instructions.

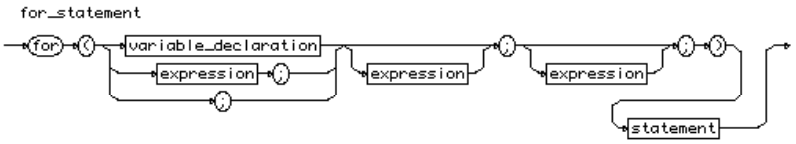


Diagramme 5.7 for_statement

5.4 Identifier une instruction

Il est possible d'attribuer un label (*identifier*, diagramme 5.8) à une instruction ou à un bloc d'instructions afin de l'identifier. Cette possibilité est utilisée en conjonction avec les instructions de rupture (*break*) et de continuation d'itération (*continue*). Exemple de label (*boucle1*) assigné à une boucle :

```
class LabelInst{
   public static void main (String args[]) {
      boucle1:
         for (int i=0;i<10;i++)
            for (int j=0; j<10;j++){
               System.out.println(i+"/"+j);
            }
   }
}
```

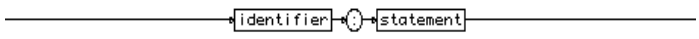


Diagramme 5.8 identifier:statement

5.5 Rupture

L'instruction *break*, déjà rencontrée dans l'instruction *switch* (voir le point « Au cas où (switch) », p. 67), peut également être utilisée dans les itérateurs *while*, *do* et *for* : elle ordonne alors de quitter la boucle dans laquelle elle est utilisée. Il est même possible de remonter de plusieurs niveaux en accompagnant l'instruction *break* du libellé identifiant le bloc d'instructions à quitter.

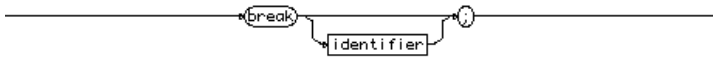


Diagramme 5.9 break

```
class TestBreak{
   public static void main (String args[]) {
      bloc1:
         for (int i=0;i<10;i++)
            for (int j=0; j<10;j++){
               System.out.println(i+"/"+j);
               // quitte la boucle locale
               if (j==1) break;
               // quitte la boucle for i ...
               if (i==2) break bloc1;
            }
   }
}
```

Exécution du programme *TestBreak* :

```
0/0
0/1
1/0
1/1
2/0
```

Dans l'exemple *TestBreak*, le premier *break* (sans libellé) force le programme à quitter la boucle locale (*j*) tandis que le second (*break bloc1*) le force à quitter le bloc complet identifié par le libellé *bloc1*.

Dans l'exemple *TestBreak2* ci-dessous, l'instruction d'affichage sera ignorée car elle se trouve dans le bloc d'instructions identifié par le libellé *bloc2* (et non dans la boucle elle-même!) :

```
class TestBreak2{
   public static void main (String args[]) {
      bloc2:{
         for (int i=0;i<10;i++)
            if (i==2) break bloc2;
         System.out.println("ne sera pas écrit");
      }
   }
}
```

5.6 Continuation

L'instruction *continue* est utilisée dans les itérateurs *while*, *do* et *for*. Elle provoque l'interruption du déroulement normal de la boucle, le contrôle étant repris après la dernière instruction de celle-ci (on recommence donc une nouvelle itération). Comme pour la rupture, il est possible d'indiquer un libellé : dans ce cas, le contrôle est repris dans la boucle dépendant immédiatement de ce libellé, comme le montre l'exemple *TestContinue* :

```
class TestContinue{
   public static void main (String args[]) {
      bloc3:
         for (int i=0;i<10;i++)
            for (int j=0; j<10;j++){
               // quitte la boucle locale
               if (j>1) continue;
               // quitte la boucle for i ...
               if (i>2) continue bloc3;
               System.out.println(i+"/"+j);
            }
       System.out.println("en dehors des boucles");
  }
}
```

Exécution du programme *TestContinue* :

```
0/0
0/1
1/0
1/1
2/0
2/1
en dehors des boucles
```

Malgré leur apparente ressemblance, les instructions *break* et *continue* sont très différentes : dans le cas du *break*, on **quitte** réellement la boucle, alors que dans le cas du *continue,* on saute seulement le bloc d'instructions et on **continue** quand même à exécuter les itérations.

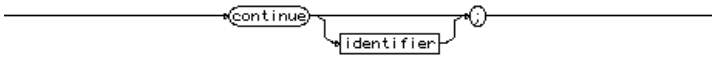


Diagramme 5.10 continue

5.7 Exemples d'algorithmes en Java

Nos connaissances actuelles devraient être suffisantes pour effectuer des calculs ou programmer des procédures algorithmiques telles que des tris. Nous vous proposons donc deux petits exemples destinés à utiliser nos acquis en matière de syntaxe Java.

Recherche d'intervalles sans nombre premier

Le programme *TestPremier* recherche les plus grands intervalles (suites de nombres croissants entre 1 et un million) dans lesquels ne figure aucun nombre premier (exemple : entre 7 et 11, nous avons un « trou » de 4). Afin d'accélérer le processus, on stocke les nombres premiers de 1 à 1 100 (*boucle1*), avant de rechercher les intervalles (*boucle2*).

Dans la *boucle1*, on cherche les nombres premiers jusqu'à 1 100 et on les stocke dans un vecteur *p*. Quelques remarques à propos de cet exemple :

- *i+=2;* permet une incrémentation par 2;
- *j< i / j;* la condition d'arrêt évolue avec l'indice;
- *continue boucle1;* interrompt les tests dès qu'un diviseur de *i* est trouvé;
- seuls sont conservés les nombres n'ayant pas quitté la boucle prématurément (en d'autres termes, ceux ayant subi le test avec succès).

Dans la *boucle2*, on cherche les intervalles les plus grands sans nombre premier ; on ne retient un intervalle que s'il est plus long que ses prédécesseurs :

- la boucle portant sur l'indice *j* teste la primalité d'un nombre en utilisant uniquement les nombres stockés dans le vecteur *p* (crible d'Ératosthène);
- *p[j]<i/p[j];* la condition d'arrêt ne porte pas directement sur l'indice *j* mais sur les valeurs référencées;
- *continue boucle2;* on quitte la boucle de test si l'on trouve un diviseur;
- quand on trouve un nombre premier, on teste sa distance avec le précédent. Si elle est plus grande que la dernière trouvée, alors on l'imprime.

```
class TestPremier{
   public static void main (String args[]) {
   int p[]=new int[200];
   int dernierP=1;
      p[0]=2;
      p[1]=3;
      boucle1:
         for (int i=5;i<1100;i+=2){
            for (int j=3; j<=i/j;j+=2)
               if ((i%j)==0) continue boucle1;
            dernierP++;
            p[dernierP]=i;
         }
      System.out.println("P entre 2 et 1100: "+dernierP);

   int dp=11, trou=4;
      boucle2:
         for (int i=13;i<1000000;i+=2){
            for (int j=1; p[j]<i/p[j];j++)
               if ((i%p[j])==0) continue boucle2;
```

```
            if (i-dp>trou) {
               trou=i-dp;
               System.out.println(dp+".."+i+" = "+trou);
            }
            dp=i;
         }
      System.out.println("fin!");
      }
}
```

Exécution du programme *TestPremier* :

```
P entre 2 et 1100: 193
31..37 = 6
89..97 = 8
139..149 = 10
199..211 = 12
293..307 = 14
887..907 = 20
1129..1151 = 22
1327..1361 = 34
9551..9587 = 36
15683..15727 = 44
19609..19661 = 52
31397..31469 = 72
155921..156007 = 86
360653..360749 = 96
370261..370373 = 112
492113..492227 = 114
fin!
```

Recherche de nombres amicaux

Ce deuxième exemple (*TestAmicaux*) recherche les nombres amicaux (nombres dont la somme des diviseurs de l'un est égale à l'autre et réciproquement). Exemple : les nombres 220 et 284 sont amicaux car :

- somme_div (220) = 1+2+4+5+10+11+20+22+44+55+110 = 284;
- somme_div (284) = 1+2+4+71+142 = 220.

Dans la *boucle1*, on calcule (pour les nombres jusqu'à 10000) la valeur de la somme de leurs diviseurs que l'on mémorise dans un vecteur *p*. Le code est très semblable à celui de la recherche des nombres premiers, sauf qu'ici nous désirons conserver la valeur des diviseurs.

Dans la *boucle2*, on cherche les nombres amicaux. On remarquera :

- *((p[i]<10000)&&(p[p[i]]==i))*; la condition du test avec abandon : cela permet de tester si la somme des diviseurs est plus grande que 9999; si c'est le cas, on n'effectue pas le test d'amitié entre les nombres (l'utilisation de && permet ainsi d'éviter de sortir des valeurs autorisées pour le vecteur *p)*.

```
class TestAmicaux{
   public static void main (String args[]) {
  int p[]=new int[10000];
  int dernierP=1;
      p[1]=1;p[2]=1;
      p[3]=1;p[4]=3;
      p[5]=1;p[6]=6;
         // boucle 1
         for (int i=7;i<10000;i++)
            for (int j=1; j<=i/2;j++)
               if ((i%j)==0) p[i]+=j;
         // boucle 2
         for (int i=2;i<10000;i++)
               if ((p[i]<10000)&&(p[p[i]]==i))
       System.out.println(i+" "+p[i]);
   System.out.println("fin!");
  }
}
```

Exécution du programme *TestAmicaux* :

```
6 6
28 28
220 284
284 220
496 496
1184 1210
1210 1184
2620 2924
2924 2620
5020 5564
5564 5020
6232 6368
6368 6232
8128 8128
fin!
```

Les nombres qui sont amicaux avec eux-mêmes sont appelés « nombres parfaits ». Parmi eux on peut citer : 6 et 28.

Chapitre 6

Une introduction aux objets

Nous avons examiné jusqu'ici la partie syntaxique de Java, qui permet d'exprimer des algorithmes. Avant d'aborder celle autorisant la déclaration des classes et des méthodes, il nous faut découvrir ce qu'est un **objet**.

6.1 Comportement des objets

Pour bien comprendre la notion d'objet, une vision basée sur le comportement peut se révéler utile. Dans cet esprit, nous obtenons les définitions suivantes :

- une **classe** est un archétype qui conditionne tous les comportements;
- un **objet** est une instance d'une et une seule classe. C'est un individu qui possède tous les comportements de sa classe;
- une **méthode** définit une action élémentaire que l'on peut effectuer sur un objet. L'ensemble des méthodes d'une classe définit le comportement de chaque objet de la classe;
- un **message** est une demande d'exécution d'une méthode à un objet.

Dans la figure 6.1 ci-après, on distingue deux classes, l'une définissant les rectangles et l'autre les disques. La classe sert de moule pour former ses instances. Ainsi, les objets *r1*, *r2* et *r3* sont des instances de *Rectangle* tandis que *d1* et *d2* sont des instances de *Disque*. Les méthodes définies dans les classes indiquent les services que l'on peut demander aux objets de ces classes.

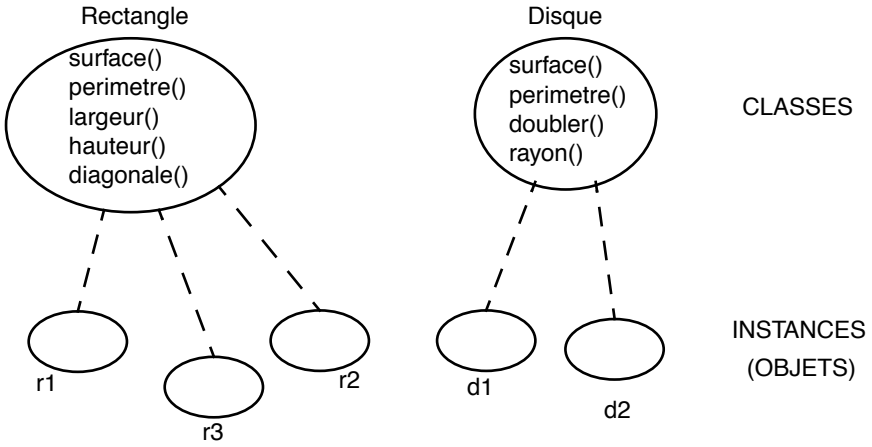


Figure 6.1 Classes, méthodes et instances

6.2 Classes et objets en Java

En Java, l'énoncé suivant permet de déclarer une classe :

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

Voyons ce qu'on trouve à l'intérieur d'une déclaration de classe.

Variables et méthodes d'instance

Une classe peut posséder des **variables d'instance**. Comme leur nom l'indique, les variables d'instance sont propres à chaque instance (objet) de la classe. Elles constituent la « mémoire » de chaque objet. Dans notre cas, *largeur* et *hauteur* peuvent être déclarées comme variables d'instance de la classe *Rectangle*, qui servira ensuite de moule pour créer nos objets :

```
class Rectangle {
   double largeur, hauteur;
   ...
}
```

Les **méthodes** de *Rectangle* déterminent le comportement des objets de ce type. La déclaration d'une méthode est composée des éléments suivants :

- le type du résultat produit, ou *void* si elle ne produit pas de résultat;
- le nom de la méthode;
- le type et le nom des paramètres, placés entre ();
- un bloc d'instructions qui constitue le corps de la méthode.

```
class Rectangle extends Object{
   ...
```

```
   double perimetre() {return 2*(largeur+hauteur);}
   double surface() {return largeur*hauteur;}
   double diagonale() {
      return Math.sqrt(largeur*largeur+hauteur*hauteur);}
   void doubler() {largeur*=2; hauteur*=2;}
}
```

On remarquera l'accès systématique aux variables d'instance dont chaque *Rectangle* est muni et qui reflètent son état (sa *largeur* et sa *hauteur*). L'instruction *return* calcule le résultat et termine l'exécution de la méthode.

Le type du résultat, le nom de la méthode et le nom et le type des paramètres forment ce qu'on appelle la **signature** de la méthode.

Dans le bloc d'instructions d'une méthode, on peut utiliser la pseudovariable **this** pour désigner l'objet courant, c'est-à-dire l'objet sur lequel la méthode est en train de travailler.

Constructeurs

Finalement, il ne nous reste plus qu'à définir le constructeur de la classe : une méthode particulière qui indique comment initialiser une instance. La règle de nommage d'un constructeur est simple, il doit porter le même nom que la classe concernée :

```
class Rectangle {
   ...
Rectangle(double initL, double initH){
     largeur=initL;
     hauteur=initH;
   }
   ...
}
```

Ce constructeur initialise les deux variables d'instance du nouvel objet avec les valeurs des deux paramètres *initL* et *initH*.

Voici donc le code complet de notre classe *Rectangle* :

```
class Rectangle {
   double largeur, hauteur;

   Rectangle(double initL, double initH){
     largeur=initL;
     hauteur=initH;
   }
   double perimetre() {return 2*(largeur+hauteur);}
   double surface() {return largeur*hauteur;}
   double diagonale() {
      return Math.sqrt(largeur*largeur+hauteur*hauteur);}
   void doubler() {largeur*=2; hauteur*=2;}
}
```

Nous procédons de la même manière pour la classe *Disque* (variables d'instance, constructeur, méthodes) et obtenons le code suivant :

```
class Disque {
   double diametre;
   static final double pi=3.14159;

   Disque(double initD){
      diametre=initD;
   }
   double perimetre() {return pi*diametre;}
   double surface() {return (pi*diametre*diametre)/4;}
   double rayon() {return diametre/2;}
   void doubler() {diametre*=2;}
}
```

Notons encore que si aucun constructeur n'est défini, Java crée un constructeur qui n'a aucun paramètre et n'initialise aucune variable d'instance.

Création d'instances et assignation

Nous venons de déclarer les classes *Rectangle* et *Disque*, mais nous n'avons pas encore créé d'instances de ces classes (les fameux *objets*). Pour créer une instance, on invoque un constructeur à l'aide de l'instruction *new*. Par exemple :

```
new Disque(2.5)
```

crée un objet de la classe *Disque* et invoque le constructeur qui initialise la variable *diametre* à 2.5.

On peut assigner un objet à une variable : pour cela il faut déclarer une variable du type correspondant. La déclaration de type s'effectue comme pour les types de base (*int*, *char*), en faisant précéder la variable par le nom de la classe :

```
Disque d1,d2;       // déclaration de 2 variables de type Disque
d1=new Disque(2);   // invoque le constructeur et assigne l'objet
d2=new Disque(4);
```

Invocation des méthodes

L'invocation d'une méthode s'effectue en nommant l'instance et en la faisant suivre du nom de la méthode et d'une liste, éventuellement vide, d'expressions :

```
instance.méthode(expression1, ...)
```

Ainsi, pour obtenir la diagonale du *Rectangle r1*, nous écrirons :

```
r1.diagonale()
```

Si la méthode possède des paramètres, les expressions données lors de l'invocation (arguments) sont évaluées et affectées aux paramètres correspondants de la méthode.

Si l'on veut invoquer une méthode sur l'instance courante, on écrira :

```
this.méthode(expression1, ...)
```

qui peut s'abréger en :

```
méthode(expression1, ...)
```

Un programme complet

Nous pouvons écrire maintenant un petit programme *(TestRectDisk1)* utilisant les classes *Rectangle* et *Disque* (on notera au passage que les noms commençant par une majuscule sont réservés, par convention, aux identifiants des classes, ceux des méthodes commençant par une minuscule).

```
class TestRectDisk1{
   public static void main (String args[]) {

      Rectangle r1=new Rectangle(2,4);
      Rectangle r2=new Rectangle(3,4);
      System.out.println("diagonale de r1: "+r1.diagonale());
      System.out.println("périmètre de r2: "+r2.perimetre());
      r2.doubler();
      System.out.println("périmètre de r2: "+r2.perimetre());

      Disque d1,d2;
      d1=new Disque(2);
      d2=new Disque(4);
      System.out.println("rayon de d1: "+d1.rayon());
      System.out.println("périmètre de d2: "+d2.perimetre());
      d2.doubler();
      System.out.println("périmètre de d2: "+d2.perimetre());
   }
}
```

Exécution du programme *TestRectDisk1* :

```
diagonale de r1: 4.47214
périmètre de r2: 14
périmètre de r2: 28
rayon de d1: 1
périmètre de d2: 12.5664
périmètre de d2: 25.1327
```

Déclaration de plusieurs constructeurs

Il est pratique de proposer plusieurs constructeurs pour une même classe (afin de permettre différentes formes de représentation selon les utilisateurs). Il suffit pour cela de créer un nouveau constructeur, devant obligatoirement se distinguer des autres par le nombre et/ou le type de ses paramètres.

Nous avons ainsi ajouté un constructeur à la classe *Rectangle* qui, à partir des coordonnées du coin supérieur gauche *(csgx,csgy)* et du coin inférieur droit *(cidx,cidy)*, détermine la *largeur* et la *hauteur* de l'instance à créer. Nous obtenons le code suivant :

```
class Rectangle {
```

```
    ...
     Rectangle(double initL, double initH){
       largeur=initL;
       hauteur=initH;
     }

     Rectangle(double csgx, double csgy,
               double cidx, double cidy){
       largeur=Math.abs(csgx-cidx);
       hauteur=Math.abs(csgy-cidy);
     }
     ...
}
```

Nous pouvons maintenant choisir comment créer des instances de *Rectangle* :

```
class TestRectDisk2{
   public static void main (String args[]) {

      Rectangle r1=new Rectangle(1,1,3,5);// par les coins
      Rectangle r2=new Rectangle(3,4);    // par largeur et hauteur
   ...
}
```

Si une classe possède plusieurs constructeurs, ceux-ci peuvent s'invoquer les uns les autres grâce à l'instruction *this(...)*. Par exemple :

```
     Rectangle(double initLH){
       this(initLH, initLH);
```

qui invoque *Rectangle(double initL, double initH).*

Destruction d'objets

Il n'y a en général pas besoin de s'occuper de détruire les objets car la machine virtuelle Java s'en charge. Dès qu'un objet n'est plus référencé par aucune variable ou aucun autre objet, le gestionnaire de mémoire le détruit et récupère l'espace mémoire qu'il occupait. Cependant, dans le cas où l'objet avait, au travers de ses méthodes, ouvert des canaux de communication ou des fichiers, il faut les fermer explicitement. C'est pourquoi le système invoque la méthode *finalize()*, si elle existe, juste avant la destruction de l'objet. On peut donc placer dans cette méthode le code nécessaire à la restitution des ressources occupées par l'objet.

Surcharge des méthodes

On aimerait parfois pouvoir proposer plusieurs méthodes dans une même classe effectuant une action similaire, sans avoir à leur donner des noms différents. Pour cela, on peut créer une nouvelle méthode portant le même nom qu'une autre, pour autant qu'elle s'en distingue par le nombre et/ou le type de ses paramètres.

Nous avons ainsi ajouté une méthode *doubler()* à la classe *Rectangle* permettant

de doubler la *largeur* et la *hauteur* de l'instance de manière indépendante. Nous obtenons le code suivant :

```
class Rectangle extends Object{
   ...
   void doubler() {
      largeur*=2;
      hauteur*=2;}

   void doubler(boolean l, boolean h) {
      if (l) largeur*=2;
      if (h) hauteur*=2;
   }
}
```

Nous pouvons alors doubler les instances de *Rectangle* selon nos besoins :

```
class TestRectDisk2{
   public static void main (String args[]) {

      r2.doubler();            // largeur et hauteur
      r2.doubler(false,true);  // uniquement la hauteur
}
```

On parle dans un tel cas de **surcharge** des méthodes. Le compilateur examine les types de paramètres et leur nombre au moment de la déclaration de l'appel, afin de déterminer la « bonne » méthode à invoquer. L'exemple suivant montre un autre cas de surcharge vu précédemment :

```
      r2.doubler(); // r2 est un Rectangle
      d2.doubler(); // d2 est un Disque
```

Dans ce cas, le compilateur sait (grâce à la déclaration de type des variables) que pour *r2*, il doit invoquer la méthode *doubler()* de la classe *Rectangle* alors que pour *d2*, il doit invoquer celle de la classe *Disque*.

Variables de classe

Une classe peut également posséder ses propres *variables de classe* (qui ne seront pas créées dans ses instances), lui permettant de gérer des informations à propos de ses instances, comme par exemple la hauteur moyenne des *Rectangles*. Les variables de classe sont déclarées statiques *(static)*. Elles sont initialisées au moment du chargement de la classe.

La constante *pi,* déclarée *static* dans la classe *Disque*, est un autre exemple de variable de classe. En effet, il n'est pas nécessaire (ni même utile) que chaque instance de *Disque* possède sa propre copie de la variable *pi*.

Visibilité des variables d'instance et de classe

En ce qui concerne la visibilité, les variables d'une instance sont accessibles :

- sans être préfixées, depuis toutes les méthodes de la classe;

- en étant préfixées par le nom de l'objet, depuis l'extérieur de la classe.

Ainsi, l'instruction suivante imprimera les dimensions de *r2* :

```
System.out.println("dimension:"+r2.largeur+"/"+r2.hauteur);
r2.hauteur=45;
```

Il est bien entendu possible de « cacher » les variables d'instance (surtout pour éviter qu'on ne les modifie de l'extérieur) grâce à deux modificateurs, *private* et *protected*, que nous étudierons plus en détail au point 24.2.

Les variables de classe, quant à elles, sont accessibles depuis l'extérieur de la classe en les préfixant du nom de la classe. Le fameux *System.out* n'est autre que la variable de classe *out* de la classe *System* et *System.out.println(...)* invoque la méthode *println()* de cet objet.

Méthodes de classe

Les méthodes de classe sont destinées à agir sur la classe plutôt que sur ses instances : elles permettent de manipuler les variables de classe. Nous avons ainsi ajouté à notre exemple une méthode de classe *modifierEchelle()* que nous devons déclarer *static*. L'appel à cette méthode se fera en la préfixant par le nom de la classe.

Lorsque l'on désire effectuer des initialisations complexes des variables de la classe, on peut lui associer un bloc d'instructions qui sera exécuté lors du chargement de la classe. Un tel bloc doit avoir la forme suivante (on notera l'absence de nom de méthode) :

```
static { instruction; ...}
```

Nous obtenons le code *TestRectDisk3* suivant :

```
class Rectangle extends Object{
   static double echelle=1.0;          // var de classe
   double largeur, hauteur;            // var d'instance
   ...
static void modifierEchelle(double e){ // méthode de classe
      echelle=e;
   }

static {         //initialisation complexe des var de classe
      echelle=0.5;
   }
}

class TestRectDisk3{
   public static void main (String args[]) {

      Rectangle r1=new Rectangle(2,4);
      Rectangle r2=new Rectangle(3,4);
      System.out.println("dim. r2:"+r2.largeur+"/"+r2.hauteur);
```

```
        Rectangle.modifierEchelle(4);
        System.out.println("echelle "+Rectangle.echelle);
}}
```

Exécution du programme *TestRectDisk3* :

```
dim. r2: 3/4
echelle 4
```

6.3 Et si l'on parlait d'héritage

Jusqu'à maintenant, nous avons parlé de classes sans mentionner la relation d'héritage qui existe entre elles. L'héritage permet de conserver certains acquis d'une classe et d'en faire bénéficier une autre, sans avoir à tout redéfinir (comme dans la vie courante où l'on hérite de ce fameux oncle d'Amérique).

Définition de sous-classes

Admettons que nous voulions créer une classe *Carré*. Sans le mécanisme d'héritage, nous devrions redéfinir les méthodes *perimetre(), surface(), diagonale(), etc*., alors qu'un carré, c'est « presque » un rectangle.

Nous allons donc profiter du fait que Java supporte la notion d'héritage (et que nous avons déjà défini une classe *Rectangle*) pour créer une classe *Carré* qui **étend** le comportement de *Rectangle*, héritant ainsi de ses variables d'instance et de ses méthodes. Nous dirons alors que *Carré* est **une sous-classe** de *Rectangle* et que *Rectangle* est **la super-classe** de *Carré*.

Nous définissons un constructeur pour la classe *Carré*, qui va invoquer le constructeur de la classe dont elle hérite (*Rectangle*). On utilise à cet effet le désignateur *super(...)*.

Nous ajoutons ensuite une méthode *cote()* à la classe *Carré*.

```
class Carre extends Rectangle{

   Carre(double initC){
      super(initC,initC);
   }

   double cote() {return this.hauteur;}
}
class TestRectDisk4{
   public static void main (String args[]) {

      Carre c1=new Carre(4);
      System.out.println("dim. c1: "+c1.largeur+"/"+c1.hauteur);
      System.out.println("surf. c1: "+c1.surface());
      System.out.println("coté de c1 "+c1.cote());
   }}
```

Exécution de *TestRectDisk4* :

```
dim. c1: 4/4
surf. c1: 16
coté de c1 4
```

Assignation, sous-classes et redéfinition

Nous avons vu qu'il faut déclarer une variable de type *Rectangle* pour pouvoir lui assigner un objet de la classe *Rectangle*. En fait, une telle variable peut également recevoir un objet de n'importe quelle sous-classe de *Rectangle*, comme dans le fragment de code ci-dessous :

```
Rectangle r1, r2;
r1 = new Rectangle(5, 6);
r2 = new Carre(4);
```

Comme *r2* est de type *Rectangle*, et bien qu'elle désigne un *Carre*, son comportement est restreint à celui d'un *Rectangle;* on ne peut lui appliquer que les méthodes propres à *Rectangle*.

```
double s = r2.surface(); // OK
double c = r2.cote();    //*** FAUX *** erreur à la compilation
```

Par contre, si une méthode est redéfinie dans une sous-classe, c'est bien la méthode de la sous-classe qui est appelée. Par exemple, s'il existe une méthode *imprimer()* définie dans *Rectangle,* qui imprime la largeur et la hauteur d'un rectangle, et une méthode *imprime()* dans *Carre*, qui imprime le côté, l'exécution de

```
Rectangle r
r = new Rectangle(5, 6);
r.imprimer();
r = new Carre(4);
r.imprimer();
```

produira :

```
largeur: 5 hauteur: 6
cote: 4
```

Le mécanisme qui, lors de l'exécution du programme, choisit la bonne méthode à exécuter en fonction de la classe de l'objet désigné, s'appelle **liaison dynamique;** nous y reviendrons en détail au point 22.1, p. 313.

Mentionnons aussi que la pseudovariable **super** sert à accéder aux méthodes de la super-classe, même si elles ont été redéfinies. Cela est utile lorsqu'une méthode de la sous-classe veut utiliser la méthode qu'elle redéfinit. Par exemple :

```
class CarreCouleur extends Carre {
   String couleur;

   CarreCouleur(int a, String clr) {... } // constructeur
```

```
    ...
    void imprime () {
       super.imprime(); // utilise la méthode imprime de Carre
       System.out.println(couleur); // et ajoute la couleur
    }
}
```

L'exécution des instructions

```
CarreCouleur cc = new CarreCouleur(3, "rouge");
cc.imprime();
```

produira :

```
cote: 3
rouge
```

Assignation et compatibilité des types

Il nous faut encore mentionner un point particulier concernant l'affectation des objets. Supposons qu'une variable soit déclarée de type *C*; on peut alors lui affecter un objet de la classe *C* ou d'une sous-classe de *C*, comme nous l'avons fait ci-dessus.

```
Rectangle r;
r = new Carre(4);        // Carre est sous-classe de Rectangle
```

L'inverse n'est par contre pas admis :

```
Carre c;
c = new Rectangle(7, 2); // *** Erreur à la compilation
```

Cette règle peut parfois s'avérer trop sévère et empêcher des affectations correctes, comme par exemple :

```
Rectangle r;
Carre c;
r = new Carre(5);   // r désigne bien un Carre
...
c = r;              // *** Refusé à la compilation
```

Le compilateur a raison de refuser la dernière instruction car *r* pourrait très bien désigner autre chose qu'un *Carre*. Lorsque l'on sait qu'une variable *r* (ou une expression) de type *Rectangle* désigne en fait un *Carre*, on peut écrire :

```
(Carre) r
```

afin de changer le type de l'expression en *Carre*, permettant ainsi d'écrire l'assignation prévue :

```
c = (Carre) r;
```

Le compilateur admet cette instruction, mais Java étant sûr, il y aura un contrôle au moment de l'exécution, afin de vérifier que *r* désigne bien un objet de *Carre* (ou de l'une de ses sous-classes). Si ce n'est pas le cas, une erreur d'exécution se produira.

Constructeurs de sous-classes

L'une des règles de Java impose que tout constructeur appelle directement ou indirectement un constructeur de sa super-classe. Si la première instruction d'un constructeur n'est pas *super()* ou *this()*, un appel *super()* est automatiquement ajouté avant la première instruction. Dans ce cas, il faut qu'un constructeur sans paramètre existe dans la super-classe.

Examinons encore un exemple : admettons que l'on souhaite étendre le comportement des rectangles à celui des parallélépipèdes. Nous agissons comme dans le cas précédent, en étendant la classe *Rectangle* afin d'ajouter une variable d'instance pour mémoriser la profondeur d'un *Parallelepipede* et en déclarant le constructeur et les méthodes propres à cette nouvelle classe :

```
class Parallelepipede extends Rectangle{
  double profondeur;

  Parallelepipede(double x, double y, double z){
     super(x,y);
     profondeur=z;
   }
   double volume(){return surface()*profondeur;}
}

class TestRectPar{
   public static void main (String args[]) {

   Parallelepipede p1=new Parallelepipede(2,3,4);

   System.out.println("prof. p1:"+p1.profondeur);
   System.out.println("surf. p1:"+p1.surface());
   System.out.println("vol.  p1:"+p1.volume());
  }
}
```

Exécution de *TestRectPar* :

```
prof. p1:4
surf. p1:6
vol.  p1:24
```

Il y a quelques pages encore, nous ne savions rien de l'héritage et voilà que nous avons créé une véritable petite famille.

En Java, toutes les classes ont pour origine la classe *Object*. Une classe ne pouvant étendre qu'une seule classe (appelée sa *super-classe*), nous avons donc une hiérarchie stricte que l'on peut représenter sous forme d'un arbre d'héritage, dont la racine est la classe *Object* (voir figure 6.2).

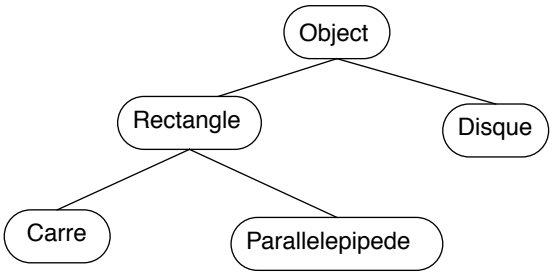


Figure 6.2 Arbre d'héritage des classes

6.4 Classes et méthodes abstraites

Les classes abstraites sont des classes dont certaines méthodes, dites abstraites, sont déclarées (nom et paramètres) mais ne possèdent pas de corps d'instructions. Une classe abstraite n'est pas instanciable (l'opération *new* est interdite). Si une sous-classe d'une classe abstraite redéfinit toutes les méthodes abstraites de celle-ci, elle devient concrète, sinon elle est également abstraite.

À l'inverse des classes abstraites, les classes finales se situent forcément « en bas » des hiérarchies de classes car elles interdisent toute définition de sous-classes. Une classe peut ne pas être finale mais déclarer des méthodes comme finales; il est alors interdit de redéfinir ces méthodes.

Nous reviendrons au point 22.3 sur l'utilité des classes abstraites dans la conception et l'architecture des programmes. Contentons-nous pour l'instant d'un exemple.

Dans notre famille « géométrique », nous constatons que nous avons des méthodes communes aux classes *Rectangle* et *Disque* (*perimetre()*, *surface()*) qui pourraient très bien être définies pour d'autres éléments de la géométrie plane. De même, la constante *pi* pourrait leur être utile. Nous avons donc défini une classe abstraite *GeometriePlane* munie de deux méthodes abstraites : *perimetre()* et *surface()* et d'une constante *pi*. Les classes *Rectangle* et *Disque* étendent cette nouvelle classe en fournissant une implantation pour les méthodes abstraites, en plus de leurs propres méthodes :

```
abstract class GeometriePlane{
  static final double pi=3.14159;
  abstract double perimetre();
  abstract double surface();
}
class Rectangle extends GeometriePlane{
   ...
   double perimetre() {
```

```
        return 2*(largeur+hauteur);
    }
    double surface(){
        return largeur*hauteur;
    }
    ...
}
class Disque extends GeometriePlane{
    ...
    double perimetre() {
        return GeometriePlane.pi*diametre;
    }
    double surface() {
        return (GeometriePlane.pi*diametre*diametre)/4;
    }
    ...
}
```

La classe abstraite n'apporte pas grand-chose de nouveau dans le comportement des objets. Par contre, en observant la structure, nous savons désormais que toutes les sous-classes d'une classe abstraite implantent ses méthodes. Exemple :

```
class Triangle extends GeometriePlane{...}
```

Cette déclaration nous assure que des méthodes *perimetre()* et *surface()* adaptées aux triangles pourront être invoquées sur des instances de la classe *Triangle*. Nous avons donc structuré notre monde d'objets géométriques.

6.5 Classes internes

Depuis la version 1.1 de Java, il est possible de déclarer une classe I à l'intérieur d'une classe C. Dans ce cas tout objet de I est dépendant d'un objet, appelé objet englobant, de C. Les objets de I sont des sous-objets de C. L'exemple ci-dessous montre l'usage d'une classe interne pour définir la structure des sommets d'un polygone :

```
class Polygone {
    int nbSommets;
    Sommet[] sommets;

    class Sommet {
        double x,y,poids;

        void mirroirY() {x = -x;}

        double poidsRelatif() {return poids/nbSommets;}
    }

    Polygone(int ns) { // crée un polygone de ns sommets
        nbSommets = ns;
        sommets = new Sommet[nbSommets];
```

```
        for (int i=0; i<nbSommets; i++) sommets[i] = new Sommet();
    }
...
}
```

On voit sur cet exemple que les méthodes de la classe interne peuvent accéder aux variables d'instance de leur objet englobant. L'expression *poids/nbSommets* signifie : diviser la variable *poids* du sommet courant par la variable *nbSommet* du polygone auquel appartient le sommet.

L'instruction *new Sommet(...)* du constructeur de *Polygone* crée des objets de *Sommet* dépendant du *Polygone* en construction.

On peut également créer des objets de *Sommet* depuis l'extérieur de la classe *Polygone*, pour autant qu'on indique de quel *Polygone* ils dépendent. On utilise pour cela une nouvelle syntaxe de l'instruction *new* :

```
Polygone po = new Polygone();
Polygone.Sommet spo = po.new Sommet();
```

6.6 Classes anonymes

Les classes anonymes sont une variante des classes internes que l'on utilise de préférence lorsqu'on veut créer un objet particulier sans avoir à définir un nom de classe. Une déclaration de classe anonyme est une extension de l'instruction *new*. Par exemple :

```
Sommet sommetSpecial = new Sommet() {
                          double poidsRelatif() {
                             return poids * 6.25;
                          }
```

Dans ce cas, la classe anonyme est une sous-classe de *Sommet* et l'objet *sommetSpecial* se comporte comme un *Sommet* mais avec une méthode *poidsRelatif* redéfinie.

Lorsque nous étudierons la gestion des événements, nous verrons que les classes anonymes et les classes internes peuvent simplifier l'écriture du code.

6.7 Interfaces

Une interface est assez similaire à une classe abstraite, puisqu'elle déclare un ensemble de méthodes abstraites et de constantes. On ne peut donc pas instancier une interface (pas de *new()*), par contre l'interface définit un type (on peut déclarer des variables de ce type).

Une classe peut indiquer dans sa définition qu'elle **implante** une ou plusieurs interfaces. Elle s'engage alors à fournir une définition pour **toutes** les méthodes définies dans l'interface.

Une interface peut hériter d'une ou plusieurs autres interfaces, les interfaces formant une hiérarchie séparée de celle des classes. Revenons à la géométrie et déclarons une hiérarchie d'interfaces :

```
interface Geometrie{
  static final double pi=3.14159;
}
interface Courbe extends Geometrie{
  double longueur();
  void doubler();
}
interface Surface extends Geometrie{
   double surface();
}
```

Redéfinissons maintenant les classes *Rectangle* et *Disque* en précisant qu'elles implantent ces interfaces :

```
class Rectangle extends GeometriePlane
      implements Courbe, Surface {
   ... comme avant
}
class Disque extends GeometriePlane
      implements Courbe, Surface{
   ... comme avant
}
```

Les hiérarchies de classes et d'interfaces ainsi que les liens d'implantation sont illustrés sur la figure 6.3 :

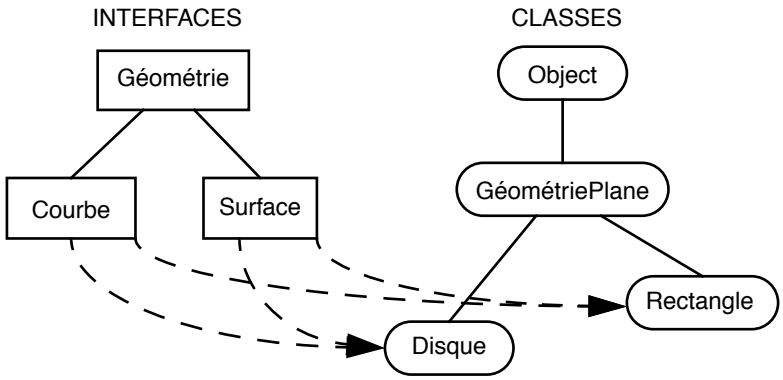


Figure 6.3 Hiérarchie des classes et hiérarchie des interfaces

Dans le programme *TestRectDisk7* ci-après, nous montrons qu'il est possible de manipuler les rectangles et les disques comme des courbes ou des surfaces. On parle alors de *polymorphisme* :

```
class TestRectDisk7{
   public static void main (String args[]) {
```

```
    Courbe c[]=new Courbe[10];
    Surface s[]=new Surface[10];
    Rectangle r[]=new Rectangle[5];
    Disque d[]=new Disque[5];

    // créer des rectangles
    for (int i=0;i<5;i++){
      r[i]=new Rectangle(i+1,i+1);
      c[i]=r[i];
      s[i]=r[i];
    }
    // créer des disques
    for (int i=0;i<5;i++){
      d[i]=new Disque(i+1);
      c[i+5]=d[i];
      s[i+5]=d[i];
    }

    // manipuler les objets de chaque classe
    System.out.println("surf. r[2]: "+r[2].surface());
    System.out.println("surf. s[2]: "+s[2].surface());
    System.out.println("long. r[2]: "+r[2].longueur());
    System.out.println("long. c[2]: "+c[2].longueur());

    // considérer les objets comme des éléments géométriques
    // surface de tous les objets
    double surfTotal=0;
    for (int i=0;i<10;i++)surfTotal+=s[i].surface();
    System.out.println("surfTotal: "+surfTotal);
    // doubler tous les périmètres
    for (int i=0;i<10;i++)c[i].doubler();
    //surface de tous les objets
    surfTotal=0;
    for (int i=0;i<10;i++)surfTotal+=s[i].surface();
    System.out.println("surfTotal: "+surfTotal);
    }
}
```

Exécution de *TestRectDisk7* :

```
surf. r[2]: 9
surf. s[2]: 9
long. r[2]: 12
long. c[2]: 12
surfTotal: 98.1969
surfTotal: 392.787
```

6.8 Packages

Un package permet de regrouper « physiquement » des classes concernant un domaine particulier (exemple : accès au réseau), afin de mieux les organiser. Le package définit un système de nommage (*nomDePackage.nomD'element*);

celui-ci devra être utilisé dans tout programme désirant accéder aux éléments (classes, constantes, etc.) du package.

En résumé, les trois mécanismes proposés par Java pour structurer et organiser le code sont :

- l'héritage, structure des classes;
- les interfaces, structure des comportements;
- les packages, structure du développement.

6.9 Référence, copie et égalité

Il est important de bien distinguer l'assignation et la copie d'objets. L'assignation d'un objet à une variable place dans la variable une **référence** à cet objet. Après avoir effectué les opérations :

```
Rectangle r1=new Rectangle(2,4);
Rectangle r2=r1
```

La situation est celle décrite par la figure 6.4 ci-dessous : les deux variables *r1* et *r2* font référence au même objet :

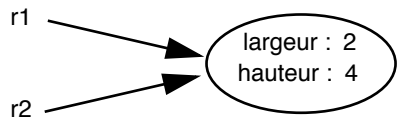


Figure 6.4 Deux variables référençant le même objet Rectangle

La méthode *clone()*, héritée d'*Object*, peut être invoquée pour créer une copie d'un objet :

```
Rectangle r3 = r1.clone()
```

Les deux objets obtenus après un tel clonage sont égaux (ils ont les mêmes valeurs dans leurs variables d'instance) mais ne sont pas identiques.

```
r3 == r1         // retourne false
r3.equals(r1)    // retourne true
```

La méthode *equals()* teste l'égalité du contenu des objets alors que l'opérateur == teste l'identité (s'agit-il du même objet?). Nous reviendrons plus en détail sur ces questions au point 23.4.

Comparaison des chaînes de caractères

Ce que nous venons de présenter s'applique en particulier aux chaînes de caractères (*String*). Les chaînes de caractères étant des objets à part entière, il faut utiliser la méthode *equals()* pour tester l'égalité de deux chaînes (mêmes caractères) et non pas == (même objets).

On peut également tester l'ordre lexicographique entre deux chaînes avec la méthode *compareTo()* de la classe *String*.

6.10 Types simples et classes enveloppes

Les valeurs des types simples – *short*, *byte*, *int*, *boolean*, etc. – ne sont pas des objets. Or, il arrive que certaines méthodes ne traitent que des objets : par exemple, les méthodes d'instance de la classe *java.util.Vector,* qui gère des séquences d'objets.

C'est pourquoi il existe des classes « enveloppes » dont le seul but est d'encapsuler un type simple dans un objet. Ces classes sont : *Boolean, Character, Double, Float, Integer* et *Long*. Leurs constructeurs prennent comme paramètre une valeur du type simple à encapsuler.

```
boolean b = true;
int p = 20292;
Boolean bObj = new Boolean(b);
Integer pObj = new Integer(p);
```

Chaque classe fournit une méthode pour extraire la valeur simple de l'objet enveloppant :

```
boolean b2 = bObj.booleanValue();
int p2 = pObj.integerValue();
```

Nous sommes maintenant en mesure d'utiliser la classe *Vector* pour créer une séquence d'entiers :

```
Vector v = new Vector;
v.addElement(new Integer(5)); v.addElement(new Integer(2));
v.addElement(new Integer(90));
```

et pour en extraire le dernier élément :

```
int i = (v.lastElement()).integerValue();
```

Chapitre 7

Les structures

Classes, interfaces, packages

Les différents éléments permettant d'organiser et de structurer un programme Java, à savoir les *classes*, les *interfaces* et les *packages* ont été introduits au chapitre précédent. Nous allons maintenant y revenir afin d'examiner en détail la syntaxe de leur déclaration.

7.1 Unité de compilation

Une unité de compilation est un fichier de code source (avec une extension *.java)* qui contient un ensemble de déclarations de classes et d'interfaces qui appartiennent au même package. Le compilateur placera le résultat de la compilation de chaque interface et chaque classe dans un fichier séparé, portant le nom de celle-ci (suivi de l'extension *.class*). Cette séparation en différents fichiers permettra de charger dynamiquement les classes nécessaires, au moment de l'exécution (ou de la compilation lorsque les classes sont importées dans un autre programme). Il ne peut y avoir qu'un seul élément *public* (classe ou interface) dans une unité de compilation

Les points suivants (7.2 à 7.4) décrivent les éléments nécessaires à la déclaration d'une unité de compilation, schématisée par le diagramme 7.1.

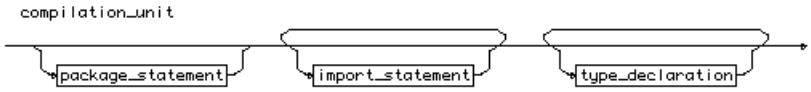


Diagramme 7.1 compilation_unit

7.2 Déclaration de package

La déclaration d'un package (voir diagramme 7.2) doit être la première instruction d'un programme (hormis des commentaires). Elle permet de regrouper l'ensemble des classes déclarées dans le même fichier sous un dénominateur commun. Ces classes devront être importées individuellement ou de manière globale (package entier) par toute autre unité de compilation qui désirerait les utiliser.

Les règles de visibilité sont différentes pour des objets appartenant à un même package de celles qui sont en vigueur pour des objets de packages différents.

Diagramme 7.2 package_statement

7.3 Importation des packages et des classes

Le mécanisme d'importation permet d'abréger localement le nom complet d'une classe. Pour qu'une classe soit accessible, il faut :

- que le package qui la contient ait été compilé avec succès;
- que le chemin *(pathclass)* du compilateur puisse y accéder;
- que le fichier *(.class)* soit accessible en lecture.

Dans l'exemple qui suit, nous n'avons pas déclaré d'importation. Nous devons donc indiquer le nom complet de la classe *Point*. Dans ce cas, le nom complet reste relativement court car le package réside localement.

```
class TesImport1{
   public static void main (String args[]) {
      java.awt.Point p=new java.awt.Point(1,2);
      System.out.println(p.toString());
   }
}
```

L'importation peut également être utilisée pour abréger le nom des interfaces. Sun Microsystems a proposé un schéma de nommage unique pour les packages, un peu à la façon des URL. On trouve dans l'identifiant le nom de domaine assigné au site où réside le package, suivi d'un chemin dans l'arborescence des répertoires menant au package (ou au sous-package). Exemple :

```
CH.Unige.Cui.java.projet3.pck4.classe
```

Par convention, le premier composant est écrit entièrement en majuscules (*COM*, *GOV*, *FR*, etc.), les suivants (commençant par une majuscule) faisant partie du nom de domaine assigné au site (dans l'ordre inverse du format d'adresses Internet). Ce schéma permet d'identifier de manière unique chaque

classe et évite toute collision. Ainsi chaque développeur peut créer sa propre classe *Rectangle!*

Il existe trois formes pour spécifier l'importation. Le diagramme 7.3réunit leurs syntaxes respectives tandis que les exemples ci-dessous illustrent leur utilisation :

Forme 1 : *import package.comp1.....compn;*

Dans cette forme, les classes du dernier composant (*compn)* peuvent être utilisées en les préfixant par *compn*. Nous pouvons ainsi réécrire notre exemple de la manière suivante :

```
import java.awt;
class TesImport2{
   public static void main (String args[]) {
    awt.Point p=new awt.Point(1,2);
    System.out.println(p.toString());
   }
}
```

Forme 2 : *import package.comp1.....compn.Class;*

Dans ce cas, une seule classe est importée. Elle est identifiée directement par son nom. Nous pouvons ainsi réécrire notre exemple :

```
import java.awt.Point;
class TesImport3{
   public static void main (String args[]) {
    Point p=new Point(1,2);
    System.out.println(p.toString());
   }
}
```

Forme 3 : *import package.comp1.....compn.*;*

Dans cette dernière forme, toutes les classes du package sont importées. Elles sont identifiées directement par leur nom. Nous pouvons réécrire une dernière fois notre exemple :

```
import java.awt.*;
class TesImport4{
   public static void main (String args[]) {
    Point p=new Point(1,2);
    Rectangle r; // une autre classe de awt
    System.out.println(p.toString());
   }
}
```

Remarques :

- si plusieurs classes portent le même nom dans des packages importés, le compilateur génère une erreur;
- les classes *java.lang.** sont automatiquement importées par défaut;

c'est pourquoi nous avons pu faire référence dans nos exemples aux classes *System, Math, etc.* du package *java.lang* sans l'avoir importé.

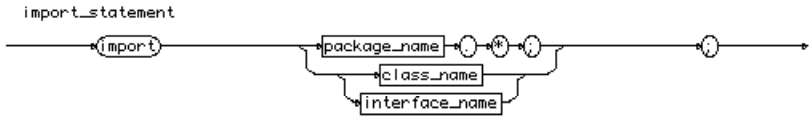


Diagramme 7.3 import_statement

7.4 Déclaration de types

Une unité de compilation est formée d'un ensemble de déclarations de types (déclarations de classes et d'interfaces). Chaque déclaration peut être précédée d'un commentaire de documentation (style /** ... */), utilisé par un outil de génération automatique de documentation. L'exemple ci-dessous et le diagramme 7.4 montrent cette possibilité :

```
/** commentaire sur la classe ...*/
class x extends ...{}
```

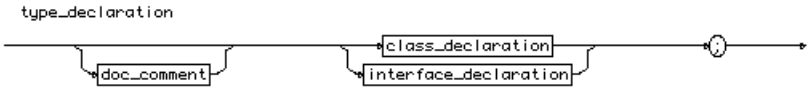


Diagramme 7.4 type_declaration

7.5 Déclaration d'une classe

La déclaration d'une classe définit un type qui porte le nom de la classe. Une classe étend une seule et unique classe (sa super-classe). Par défaut, la classe étend la classe *Object*. L'identifiant complet de la classe est défini par le nom du package et le nom de la classe *(package.classe)*. Une classe peut implanter une ou plusieurs interfaces. Une classe possède un corps qui définit ses membres (variables et méthodes).

La syntaxe complète de la déclaration d'une classe est présentée dans le diagramme 7.5.

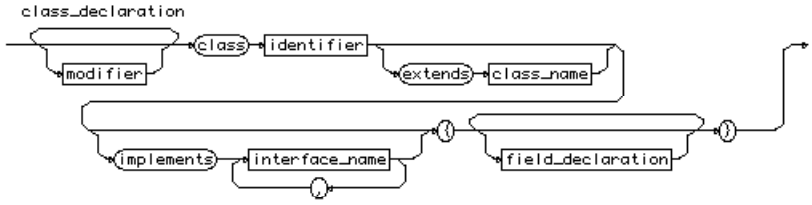


Diagramme 7.5 class_description

Les modificateurs de classe sont :

- *abstract :* indique que la classe est abstraite : certaines de ses méthodes n'ont pas de corps, elles seront implantées dans des sous-classes;
- *final* : une classe finale ne peut pas être étendue par des sous-classes (ceci permet au compilateur d'effectuer certaines optimisations dans le code, car il ne peut pas y avoir de surcharge);
- *public* : une classe est visible pour les classes des autres packages uniquement si elle est déclarée publique. Une classe non publique n'est visible que pour les classes définies à l'intérieur de son package.

Les combinaisons suivantes de modificateurs sont autorisées :

```
public; abstract; public abstract; final; public final
```

7.6 Déclaration d'une interface

La déclaration d'une interface définit un type. Une interface est définie par un nom. Elle peut étendre une ou plusieurs interfaces. L'identifiant complet de l'interface est défini par le nom du package et le nom de l'interface *(package.interface)*.

Une interface possède un corps qui définit ses membres : un ensemble de méthodes sans implantation et un ensemble de constantes. La syntaxe complète de la déclaration d'une interface est illustrée par le diagramme 7.6.

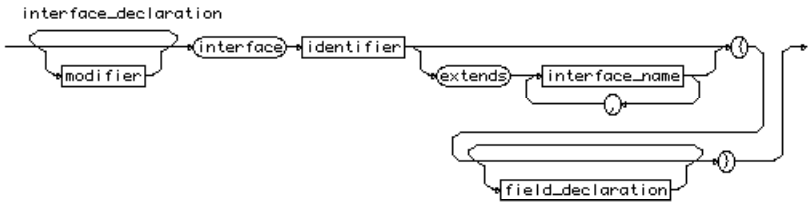


Diagramme 7.6 interface_declaration

Les modificateurs d'une interface sont :

- *abstract :* par définition une interface est abstraite;
- *public :* une interface est visible pour les classes des autres packages uniquement si elle est déclarée publique. Une interface non publique n'est visible que pour les classes définies à l'intérieur de son package.

Les combinaisons suivantes de modificateurs d'interfaces sont autorisées :

```
public; public abstract
```

Quand une interface étend une autre interface (super-interface), elle hérite de toutes ses constantes et méthodes. La classe qui implante cette interface doit implanter à la fois les méthodes de l'interface et celles de sa super-interface.

7.7 Déclaration de membres

Un membre (voir diagramme 7.7) est une composante d'une déclaration de classe ou d'interface; il peut contenir une déclaration de méthode, de constructeur ou de variable, ou un initialiseur statique. Un initialiseur statique est un bloc d'instructions qui effectue des initialisations complexes des variables statiques d'une classe. Il est exécuté juste après les initialisations simples (*variable = expression*) de ces variables.

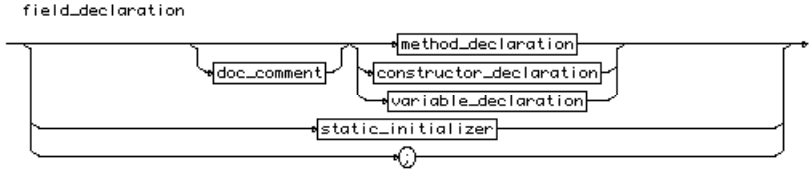


Diagramme 7.7 field-declaration

7.8 Déclaration d'une méthode et d'un constructeur

La déclaration d'une méthode peut commencer par un modificateur (*private*, *private protected*, *protected* ou *public*) qui indique son degré de visibilité. Contentons-nous pour l'instant d'indiquer que :

- par défaut, la visibilité s'étend au package;
- *private* limite la visibilité à la classe;
- *public* étend la visibilité à tous les packages;
- *private protected* étend la visibilité aux sous-classes du même package;
- *protected* étend la visibilité à toute sous-classe.

Nous étudierons tout ceci en détail au point 24.2, p. 343.

Il faut également spécifier le type du résultat retourné, ou *void* s'il n'y en a pas, le nom de la méthode, la liste des paramètres (de la forme *type identificateur*), puis les instructions à exécuter.

Si le résultat est un tableau, la liste de paramètres doit être suivie de *[ ]*, autant de fois que le tableau a de dimensions.

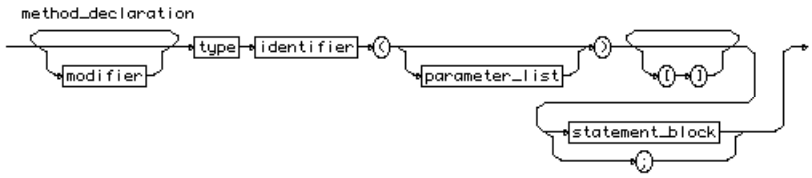


Diagramme 7.8 method_declaration

La déclaration d'un constructeur suit le même principe que celle d'une méthode. Il n'y a cependant pas de type de résultat et le nom du constructeur doit être identique à celui de la classe.

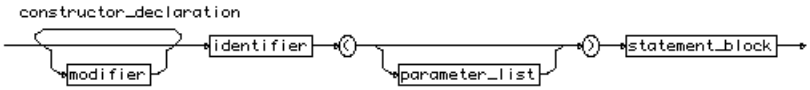


Diagramme 7.9 constructor_declaration

7.9 Création d'objets

Le mot réservé *new* suivi d'un nom de classe et d'une liste d'arguments crée un objet de cette classe et invoque le constructeur dont les types des paramètres correspondent aux types des arguments de la liste (voir diagramme 7.10).

La création d'un tableau (vecteur) s'effectue en donnant le type des éléments suivi de la taille du tableau entre [et]. Si les éléments sont eux-mêmes des tableaux, il faut ajouter [ ] ([ ] [ ] pour des tableaux de tableaux, et ainsi de suite).

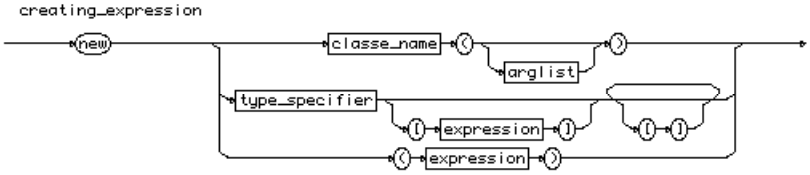


Diagramme 7.10 creating_expression

Chapitre 8

Exceptions et processus

8.1 Exceptions

Pour traiter de manière structurée les situations exceptionnelles, Java offre un mécanisme d'exceptions. Une exception est un objet particulier de la classe *Exception* ou d'une de ses sous-classes qui est créé au moment où une situation anormale est détectée. Une fois créée, l'exception est lancée (ou levée) à l'aide de l'instruction *throw*, ce qui a pour effet d'arrêter l'exécution normale du programme et de propager l'exception jusqu'au premier bloc d'instructions capable de la capter et de la traiter (*catch*). Le traitement d'exceptions introduit une nouvelle structure d'instruction décrite par le diagramme 8.1 ci-dessous.

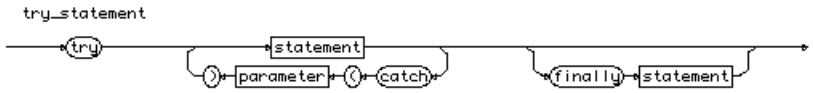


Diagramme 8.1 try_statement

En temps normal, l'instruction suivant le *try* est exécutée. Si une exception survient, on saute directement au *catch* dont le type du paramètre correspond au type de l'exception. Si aucun *catch* ne convient, l'exception est propagée aux blocs englobants puis, s'ils ne la captent pas, à la méthode qui a invoqué celle où se trouve le *try*, et ainsi de suite, éventuellement jusqu'au système d'exécution Java qui interrompt alors le programme. Dans tous les cas de figure, l'instruction suivant le *finally* est exécutée avant de quitter le bloc (même si l'exception n'a pas été captée et doit se propager).

Quelques exemples classiques

L'opération de conversion d'une chaîne de caractères en un entier ne réussit pas toujours, par exemple si la chaîne contient "*çH\!aaa". Dans ce cas, une excep-

tion de type *NumberFormatExeception* est lancée par la méthode *Integer.ParseInt()* :

```
String s = ... ;
int i;
try i = Integer.ParseInt(s);
catch (NumberFormatExeception e) {
   System.out.prinln("erreur: la chaîne " + s
                     + " ne peut pas être convertie en un entier");
   i = 0; }
```

Un processus (voir point suivant) peut se mettre en sommeil pour une certaine durée. Ce sommeil peut être interrompu pour diverses raisons qui génèrent alors une exception de type *InterruptedException* :

```
try {Thread.sleep(2000);} // dort 2 secondes
catch(InterruptedException signal) {} // continue simplement
```

Un problème peut survenir lors d'une opération d'entrée-sortie, créant une exception de type *IOException* :

```
try {x=System.in.read();
        ...
catch (IOException e) { ... }
```

La tentative d'accès à un élément hors des limites d'un tableau cause également une exception :

```
try z = tab[i];
catch (ArrayIndexOutOfBounds ai) {
   System.out.println(ai.getMessage());
   z = -1; }
```

L'exception est un objet qui peut contenir un texte informatif précisant la nature du problème. La méthode *getMessage()* récupère ce texte.

Générer des exceptions

Il est possible de créer ses propres types d'exceptions en étendant la classe *Exception* et de lancer des exceptions à l'aide de l'instruction *throw*. Si une méthode lance ou propage des exceptions, elle doit l'indiquer dans sa déclaration, sauf s'il s'agit d'exceptions d'une sous-classe de *Error* ou de *RuntimeException*.

```
class CalculImpossible extends Exception {
   public CalculImpossible { super(); }
}
class UneClasse {

   void uneMethode(int a, int b) throws CalculImpossible {
      ...
      if (a == b) throw new CalculImpossible();
      ...
   }
}
```

Il existe déjà toute une hiérarchie d'exceptions dont le sommet est la classe *Throwable*.

8.2 Processus et synchronisation

Le langage Java a été prévu dès le départ pour supporter l'exécution concurrente de plusieurs processus. Un processus est un objet de la classe *java.lang.Thread* qui possède son propre environnement d'exécution.

Création de processus

Pour faire fonctionner un processus, il faut :

1. Créer un objet de la classe *Thread* (ou de l'une de ses sous-classes).
2. Démarrer l'exécution du processus en invoquant la méthode *start()*.

Il y a deux manières de créer des processus en Java :

- définir une sous-classe de *Thread* qui redéfinit la méthode *run()*;

```
class Pro1 extends Thread {
   public void run() {
      // ICI ce que fait le processus //
   }
}
...
Pro1 p1 = new Pro1();
p1.start();   // démarre un processus qui exécute p1.run()
```

- créer un objet de *Thread* avec le constructeur *Thread(Runnable obj)* auquel on passe un objet qui implante *Runnable*, c'est-à-dire qui possède une méthode *run()*.

```
class Pro2 implements Runnable {
public void run() {
   // ICI ce que fait le processus //
}
...
Pro2 q = new Pro2();
Thread p2 = new Thread(q);
p2.start();   // démarre un processus qui exécute q.run()
```

Synchronisation

Deux processus peuvent vouloir accéder aux mêmes objets en même temps, ce qui peut causer des problèmes de cohérence. Pensons à un processus qui calcule la somme des nombres contenus dans un tableau alors que, simultanément, un autre processus déplace les éléments du tableau pour les trier. Pour prévenir ce type de situation, un processus peut s'assurer l'accès exclusif à un objet.

L'instruction *synchronized()* réserve un objet pour la durée de l'exécution de l'instruction qui suit puis le libère. Si l'objet est déjà réservé (par un autre processus), *synchronized()* met le processus en attente jusqu'à ce que l'objet soit à nouveau libre.

À l'intérieur d'un *synchronized(o),* le processus peut libérer temporairement l'objet et se mettre en attente en appelant *o.wait()*. Il attendra jusqu'à ce qu'un autre processus exécute un *o.notify()* puis libère l'objet *o*. Les méthodes *wait()* et *notify()* ne peuvent être invoquées que par un processus qui a réservé l'objet.

Une méthode peut être déclarée *synchronized*, ce qui équivaut à englober le corps de la méthode avec *synchronized (this) {...}*.

Nous verrons plusieurs exemples de processus dans la partie III et nous expliquerons plus en détail la synchronisation et la concurrence au chapitre 25.

Partie III

Utiliser les classes de l'environnement Java

> Car de quelqu'un n'ayant pas toute sa tête peut-on raisonnablement affirmer qu'il se le demande et qui plus est sous peine d'incohérence s'acharne sur ce casse-tête avec tout ce qui lui reste de raison?
>
> Samuel Beckett, *Soubresauts*.

Chapitre 9

L'API Java

L'environnement de développement Java est fourni avec une *interface de programmation d'applications* (API : *Application Programming Interface*) appelée *Core API*, regroupant un ensemble de classes prédéfinies d'usage général que l'on retrouve sur toutes les plates-formes.

Cette API définit les briques de base à utiliser dans tout programme Java afin de lui assurer une portabilité maximale de son code.

Le *Core package* de Java est actuellement constitué d'une soixantaine de packages, les 15 de premier niveau figurant dans la table ci-dessous. Chaque package est spécialisé dans un domaine particulier. Tous proposent des solutions à de nombreux problèmes auxquels tout programmeur Java est tôt ou tard confronté.

Package	Description	Java1.0	Traité
java.applet	Fournit les classes nécessaires pour créer une applet et les classes pour que l'applet puisse communiquer dans son contexte d'exécution.	Présent	10
java.awt	Fournit toutes les classes pour créer des interfaces avec l'utilisateur pour peindre et afficher des images.	Présent en partie	11 à 17
java.beans	Contient les interfaces et les classes pour le développement de composants logiciels JavaBeans.		24
java.io	Fournit les interfaces et les classes pour la gestion des entrées/sorties à travers des canaux de données, la sérialisation et les systèmes de gestion de fichiers.	Présent (mais remaniée depuis)	19

Tableau 9.1 Les packages de l'API Java 2

Package	Description	Java1.0	Traité
java.lang	Fournit les classes fondamentales pour la conception du langage de programmation Java.	Présent	partout
java.math	Fournit les interfaces et les classes pour effectuer des opérations en entier et décimal d'une précision arbitraire (aussi longue que désirée).		Exercice dans ce chapitre
java.net	Fournit les classes pour la gestion du réseau et des communications.	Présent	20
java.rmi	Fournit les services d'invocation à distance des méthodes (*Remote Method Invocation*).		29
java.security	Fournit les interfaces et les classes dans le cadre de la sécurité.		utilisé dans 29 et 30
java.sql	Fournit les services JDBC (Connection aux bases de données).		28
java.text	Fournit les interfaces et les classes pour la manipulation des textes, des dates, des nombres et des messages d'une manière indépendante des langues naturelles.		
java.util	Contient la gestion de quelques structures de données des utilitaires pour la manipulation des dates, un modèle d'événement, etc.	Présent sans les collections	23 et ailleurs
javax.accessibility	Définit un contrat entre les composants interface-utilisateur et des technologies d'assistance à l'utilisateur (loupe, vocaliseur...)		
javax.swing	Fournit un ensemble de *composants légers* (entièrement écrits en Java, pour la gestion des interfaces, ils sont donc complètement indépendants de la plate-forme et ont un comportement similaire sur toutes les plate-formes).		18
org.omg.CORBA	Fournit la correspondance entre l'API CORBA de l'OMG et le langage Java. Elle contient la classe ORB qui est un courtier d'objets (*Object Request Broker*) directement utilisable.		30

Tableau 9.1 Les packages de l'API Java 2

Nous indiquons le chapitre de ce livre où ce package est traité.

9.1 Méthode de travail

Pour aborder cet environnement complexe, notre approche va être la suivante.

Tout d'abord faire un tour relativement complet des packages de base (java.applet, java.awt, java.io, java.lang, java.net, java.util). Ensuite, examiner les principes et les services spécialisés rendus principalement dans le cadre de la programmation distribuée :

- accès aux bases de données (java.sql);
- invocation des méthodes à distance (java.rmi);
- utilisation d'un ORB (approche CORBA avec org.omg.CORBA);
- programmation des interfaces (javax.swing).

Nous considérons ces ensembles de classes comme notre boîte à outils dont il va falloir apprendre à se servir. Ainsi, avant de construire de nouveaux outils, il va être important de bien connaître ceux dont nous disposons déjà. Cette partie est consacrée à l'apprentissage des fonctionnalités proposées par l'API.

Nous avons choisi de ne pas traiter les packages un à un, mais de travailler autour de différents thèmes :

- comment dessiner, écrire?
- comment animer une applet?
- comment interagir avec la souris, avec le clavier?
- comment gérer une fenêtre et son contenu?
- comment gérer des fichiers?
- comment communiquer sur un réseau?

Pour découvrir cette boîte à outils, nous vous proposons quelques éléments de méthode (à prendre comme des conseils d'ami) :

- comprendre les concepts définis dans les packages;
- se familiariser avec les principales classes de ces packages;
- ne **pas** apprendre toutes les méthodes définies par ces classes;
- chercher, fouiller, butiner dans la documentation (très complète), qui permet par exemple de trouver toutes les méthodes d'une classe;
- examiner également les méthodes héritées (examiner la super-classe).

En résumé, pas de mémorisation systématique mais plutôt de l'intuition concernant le comportement général d'une classe.

La méthode proposée ici est la seule qui vous permettra de devenir autonome. En effet, les packages de base semblent déjà consistants mais l'ensemble de l'API java comporte plus de mille classes et plusieurs milliers de méthodes (7000). Seule une utilisation systématique de la documentation fournie par Sun avec le JDK (*Java Development Kit*) vous permettra de survivre.

9.2 La vraie classe *Rectangle*

Nous allons maintenant examiner de manière complète une classe et ses différentes méthodes. Nous avons choisi la classe *Rectangle* du package *Java.awt* (la vraie! À ne pas confondre avec celle utilisée dans nos exemples du chapitre 6).

La documentation nous donne la filiation de *Rectangle* :

```
java.lang.Object
   java.awt.geom.RectangularShape
      java.awt.geom.Rectangle2D
       java.awt.Rectangle
```

Ainsi que les sous-classes connues dérivées de *Rectangle* :

```
DefaultCaret
```

Les interfaces qui sont implémentées par *Rectangle* :

```
Shape, Serializable
```

Rectangle hérite de la classe *Rectangle2D* des variables suivantes (des masques binaires pour situer un point *(x,y)* par rapport au rectangle dans l'espace) :

```
OUT_BOTTOM, OUT_LEFT, OUT_RIGHT, OUT_TOP
```

Un *Rectangle* est défini à l'aide des variables d'instance *x*, *y*, *width* et *height* :

Variable	Description
`height`	Hauteur du rectangle
`width`	Largeur du rectangle
`x`	Coordonnée en x du rectangle
`y`	Coordonnée en y du rectangle

Tableau 9.2 Variables d'instances de la classe Rectangle

L'observation du package *Java.awt* nous indique que ces variables sont de type *int* et qu'elles sont *public*, ce qui signifie qu'on peut y accéder dans toute instance de *Rectangle*. Le tableau 9.3 énumère les constructeurs de cette classe.

Constructeur	Description
`Rectangle()`	Construit un nouveau rectangle.
`Rectangle(int,int)`	Construit un nouveau rectangle et l'initialise avec les paramètres *width* et *height* spécifiés.
`Rectangle(int,int, int,int)`	Construit un nouveau rectangle et l'initialise avec les paramètres *x*, *y*, *width* et *height* spécifiés.

Tableau 9.3 Constructeurs de la classe Rectangle

Constructeur	Description
`Rectangle(Dimension)`	Construit un nouveau rectangle et l'initialise avec le paramètre *Dimension* (un autre objet).
`Rectangle(Point)`	Construit un nouveau rectangle et l'initialise avec le paramètre *Point* (un autre objet).
`Rectangle(Point, Dimension)`	Construit un nouveau rectangle et l'initialise avec les paramètres *Point* et *Dimension* (deux objets).
`Rectangle(Rectangle )`	Construit un nouveau *Rectangle*, initialisé avec les données du rectangle spécifié.

Tableau 9.3 Constructeurs de la classe Rectangle

Nous constatons que l'on peut travailler soit directement sur les variables d'instance des objets *Rectangle*, soit dans un modèle plus élaboré basé sur les concepts de *Point* et de *Dimension*. Les différentes méthodes déclarées dans la classe *Rectangle* (tableau 9.4) fournissent des opérations de composition, de manipulation et de comparaison.

Méthode	Description
`add(int, int)`	Ajoute une coordonnée (x,y) au rectangle, redéfinit les dimensions du rectangle pour que celui-ci contienne ce point.
`add(Point)`	Ajoute un *Point* au rectangle, redéfinit les dimensions du rectangle pour que celui-ci contienne ce point.
`add(Rectangle)`	Ajoute un *Rectangle* au rectangle, redéfinit les dimensions du rectangle pour que celui-ci contienne ce rectangle.
`contains(int, int)`	Détermine si le rectangle contient le point de coordonnées (x,y).
`contains(int, int, int, int)`	Détermine si le rectangle contient entièrement le rectangle spécifié par les coordonnées (x,y) et les dimensions (w,h).
`contains(Point)`	Détermine si le rectangle contient le point.
`contains(Rectangle)`	Détermine si le rectangle contient entièrement le rectangle passé en paramètre.
`createIntersection(Rectangle2D)`	Retourne un nouveau rectangle (type *Rectangle2D*) représentant l'intersection des deux rectangles.

Tableau 9.4 Méthodes de la classe Rectangle

Méthode	Description
`createUnion(Rectangle2D)`	Retourne un nouveau rectangle (type *Rectangle2D*) représentant l'union des deux rectangles (le rectangle qui contient les deux !).
`equals(Object)`	Vérifie si deux rectangles sont égaux.
`getBounds()`	Retourne le rectangle contenant ce rectangle (pour satisfaire la complétude demandée par component !).
`getBounds2D()`	Retourne le rectangle (haute précision) contenant ce rectangle.
`getHeight()`	Retourne la hauteur en double précision du rectangle.
`getLocation()`	Retourne le point localisant ce rectangle.
`getSize()`	Retourne les dimensions de ce rectangle.
`getWidth()`	Retourne la largeur en double précision du rectangle.
`getX()`	Retourne la coordonnée X en double précision du rectangle.
`getY()`	Retourne la coordonnée Y en double précision du rectangle.
`grow(int, int)`	Agrandit le rectangle en x et en y.
`inside(int,int)`	Dépréciée. À partir de la version JDK 1.1, remplacée par contains (int, int).
`intersection(Rectangle)`	Calcule l'intersection de deux rectangles.
`intersects(Rectangle)`	Vérifie si deux rectangles ont une intersection.
`isEmpty()`	Détermine si le rectangle est vide.
`move(int, int)`	Dépréciée. À partir de la version JDK 1.1, remplacée par setLocation (int,int).
`outcode(double,double)`	Détermine où les coordonnées spécifiées ne s'appliquent pas avec ce rectangle (voir les champs OUT...).
`reshape(int,int,int,int)`	Redéfinit le rectangle (coordonnées et taille).
`resize(int, int)`	Redéfinit la taille du rectangle.

Tableau 9.4 Méthodes de la classe Rectangle

Méthode	Description
`setBounds(int, int, int , int)`	Définit les bornes du rectangle avec les coordonnées (x,y) et les dimensions (w,h).
`setBounds(Rectangle)`	Définit les bornes du rectangle avec celles du rectangle données en paramètre.
`setLocation(int,int)`	Déplace ce rectangle à la position (x,y).
`setLocation(Point)`	Déplace ce rectangle aux coordonnées du point.
`setRect(double , double, double, double )`	Définit les bornes du rectangle avec les coordonnées (x,y) et les dimensions (w,h) données en double précision.
`setSize(Dimension)`	Définit ce rectangle à la dimension donnée.
`setSize(int,int)`	Définit ce rectangle aux dimensions (w,h).
`toString()`	Retourne sous forme de *String* les valeurs définissant le rectangle.
`translate(int, int)`	Effectue une translation sur le rectangle.
`union(Rectangle)`	Calcule l'union de deux rectangles.

Tableau 9.4 Méthodes de la classe Rectangle

On peut constater que l'outil *Rectangle* est bien affûté, mais qu'il peut encore être étendu (si le cœur vous en dit) en définissant sa surface, son périmètre, etc.

Mais ce n'est pas tout, un objet hérite de nombreuses méthodes de sa filiation. Examinons cette dernière.

Méthodes héritées de la classe *java.awt.geom.Rectangle2D* :

```
add, add, add, contains, contains, getPathIterator, getPathIterator,
hashCode, intersect, intersects, intersectsLine, intersectsLine,
outcode, setFrame, setRect, union
```

Méthodes héritées de la classe *java.awt.geom.RectangularShape* :

```
clone, contains, contains, getCenterX, getCenterY, getFrame,
getMaxX, getMaxY, getMinX, getMinY, intersects, setFrame, setFrame,
setFrameFromCenter, setFrameFromCenter, setFrameFromDiagonal,
setFrameFromDiagonal
```

Méthodes héritées de la classe *java.lang.Object* :

```
finalize, getClass, notify, notifyAll, wait, wait, wait
```

Comment se rappeler ce qu'il y a dans cette classe ? Fermez les yeux, imaginez un monde de *Rectangles* : ils se créent, grandissent, se déplacent, interagissent, fusionnent, etc. Nous avons bien là les bonnes méthodes, celles qui régissent le

monde des *Rectangles*. Aussi vaut-il mieux **comprendre** ce monde que tenter de mémoriser les détails des méthodes de cette classe.

Dans la suite de cette partie, nous examinerons plusieurs autres « mondes » sans pour autant être exhaustif, car nous nous intéresserons principalement aux **interactions** entre tous ces mondes.

Dans la documentation de Sun, on trouvera une description détaillée de chacune des classes ou chacun des champs que nous avons cités. Nous donnons sans le traduire un exemple de cette documentation pour la méthode *translate*.

```
public void translate(int x,
                      int y)
Translates this Rectangle the indicated distance, to the right along
the x coordinate axis, and downward along the y coordinate axis.
     Parameters:
          dx - the distance to move this Rectangle along the x axis
          dy - the distance to move this Rectangle along the y axis
     See Also:
          setLocation(int, int), setLocation(java.awt.Point)
```

À la fin de chaque chapitre, nous inviterons le lecteur à continuer son exploration en lui proposant quelques exercices.

9.3 Invitation à explorer

Pour le package java.math, nous n'avons pas prévu d'explication particulière. Par contre, un petit exercice peut être aussi utile :

- dresser un petit tableau des possibilités offertes par le package *java.math;*
- calculer avec une précision arbitraire 100 !,1000 ! (rappelons que n ! (factoriel) est le produit des nombres entiers de 1 à n, par exemple 5 ! =1*2*3*4*5=120).

Chapitre 10

Retour sur les applets

Les programmes Java peuvent prendre deux formes : les applications et les applets.

Les *applications* sont des programmes au sens classique du terme, c'est-à-dire qu'elles sont stockées sur la machine qui va les exécuter.

Les *applets*[1] quant à elles sont destinées à être chargées depuis une machine (le serveur) et exécutées sur une autre machine (le poste du client). L'invocation d'une applet est réalisée depuis l'intérieur d'un document HTML. Ainsi, toute page HTML peut faire appel à un programme (une applet) dont la taille est souvent moins importante que celle des images décorant les pages HTML actuelles.

L'exécution de l'applet est confiée à un butineur muni d'un interpréteur Java (comme par exemple HotJava, Netscape Navigator ou Internet Explorer). Cette configuration, combinant le meilleur du Web et de Java, permet de distribuer des applications indépendamment de la plate-forme utilisée pour leur exécution.

Le butineur sait quand il doit charger une applet, un nouveau délimiteur <*APPLET*> ayant été défini à cet effet. Il lui indique l'emplacement du code Java à charger et les différents paramètres de mise en pages de l'applet. Ainsi, seule une référence au code Java est incluse dans la page HTML (contrairement à JavaScript qui inclut du code dans la page HTML).

Afin d'assurer au client une exécution sûre et lui éviter certains ennuis (virus, cheval de Troie), le comportement d'une applet est quelque peu bridé : une applet ne peut pas lire ou écrire de fichiers sur le système du client (ou alors uniquement dans des répertoires prédéfinis) :

1. Prononcez *applette*.

- une applet ne peut pas communiquer avec d'autres machines que celle depuis laquelle elle a été chargée;
- une applet ne peut pas démarrer l'exécution de programmes sur le système du client;
- une applet ne peut pas charger de programmes natifs sur le système du client (DDL, driver, etc.).

Le client peut également configurer le degré de sécurité qu'il désire. Cette souplesse provient du fait que le code Java est destiné à être exécuté sur une machine virtuelle. Celle-ci (incorporée dans le butineur Web) est le passage obligé entre le système du client et le code du programme chargé. Il est ainsi possible de contrôler le comportement de n'importe quel programme : toute opération d'accès aux ressources (fichiers, communications) devra obligatoirement transiter par la machine virtuelle, qui peut la refuser si elle n'est pas autorisée. Nous reviendrons plus en détail sur ces aspects sécurité au chapitre 27.

10.1 Comment créer une applet

Une applet est une classe Java déclarée *public* et étendant la classe *java.applet.Applet*. Nos applets auront donc toutes la forme suivante :

```
public nomApplet extends java.applet.Applet {}
```

Une applet, comme n'importe quelle classe, peut être plus ou moins complexe et faire référence à d'autres classes. Elle peut également implanter des interfaces. L'exemple suivant est une applet qui dessine un carré noir (voir applet 1) :

```
import java.awt.Graphics;

public class ComportementApplet extends java.applet.Applet {
   public void paint(Graphics g) {
      g.fillRect(5,5,40,40);
   }
}
```

Pour pouvoir exécuter cette applet, il faut tout d'abord la compiler. Le résultat de la compilation (une fois les erreurs corrigées) est un fichier portant le nom de l'applet suivi de l'extension *.class* (*ComportementApplet.class*) indiquant qu'il s'agit de code exécutable.

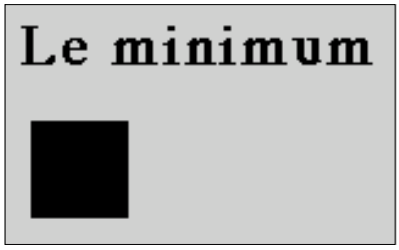
Il faut ensuite incorporer un appel à ce fichier de code dans une page HTML, à l'aide du délimiteur <*APPLET*>. L'exemple qui suit montre les paramètres à déclarer, dans le cas où la page HTML et le code de l'applet résident dans le même répertoire. Nous devons obligatoirement préciser :

- le nom du fichier de code de l'applet (*ComportementApplet.class*);
- la taille de la surface occupée par l'applet dans la page HTML lors de son exécution.

Page HTML minimale pour invoquer l'exécution de notre applet :

```
<html>
<head><title>Comportement Applet</title></head>
<body><h1>Le minimum</h1>
<applet code=ComportementApplet.class width=50 height=50>
</applet>
</body>
</html>
```

Il ne reste alors plus qu'à charger la page HTML dans un butineur capable d'exécuter du code Java pour obtenir notre première applet :



Applet 1 : Une applet minimum

Que d'efforts pour un résultat bien modeste, alors qu'une simple image *gif* ferait aussi bien l'affaire! Mais que de promesses également, si l'on y réfléchit :

- il est possible de modifier le programme de l'applet afin de lui donner un nouveau comportement sans modifier la page HTML et sans avoir à distribuer la nouvelle version aux clients;
- il est possible d'incorporer cette applet dans plusieurs pages HTML et de lui donner des comportements différents, selon les paramètres qui lui sont associés;
- il est possible que d'autres serveurs fassent référence à cette applet;
- il est possible qu'un programmeur développe sur un Macintosh du code destiné à des utilisateurs de PC ou de stations Unix (et vice versa);
- il est possible de transformer une page HTML en tableur, en modeleur 3D, en cockpit d'avion, en machine à sous, etc.

10.2 Le délimiteur <*Applet*> en détail

Nous venons de constater qu'il est nécessaire de préciser dans le délimiteur <*APPLET*> la source du code à charger et la taille de l'espace occupé par l'applet dans la page HTML lors de son exécution. Le diagramme 10.1 indique qu'il est aussi possible d'associer des paramètres à l'applet qu'elle pourra récupérer au moment de son exécution.

Certains butineurs étant encore incapables d'exécuter du code Java, il faut pré-

voir dans le code HTML un texte de substitution à l'applet. Deux solutions existent :

- placer du code HTML entre les deux délimiteurs d'invocation <*APPLET*> et <*/APPLET*>. Ce code sera alors interprété par le butineur;
- ajouter l'option *alt=texte* dans le délimiteur. Ce texte sera affiché dans le cas où le butineur connaît le délimiteur <*APPLET*> mais qu'il ne peut pas exécuter l'applet.

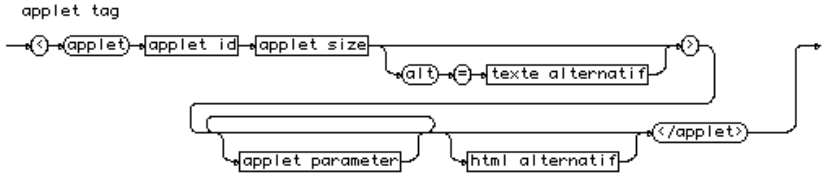


Diagramme 10.1 applet_tag

Le diagramme 10.2 décrit les règles d'identification de l'applet. L'option *code* est obligatoire, elle définit le nom du fichier contenant le code de l'applet. Le chemin d'accès doit être impérativement défini dans l'option *codebase* s'il est différent de celui de la page HTML contenant l'invocation de l'applet. L'option *name* permet de donner un nom à l'applet lors de son exécution dans le butineur. Plusieurs applets peuvent ainsi cohabiter dans une même page et éventuellement collaborer à une tâche (échanger des informations, se synchroniser).

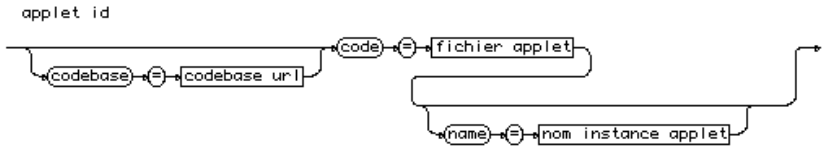


Diagramme 10.2 applet_id

La taille de l'applet est définie à l'aide des deux paramètres *width* (largeur) et *height* (hauteur), indiqués en pixels. On peut également préciser l'alignement de l'applet dans la page et l'espace à laisser vide autour de l'applet (diagramme 10.3).

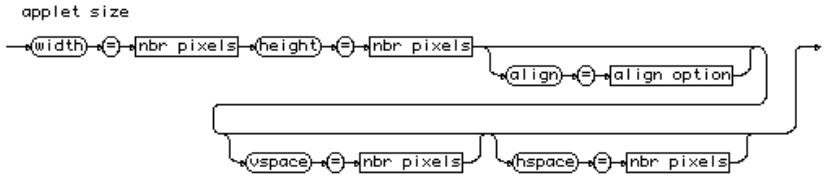


Diagramme 10.3 applet size

Examinons ces différentes options à l'aide d'exemples. Ainsi, l'applet 2 montre que :

- l'espace occupé par l'applet se comporte de façon similaire à l'espace occupé par une image;
- l'alignement horizontal situe l'applet par rapport au texte;
- la même applet peut être invoquée plusieurs fois dans une même page.

```
<html>
<head><title>Comportement Applet</title></head>
<h1>comme une image</h1>
<applet code=ComportementApplet.class width=50 height=50
align=left>
   votre butineur ne sait pas exécuter du Java, dommage!<p>
</applet>
Ce texte doit continuer à droite de l'applet et couler sans problème
comme avec une image.<p>
<applet code=ComportementApplet.class width=50 height=50
align=right>
</applet>
Ce texte doit continuer à gauche de l'applet et couler sans problème
comme avec une image.<p>
</body>
</html>
```

Applet 2 : Emplacements occupés par une applet dans une page HTML

L'applet 3 illustre l'utilisation des options *vspace* et *hspace* qui permettent de définir un cadre vide autour de la surface occupée par l'applet :

```
<html>
<head><title>Comportement Applet</title></head>
<applet code=ComportementApplet.class width=50 height=50
     align=left vspace=50 hspace=30>
votre butineur ne sait pas executer du Java, dommage!<p>
</applet>
<h1>alignement</h1>
Ce texte doit continuer à droite du cadre de l'applet et couler sans
problème comme avec une image.<p>
</body>
</html>
```

Enfin, examinons les possibilités d'alignement vertical (par le haut, par le milieu et par le bas). Chaque alignement peut agir de deux manières : l'une relative au

texte actuellement utilisé dans la ligne, l'autre absolue. Cette seconde manière permet de s'aligner sur les éléments les plus grands de la ligne courante.

Applet 3 : Cadre vide autour d'une applet

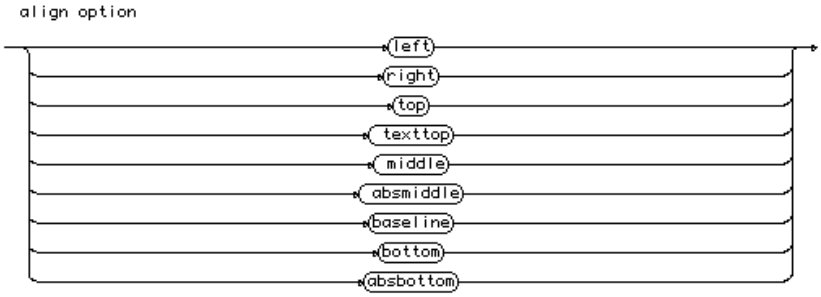
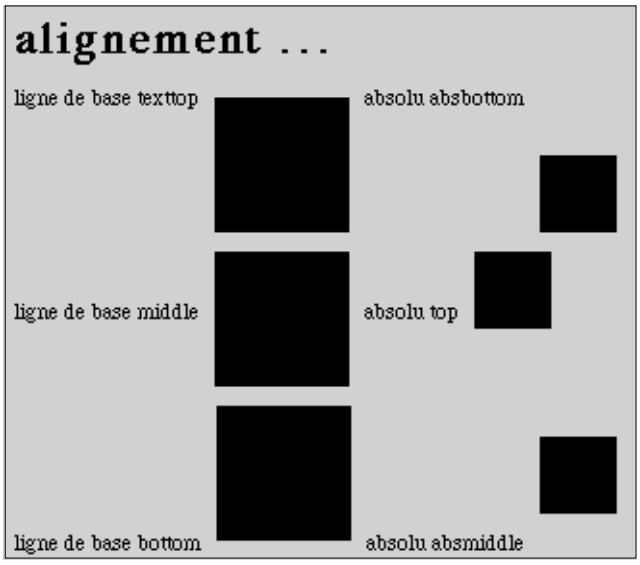


Diagramme 10.4 align options

L'applet 4 illustre ces possibilités à l'aide d'une deuxième applet dessinant un carré plus grand, afin de créer des différences de taille dans la même ligne.

```
<html>
<head><title>Comportement Applet</title></head>
<h1>alignement ...</h1>
ligne de base texttop
<applet code=ComportementApplet2.class width=80 height=80
align=texttop></applet>
absolu absbottom
<applet code=ComportementApplet.class width=50 height=50
align=absbottom></applet><br>
ligne de base middle
<applet code=ComportementApplet2.class width=80 height=80
align=middle></applet>
absolu top
<applet code=ComportementApplet.class width=50 height=50
align=top></applet><br>
ligne de base bottom
<applet code=ComportementApplet2.class width=80 height=80
align=bottom></applet>
absolu absmiddle
<applet code=ComportementApplet.class width=50 height=50
align=absmiddle></applet><br>
</body>
</html>
```



Applet 4 : Alignement d'applets dans une page HTML

Comme nous l'avons indiqué au point 10.1, une applet peut recevoir des paramètres inclus dans la page HTML, ce qui permet d'étendre son comportement (ainsi que celui de HTML). Chaque paramètre est enfermé dans un délimiteur, auquel on associe un nom et une valeur (diagramme 10.5).

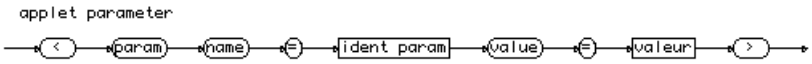


Diagramme 10.5 applet parameter

Dans l'applet 5, nous utilisons Java pour dessiner des rectangles noirs de différentes tailles données en paramètres.

Pour accéder à un paramètre depuis l'applet, nous faisons appel à une méthode *getParameter("nom_du_paramètre")* qui permet de récupérer (dans une chaîne de caractères) la valeur associée à ce paramètre. Ensuite, nous transformons cette chaîne de caractères en un entier à l'aide de la méthode appropriée.

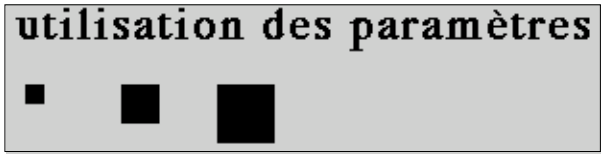
Ce programme utilise également la méthode *resize()* qui demande au butineur de redimensionner la taille occupée par l'applet (les paramètres *width* et *height* sont alors redéfinis).

```
import java.awt.Graphics;

public class BlackRect extends java.applet.Applet {

   String parametre;
   int tailleRect;
```

```
    public void init() {
       parametre = getParameter("Rect");
       if (parametre == null)
          tailleRect = 50;
       else
          tailleRect = Integer.parseInt(parametre);
       resize(tailleRect+10, tailleRect+10);
    }

    public void paint(Graphics g) {
       g.fillRect(5,5,tailleRect,tailleRect);
    }
}
```

Code HTML invoquant l'applet avec différentes valeurs de paramètres :

```
<html>
<head><title>Comportement Applet</title></head>
<body>
<h1>utilisation des paramètres</h1>
<applet code=BlackRect.class width=50 height=50 align=left>
   <PARAM NAME=Rect VALUE="10"></applet>
<applet code=BlackRect.class width=50 height=50 align=left>
   <PARAM NAME=Rect VALUE="20"></applet>
<applet code=BlackRect.class width=50 height=50 align=left>
   <PARAM NAME=Rect VALUE="30"></applet>
</body>
</html>
```



Applet 5 : Utilisation des paramètres d'une applet

On peut aisément imaginer d'ajouter d'autres paramètres permettant de spécifier la couleur ou la forme des objets géométriques à dessiner. On verra plus loin qu'il est possible de spécifier le chargement d'un ensemble de classes dans une applet au moyen des fichiers d'archive *.jar*.

Nous avons placé le code permettant de récupérer les paramètres dans une méthode *init()* appartenant à l'applet. Voyons cela de plus près, en étudiant le cycle de vie des applets, avant d'en programmer de plus compliquées.

10.3 Vie et mort d'une applet

Le cycle de vie d'une applet se compose de l'exécution successive des méthodes :

- *init()* : après son chargement (ou lors d'un rechargement), l'applet exécute cette méthode d'initialisation;
- *start()* : est invoquée directement après l'initialisation, ainsi qu'après un *stop(),* si le document contenant l'applet redevient à nouveau visible pour l'utilisateur;
- *stop()* : est invoquée lorsque le document contenant l'applet disparaît (changement de page HTML); en redéfinissant *stop()*, on peut interrompre un traitement (par exemple une animation) si l'applet n'est plus visible par l'utilisateur;
- *destroy()* : est invoquée avant la disparition de l'applet, afin de restituer les ressources et de quitter proprement le système.

On peut redéfinir chacune de ces méthodes si l'on tient à donner un comportement particulier à l'applet durant ces transitions.

En fait, l'applet hérite de la classe *java.awt.Component* et sa principale activité sera *paint()*. Ceci explique pourquoi notre première applet redéfinissait cette méthode.

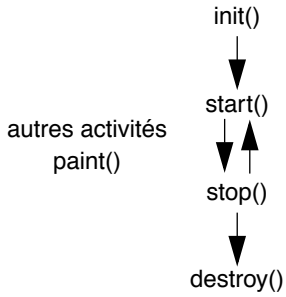


Figure 10.6 Cycle de vie d'une applet

Chapitre 11
Dessiner

Nous allons examiner comment dessiner dans l'espace occupé par une applet. Pour cela, nous décrivons en détail la classe *Graphics* qui contient de nombreuses méthodes de dessin. Elle se trouve dans le package *java.awt* et doit être importée par toute classe désirant l'utiliser.

L'activité principale d'une applet « graphique » est définie dans sa méthode *paint()*. Celle-ci est invoquée lors d'événements nécessitant de redessiner la surface occupée par l'applet. La méthode *paint()* reçoit comme paramètre un objet de la classe *Graphics* décrivant l'environnement graphique courant (couleurs, fond). Nos applets auront donc l'allure suivante :

```
import java.awt.Graphics;
public class Figure extends java.applet.Applet {
   public void paint(Graphics g) {
   ...
   }
}
```

11.1 La feuille

La feuille de dessin est représentée par une matrice de points dont chaque coordonnée est définie par un couple d'entiers. La classe *Point* (variables d'instance *x* pour horizontal et *y* pour vertical) permet de manipuler des coordonnées. La méthode get*Size()* fournit la taille de la feuille sous la forme d'un objet de la classe *Dimension* (variables d'instance *width* : largeur et *height* : hauteur). Par conséquent, les coordonnées des quatre coins de la feuille sont :

- *(0 , 0)* : le coin supérieur gauche;
- (get*Size().width* , 0) : le coin supérieur droit;

- (0 , get*Size().height*) : le coin inférieur gauche;
- (get*Sizesize().width,* get*Size().height*) : le coin inférieur droit.

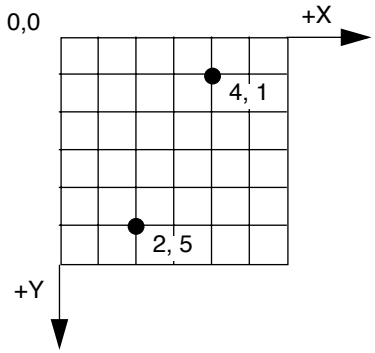


Figure 11.1 Système de coordonnées de Graphics

11.2 Dessiner une ligne

Par défaut, la feuille est remplie avec une couleur de fond (*background*) grise, les dessins apparaissant en noir. Pour dessiner une ligne, il faut spécifier son point de départ et son point d'arrivée. L'applet 6 dessine une ligne horizontale[1] :

```
public void paint(Graphics g) {
  g.drawLine(50,25,100,25);
}
```

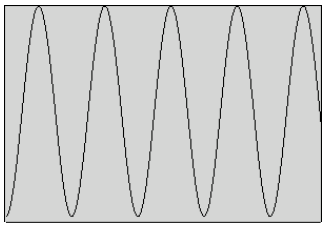
Applet 6 : Dessiner une ligne

En dessinant un ensemble de segments de droites (applet 7), on peut dessiner une fonction f (ne faites pas attention aux appels à *getSize()*, ils sont là pour cadrer la fonction cosinus dans la feuille de dessin). La boucle *for* calcule, pour chaque coordonnée *x* de la feuille, le segment de droite qui le relie au prochain point de la courbe.

```
import java.awt.Graphics;
public class Figure extends java.applet.Applet {
  double f(double x) {
    return (Math.cos(x/10)+1) * getSize().height / 2;
  }
  public void paint(Graphics g) {
```

1. On ne présente que les extraits significatifs des applets. Le texte complet de l'applet contient : import...; class...; etc.

```
      for (int x = 0 ; x < getSize().width ; x++) {
        g.drawLine(x, (int)f(x), x + 1, (int)f(x + 1));
      }
    }
  }
```



Applet 7 : Dessiner une fonction (cosinus)

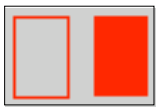
11.3 Rectangles

La classe *Graphics* propose de nombreuses méthodes capables de dessiner des formes géométriques : rectangles, polygones, ovales et arcs. Pour chacune de ces formes, il existe une paire de méthodes : *draw...* qui dessine le contour de la forme et *fill...* qui remplit la forme. Dans le cas des rectangles, nous devons indiquer les paramètres suivants dans l'ordre :

- *int x* : la coordonnée x du coin supérieur gauche;
- *int y* : la coordonnée y du coin supérieur gauche;
- *int width* : la largeur du rectangle;
- *int height* : la hauteur du rectangle.

L'applet 8 ci-dessous dessine deux rectangles côte à côte :

```
public void paint(Graphics g) {
  g.drawRect(50,30,20,30);
  g.fillRect(80,30,20,30);
}
```



Applet 8 : Dessiner des rectangles

La classe *Graphics* propose également des rectangles aux coins arrondis; il faut alors ajouter deux paramètres :

- *int arrondiX* : largeur de l'arrondi (axe des X);
- *int arrondiY* : hauteur de l'arrondi (axe des Y).

L'applet 9 le montre en dessinant deux rectangles arrondis côte à côte :

```
public void paint(Graphics g) {
  g.drawRoundRect(50,70,20,30,15,15);
  g.fillRoundRect(80,70,20,30,15,20);
}
```

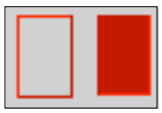
Applet 9 : Dessiner des rectangles aux coins arrondis

Enfin, des rectangles peuvent être dessinés en relief (peu visible selon les butineurs). Un paramètre supplémentaire de type booléen indique :

- *false* : le rectangle est en relief;
- *true* : le rectangle est en creux.

L'applet 10 dessine un rectangle en creux (à gauche) et un en relief (à droite) :

```
public void paint(Graphics g) {
  g.draw3DRect(50,110,20,30,true);
  g.fill3DRect(80,110,20,30,false);
}
```



Applet 10 : Dessiner des rectangles en relief

11.4 Polygones

Le dessin d'un polygone requiert deux tableaux de coordonnées. L'un pour les valeurs en X et l'autre pour les valeurs en Y. Un polygone n'est pas fermé automatiquement.

Le nombre de points à dessiner doit être indiqué au moment de l'appel à la méthode de dessin.

L'applet 11 dessine deux polygones, dont l'un est *rempli* en invoquant la méthode *fillPolygon()*. On notera au passage la manière d'initialiser des tableaux et celle de récupérer leur taille grâce à la méthode *length() :*

```
public void paint(Graphics g) {
  int listeX[]={50,40,80,100,55};
  int listeY[]={150,170,200,170,160};
  int nbrXY=listeX.length;  // nombre de points à dessiner
  g.drawPolygon(listeX, listeY, nbrXY);
  int listeY2[]={200,220,250,220,210};
  g.fillPolygon(listeX, listeY2, nbrXY);
}
```

11.5 Cercles, ovales, arcs

Les formes ovales (cercle, ovale, arc) pouvant toutes s'inscrire dans un rectangle, leurs méthodes de dessin se basent sur les dimensions de celui-ci. Les paramètres utilisés sont : *x, y, largeur, hauteur* (*x* et *y* indiquant le coin supérieur gau-

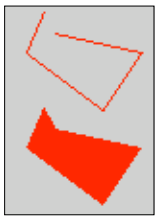
che du rectangle). À noter que pour obtenir un cercle, *hauteur* doit être égale à *largeur*. L'applet 12 dessine ainsi un ovale et un cercle :

```
public void paint(Graphics g) {
  g.drawOval(120,30,20,30);
  g.fillOval(150,30,30,30);
}
```

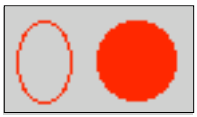
Il est également possible de dessiner des arcs. Deux paramètres supplémentaires (exprimés en degrés) sont nécessaires : le premier indique l'angle de départ de l'arc, le second les degrés nécessaires à la réalisation du dessin.

L'applet 13 dessine deux arcs. On remarquera que l'on peut associer des valeurs négatives aux paramètres exprimés en degrés.

```
public void paint(Graphics g) {
  g.drawArc(120,70,20,30,45,180);
  g.fillArc(150,70,30,30,45,-150);
}
```



Applet 11 : Dessiner des polygones



Applet 12 : Dessiner des cercles et des ovales

Applet 13 : Dessiner des arcs

11.6 Gommer, copier

La méthode *clearRect()* efface une zone rectangulaire :

```
g.clearRect (x, y, largeur, hauteur);
```

La méthode *copyArea()* copie une zone rectangulaire à une nouvelle coordonnée *(nx, ny)* :

```
g.copyArea(x, y, largeur, hauteur, nx, ny);
```

11.7 Colorier

Le package *awt* définit une classe *Color* pour représenter les couleurs. Une couleur est codée sur 24 bits : huit pour chacune des trois couleurs fondamentales de la vidéo (rouge, vert, bleu), ce qui représente environ seize millions de possibilités. Le nombre de couleurs effectivement affichées dépend bien entendu des capacités de votre écran et du butineur. La classe *Color* propose également une palette de couleurs prédéfinies (voir tableau 11.1).

Couleur	Nom de couleur	Rouge, Vert, Bleu
Blanc	`Color.white`	`255,255,255`
Gris pâle	`Color.lightGray`	`192,192,192`
Gris	`Color.gray`	`128,128,128`
Gris foncé	`Color. darkGray`	`64,64,64`
Noir	`Color.black`	`0,0,0`
Rouge	`Color.red`	`255,0,0`
Vert	`Color.green`	`0,255,0`
Bleu	`Color.blue`	`0,0,255`
Jaune	`Color.yellow`	`255,255,0`
Magenta	`Color. magenta`	`255,0,255`
Cyan	`Color. cyan`	`0,255,255`
Rose	`Color.pink`	`255,175,175`
Orange	`Color.orange`	`255,200,0`

Tableau 11.1 Couleurs prédéfinies dans la classe Color

On peut définir de nouvelles couleurs et étendre la palette à l'aide de la méthode *Color(R, V, B)* où les valeurs de *R*, *V* et *B* sont comprises entre 0 et 255 (ou entre 0.0 et 1.0). Exemple :

```
Color bleuPale = new Color(0,0,80);
```

La couleur du fond (arrière-plan) est définie par la méthode *setBackground(Color)* de la classe *Component* (*Applet* hérite de cette classe abstraite). L'exemple suivant impose un fond blanc :

```
setBackground(Color.white);
```

La méthode *getBackground()* de la classe *Component* retourne la couleur ac-

tuelle du fond tandis que la méthode *setForeground(Color)* impose une couleur de dessin, par exemple le rose :

```
setForeground(Color.pink);
```

La classe *Graphics* propose encore les méthodes *setColor(Color)* pour redéfinir la couleur avec laquelle on dessine et *getColor()* pour connaître la couleur de dessin courante.

Regardez la documentation pour les autres constructeurs de *Color()*. Il existe deux méthodes *brigther()* et *darker()* qui permettent d'obtenir des couleurs plus claires et plus foncées ayant la même tonalité que la couleur sur laquelle elles sont appliquées.

11.8 Écrire

La classe *Graphics* traite non seulement des formes géométriques mais également du texte de différentes tailles, fontes, couleurs, etc.

La méthode *drawString()* permet de dessiner une chaîne de caractères à partir d'une coordonnée *(x,y)*. L'applet 14 montre que la couleur courante est également utilisée pour le dessin des caractères.

```
import java.awt.Graphics;
import java.awt.Color;

public class MaPremiereApplet extends java.applet.Applet {

  public void paint(Graphics g) {
    g.setColor(Color.black);
    g.drawString("Ma première applet politisée!", 50, 30);
    g.setColor(Color.blue);
    g.drawString("Liberté", 50, 60);
    g.setColor(Color.white);
    g.drawString("Egalité", 50, 90);
    g.setColor(Color.red);
    g.drawString("Fraternité", 50, 120);
  }
}
```

Applet 14 : Dessiner du texte en couleur

11.9 Les polices

Les polices de caractères *(f*ontes) déterminent l'apparence des caractères et des textes affichés. Elles sont gérées par la classe *java.awt.Font*. Pour construire une nouvelle fonte, il faut instancier un objet *Font* et l'initialiser avec les paramètres suivants :

- le nom de la police : *Helvetica, TimesRoman, Courier* par exemple sont des fontes disponibles sur tous les butineurs;
- le style des caractères : trois constantes sont à disposition – *Font.PLAIN* (normal), *Font.BOLD* (**gras**), Font.ITALIC (*italique*). Il est possible de spécifier des styles composés en additionnant ces constantes;
- la taille des caractères : elle est indiquée en nombre de points.

Comme pour les couleurs, il est nécessaire de définir la police à utiliser, ce que réalise la méthode *setFont()*.

Dans l'applet 15, nous définissons plusieurs polices que nous utilisons au fur et à mesure de nos besoins.

```
import java.awt.Graphics;
import java.awt.Color;
import java.awt.Font;

public class MaPremiereApplet extends java.applet.Applet {

  public void paint(Graphics g) {
    Font helvetica14Normal = new
         Font("Helvetica",Font.PLAIN,14);
    Font courier12Gras = new
         Font("Courier",Font.BOLD,12);
    Font timesRoman18Italic = new
         Font("TimesRoman",Font.ITALIC,18);
    Font timesRoman18ItalicGras = new
         Font ("TimesRoman",Font.ITALIC+Font.BOLD,18);
    g.setColor(Color.black);
    g.setFont(helvetica14Normal);
    g.drawString("Le même avec emphase!", 50, 30);

    g.setColor(Color.blue);
    g.setFont(courier12Gras);
    g.drawString("Liberté", 50, 60);

    g.setColor(Color.white);
    g.setFont(timesRoman18Italic);
    g.drawString("Egalité", 50, 90);

    g.setColor(Color.red);
    g.setFont(timesRoman18ItalicGras);
    g.drawString("Fraternité", 50, 120);
  }
}
```

Applet 15 : Dessiner des textes dans plusieurs fontes

La classe *Font* fournit des informations à propos d'une fonte (voir tableau 11.2). Ainsi, la police courante est obtenue à l'aide de la méthode *getFont()* appliquée à un composant graphique.

Méthode	Description (information à propos de la fonte)
`getName()`	Retourne un string indiquant le nom de la fonte.
`getSize()`	Retourne un entier indiquant la taille de la fonte.
`getStyle()`	Retourne un entier indiquant le style (0, **1**, 2, *3*).
`isPlain()`	Retourne un booléen, vrai si le style est normal.
`isBold()`	Retourne un booléen, vrai si le style est **gras**.
`isItalic()`	Retourne un booléen, vrai si le style est *italique*.

Tableau 11.2 Méthodes de la classe Font

Java 2 a particulièrement développé la class *Font*. Les concepts de caractère et de glyphe sont entièrement séparés. Il est donc possible d'utiliser les ligatures, par exemple les deux caractères, *f* suivi de *i* deviennent un seul glyphe *fi*. Les police connues par Java sont toutes celles connues par le système sur lequel s'exécute le code. La méthode *getAllFonts()* permet d'obtenir cette liste. Il existe des méthodes pour créer, dériver et transformer des polices existantes et ainsi d'en obtenir de nouvelles. Il n'existe donc à priori aucune limitation typographique pour le développeur Java. Il ne lui suffit que d'un peu de courage pour entrer dans cet univers!

En utilisant des polices connues d'un seul système d'exploitation, on risque de perdre la mobilité du code Java. Il faut donc obtenir un équilibre entre la mobilité et l'esthétique.

Pour permettre la justification précise des caractères, la classe *java.awt.FontMetrics* calcule la taille occupée par une chaîne de caractères. Il sera donc possible, par exemple, de centrer un texte dans une applet dont la taille varie. Précisons que toutes ces méthodes utilisent le **point** comme unité.

Méthode	Description (information à propos d'une chaîne)
`stringWidth()`	Retourne un entier indiquant la largeur d'une chaîne.
`charWidth()`	Retourne un entier indiquant la largeur d'un caractère.
`getAscent()`	Retourne un entier indiquant la hauteur de la fonte au-dessus de la ligne de base.
`getDescent()`	Retourne un entier indiquant la hauteur de la fonte au-dessous de la ligne de base.
`getLeading()`	Retourne un entier indiquant l'espace entre les lignes.
`getHeight()`	Retourne un entier indiquant la hauteur totale de la fonte.

Tableau 11.3 Méthodes de la classe FontMetrics

Dans l'applet 16, nous désirons centrer nos chaînes de caractères. Nous construisons à cet effet une méthode *writeCenter()* qui reçoit pour paramètres :

- *g* le composant graphique dans lequel elle va dessiner;
- *s* la chaîne à écrire;
- *y* la hauteur de cette chaîne.

La méthode *getFontMetrics(g.getFont())* permet de connaître la métrique actuellement utilisée par le composant graphique.

L'instruction *java.awt.** importe toutes les classes du package *awt* :

```
import java.awt.*;

public class MonPremierApplet extends java.applet.Applet {
    Font helvetica18Normal = new
         Font("Helvetica",Font.PLAIN,18);

  public void writeCenter(Graphics g, String s, int y){
    FontMetrics metriqueCourante=getFontMetrics(g.getFont());
    g.drawString(
         s,
         (getSize().width-metriqueCourante.stringWidth(s))/2,
         y);
    }

  public void paint(Graphics g) {
    g.setColor(Color.blue);
    g.setFont(helvetica18Normal);
    writeCenter(g, "Le même avec emphase au centre!", 30);
    writeCenter(g, "Liberté",  60);
    writeCenter(g, "Egalité",  90);
    writeCenter(g, "Fraternité",  120);
  }
}
```

Applet 16 : Dessiner du texte centré

Cet exemple est un peu simplifié. Dans l'absolu, on ne connaît pas la hauteur des caractères qui vont s'imprimer et nous aurions dû interroger la fonte sur sa taille avec les méthodes de *FontMetrics* et non utiliser un interligne fixe. La classe *FontMetrics* propose d'autres méthodes que celles décrites ci-dessus pour mesurer l'espace pris par une chaîne de caractères. À noter la méthode *getStringBounds()* qui existe en plusieurs variantes et qui retourne un objet *Rectangle2D* donnant les dimensions du texte à afficher dans un objet *Graphics* (une applet, par exemple).

Dans l'exemple suivant, on a utilisé cette méthode pour redéfinir la méthode *writeCenter()*. On remarquera la conversion de type (casting) vers le type *Rectangle* qui possède la méthode *width()* et qui est sous-classe de *Rectangle2D*.

```
public void writeCenter(Graphics g, String s, int y){
   FontMetrics metriqueCourante= getFontMetrics(g.getFont());
   g.drawString(s,(getSize().width-
      ((Rectangle)metriqueCourante.getStringBounds(s,g)).width)/2,
y);
```

11.10 Représenter la troisième dimension en couleur

Nous allons maintenant étudier un exemple qui utilise largement les couleurs. Notre objectif est de représenter la troisième dimension par l'intensité d'une couleur.

Nous importons tout d'abord les classes nécessaires :

```
import java.awt.Graphics;
import java.awt.Color;
```

Notre programme sera une applet :

```
public class Figure extends java.applet.Applet {
```

La fonction *f* à représenter est le sinus du carré de la distance d'un point sur le plan par rapport à l'origine (ou plus simplement : une sorte de vague oscillant de plus en plus vite). La valeur de *f* est comprise entre 1.0 et -1.0.

```
  double f(double x, double y) {
    return (Math.sin((x*x+y*y)/5000));
  }
```

Nous redéfinissons la méthode *paint()* et déclarons une variable pour chacune des couleurs de base. *offsetx* et *offsety* sont des constantes nous permettant de recentrer le dessin dans l'espace occupé par l'applet :

```
public void paint(Graphics g) {

  int rouge, vert, bleu;
  int offsetx, offsety;
```

Nous initialisons nos variables (on peut aussi le faire lors de la déclaration!). *getSize().width* et *getSize().height* permettent de récupérer la taille de la zone de dessin.

```
  vert=0;
  bleu = 128;
  offsetx=getSize().width/2;
  offsety=getSize().height/2;
```

Nous allons uniquement faire varier la nuance de la couleur rouge (de 0 à 255) en fonction de la valeur de *f* pour chaque coordonnée du plan.

La première boucle balaie les valeurs *x* (axe horizontal) :

```
for (int x = 0 ; x < getSize().width ; x++) {
```

La deuxième boucle balaie les valeurs *y* (axe vertical) :

```
    for (int y = 0 ; y < getSize().height ; y++) {
```

Pour chaque coordonnée *(x,y),* on calcule la valeur de la fonction et on la convertit en une nuance de rouge :

```
      rouge= (int) ((f(x-offsetx,y-offsety)+1)*128);
```

Nous pouvons alors définir une nouvelle couleur et l'utiliser pour dessiner :

```
      g.setColor(new Color(rouge,vert,bleu));
```

Nous dessinons le point *(x,y)* (une ligne dont les extrémités sont confondues) :

```
      g.drawLine(x,y,x,y);
```

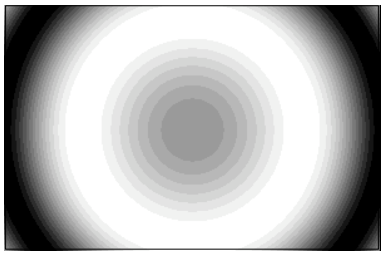
Nous refermons tous les blocs d'instructions ouverts :

```
    }
  }
 }
}
```

Le programme complet de représentation de la troisième dimension à l'aide de nuances de rouge est donné ci-dessous (voir applet 17) :

```
import java.awt.Graphics;
import java.awt.Color;

public class Figure extends java.applet.Applet {
```

```
  double f(double x, double y) {
    return (Math.sin((x*x+y*y)/5000));
  }

  public void paint(Graphics g) {
    int rouge, vert, bleu;
    int offsetx, offsety;

    vert=0;
    bleu = 128;
    offsetx=getSize().width/2;
    offsety=getSize().height/2;

    for (int x = 0 ; x < getSize().width ; x++) {
      for (int y = 0 ; y < getSize().height ; y++) {
        rouge= (int) ((f(x-offsetx,y-offsety)+1)*128);
        g.setColor(new Color(rouge,vert,bleu));
        g.drawLine(x,y,x,y);
      }
    }
  }
}
```



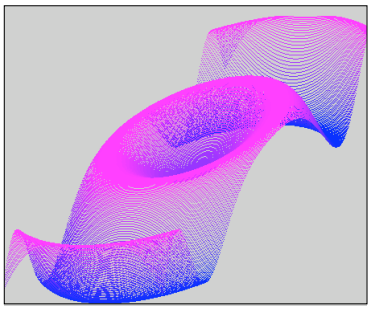
Applet 17 : Représenter la 3e dimension à l'aide de couleurs

11.11 Représenter la 3D par une projection en 2D

L'applet 18 est basée sur le même principe que la précédente : la couleur met en évidence le relief. Cependant, nous avons effectué une projection à 45 degrés permettant de dessiner la courbe en 3D (3 dimensions). Nous avons déplacé les variables hors de la déclaration de *paint()*, sauf pour *offsetx* et *offsety* dont les valeurs sont initialisées avec la taille de la feuille de dessin, disponible uniquement dans *paint()*.

```
import java.awt.Graphics;
import java.awt.Color;

public class Figure extends java.applet.Applet {
  int rouge;
  int vert=0;
  int bleu= 255;
  int x3,y3;
```

```
    double p45=0.707;

    double f(double x, double y) {
      return  (Math.sin((x*x+y*y)/5000));
    }

    public void paint(Graphics g) {
      int offsetx=(getSize().width+(int)( p45*getSize().height))/2;
      int offsety=(getSize().height)/2;

      for (int x=(int)(p45*getSize().height);x<getSize().width;x++){
         for (int y = 0; y<getSize().height; y++) {
           x3=(int) (x-y*p45);
           y3=(int) (y*p45-50*f(x-offsetx,y-offsety)
                     +offsety*p45/2);
           rouge= (int) ((f(x-offsetx,y-offsety)+1)*128);
           g.setColor(new Color(rouge,vert,bleu));
           g.drawLine(x3,y3,x3,y3);
         }
      }
    }
}
```



Applet 18 : Projeter une figure en 3D sur une surface 2D

11.12 Des rectangles multicolores

Nous allons maintenant dessiner des rectangles dont la taille, la position et la couleur sont choisies au hasard. Nous utilisons à cet effet le générateur de nombres aléatoires *Math.random()* du package *java.lang*. Ce générateur retourne des nombres flottants entre 0.0 et 1.0 qui nous serviront à générer les couleurs et les rectangles.

Nous importons les classes nécessaires, dont la classe *Rectangle* :

```
import java.awt.Graphics;
import java.awt.Color;
import java.awt.Rectangle;
```

Nous déclarons trois variables (une pour chaque couleur de base) et une instance de la classe *Rectangle* :

```
public class Figure extends java.applet.Applet {
  int rouge,vert,bleu;
  Rectangle  r = new Rectangle ();
```

Nous allons dessiner cent rectangles :

```
  public void paint(Graphics g) {
    for (int i = 0 ; i < 100 ; i++) {
```

Pour chaque rectangle, nous générons une couleur à partir de trois intensités tirées au hasard (notez les conversions en nombres entiers) :

```
      vert  = (int)(Math.random()*255.99);
      bleu  = (int)(Math.random()*255.99);
      rouge = (int)(Math.random()*255.99);
      g.setColor(new Color(rouge,vert,bleu));
```

Pour chaque rectangle, nous tirons au hasard *x* et *y*, (la coordonnée de son coin supérieur gauche) et lui attribuons une taille nulle :

```
      r.x= (int)(Math.random()*getSize().width);
      r.y= (int)(Math.random()*getSize().height);
      r.width=0;
      r.height=0;
```

Ensuite, nous utilisons la méthode *add()* qui permet de redimensionner le rectangle afin qu'il contienne un point tiré lui aussi au hasard :

```
      r.add((int)(Math.random()*getSize().width),
            (int)(Math.random()*getSize().height));
```

Finalement, nous dessinons le rectangle. On remarquera que l'on ne peut pas le dessiner directement, mais que l'on doit faire appel à ses variables d'instance (*x, y, largeur, hauteur*) qui, elles, le définissent :

```
      g.fillRect(r.x,r.y,r.width,r.height);
    }
   }
}
```

Le code complet de l'applet 19 est le suivant :

```
import java.awt.Graphics;
import java.awt.Color;
import java.awt.Rectangle;

public class Figure extends java.applet.Applet {
  int rouge,vert,bleu;
  Rectangle  r = new Rectangle ();

  public void paint(Graphics g) {
    for (int i = 0 ; i < 100 ; i++) {
      vert  = (int)(Math.random()*255.99);
```

```
        bleu  = (int)(Math.random()*255.99);
        rouge = (int)(Math.random()*255.99);
        g.setColor(new Color(rouge,vert,bleu));
        r.x= (int)(Math.random()*getSize().width);
        r.y= (int)(Math.random()*getSize().height);
        r.width=0; r.height=0;
        r.add((int)(Math.random()*getSize().width),
              (int)(Math.random()*getSize().height));
        g.fillRect(r.x,r.y,r.width,r.height);
      }
    }
}
```

Applet 19 : Dessiner des rectangles aléatoires

11.13 Esquisser un rectangle à l'aide d'ovales

Ce programme (applet 20) est une variante du précédent. Nous avons déclaré une zone interdite et nous testons systématiquement s'il y a intersection entre le rectangle généré et la zone interdite. Si ce n'est pas le cas, on dessine l'**ovale** inscrit dans le rectangle généré.

```
import java.awt.Graphics;
import java.awt.Color;
import java.awt.Rectangle;

public class Figure extends java.applet.Applet {
  public void paint(Graphics g) {
    int rouge,vert,bleu;
    Rectangle  r = new Rectangle ();
    Rectangle  interdit = new Rectangle (
                              (int)(getSize().width/3),
                              (int)(getSize().height/3),
                              (int)(getSize().width/3),
                              (int)(getSize().height/3));
    for (int i = 0 ; i < 1000 ; i++) {
      vert  = (int)(Math.random()*255.99);
      bleu  = (int)(Math.random()*255.99);
      rouge = (int)(Math.random()*255.99);
```

```
        g.setColor(new Color(rouge,vert,bleu));
        r.x= (int)(Math.random()*getSize().width);
        r.y= (int)(Math.random()*getSize().height);
        r.width=0; r.height=0;
        r.add((int)(Math.random()*getSize().width),
          (int)(Math.random()*getSize().height));
        if (!r.intersects(interdit)) // test d'intersection
            g.fillOval(r.x,r.y,r.width,r.height);
      }
    }
}
```

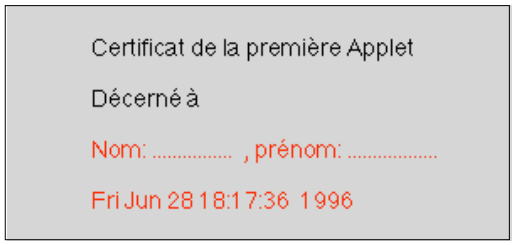
Applet 20 : Dessiner un rectangle à l'aide d'ovales

11.14 Première et dernière

Vous nous avez suivis jusqu'ici, alors félicitations! Nous sommes heureux de vous décerner un prix : la dernière applet de cette partie. Nous en profitons pour vous présenter une nouvelle classe, *java.util.Date*, qui permet de manipuler les dates, les heures, etc. Cette classe servira d'exemple pour nos futures animations. Le constructeur *new Date()* retourne la date courante maintenue dans l'ordinateur qui exécute l'applet. La méthode *toString()* appliquée à un objet *Date* permet de convertir cette date en une chaîne de caractères.

```
import java.awt.Graphics;
import java.awt.Color;
import java.awt.Font;
import java.util.Date;

public class MaPremiereApplet extends java.applet.Applet {
  Font helvetica14Normal = new
       Font("Helvetica",Font.PLAIN,14);
  Date maintenant;

  public void paint(Graphics g) {
    g.setColor(Color.black);
    g.setFont(helvetica14Normal);
    g.drawString("Certificat de la première Applet", 50, 30);
    g.drawString("Décerné à", 50, 60);
```

```
        g.setColor(Color.red);
        g.drawString("Nom: ..........  , prénom: .........", 50, 90);
        maintenant = new Date();
        g.drawString(maintenant.toString(), 50, 120);
        }
    }
```



Applet 21 : Certificat de la 1re applet

11.15 Invitation à explorer

Examinez les méthodes de la classe *java.awt.Color : brighter(), darker(), getBlue(), getColor(String), getColor(String,Color), getGreen(), getHSBColor(float, float, float)*, etc.

Esquissez un ovale à l'aide de rectangles, étudier l'effet de la taille maximum des rectangles sur l'aspect de l'ovale.

Dessinez une spirale à l'aide de quarts de cercles, en modifiant leur rayon.

Dessinez une spirale dont la couleur varie de manière dégradée ou aléatoire.

Chapitre 12
Animer

Avant de découvrir comment faire de l'animation en Java, il est important de bien comprendre le cycle de vie de l'applet. Nous rappelons que celui-ci est :

init() → *start()* ↔ *stop()* → *destroy()*

Après l'activité *start()*, l'applet exécute la méthode *paint()* du composant graphique. C'est en redéfinissant cette méthode que nous avons introduit nos propres actions. Vous avez sans doute remarqué que si l'applet était partiellement cachée par une autre fenêtre, celle-ci se redessinait quand elle redevenait visible. En effet, le composant graphique reçoit l'ordre de repeindre le composant. Il exécute alors la méthode *repaint()*, qui elle-même invoque *update()*, cette dernière faisant alors appel à la méthode *paint()*. (Voir l'applet 54, p. 229.)

Le programme ci-dessous doit vous permettre de tester ces différents comportements. À chaque fois que le document disparaît, il doit changer de couleur lors de son prochain affichage.

```
import java.awt.Graphics;
import java.awt.Color;

public class ComportementApplet extends java.applet.Applet {
  boolean ok;

  public void init() {
    ok=true;
  }
  public void start() {
    repaint();
  }
  public void stop() {
    ok=!ok;}
```

```
  public void paint(Graphics g) {
    if (ok) g.setColor(Color.green);
      else  g.setColor(Color.red);
    g.fillRect(0,0,getSize().width,getSize().height);
  }
}
```

Maintenant que nous avons bien compris le comportement d'une applet, nous allons faire plusieurs tentatives d'animation. Pour simplifier, nous prendrons comme exemple l'affichage de l'heure (quelque chose de bien suisse et qui change en permanence!). Nous avons repris le programme de notre certificat (applet 21) en retirant tout ce qui ne concerne pas l'affichage de la date.

12.1 Mon applet est un seul processus

Cette première version de l'horloge (applet 22) n'est pas une animation, mais elle montre bien l'effet des événements extérieurs au composant graphique de l'applet, qui la forcent à se repeindre et donc à rafraîchir l'heure (essayez cette applet en la masquant avec une autre fenêtre et en la rendant à nouveau visible).

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

public class Animation extends java.applet.Applet {
  Font timesRoman24Gras = new
         Font("TimesRoman",Font.BOLD,24);
  Date maintenant;

  public void paint(Graphics g) {
    g.setColor(Color.red);
    g.setFont(timesRoman24Gras);
    maintenant = new Date();
    g.drawString(maintenant.toString(), 10, 40);
  }
}
```

Applet 22 : Animation : une horloge digitale

L'idée de l'essai ci-dessous est de dessiner continuellement l'heure en insérant l'appel à *drawString()* dans une boucle sans fin :

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

public class Animation extends java.applet.Applet {
  Font timesRoman24Gras = new
         Font("TimesRoman",Font.BOLD,24);
  Date maintenant;
```

```
  public void paint(Graphics g) {
    g.setColor(Color.red);
    g.setFont(timesRoman24Gras);
    while (true) {
       maintenant = new Date();
       g.drawString(maintenant.toString(), 10, 40);
     }
  }
}
```

Le résultat est assez surprenant : on perd pratiquement tout contrôle, car la machine se consacre entièrement à sa nouvelle tâche : peindre et repeindre l'heure. De plus, on recouvre continuellement ce que l'on a déjà dessiné, de ce fait on ne voit plus rien! Ce dernier problème peut être évité en effaçant la surface que l'on vient de dessiner avant d'afficher une nouvelle heure. La boucle devient alors :

```
while (true) {
  g.clearRect(0,0,size().width,size().height);
  maintenant = new Date();
  g.drawString(maintenant.toString(), 10, 40);
}
```

Le résultat est un clignotement très intense de l'heure, qui lutte pour être affichée mais qui est immédiatement effacée. Néanmoins, on voit les secondes défiler.

La remarque suivante doit certainement vous venir à l'esprit : pourquoi vouloir afficher si souvent un phénomène ne changeant que toutes les secondes? Penchons-nous sur la notion de *Thread* (détaillée au chapitre 25) afin de découvrir une meilleure manière de réaliser une horloge ou toute autre animation.

En Java, tout processus est un *Thread*, y compris celui exécutant notre applet. Nous pouvons donc lui appliquer les méthodes de cette classe, en particulier celle qui met le processus en veille pendant un certain temps (*Thread.sleep()*), exprimé en millisecondes. Nous en profitons pour entourer l'appel à *Thread.sleep()* de deux nouvelles instructions, *try* et *catch*, que n ous avons vues au point 8.1.

```
while (true) {
  g.clearRect(0,0,size().width,size().height);
  maintenant = new Date();
  g.drawString(maintenant.toString(), 10, 40);
  try {Thread.sleep(1000);}
  catch(InterruptedException signal) {}
}
```

Le résultat est acceptable : chaque seconde, l'heure est affichée. Cependant, nous n'avons pas vraiment réalisé une animation. L'applet a perdu son comportement global, elle ne reçoit plus les appels à *repaint(), stop()*, etc. car l'exécution du processus est mise en veille dans la méthode *paint()*.

Résumons notre situation :

- nous voulons conserver le comportement normal de l'applet;
- nous voulons **quelque chose** qui force l'applet à se redessiner toutes les secondes.

Ce *quelque chose* ne peut malheureusement pas provenir de l'applet elle-même. En effet, notre applet n'est composée que d'un seul processus, comme l'indique le titre de cette section.

12.2 Multithread

Pour contourner ces difficultés, il est nécessaire d'utiliser un autre processus (*Thread*) possédant son propre flot d'instructions indépendant de l'applet. Le cycle de vie d'un *Thread* est le suivant :

- *start()* : il est activé et commence à exécuter la méthode *run()*;
- *run()* : cette méthode constitue la tâche à effectuer par le processus;
- pour *stopper* un processus, il faut le laisser terminer lui-même sa méthode run().

Pendant qu'il exécute son activité (*run()*), il est possible qu'il soit :

- *sleep()* : endormi pour une certaine durée;
- *join()* : mis en attente de la fin d'un autre processus.

Cependant, aucun de ces événements ne peut modifier le flot d'instructions décrit dans la méthode *run()* qu'il exécute. Tout au plus, il peut être ralenti, synchronisé ou temporisé par rapport à l'exécution normale de ce flot.

12.3 Mon applet lance un deuxième processus

Dans cette version de l'horloge, l'applet va exécuter son cycle de vie normal *(init(), start(), paint(), repaint(),..., stop())*. En déclarant qu'elle implante l'interface *Runnable*, elle s'engage à implanter une méthode *run()* pour les processus qu'elle active.

Dans la méthode *start()*, l'applet crée le nouveau processus (*new Thread(this)*) et le fait démarrer (*actif.start()*). Celui-ci exécute alors automatiquement la méthode *run()* implantée par l'applet. On utilise donc la seconde manière de créer un processus, présentée au point 8.2.

La méthode *run()* invoque la méthode *repaint()* de l'applet toutes les secondes.

Finalement, si l'applet reçoit l'ordre de s'arrêter *stop()*, elle met la variable stop à *true* (à vrai), ce qui déclenchera l'arrêt de la boucle *while(!stop)* du processus actif.

La figure 12.1 ci-dessous schématise ces diverses interactions.

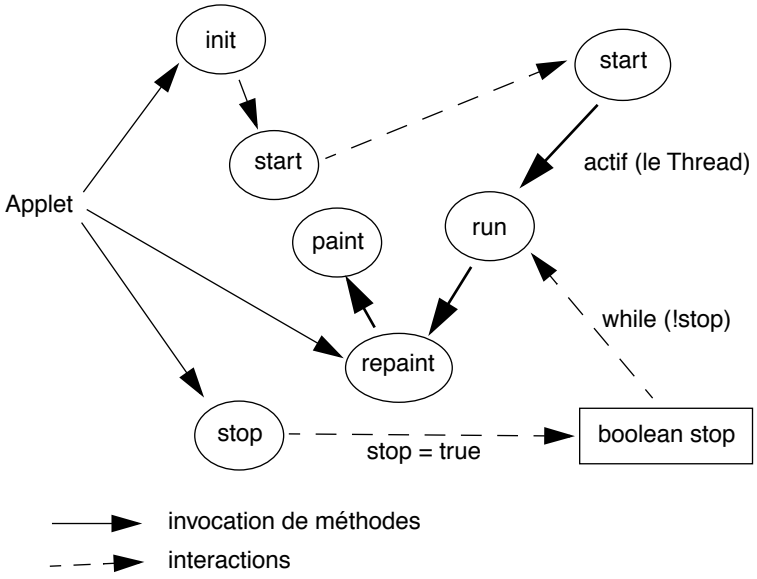


Figure 12.1 Interactions entre les processus

Code de l'horloge utilisant les mécanismes de multiprocessus :

```
import java.awt.Graphics;
import java.awt.Color;
import java.awt.Font;
import java.util.Date;

public class Animation extends java.applet.Applet
   implements Runnable {

   boolean stop=false;

   Font timesRoman24Gras = new Font("TimesRoman",Font.BOLD,24);
   Date maintenant;
   Thread actif;

   public void start() {
      actif = new Thread(this);
      actif.start();
   }
   public void stop() {
      stop = true;
      actif = null;
   }

   public void run() {
```

```
        while (!stop) {
        repaint();
           try {Thread.sleep(1000);}
           catch(InterruptedException signal) {}
        }
     }

     public void paint(Graphics g) {
        g.setColor(Color.red);
        g.setFont(timesRoman24Gras);
        maintenant = new Date();
        g.drawString(maintenant.toString(), 10, 40);
     }
  }
```

Notre applet est *habitée* par deux processus : le processus habituel qui appelle *paint*, *start*, *stop* suivant les circonstances et le processus *actif* qui fonctionne dans *run* et appelle aussi *paint* (toutes les secondes).

12.4 Un squelette général pour les animations

Nous pouvons dériver de notre horloge le squelette général suivant pour les animations simples (une seule activité) :

```
import ...

public class Animation extends java.applet.Applet
   implements Runnable {

   boolean stop = false; // la variable de controle du thread
   // déclaration des variables de l'applet
   ...
   Thread actif;

   public void start() {
   // activation du processus d'animation
      if (actif==null); {
         actif = new Thread(this);
         actif.start();
      }
   }

   public void stop() {
   // arrêt du processus d'animation
      if (actif!=null); {
         stop = true;
      actif = null;
      }
   }

   public void run() {
```

```
        // déclaration des variables du processus
        ...
        // boucle d'animation
        while (!stop) { // attend la fin
           // modification des variables de l'applet
           ...
           // demande de mise à jour update() ou repaint()
           ...
           // réglage de la vitesse de l'animation
           try {Thread.sleep(...);}
           catch(InterruptedException signal) {}
        }
     }

     public void paint(Graphics g) {
        // exécution de l'animation
        ...
     }
  }
```

Nous avons légèrement modifié les méthodes *start()* et *stop()* afin de vérifier que nous ne lançons pas un processus déjà actif et que nous n'arrêtons pas un processus inexistant (par exemple, si l'on arrête l'applet avant qu'elle exécute la méthode *start()*).

Nous allons maintenant nous familiariser avec ce squelette que nous appliquerons dans plusieurs exemples tout au long des chapitres suivants.

12.5 Une horloge avec des aiguilles

Reprenons notre exemple de l'horloge et incorporons nos connaissances sur le dessin afin de réaliser une horloge avec des aiguilles (une horloge analogique).

Dans ce cas, notre problème est d'arriver à :

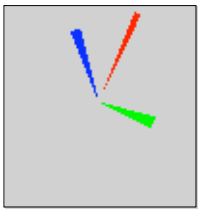
- convertir la valeur de l'heure digitale en heure analogique;
- trouver pour les heures, les minutes et les secondes l'angle de la position des aiguilles;
- représenter les aiguilles de l'horloge le plus simplement possible, tout en offrant une bonne lisibilité;
- animer l'horloge toutes les secondes.

Dans l'horloge analogique (voir applet 23), nous avons décidé de représenter les aiguilles à l'aide d'arcs de disques que nous savons dessiner avec *fillArc()*.

Pour chaque aiguille (*s, m, h*), nous définissons son origine (*ox, oy*), sa largeur (*i*), sa hauteur (*h*) et son arc de cercle (*d*). Nous obtenons la série de données suivante :

```
int oxs = 0, oys = 0, ls=99, hs=99, ds=2;
```

```
int oxm = 10, oym = 10, lm=79, hm=79, dm=4;
int oxh = 20, oyh = 20, lh=59, hh=59, dh=6;
```



Applet 23 : Animation, une horloge analogique (15h 56mn 7s)

En examinant la classe *Date*, on remarque que les méthodes *getHours()*, *getMinutes()*, et *getSeconds()* nous permettent de récupérer les valeurs utiles pour notre horloge.

La seule difficulté est d'associer une valeur horaire à une position du cercle trigonométrique :

- le sens horaire et le sens trigonométrique étant opposés, on utilise l'inversion (le signe moins) pour rétablir la situation;
- le zéro horaire et le zéro trigonométrique n'étant pas situés au même endroit, on utilise une constante (450) pour les décaler. Douze heures devant correspondre à 360 degrés, on multiplie par 30. Soixante minutes ou secondes devant correspondre à 360 degrés, on multiplie par 6. On ajoute un petit coup de modulo 360 pour le cas où l'on fait plusieurs tours et l'affaire est réglée.

Nous avons intégré la valeur des minutes dans le calcul de la position des heures, afin que le mouvement de l'aiguille soit continu (sinon elle ne se déplacerait qu'une fois toutes les heures). En utilisant ce qui vient d'être dit, nous obtenons les formules suivantes :

```
h=(-maintenant.get(Calendar.HOUR_OF_DAY)*30-
        (maintenant.get(Calendar.MINUTE)*30)/60+450)%360;
m=(-maintenant.get(Calendar.MINUTE)*6+450)%360;
s=(-maintenant.get(Calendar.SECOND)*6+450)%360;
```

Il ne nous reste plus qu'à utiliser notre squelette d'animation et à décrire complètement la méthode *paint()* afin qu'elle demande l'heure au système, calcule la position des aiguilles et les dessine chacune dans une couleur différente. Nous obtenons alors l'applet suivante :

```
import java.awt.Graphics;
import java.awt.Color;
import java.util.*;

//<pre>
public class Animation extends java.applet.Applet
```

```
    implements Runnable {

   GregorianCalendar maintenant; // on craint pas le bug de l'an 2000
    Thread actif;
    boolean stop = false;

    int h,m,s;
    int oxs = 0, oys = 0, ls=99, hs=99, ds=2;
    int oxm = 10, oym = 10, lm=79, hm=79, dm=4;
    int oxh = 20, oyh = 20, lh=59, hh=59, dh=6;

    public void start() {
      if (actif==null); {
        actif = new Thread(this);
        actif.start();
      }
    }

     public void stop() {
      stop = true;
      actif = null;

    }

    public void run() {
       while (!stop) {
          repaint();
          try {Thread.sleep(1000);}
          catch(InterruptedException signal) {}
       }
    }

    public void paint(Graphics g) {
      maintenant = new GregorianCalendar();
      g.setColor(Color.green);
      h=(-maintenant.get(Calendar.HOUR_OF_DAY)*30-
(maintenant.get(Calendar.MINUTE)*30)/60+450)%360;
        g.fillArc(oxh,oyh,lh,hh,h+dh,-2*dh);

      g.setColor(Color.blue);
      m=(-maintenant.get(Calendar.MINUTE)*6+450)%360;
        g.fillArc(oxm,oym,lm,hm,m+dm,-2*dm);

      g.setColor(Color.red);
      s=(-maintenant.get(Calendar.SECOND)*6+450)%360;
        g.fillArc(oxs,oys,ls,hs,s+ds,-2*ds);

     }
}
```

12.6 Plusieurs activités indépendantes

L'inconvénient de notre squelette est qu'il ne gère qu'une seule activité (ou plusieurs activités ayant toutes le même comportement). Dans l'exemple qui suit, nous allons créer deux activités supplémentaires : la première sera en charge du contrôle de la coordonnée *x* d'un rectangle, l'autre de la coordonnée *y*.

Nous importons les classes *Rectangle* et *Graphics* :

```
import java.awt.Graphics;
import java.awt.Rectangle;
```

La classe *Actif1* étend la classe *Thread*. Nous devons donc redéfinir la méthode *run()*. Comme variable d'instance, nous lui fournissons le rectangle dont elle doit contrôler la coordonnée *x*; ceci est effectué lors de l'instanciation d'un objet de la classe *Actif1* (dans le constructeur).

La méthode *run()* définit deux boucles (aller et retour) qui modifient la coordonnée *x* du rectangle de manière incrémentale. L'attente entre deux modifications est proportionnelle à l'éloignement de l'origine (ce qui donnera une impression d'accélération du mouvement de l'objet) lors de son affichage. On remarquera la méthode *arret()* qui permet de mettre à vrai la variable *stop* et qui permet de terminer la méthode *run()*.

Code de la classe *Actif1* :

```
//<pre>
import java.awt.Graphics;
import java.awt.Rectangle;

class Actif1 extends Thread {
   private Rectangle r1;
   boolean stop = false;

      Actif1(Rectangle r){
        r1=r;
      }

      public void run() {
        while (!stop) {
          for (int i=0;i<80;++i){
            r1.x=i;
            try {Thread.sleep(i+5);}
            catch(InterruptedException signal) {}
          }
          for (int i=80;i>0;--i){
            r1.x=i;
            try {Thread.sleep(i+5);}
            catch(InterruptedException signal) {}
          }
        }
      }
```

```
        public void arret() {stop=true;}
}
```

La classe *Actif2* est très semblable, mais elle modifie l'axe *y*.

Code de la classe *Actif2* :

```
class Actif2 extends Thread {
   private Rectangle r1;
   boolean stop = false;

      Actif2(Rectangle r){
        r1=r;
      }
      public void run() {
        while (!stop) {
          for (int i=0;i<80;++i){
            r1.y=i;
            try {Thread.sleep(i+5);}
            catch(InterruptedException signal) {}
          }
          for (int i=80;i>0;--i){
            r1.y=i;
            try {Thread.sleep(i+5);}
            catch(InterruptedException signal) {}
          }
        }
      }
      public void arret() {stop=true;}
}
```

Décrivons maintenant notre nouvelle applet, 24e du nom.

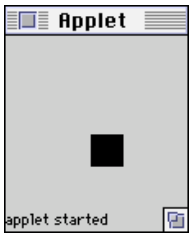
Nous déclarons deux objets *a1* et *a2* de type *Actif1* et *Actif2*. Nous conservons le squelette original d'animation.

La méthode *run()* de l'animation va initialiser nos deux processus en leur fournissant le rectangle sur lequel elles doivent travailler puis elle va lancer ces processus. Nos deux processus vont travailler sur le même rectangle, l'un le déplaçant horizontalement, l'autre verticalement. Ensuite, elle demandera régulièrement que l'on redessine l'applet. La méthode *paint()* dessine le rectangle.

```
public class Animation extends java.applet.Applet
    implements Runnable {

    Thread actif;
    Actif1 a1;
    Actif2 a2;
    Rectangle r= new Rectangle(0,0,20,20);

    public void start() {
```

```
        if (actif==null); {
          actif = new Thread(this);
          actif.start();
        }
      }

       public void stop() {
        if (a1!=null) a1.arret();
        if (a2!=null) a2.arret();
      }

      public void run() {
         a1=new Actif1(r);
         a1.start();
         a2=new Actif2(r);
         a2.start();
         while (true) {
            repaint();
            try {Thread.sleep(10);}
            catch(InterruptedException signal) {}
         }
      }

      public void paint(Graphics g) {
          g.fillRect(r.x,r.y,r.width,r.height);

        }
}
```

Le résultat est un rectangle noir qui oscille le long de la diagonale, en ayant une vitesse variable. En laissant fonctionner l'applet assez longtemps, une dérive peut apparaître : le rectangle ne reste plus sur la diagonale. Ce comportement prouve bien que les processus ne sont pas complètement synchrones.



Applet 24 : Animation, un rectangle sur une diagonale

12.7 Encore plus d'activités indépendantes

Nous avons repris les classes *Actif1* et *Actif2* et nous avons modifié les lignes de mise en attente de la façon suivante :

```
try {Thread.sleep((int)Math.random()*40+5);}
```

Nous avons ainsi une attente aléatoire qui va faire dévier la trajectoire du rectangle autour de la diagonale. Modifions notre applet afin de contrôler trois rectangles (voir applet 25) :

- les processus *a1* et *a2* contrôlent le rectangle *ra;*
- les processus *b1* et *b2* contrôlent le rectangle *rb;*
- les processus *c1* et *c2* contrôlent le rectangle *rc*.

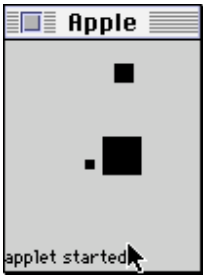
La méthode *paint()* est redéfinie afin de dessiner les trois rectangles.

Code de l'applet avec trois rectangles :

```
public class Animation extends java.applet.Applet
   implements Runnable {
   boolean stop = false;
   Thread actif;
   Actif1 a1,b1,c1;
   Actif2 a2,b2,c2;
   Rectangle ra= new Rectangle(0,0,20,20);
   Rectangle rb= new Rectangle(0,0,10,10);
   Rectangle rc= new Rectangle(0,0,5,5);

   public void start() {
      if (actif==null); {
         actif = new Thread(this);
         actif.start();
      }
   }

   public void stop() {
      if (actif!=null); {
         stop=true;
         if (a1!=null) a1.arret();
         if (b1!=null) b1.arret();
         if (c1!=null) c1.arret();
         if (a2!=null) a2.arret();
         if (b2!=null) b2.arret();
         if (c2!=null) c2.arret();
         actif = null;
      }
   }

   public void run() {
      a1=new Actif1(ra);a1.start();
      a2=new Actif2(ra);a2.start();
      b1=new Actif1(rb);b1.start();
      b2=new Actif2(rb);b2.start();
      c1=new Actif1(rc);c1.start();
      c2=new Actif2(rc);c2.start();
      while (!stop) {
         repaint();
         try {Thread.sleep(10);}
```

```
            catch(InterruptedException signal) {}
         }
      }

      public void paint(Graphics g) {
         g.fillRect(ra.x,ra.y,ra.width,ra.height);
         g.fillRect(rb.x,rb.y,rb.width,rb.height);
         g.fillRect(rc.x,rc.y,rc.width,rc.height);
      }
   }
```

Nous avons maintenant trois rectangles se déplaçant dans le cadre de l'applet.



Applet 25 : Animation, trois rectangles gérés chacun par deux threads

Ces exemples montrent que l'on peut gérer de nombreux processus : on peut ainsi déclarer des matrices de processus. Cependant, cela peut poser quelques difficultés de gestion. Dans notre cas, chaque processus travaillant sur une variable non partagée (*x* ou *y*), il n'y avait pas de risque de collision dans les activités. Nous verrons plus loin que l'on doit utiliser des techniques de synchronisation pour gérer des coopérations plus intenses entre les processus.

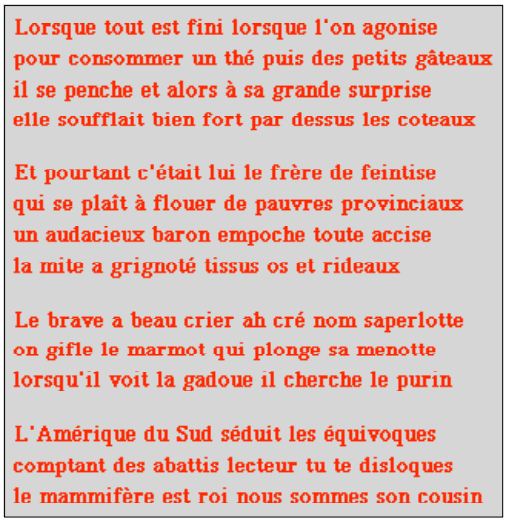
12.8 Un peu de poésie

« ... En comptant 45 secondes pour lire un sonnet et 15 secondes pour changer les volets, à 8 heures par jour, 200 jours par an, on a pour plus d'un million de siècles de lecture... »

Raymond Queneau dans *100 000 000 000 000 poèmes*, Gallimard, 1982.

Ce magnifique livre se compose de dix pages cartonnées; sur chacune d'elles est imprimé un sonnet de quatorze vers. Les pages sont découpées de manière à ce que chaque vers soit libre. Il est alors possible de sélectionner au hasard un vers dans une des dix planches et de constituer ainsi un nouveau sonnet.

L'applet 26 que nous proposons ci-dessous réalise le rêve de Raymond Queneau en affichant un nouveau sonnet toutes les minutes. Les résultats sont surprenants et pourraient correspondre à ce qui suit :



Applet 26 : Un poème généré à partir de la structure de R. Queneau

Pour réaliser notre idée, il nous faut un endroit pour mémoriser les dix planches de quatorze sonnets. Une matrice de *String* fera parfaitement l'affaire :

```
String  q[] [] = new String[10][14];
```

Dans la méthode *init()* de l'applet, nous initialisons notre matrice :

```
public void init() {
   // la première planche
   q[0][0]="Le roi de la pampa retourne sa chemise";
   q[0][1]="pour la mettre à sécher aux cornes des taureaux";
```

La méthode *run()* doit, pour composer un sonnet, tirer au hasard une des dix planches. Ensuite, elle demande à redessiner le sonnet nouvellement composé et elle attend une minute.

Le résultat du tirage au sort est mémorisé dans le vecteur *p* :

```
int p[] = new int[14];

public void run() {
   while (!stop) {
      for (int i=0; i<p.length;i++)
         p[i]=(int)(Math.random()*9.99999);
      repaint();
      try {Thread.sleep(60000);}
      catch(InterruptedException signal) {}
   }
}
```

La méthode *paint()* doit redessiner le sonnet généré (stocké dans *p*). Pour chaque vers *i*, elle recherche la planche *p[i]* qui lui donne le vers *q[p[i]][i]*. Le vecteur *strophe[]* est utilisé pour faire apparaître la structure du sonnet (4-4-3-3).

```
    public void paint(Graphics g) {
       g.setColor(Color.red);
       g.setFont(timesRoman14Gras);
       for (int i=0; i<14;i++)
          g.drawString(q[p[i]][i], 10, 40+i*20+strophe[i]);
    }
}
```

Comme toujours, nous utilisons notre squelette pour gérer l'animation de l'applet.

```
import java.awt.Graphics;
import java.awt.Color;
import java.awt.Font;

public class Animation extends java.applet.Applet implements
Runnable {

   boolean stop=false;
   Font timesRoman14Gras = new Font("TimesRoman",Font.BOLD,14);
   int p[] = new int[14];
   String  q[] [] = new String[10][14];
   int strophe[] = {0,0,0,0,15,15,15,15,30,30,30,45,45,45};

   Thread actif;

   public void start() {
      if (actif==null); {
         actif = new Thread(this);
         actif.start();
      }
   }

   public void stop() {
      if (actif!=null); {
         stop=true;
         actif = null;
      }
   }

   public void run() {
      while (!stop) {
         for (int i=0; i<p.length;i++)
            p[i]=(int)(Math.random()*9.99999);
         repaint();
         try {Thread.sleep(60000);}
         catch(InterruptedException signal) {}
      }
```

```
    }

    public void init() {
       // la première planche
       q[0][0]="Le roi de la pampa retourne sa chemise";
       q[0][1]="pour la mettre à sécher aux cornes des taureaux";
       q[0][2]="le cornédbif en boîte empeste la remise";
       ...
       // la deuxième planche
       q[1][0]="Le cheval Parthénon s'énerve sur sa frise";
       q[1][1]="depuis que lord Elgin négligea ses naseaux";
       q[1][2]="le Turc de ce temps-là pataugeait dans sa crise";
       ...
       // les autres planches
    }

    public void paint(Graphics g) {
       g.setColor(Color.red);
       g.setFont(timesRoman14Gras);
       for (int i=0; i<14;i++)
          g.drawString(q[p[i]][i], 10, 40+i*20+strophe[i]);
    }
}
```

Le lecteur patient pourra se procurer l'intégrale des sonnets chez Gallimard afin de compléter (pour son usage personnel) le code de la méthode *init()*. Il possédera ainsi une source inépuisable de poésie!

12.9 Invitation à explorer

Nous invitons le lecteur à explorer les animations basées sur des processus coopérant entre eux, comme par exemple une applet dessinant un carré qui poursuit un cercle.

Chapitre 13
Interagir

Jusqu'à présent, nos programmes étaient insensibles à toutes les sollicitations extérieures. Dans ce chapitre, nous allons apprendre à gérer les interactions provenant de la souris et du clavier (pour d'autres périphériques, il faut écrire les classes capables de gérer les événements qu'ils génèrent).

Entre les versions 1.0 et 1.1 de Java, la gestion des événements a été profondément modifiée. Étant donné qu'il existe encore beaucoup de codes écrits selon la version 1.0 et que tous les utilisateurs n'ont pas encore de butineurs adaptés à la version 1.1, nous avons choisi de présenter les deux modèles de gestion d'événements. Si vous n'avez pas à gérer du code écrit pour la version 1.0, vous pouvez aller directement au point 13.7.

13.1 Le modèle d'événements de Java 1.0

Les interactions sont gérées comme des événements fournis à un objet (dans notre cas, le composant graphique associé à l'applet). Pour traiter l'événement, il suffit de redéfinir la méthode correspondant à son type, cette méthode devant être rattachée à l'objet recevant l'événement.

Event est une classe qui définit un ensemble de constantes utiles à la gestion du clavier et de la souris et qui a pour variables d'instance les valeurs décrivant l'événement (heure, position, état des touches de fonction, etc.).

Prenons tout de suite un exemple afin de clarifier les choses.

Le programme suivant (applet 27) permet d'enregistrer les mouvements de la souris (lorsque le bouton n'est pas enfoncé). *mouseMove()* est une méthode de *Component* dont hérite la classe *Applet*. Nous redéfinissons donc *mouseMove()* dans notre applet afin de récupérer les événements provoqués par les

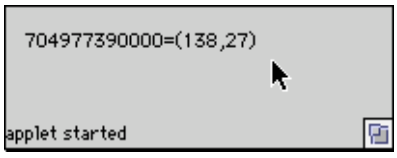
mouvements de la souris : chaque fois que cette méthode sera invoquée, nous recevrons une instance de l'événement qui lui est associée. Nous pourrons alors connaître l'instant (*e.when*) et la position (*x,y*) auxquels l'événement s'est produit. Nous conservons ces valeurs dans des variables de l'applet (*sourisX, sourisY* et *quand*).

La méthode *mouseMove()* doit retourner un booléen. Si celui-ci est vrai, cela signifie que l'événement est considéré comme traité, sinon cet événement peut être transmis à un autre objet pouvant aussi le traiter (par exemple si plusieurs objets sont empilés sur la même coordonnée, il peut être souhaitable que chacun d'entre eux reçoive l'événement).

Code permettant d'afficher les mouvements de la souris :

```
import java.awt.Graphics;
import java.awt.Event;

public class Interaction extends java.applet.Applet {
   int sourisX, sourisY;
   long quand;

   public boolean mouseMove(Event e, int x, int y){
      quand=e.when;
      sourisX=x;
      sourisY=y;
      repaint();
      return true;
   }

    public void paint(Graphics g) {
      g.drawString(quand+"=("+sourisX+","+sourisY+")"
      ,10,20);
   }
}
```

En déplaçant la souris, les coordonnées du curseur sont affichées, une estampille indiquant l'instant de l'événement (en millisecondes) :



Applet 27 : Afficher les coordonnées de la souris

13.2 Gestion de la souris

Les méthodes suivantes gèrent les événements concernant la souris :

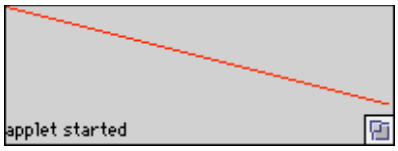
- *public boolean mouseDown(Event e, int x, int y)* : invoquée quand le

bouton de la souris est enfoncé dans la surface de l'applet;

- *public boolean mouseUp(Event e, int x, int y)* : invoquée quand le bouton de la souris est relâché dans la surface de l'applet;
- *public boolean mouseMove(Event e, int x, int y)* : invoquée quand la souris se déplace, le bouton étant relâché;
- *public boolean mouseDrag(Event e, int x, int y)* : invoquée quand la souris se déplace, le bouton étant enfoncé;
- *public boolean mouseEnter(Event e, int x, int y)* : invoquée quand la souris entre dans la surface de l'applet;
- *public boolean mouseExit(Event e, int x, int y)* : invoquée quand la souris sort de la surface de l'applet.

Exemple de programme traitant les événements de la souris : dans celui-ci, la position du bouton de la souris détermine la couleur du dessin (enfoncé = blanc, relâché = rouge). Le résultat est représenté ci-dessous (applet 28) :

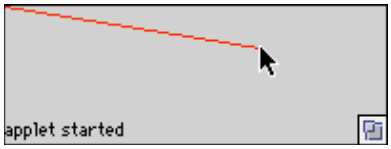
```
import java.awt.Graphics;
import java.awt.Color;
import java.awt.Event;

public class Interaction extends java.applet.Applet {
   Color c=new Color(0,0,255);

   public boolean mouseDown(Event e, int x, int y){
      c= new Color(255,255,255);
      repaint();
      return true;
   }
   public boolean mouseUp(Event e, int x, int y){
      c= new Color(255,0,0);
      repaint();
      return true;
   }
   public void paint(Graphics g) {
      g.setColor(c);
      g.drawLine(0,0,size().width ,size().height);
   }
}
```



Applet 28 : Capter l'état du bouton de la souris

En ajoutant le traitement de l'événement *mouseDrag()*, on peut modifier l'extrémité du trait. Il suit alors tous les mouvements de la souris pendant que le bouton est enfoncé (voir applet 29).

```
import java.awt.Graphics;
import java.awt.Color;
import java.awt.Event;

public class Interaction extends java.applet.Applet {
   Color c=new Color(0,0,255);
   int dx=size().width;
   int dy=size().height;

   public boolean mouseDown(Event e, int x, int y){
      c= new Color(255,255,255);
      repaint();
      return true;
   }
   public boolean mouseUp(Event e, int x, int y){
      c= new Color(255,0,0);
      repaint();
      return true;
   }
   public boolean mouseDrag(Event e, int x, int y){
      dx=x;
      dy=y;
      repaint();
      return true;
   }
   public void paint(Graphics g) {
      g.setColor(c);
      g.drawLine(0,0,dx,dy);
   }
}
```



Applet 29 : Capter la position de la souris

13.3 Une alternative à la gestion des événements

La méthode *handleEvent(Event e)* est une autre méthode permettant de gérer les événements. En redéfinissant cette méthode, il est possible de collecter tous les événements adressés à la surface graphique de l'applet.

La détermination du type d'un événement se fera en examinant sa variable d'instance *id* et en la comparant aux constantes définies dans la classe *Event*. Pour récupérer les coordonnées *(x,y)* de la souris, on examinera les variables d'instance *x* et *y* de l'événement.

Le programme précédent peut être modifié afin d'utiliser les constantes *Event.MOUSE_DOWN*, *Event.MOUSE_UP* et *Event.MOUSE_DRAG* pour dé-

terminer le type de l'événement. Dans le cas où l'on ne sait pas comment traiter cet événement, on fait appel à la méthode de la super-classe.

```
import java.awt.Graphics;
import java.awt.Color;
import java.awt.Event;

public class Interaction extends java.applet.Applet {
   Color c=new Color(0,0,255);
   int dx=size().width;
   int dy=size().height;

   public boolean handleEvent(Event e){
      switch(e.id){
         case Event.MOUSE_DOWN:
            c= new Color(255,255,255);
            break;
         case Event.MOUSE_UP:
            c= new Color(255,0,0);
            break;
         case Event.MOUSE_DRAG:
            dx=e.x;
            dy=e.y;
            break;
         default: return super.handleEvent(e);
      }
      repaint();
      return true;
}
   public void paint(Graphics g) {
      g.setColor(c);
      g.drawLine(0,0,dx,dy);
   }
}
```

13.4 Une applet de brouillon

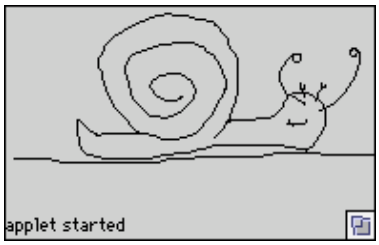
En utilisant les informations sur la position de la souris, il est possible de transformer les séquences de coordonnées *(x,y)* en segments de droites et de dessiner ainsi une trace des mouvements de la souris (voir applet 30).

Le programme suivant permet de dessiner avec la souris. On remarquera que :

- le cas *Event.MOUSE_DRAG* ne possède pas de *break*, l'exécution se poursuit donc en exécutant les instructions de *Event.MOUSE_DOWN;*
- on ne fait pas appel à la méthode *repaint()* (qui effacerait tout dans notre cas), mais on demande à la méthode *getGraphics()* de nous fournir le composant graphique sur lequel on travaille et on s'adresse directement à lui pour dessiner.

```
import java.awt.Graphics;
```

```
import java.awt.Event;

public class Interaction extends java.applet.Applet {
   int x0,y0;
   public boolean handleEvent(Event e){
      switch(e.id){
         case Event.MOUSE_DRAG:
            Graphics g= getGraphics();
            g.drawLine(x0,y0,e.x,e.y);
         case Event.MOUSE_DOWN:
            x0=e.x;
            y0=e.y;
            break;
         default: return super.handleEvent(e);}
      return true;}
}
```



Applet 30 : Une applet de brouillon pour dessiner

13.5 Gestion du clavier

La gestion des événements du clavier est très similaire à celle de la souris. On peut récupérer les événements en redéfinissant la méthode *keyDown(Event e, int touche)* (ou bien *handleEvent()*; dans ce cas la valeur de la touche du clavier se trouve dans la variable d'instance *key*). La classe *Event* définit des constantes pour les principales touches du clavier (voir tableau 13.1).

Constante	Touche représentée
`Event.UP`	Flèche haut
`Event.DOWN`	Flèche bas
`Event.LEFT`	Flèche gauche
`Event.RIGHT`	Flèche droite
`Event.HOME`	Touche « home »

Tableau 13.1 Constantes associées aux principales touches

Constante	Touche représentée
`Event.END`	Touche « fin »
`Event.PGUP`	Défilement page haut
`Event.PGDN`	Défilement page bas
`Event.F1 à Event.F12`	Touches de fonction F1 à F12

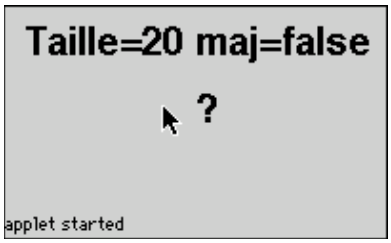
Tableau 13.1 Constantes associées aux principales touches

Pour tester les touches de modification du clavier, il existe trois méthodes :

- *public boolean shiftDown()* : la touche majuscule est-elle enfoncée?
- *public boolean controlDown()* : la touche de contrôle est-elle enfoncée?
- *public boolean metaDown()* : la touche de méta *(alt)* est-elle enfoncée?

L'applet suivante (applet 31) montre comment gérer le clavier. Ce programme modifie la taille des caractères affichés en utilisant les touches *flèche haut* et *flèche bas*. Il teste également l'état de la touche *majuscule*.

```
import java.awt.Graphics;
import java.awt.Font;
import java.awt.Event;

public class Interaction extends java.applet.Applet {
   int taille=12;
   char toucheCourante='?';
   boolean majuscule;

   public boolean keyDown(Event e, int touche){
      switch(touche){
         case Event.DOWN:--taille;break;
         case Event.UP: ++taille;break;
         default: toucheCourante=(char)touche ;
      }
      setFont(new Font("Helvetica",Font.BOLD,taille));
      majuscule=e.shiftDown();
      repaint();
      return true;
   }

    public void paint(Graphics g) {
      g.drawString("Taille="+taille+" maj="+majuscule,10,25);
      g.drawString(String.valueOf(toucheCourante),100,60);
   }
}
```



Applet 31 : Gestion du clavier

13.6 Souris à trois boutons

Les événements se rapportant à la souris peuvent tester les touches de modification (*maj*, *ctrl*, *meta*). Il est donc possible de distinguer un clic souris avec ou sans une ou plusieurs de ces touches enfoncées.

Afin de rendre Java indépendant des différentes plates-formes matérielles, seule une souris à un bouton est supportée. Cependant, les conventions suivantes (voir tableau 13.2) permettent de réintroduire les autres boutons. La variable d'instance *modifiers* d'un événement contiendra éventuellement une valeur de modificateur.

Bouton de la souris	Modificateur
Gauche	pas de modificateur
Milieu	`Event.ALT_MASK`
Droite	`Event.ALT_META`

Tableau 13.2 Modificateurs pour une souris à trois boutons

13.7 Le modèle d'événements à partir de Java 1.1

À partir de la version 1.1 de l'API Java, le modèle de gestion des événements est basé sur le principe de la délégation. C'est un principe très général qui ne s'applique pas seulement aux événements physiques (déplacement de la souris, frappe au clavier, etc.), mais aussi à la communication entre composants d'un système. Chaque événement est représenté par un objet qui constitue la *mémoire* de l'événement. Ce dernier contient des informations générales telles que la source de l'événement ou le moment auquel il a eu lieu et des informations spécifiques dépendant du type d'événement.

N'importe quel objet peut recevoir des événements de son choix, il lui suffit pour cela :

- de s'inscrire comme écouteur auprès d'un objet source d'événement;

- de posséder les méthodes requises pour traiter ce type d'événement.

De même, tout objet peut être source d'événements, il doit alors :

- posséder une méthode d'inscription et de désinscription d'écouteurs;
- transmettre les événements en appelant les méthodes requises des écouteurs.

En général, une source peut avoir plusieurs écouteurs différents qui lui sont attachés et un écouteur peut être inscrit auprès de plusieurs sources différentes.

13.8 Gestion de la souris (trois manières)

Prenons le cas des événements générés par les mouvements de la souris. Un objet qui veut recevoir ces événements, de type *MouseEvent*, doit posséder toutes les méthodes définies dans l'interface *MouseMotionListener*, c'est-à-dire *mouseMoved()* et *mouseDragged()*. En d'autres termes, sa classe doit implémenter *MouseMotionListener*.

L'objet doit également s'inscrire auprès d'une source d'événements de type *MouseMotion* en appelant la méthode *addMouseMotionListener()* de cette source. En général, on effectue cet appel lors de l'initialisation de l'objet.

Le cas des applets est un peu particulier car une applet est source d'événements souris qu'elle va consommer elle-même. On écrira donc l'appel d'inscription :

```
this.addMouseMotionListener(this);
```

Nous voilà prêts pour écrire une applet qui affiche en permanence les coordonnées de la souris :

```
import java.awt.Graphics;
import java.awt.Event;
import java.awt.event.*;

public class Interaction extends java.applet.Applet
   implements MouseMotionListener {
      int sourisX, sourisY;
      long quand;

   public void init() {
      this.addMouseMotionListener(this); // inscription
   }

   public void mouseMoved(MouseEvent e){
      // extrait les informations de l'événement
      quand=e.getWhen(); sourisX=e.getX(); sourisY=e.getY();
      repaint();
      }
   public void mouseDragged(MouseEvent e)  {}

   public void paint(Graphics g) {
```

```
        g.drawString(quand+"=("+sourisX+","+sourisY+")",10,20);
    }
}
```

On obtient le même comportement que l'applet 27 p.166.

Cette manière d'écrire une applet qui répond à des événements est la plus proche de celle utilisée avec JDK 1.0, mais elle n'est pas la plus élégante car il faut définir toutes les méthodes de l'interface d'écoute (en l'occurrence *MouseMotionListener*), même celles qu'on n'utilise pas.

Pour simplifier l'écriture des écouteurs, le package *java.awt.event* propose des classes toutes faites, appelées **adaptateurs**, qui réalisent chacune un écouteur inactif. On définira alors un écouteur par extension d'un adaptateur, ce qui nous donne le code suivant :

```
import java.awt.Graphics;
import java.awt.Event;
import java.awt.event.*;

public class InteractionI extends java.applet.Applet{

   int sourisX, sourisY;
   long quand;

   class EcouteSouris extends MouseMotionAdapter {
      public void mouseMoved(MouseEvent e){
         quand=e.getWhen(); sourisX=e.getX(); sourisY=e.getY();
         repaint();
      }
   }
   public void init() {
      this.addMouseMotionListener(new EcouteSouris());
   }
   public void paint(Graphics g) {
      g.drawString(quand+"=("+sourisX+","+sourisY+")",10,20);
   }
}
```

Ici, l'adaptateur est une classe interne, ce qui lui permet d'accéder aux variables de l'applet. Lors de l'appel d'enregistrement, on crée l'objet écouteur (*new EcouteSouris()*).

Finalement, on peut utiliser le mécanisme des classes anonymes pour éviter de définir explicitement une classe écouteur :

```
public class InteractionA extends java.applet.Applet {
   int sourisX, sourisY;
   long quand;

   public void init() {
      this.addMouseMotionListener(
```

```
        new MouseMotionAdapter(){
           public void mouseMoved(MouseEvent e){
              quand=e.getWhen(); sourisX=e.getX();
sourisY=e.getY();
              repaint();
           }
        });
    }


    public void paint(Graphics g) {
       g.drawString(quand+"=("+sourisX+","+sourisY+")",10,20);
    }
}
```

L'expression *new MouseMotionAdapter() { ... }* définit une classe qui étend *MouseMotionAdapter* et crée un objet de cette classe (l'écouteur). L'écriture est plus compacte mais n'est plus très lisible car on mélange des déclarations de méthodes à l'intérieur d'expressions. Il vaut mieux éviter d'abuser de ce mécanisme.

Si l'interface *MouseMotionListener* permet de s'inscrire pour recevoir les événements de déplacement de la souris, l'interface *MouseListener* permet de recevoir les clics souris (méthodes *mousePressed()*, *mouseReleased()* et *mouseClicked()*) et les entrées/sorties du curseur sur la surface de composant (*mouseEntered()* et *mouseExited()*).

L'événement de type *MouseEvent* reçu contient différentes informations que l'on peut extraire à l'aide de méthodes telles que :

- *getWhen()* : le temps auquel s'est produit l'événement;
- *getX()*, *getY()* : la position du pointeur;
- *getClickCount()* : le nombre de clics sur ce même point;
- *isAltDown()*, *isAltGraphDown()*, *isControlDown()*, *isMetaDown()*, *isShiftDown()* : vrai si la touche (modificateur) Alt, AltGr, Ctrl, Meta ou Shift était enfoncée lors de l'événement;
- *getModifiers()* : un codage sous forme d'un *int* des touches enfoncées.

Dans le cas d'une souris à plusieurs boutons, on peut tester quel bouton a déclenché l'événement avec l'expression :

```
(getModifiers() & BUTTONi_MASK) // i = 1, 2 ou 3
```

Le résultat est différent de 0 si le bouton *i* a été pressé, relâché ou cliqué (c'est-à-dire pressé puis relâché).

Cependant, pour être vraiment portable, une application ne devrait pas faire d'hypothèses sur le nombre de boutons dont est munie la souris.

13.9 Gestion du clavier

Les événements clavier sont représentés par des objets de la classe *KeyEvent* qui étend *InputEvent*. L'applet ci-dessous est la version 1.1 de l'applet 31 p.172 :

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

public class Interaction31 extends java.applet.Applet {
   int taille=12;
   char toucheCourante='?';
   boolean shiftPresse;

   class Clavier extends KeyAdapter {

      public void keyPressed(KeyEvent e){
         int touche = e.getKeyCode();
         shiftPresse = e.isShiftDown();

         switch(touche){
            case KeyEvent.VK_DOWN: --taille; break;
            case KeyEvent.VK_UP: ++taille; break;
            default: toucheCourante=e.getKeyChar();
         }
         setFont(new Font("Helvetica",Font.BOLD,taille));
         repaint();
      }
   }

   public void init() {
      this.addKeyListener(new Clavier());
   }
   public void paint(Graphics g) {
      g.drawString("Taille="+taille+" majuscule="+ shiftPresse,
                   10, 25);
      g.drawString(String.valueOf(toucheCourante),100,60);
   }
}
```

La méthode *keyPressed* est invoquée lorsqu'une touche est enfoncée. La méthode *getKeyCode* de la classe *keyEvent* fournit le code de la touche. Il s'agit de codes correspondant à un *clavier virtuel* indépendant de la machine utilisée. Ces codes sont définis par une liste de constantes *KeyEvent.VK_xxx*. Dans cet exemple, nous utilisons *VK_DOWN* et *VK_UP* qui correspondent aux flèches vers le bas et vers le haut. La méthode getKeyChar fournit un caractère lorsque la ou les touche(s) pressée(s) corresponde(nt) à un caractère.

Il existe aussi une méthode *keyTyped()* qui n'est invoquée que lorsqu'un caractère est généré par la ou les touches pressée(s). Cette méthode ne permet évidemment pas de capter les actions sur les touches de flèches, de fonctions, majuscule, ctrl, alt, etc.

De même que pour les événement souris, les événements clavier ont des méthodes *isAltDown()*, *isAltGraphDown()*, *isControlDown()*, *isMetaDown() et isShiftDown()* pour savoir quelle(s) touche(s) étai(en)t enfoncée(s) au moment du clic.

13.10 Événements et autres composants d'interaction

À l'instar de l'applet, qui est un composant d'interface graphique, il existe d'autres types d'objets (boutons, étiquettes, menus, textes, barres de défilement, etc.) qui peuvent entrer dans la composition d'une interface graphique. Ces composants, que nous décrirons dans le chapitre suivant, peuvent générer différents types d'événements en fonction des actions de l'utilisateur. La table qui suit en montre quelques exemples.

Action qui crée l'événement	Type d'événement
Un utilisateur clique sur un bouton, choisit un article dans un menu, tape la touche *Return* dans un champ de texte.	`ActionEvent`
Un utilisateur ferme une fenêtre.	`WindowEvent`
Un utilisateur presse un bouton de la souris quand le curseur est sur un composant.	`MouseEvent`
Un utilisateur bouge la souris sur un composant.	`MouseMotionEvent`
Un composant se cache ou devient visible	`ComponentEvent`
Un composant reçoit l'attention du clavier.	`FocusEvent`
Une table ou une liste de sélection est modifiée.	`ListSelectionEvent`

Tableau 13.3 Actions et événements générés

Lorsque nous étudierons les composants d'interaction graphique nous indiquerons à chaque fois quels sont les types d'événements qu'ils peuvent générer.

Il est intéressant de noter qu'on peut considérer les actions de l'utilisateur à différents niveaux de détail. Par exemple, lorsque l'utilisateur agit sur un bouton de l'interface on peut s'intéresser :

- uniquement au déclenchement de la commande associée au bouton (suite à un clic souris);
- à l'action *enfoncer le bouton* (presser sur le bouton de la souris) suivie de l'action *relâcher le bouton* (relâcher le bouton de la souris);
- à tous les mouvements du pointeur de la souris sur ce composant.

Il est donc possible de capter les événements les plus pertinents selon le degré de sophistication de l'interface que l'on doit programmer.

Finalement, rappelons que le mécanisme d'événement ne sert pas uniquement à gérer l'interface utilisateur. Il offre un moyen de communication général entre objets. On peut, par exemple, concevoir un objet qui signale à ses écouteurs tout changement de valeur de l'une de ses variables.

13.11 Invitation à explorer

Nous vous invitons à examiner de plus près le package *java.awt.event* qui contient toutes les classes d'événements relatifs à l'interface graphique.

Vous pouvez également reprendre une applet d'animation et tenter de la contrôler à l'aide de la souris.

Chapitre 14

Composants d'interaction graphique

Nous allons examiner de manière plus exhaustive le package *java.awt*. Nous avons déjà étudié les objets graphiques (*Graphics, Rectangle, Point*, etc.), leurs attributs (*Fonts*, *Color*, etc.) et la gestion des événements (*Event*). Ce chapitre présentera des composants graphiques dont le comportement est déjà établi (les boutons, les listes, etc.). L'ensemble de ces composants (voir figure 14.1) constitue l'interface graphique utilisateur (*Graphical User Interface*). Le package *awt* contient la définition conceptuelle des objets (leur comportement et leur spécification), tandis que l'aspect graphique des objets dépend quant à lui de la plate-forme d'exécution : une même applet aura donc un comportement similaire sur des plates-formes différentes, par contre sa représentation graphique sera adaptée au style en vigueur sur cette plate-forme (Macintosh, PC, Unix, etc.).

L'addition d'un composant au contenant graphique s'effectue au moyen de la méthode *add()*. Dans le cas d'une applet, l'objet adressé est du type *Panel* (panneau d'affichage). La méthode de mise en pages par défaut est *coulissante*, c'est-à-dire que les composants graphiques sont ajoutés de gauche à droite et que l'on revient à la ligne dès que nécessaire. Nous reviendrons en détail sur les conventions de mise en pages au chapitre 16.

Nous allons examiner systématiquement le paramétrage, la construction et la manipulation des principaux objets graphiques proposés dans *java.awt*.

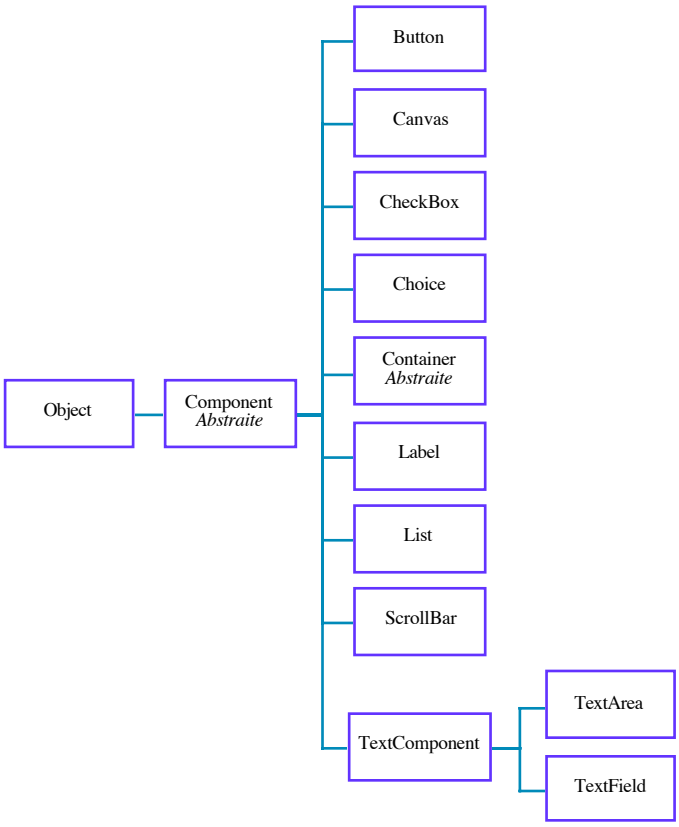


Figure 14.1 Hiérarchie des composants graphiques

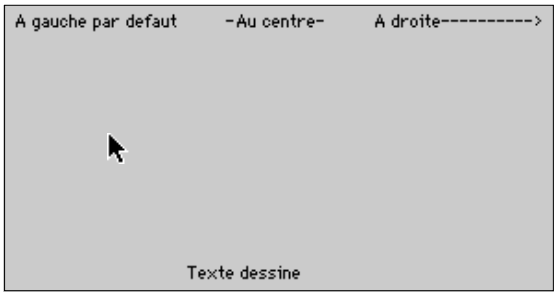
14.1 Les libellés

Les libellés sont des objets de la classe *Label*. Ils permettent d'afficher un texte dans un contenant graphique (voir applet 32). Jusqu'à présent, nous avons dessiné (peint) du texte dans le composant graphique; ici nous associons un texte et un type d'alignement à un composant graphique. La spécification d'un libellé peut être modifiée à tout moment.

L'exemple suivant montre la création de libellés et leur association à l'applet :

```
import java.awt.*;

public class Composant extends java.applet.Applet {

   public void init() {
      add(new Label ("A gauche par defaut"));
      add(new Label ("-Au centre-",Label.CENTER));
      add(new Label ("A droite---------->", Label.RIGHT));
   }
```

```
    public void paint(Graphics g) {
       g.drawString("Texte dessine", 100, 150);
    }
}
```



Applet 32 : Affichage, les libellés

Les paramètres d'alignement sont des variables de classe constantes : *Label.CENTER*, *Label.LEFT*, *Label.RIGHT*. Par défaut, l'alignement à gauche est utilisé. Le tableau 14.1 décrit les principales méthodes de la classe *Label*.

Méthode	Définition
`Label()`	Crée un libellé sans texte.
`Label(String l)`	Crée un libellé de texte. l
`Label(String l, int a)`	Crée un libellé de texte l et d'alignement a.
`getAlignment()`	Retourne l'alignement du libellé.
`getText()`	Retourne le texte du libellé.
`setAlignment(int a)`	Impose un alignement.
`setText(String label)`	Assigne un texte au label.

Tableau 14.1 Méthodes de la classe Label

14.2 Les boutons

Les boutons sont des objets de la classe *Button*. Ils correspondent à ce que l'on attend : des objets portant un titre et réagissant quand on clique sur eux. Ils émettent des événements de type *ActionEvent* qui sont envoyés aux objets qui implémentent l'interface *ActionListener* et qui se sont inscrits.

L'applet 33 montre la création de boutons (*new Button(...)*) et leur ajout à l'applet, *add(...)*), puis l'association d'un écouteur (en l'occurrence l'applet elle-même) à chaque bouton (*addActionListener(this)*). Lorsqu'un bouton est enfoncé la méthode *actionPerformed()* est appelée et on récupère la commande, en l'occurrence le nom du bouton, dans l'événement transmis. Finalement, la

méthode *paint()* choisit la fonte à utiliser en fonction de la dernière commande reçue.

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

public class Composant extends Applet implements ActionListener {

   String commande = "Normal";
   Button bn, bg, bi;
   Font fgras, fnormal, fitalic, fdef;

   public void init() {
      add(bn = new Button ("Normal"));
      add(bg = new Button("Gras"));
      add(bi = new Button("Italique"));
      bn.addActionListener(this);
      bg.addActionListener(this);
      bi.addActionListener(this);
      fgras   = new Font("Helvetica",Font.BOLD,40);
      fnormal = new Font("Helvetica",Font.PLAIN,40);
      fitalic = new Font("Helvetica",Font.ITALIC,40);
      fdef    = new Font("Helvetica",Font.PLAIN,12);
   }
   public void actionPerformed(ActionEvent e) {
      commande = e.getActionCommand();
      repaint();
   }
   public void paint(Graphics g) {
      if (commande.equals("Normal")) g.setFont(fnormal);
      if (commande.equals("Gras")) g.setFont(fgras);
      if (commande.equals("Italique")) g.setFont(fitalic);
      g.drawString("Exemple", 20, 80);
      g.setFont(fdef);
   }
}
```

Applet 33 : Affichage, les boutons

Le tableau 14.2 décrit les principales méthodes de la classe *Button*.

Méthode	Définition
`Button()`	Crée un bouton sans libellé.

Tableau 14.2 Méthodes de la classe Button

Méthode	Définition
`Button(String b)`	Crée un bouton avec libellé b.
`getLabel()`	Retourne le libellé du bouton.
`setLabel(String b)`	Assigne un libellé b.
`addActionListener (ActionListener l)`	Ajoute un écouteur d'action l.
`removeActionListener (ActionListener l)`	Supprime un écouteur.
`setActionCommand(String c)`	Définit le nom de commande qui sera donné aux événements *ActionEvent* produits (par défaut c'est le libellé du bouton).
`getActionCommand()`	Retourne le nom de commande défini.

Tableau 14.2 Méthodes de la classe Button

Pour accéder aux informations transmises par un événement de type *ActionEvent*, on utilisera les méthodes de cette classe décrites dans le tableau 14.3 ci-dessous.

Méthode	Définition
`getSource()`	Fournit l'objet qui a émis l'événement (méthode héritée de *EventObject*).
`getActionCommand()`	Retourne le String représentant la commande associée à cet événement.
`getModifiers()`	Retourne un entier qui contient les codes des modificateurs (shift, alt, ctrl...) associés à l'événement.
`paramString()`	Retourne une représentation sous forme de String de l'événement (utile pour la mise au point).

Tableau 14.3 Méthode de la classe ActionEvent

Le recours à la méthode *getSource()* est nécessaire dans le cas où un objet (par exemple notre applet) est abonné à plusieurs sources du même type d'événement (par exemple, plusieurs boutons). Il faut noter que plusieurs boutons peuvent produire la même commande et qu'un bouton peut produire différentes commandes au cours du temps.

La classe *ActionEvent* fournit quatre constantes : *ALT_MASK*, *CTRL_MASK*,

META_MASK, *SHIFT_MASK* pour tester quel(s) modificateur(s) étai(en)t enfoncé(s) au moment de l'événement. Par exemple :

```
public void actionPerformed(ActionEvent e) {
   if (e.getModifiers() & ActionEvent.SHIFT_MASK)
      { ... la touche SHIFT était enfoncée ...}
```

14.3 Les boîtes à cocher à choix multiple

Il existe deux types de boîtes à cocher : dans le premier, chaque boîte est indépendante, ce qui permet la sélection multiple d'options. Dans l'autre type (voir point 14.4), elles sont regroupées et fonctionnent en mode exclusif (une seule boîte cochée à la fois). Les boîtes à cocher sont toutes issues de la classe *Checkbox*. Lorsqu'une boîte change d'état, parce que l'utilisateur a cliqué dessus, la boîte appelle la méthode *itemStateChanged()* des objet inscrits comme *ItemListener*.

L'applet 34 montre la création de boîtes à cocher, l'inscription de l'applet en tant qu'*ItemListener*, le traitement des événements et l'affichage de résultats en fonction de l'état des boîtes.

```
public class Composant extends Applet implements ItemListener {

   String message = "", composition;
   Checkbox[] ingredients;
   double[] prix = {5.00, 5.00, 2.00, 1.50, 1.50, 1.00};

   public void init() {
      ingredients = new Checkbox[6];
      add(ingredients[0] = new Checkbox ("Fromage", null, true));
      add(ingredients[1] = new Checkbox ("Tomates", null, true));
      add(ingredients[2] = new Checkbox ("Jambon"));
      add(ingredients[3] = new Checkbox ("Champignons"));
      add(ingredients[4] = new Checkbox ("Thon"));
      add(ingredients[5] = new Checkbox ("Anchois"));
      for (int i=0; i<ingredients.length; i++)
         ingredients[i].addItemListener(this);
   }

   public void itemStateChanged(ItemEvent e)  {
      Checkbox lequel = (Checkbox)e.getItemSelectable();
      message = lequel.getLabel();
      int changement = e.getStateChange() ;
      if (changement == ItemEvent.SELECTED)
         message = "Ajout de "+message;
      else message = "Suppression de "+message;
      repaint();
   }
```

```
    public void paint(Graphics g) {
       composition = "";
       double p = 5.00;
       for (int i=0; i<ingredients.length; i++)
          if (ingredients[i].getState()) {
             composition += (" " + ingredients[i].getLabel());
             p += prix[i];
          }
       g.drawString(message, 20, 100);
       g.drawString("Une pizza du chef avec "+composition, 20, 120);
       g.drawString("Prix: "+p, 20, 140);

    }

}
```

Applet 34 : Affichage, les boîtes à cocher

Le tableau 14.4 regroupant les principales méthodes de la classe *Checkbox*.

Méthode	Définition
`Checkbox()`	Crée une boîte à cocher sans libellé.
`Checkbox (String b)`	Crée une boîte à cocher avec libellé b.
`getLabel()`	Retourne le titre de la boîte à cocher.
`setLabel(String b)`	Assigne un libellé b.
`getState()`	Retourne l'état de la boîte à cocher (booléen).
`setState(boolean c)`	Assigne l'état c à la boîte à cocher.

Tableau 14.4 Méthodes de la classe Checkbox

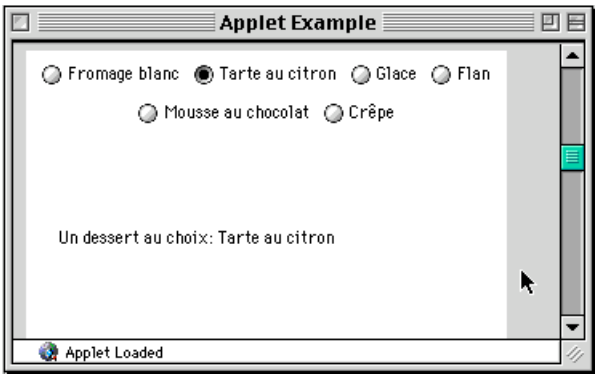
Pour reconnaître le type de changement subi par une boîte, on utilise la méthode *getStateChange()* de la classe *ItemEvent*.

14.4 Les boîtes à cocher à choix exclusif

Les boîtes à cocher de ce type travaillent dans un mode exclusif et sont associées à un groupe de boîtes. Ce groupe est spécifié à l'aide d'un objet *CheckboxGroup* (voir tableaux 14.5 et 14.6).

L'applet 35 montre la création de boîtes à cocher associées à un groupe et leur association à l'applet :

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

public class Composant extends Applet implements ItemListener {

   String message, composition;
   CheckboxGroup dessert;
   Checkbox[] ingredients;

   public void init() {
      dessert = new CheckboxGroup();
      ingredients = new Checkbox[6];
      add(ingredients[0] =
         new Checkbox("Fromage blanc", dessert, false));
      add(ingredients[1] =
         new Checkbox ("Tarte au citron", dessert, true));
      add(ingredients[2] =
         new Checkbox ("Glace", dessert, false));
      add(ingredients[3] =
         new Checkbox ("Flan", dessert, false));
      add(ingredients[4] =
         new Checkbox ("Mousse au chocolat", dessert, false));
      add(ingredients[5] =
         new Checkbox ("Crêpe", dessert, false));

      for (int i=0; i<ingredients.length; i++)
         ingredients[i].addItemListener(this);
   }

   public void itemStateChanged(ItemEvent e)  {
      repaint();
   }

   public void paint(Graphics g) {
      Checkbox choix = dessert.getSelectedCheckbox();
      g.drawString("Un dessert au choix: "+choix.getLabel(), 20,
120);

   }

}
```



Applet 35 : Affichage, les boîtes à cocher en mode exclusif

Méthode ou constructeur	Définition
`Checkbox(String b, CheckboxGroup g, boolean s)`	Crée une boîte à cocher avec libellé *b*, associée au groupe *g* dans l'état *s*.
`setCheckboxGroup(CheckboxGroup g)`	Associe la boîte au groupe *g*.

Tableau 14.5 Méthodes de la classe Checkbox utilisant la notion de groupe

Méthode ou constructeur	Définition
`CheckboxGroup()`	Crée un groupe.
`getSelectedCheckbox()`	Retourne la boîte à cocher actuellement sélectionnée.
`setSelectedCheckbox (Checkbox b)`	Sélectionne la boîte à cocher *b*.

Tableau 14.6 Méthodes de la classe CheckboxGroup

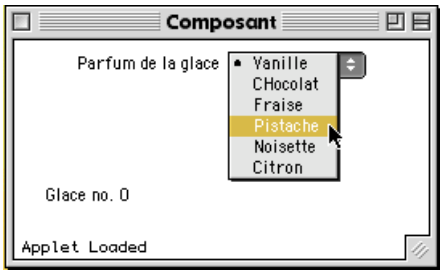
14.5 Les menus déroulants

Les menus déroulants sont des objets de la classe *Choice*. Un menu déroulant est composé d'articles *(item)*. À l'état de repos, il affiche l'article sélectionné. En cliquant sur lui, on affiche l'ensemble des articles sélectionnables et on peut alors modifier la sélection. On accède aux menus déroulants en termes de position dans le menu *(index)* ou de libellé d'un article *(label)*. Un seul article est sélectionnable à la fois. Lors d'une sélection, un événement *ItemEvent* est généré.

L'exemple suivant (applet 36) montre la création d'un menu déroulant et l'affichage de l'index de l'élément sélectionné :

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

```
public class Composant extends Applet implements ItemListener  {

   Choice parfum;

   public void init() {
      parfum = new Choice();
      parfum.addItem("Vanille");
      parfum.addItem("CHocolat");
      parfum.addItem("Fraise");
      parfum.addItem("Pistache");
      parfum.addItem("Noisette");
      parfum.addItem("Citron");
      add(new Label("Parfum de la glace"));
      add(parfum); // ajoute le menu après le libellé
      parfum.select("Vanille");
      parfum.addItemListener(this); // l'applet écoute le menu
   }
   public void itemStateChanged(ItemEvent e)  {
      repaint();
   }
   public void paint(Graphics g) {
      int parfumNo = parfum.getSelectedIndex() ;
      g.drawString("Glace no. "+parfumNo, 20, 100);
   }
```



Applet 36 : Affichage, les menus déroulants (phase de sélection)

Le tableau 14.7 décrit les principales méthodes de la classe *Choice*.

Méthode	Définition
`Choice()`	Crée un menu.
`addItem(String b)`	Ajoute un article de libellé b.
`getItem(int p)`	Retourne le libellé de l'article à la position p.
`remove(String  b)`	Supprime le premier article libellé b.
`getItemCOunt()`	Fournit le nombre d'articles du menu.

Tableau 14.7 Méthodes de la classe Choice

Méthode	Définition
`getSelectedItem()`	Fournit le libellé de l'article sélectionné.
`getSelectedIndex()`	Fournit la position de l'article sélectionné.
`select(int p)`	Sélectionne l'article à la position p.
`addItemListener(ItemListener l)` `removeItemListener(…)`	Ajoute (supprime) un écouteur de sélection d'articles.

Tableau 14.7 Méthodes de la classe Choice

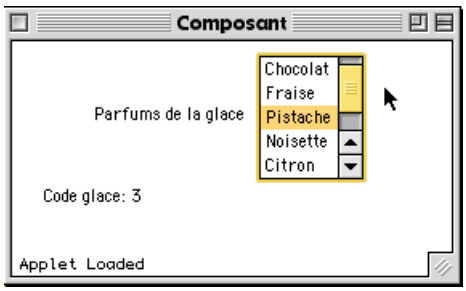
14.6 Les listes de choix

Pour les sélections multiples, les listes de choix sont appropriées. Les listes de choix sont des objets de la classe *List*. Un événement *ActionEvent* est généré lors d'un double clic sur un article ou lorsqu'on presse la touche retour alors qu'un article est sélectionné. La sélection ou la désélection d'un article (clic simple) génère un événement *ItemEvent*.

L'applet 37 montre la création d'une liste de choix et la détection des sélections et désélections.

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

public class Composant extends Applet implements ItemListener  {

   List parfum;

   public void init() {
      parfum = new List(5,true);
      parfum.add("Vanille");
      parfum.add("Chocolat");
      parfum.add("Fraise");
      parfum.add("Pistache");
      parfum.add("Noisette");
      parfum.add("Citron");
      parfum.add("Orange");
      parfum.add("Abricot");
      parfum.setMultipleMode(true);

      add(new Label("Parfums de la glace"));
      add(parfum); // ajoute la liste après le libellé
      parfum.select(3); // préselectionne Pistache
      parfum.addItemListener(this); // l'applet écoute le menu
   }
   public void itemStateChanged(ItemEvent e)  {
      repaint();
   }
   public void paint(Graphics g) {
```

```
        int[] parfumNos = parfum.getSelectedIndexes();
        String message = "Code glace: ";
        for (int i=0; i<parfumNos.length; i++)
           message = message + parfumNos[i];
        g.drawString(message, 20, 100);
     }

}
```



Applet 37 : Affichage, les listes de choix

Le tableau 14.8 regroupe les principales méthodes de la classe *List*.

Méthode ou constructeur	Définition
`List()`	Crée une liste de choix.
`List(int lignes, boolean choixMultiple)`	Crée une liste de choix de n lignes visibles spécifiant si le choix multiple est autorisé.
`add(String l)`	Ajoute un article de libellé l.
`add(String l,int p)`	Ajoute un article de libellé l à la position p.
`remove(String l)`	Supprime un article de libellé l.
`remove(int p)`	Supprime l'article à la position p.
`replaceItem(String l, int p)`	Remplace le libellé de l'article à la position *p* par *l*.
`getItemCount()`	Retourne le nombre d'articles de la liste.
`getItem(int p)`	Retourne le libellé (*String*) de l'article à la position *p*.
`setMultipleMode (boolean b)`	Autorise ou non le choix multiple.
`select(int p)`	Sélectionne l'article à la position *p*.
`deselect(int p)`	Désélectionne l'article à la position *p*.

Tableau 14.8 Méthodes de la classe List

Méthode ou constructeur	Définition
`getSelectedIndex()`	Retourne la position (*int*) de l'article sélectionné.
`getSelectedItem()`	Retourne le libellé (*String*) de l'article sélectionné.
`getSelectedIndexes()`	Retourne les positions (*int[]*) des articles sélectionnés.
`getSelectedItems()`	Retourne les libellés (*String[]*) des articles sélectionnés.
`isIndexSelected(int p)`	Retourne un booléen qui indique si l'article *p* est sélectionné ou non.
`addActionListener (ActionListener l)`	Ajoute un écouteur d'actions (double clic et retour).
`addItemListener (ItemListener l)`	Ajoute un écouteur pour les changements de sélection.

Tableau 14.8 Méthodes de la classe List

14.7 Les champs de texte sur une ligne

Les champs de texte héritent tous de la classe *TextComponent* dans laquelle sont définies les méthodes permettant la manipulation de texte (voir tableau 14.9).

Méthode	Définition
`pas de constructeur!`	Les classes *TextField* et *TextArea* ont leur propres constructeurs.
`getText()`	Retourne le texte contenu dans le champ (String).
`getSelectedText()`	Retourne le texte sélectionné (String).
`getSelectionEnd()`	Retourne la position de la fin du texte sélectionné (int).
`getSelectionStart()`	Retourne la position du début du texte sélectionné (int).
`isEditable()`	Teste si le champ est éditable.
`select(int début, int fin)`	Sélectionne le texte de la position début à la position fin.

Tableau 14.9 Méthodes de la classe TextComponent

Méthode	Définition
`selectAll()`	Sélectionne la totalité du texte.
`setText(String l)`	Assigne le texte du champ à la valeur l spécifiée.
`setCaretPosition(int pos)`	Place le symbole d'insertion (*caret*) à la position indiquée.
`getCaretPosition()`	Retourne la position du symbole d'insertion.

Tableau 14.9 Méthodes de la classe TextComponent

Le champ de texte sur une ligne est un objet de la classe *TextField*. Le contenu du champ peut être édité par l'utilisateur. La touche *retour (return)* génère un événement *ActionEvent* qui est transmis aux éventuels *ActionListener*s inscrits. L'événement généré contient la valeur du champ au moment où la touche fut pressée. De plus, chaque modification du texte génère un événement *TextEvent* transmis aux *TestListener*s par appel de leur méthode *textValueChanged()*.

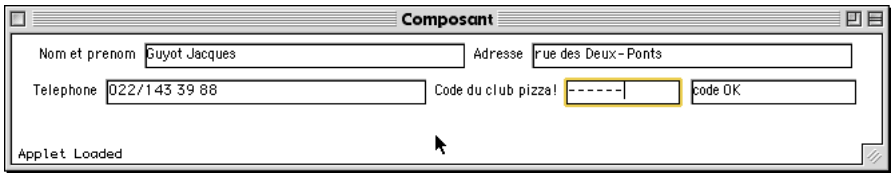
Il est possible de spécifier un caractère d'écho (exemple : '-') à afficher à la place des caractères entrés, ce qui permet par exemple de saisir des mots de passe sans qu'ils apparaissent en clair à l'écran.

Les méthodes de la classe *TextField* sont présentées dans le tableau 14.10; l'applet 38 montre la création d'un masque de saisie comportant plusieurs champs, elle écoute les événements *TextEvent* du champ *code* et vérifie si le contenu de ce champ correspond au mot de passe[1] :

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

public class Composant extends Applet implements TextListener {

   String msg;
   TextField np, adr, tel, code, message;

   public void init() {
      add(new Label("Nom et prenom"));
      add(np = new TextField(30));
      add(new Label("Adresse"));
      add(adr = new TextField(30));
      add(new Label("Telephone"));
      add(tel = new TextField("022/",30));
      add(new Label("Code du club pizza!"));
      add(code = new TextField(10));
      code.setEchoChar('-');
      add(message = new TextField(15));
```

1. Attention, ne mettez jamais un mot de passe en clair dans une applet, il pourrait facilement être retrouvé !

```
        code.addTextListener(this);
    }
    public void textValueChanged(TextEvent e) {
        if (e.getSource() == code) {
            if (code.getText().equals("secret")) // vérif. mot de passe
                message.setText("code OK");
            else message.setText("entrez le code!");
        }
    }
}
```



Applet 38 : Affichage, les champs de texte sur une ligne

La table ci-dessous présente les principales méthodes de la classe *TextField* :

Méthode ou constructeur	Définition
`TextField()`	Crée un champ.
`TextField(int n)`	Crée un champ de n caractères.
`TextField(String l)`	Crée un champ initialisé avec le texte l.
`TextField(String l,int n)`	Crée un champ de n caractères (approximativement), initialisé avec le texte l.
`echoCharIsSet()`	Teste si le caractère d'écho est défini.
`getColumns()`	Retourne la longueur (int) du champ.
`setColumns(int n)`	Définit la longueur du champ de manière à ce qu'il puisse contenir environ n caractères.
`getEchoChar()`	Retourne le caractère d'écho.
`setEchoChar(char c)`	Définit le caractère d'écho.

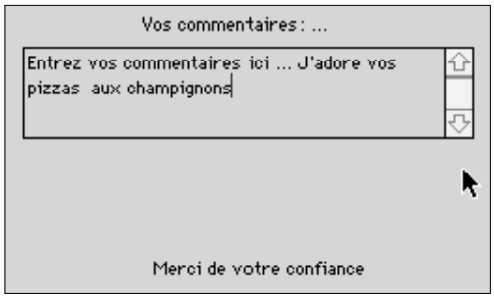
Tableau 14.10 Méthodes de la classe TextField

14.8 Les champs de texte sur plusieurs lignes

Le champ de texte sur plusieurs lignes est pour l'essentiel identique à celui sur une seule ligne. Les objets sont de la classe *TextArea*, qui comprend une gestion du nombre de lignes en plus de celle des colonnes.

L'applet 39 montre la création d'un champ sur plusieurs lignes :

```
import java.awt.*;

public class Composant extends java.applet.Applet {

   public void init() {
      add(new Label("Vos commentaires: ..."));
      add(new TextArea("Entrez vos commentaires ici ...",4,30));
   }
   public void paint(Graphics g) {
      g.drawString("Merci de votre confiance", 100, 150);
   }
}
```



Applet 39 : Affichage, les champs de texte sur plusieurs lignes

En plus des méthodes héritées de *TextComponent*, *TextArea* possède les méthodes ci-dessous :

Méthode ou constructeur	Définition
`TextArea()`	Crée un champ.
`TextArea (int n, int j)`	Crée un champ de n caractères de j lignes.
`TextArea (String t)`	Crée un champ initialisé avec t.
`TextArea (String t, int n, int j)`	Crée un champ initialisé avec t, de n caractères et j lignes.
`append(String t)`	Ajoute le texte t à la fin du champ.
`insert(String t,int n)`	Insère le texte t à la position n.
`replaceRange(String t, int d, int f)`	Remplace le texte des positions *d* à *f* par le texte *t*.

Tableau 14.11 Méthodes de la classe TextArea

Méthode ou constructeur	Définition
`getColumns()` `setColumns(int n)`	Retourne la longueur (*int*) du champ. Définit la longueur du champ.
`getRows()` `setRows(int k)`	Retourne le nombre de lignes (*int*) du champ. Définit le nombre de lignes.

Tableau 14.11 Méthodes de la classe TextArea

14.9 Les barres de défilement

Les barres de défilement sont des objets de la classe *Scrollbar*. Une barre peut être soit verticale soit horizontale, elle comporte différentes parties (l'ascenseur, les flèches, la piste) sur lesquelles l'utilisateur peut agir par des mouvements de la souris. L'état courant de la barre représente une valeur entière que l'utilisateur modifie par ses actions.

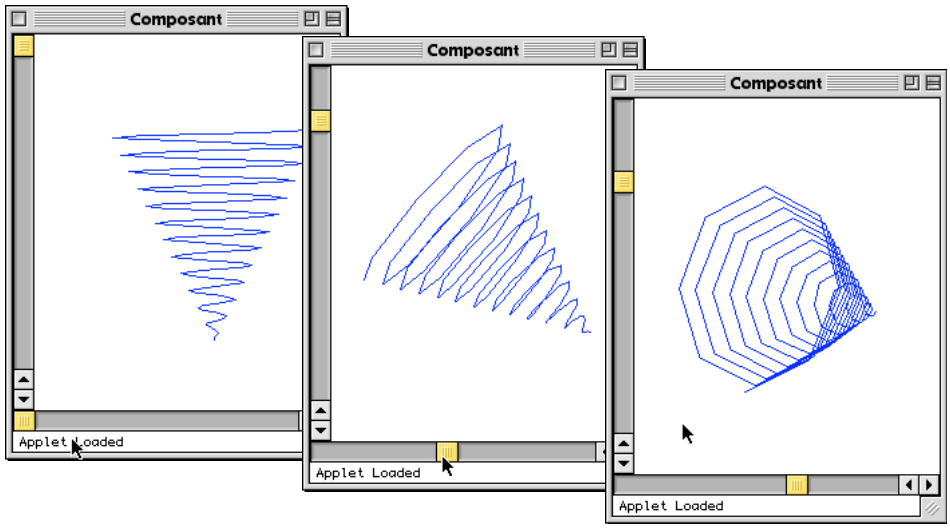
Lorsque la valeur de la barre est modifiée, celle-ci envoie un événement *AdjustmentEvent* que l'on capte en implémentant la méthode *adjustmentValueChanged()* de l'interface *AdjustmentListener*. On peut connaître le type de mouvement effectué en appliquant la méthode *getAdjustmentType()* sur l'événement reçu. Les mouvements possibles sont : *TRACK* (l'utilisateur a tiré l'ascenseur); *UNIT_INCREMENT* (clic sur la flèche vers la droite ou vers le haut); *UNIT_DECREMENT* (clic sur la flèche vers la gauche ou vers le bas); *BLOCK_INCREMENT* (clic à droite ou au dessus de l'ascenseur); *BLOCK_DECREMENT* (clic à gauche ou au-dessous de l'ascenseur).

La variable associée à la barre possède une valeur minimum et une valeur maximum que l'on indique au moment de la création. La taille de l'ascenseur peut aussi être définie et modifiée, par exemple pour représenter la proportion du texte visible dans une fenêtre.

L'applet suivante montre l'utilisation de deux barres de défilement qui servent à déterminer deux angles de rotation d'un objet en trois dimensions. Chaque modification de la valeur d'une barre entraîne le rafraîchissement du dessin pour tenir compte des nouveaux angles. Pour pouvoir placer correctement les barres, on utilise une mise en pages de type *bord* que nous décrirons dans le chapitre consacré aux protocoles de mise en pages.

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

public class Composant extends Applet implements AdjustmentListener
{

   static final int N = 100, X = 0, Y = 1, Z = 2;
```

```
   int initial = 0, visi = 1, min = 0, max = 100;
   Scrollbar sba, sbb;
   double [][] points; // points 3D de la figure
   double alpha, beta; // angles de vision

   public void init() {
      points = new double[N][3];
      // construction d'une spirale en 3D dans le cube [-80,80]^3
      for (int i = 0; i<N; i++) {
         points[i][Y] = i*160.0/N - 80.0;
         double r = 80 - i*80.0/N;
         double a = i*2*Math.PI/8;
         points[i][X] = Math.cos(a)*r;
         points[i][Z] = Math.sin(a)*r;
      }
      setLayout(new BorderLayout());
      sba = new Scrollbar(Scrollbar.VERTICAL, initial, visi,
min,max);
      add(sba, BorderLayout.WEST);
      sba.addAdjustmentListener(this);
      sbb = new Scrollbar(Scrollbar.HORIZONTAL, initial, visi,
min,max);
      add(sbb, BorderLayout.SOUTH);
      sbb.addAdjustmentListener(this);
   }
   public void paint(Graphics g) {
      double x,y,z;
      int xp = 0, yp = 0; // mémoire du point précédent
      g.setColor(Color.blue);
      for (int i = 0; i<N; i++) {
         // rotation selon z
         x = points[i][X]*Math.cos(alpha)-
points[i][Y]*Math.sin(alpha);
         y =
points[i][X]*Math.sin(alpha)+points[i][Y]*Math.cos(alpha);
         z = points[i][Z];
         // rotation selon y
         x = x*Math.cos(beta)-z*Math.sin(beta);
         z = x*Math.sin(beta)+z*Math.cos(beta);
         // projection sur (X,Y) (l'axe Z est celui qui sort de
l'écran)
         if (i>0) g.drawLine(xp, yp, (int)x+150,(int)y+150);
         xp = (int)x+150; yp = (int)y+150;
      }

   }
   public void adjustmentValueChanged(AdjustmentEvent e) {
      if (e.getSource() == sba) alpha = e.getValue()*2*Math.PI/100;
      if (e.getSource() == sbb) beta = e.getValue()*2*Math.PI/100;
      repaint();
   }
} }
```



Applet 40 : Deux barres de défilement déterminent les angles de vue

14.10 Les fonds

Les fonds sont des objets de la classe *Canvas*. Ils permettent de créer une surface de travail pour dessiner, afficher des images, etc. En étendant cette classe, il est possible de redéfinir le comportement d'un fond en surchargeant la méthode *paint()*.

Un *Canvas* est sensible aux événements générés par la souris et par le clavier.

14.11 Préférences lors de l'affichage

Nous avons omis de décrire pour chaque classe les préférences d'affichage, la taille minimum et les divers paramètres de mise en pages. Ces paramètres sont à spécifier dans le cas d'interfaces complexes destinées à être largement diffusées. Ils permettent de définir des contraintes utilisées lors du calcul automatique de la mise en pages.

14.12 Invitation à explorer

Parcourez les documents HTML décrivant ces différentes classes.

Examinez la classe *Toolkit* qui indique certains paramètres d'implantation de la plate-forme physique utilisée.

Examinez la classe *Component* de manière plus détaillée.

Chapitre 15

Gestion des conteneurs

Le dernier composant graphique qu'il nous reste à décrire est la classe abstraite *Container*. Un conteneur est un espace graphique destiné à recevoir plusieurs composants (boutons, fonds, listes, boîtes à cocher, etc.) dont la mise en pages graphique dépendra du gestionnaire de mise en pages utilisé. Les méthodes de la classe *Container* sont donc spécifiques à la gestion des composants (*add(), countComponent(), getComponent(),* etc.) et des gestionnaires de mise en pages (*layout(), getLayout()*, etc.).

À partir de *Container*, nous avons la hiérarchie de classes suivante :

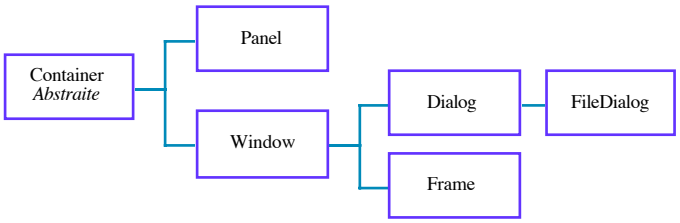


Figure 15.1 Hiérarchie des conteneurs

La classe *Panel* (panneau) représente des conteneurs destinés à être placés dans un espace existant (un autre conteneur ou un butineur dans le cas des applets).

La classe *Applet* étend la classe *Panel* (voir figure 15.2).

La classe *Window* crée des espaces indépendants (nouvelles fenêtres). Les deux classes directement utilisables sont *Dialog*, qui permet la gestion des dialogues (avec un comportement modal) et *Frame* qui crée une surface de travail dans une fenêtre afin de recevoir des composants graphiques simples. À noter que

Frame implante l'interface *MenuContainer*, ce qui signifie qu'il est possible de spécifier les interactions avec des menus propres à chaque fenêtre.

La classe *FileDialog* est une spécialisation de *Dialog* pour la gestion des interactions par rapport à un répertoire de fichiers (voir point 19.14, p. 260).

Nous allons maintenant voir quelques utilisations possibles de ces conteneurs à travers des exemples.

15.1 Mon applet est trop petite

Dans l'applet 41, nous reprenons notre programme permettant de dessiner avec la souris (applet 30, p. 170). Nous désirons ajouter un champ de texte dans lequel nous afficherons les coordonnées des segments dessinés.

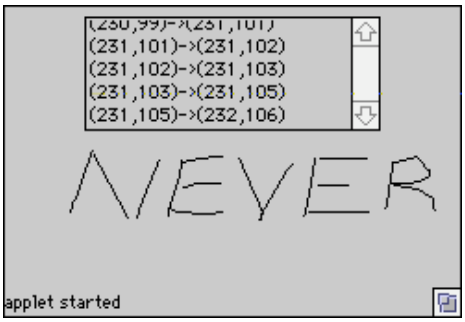
Lors de l'initialisation, nous déclarons donc un objet de type *TextArea* que nous ajoutons à notre applet (qui est également un *Panel* par héritage).

```
t=new TextArea("(x,y)",5,20);
this.add(t);
```

Ensuite, à chaque fois que l'on dessine un segment, on ajoute sa coordonnée dans le champ :

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

public class Dessin extends Applet {

   int x0 = 0, y0 = 0;
   TextArea t;

   class MSouris extends MouseMotionAdapter {
      public void mouseDragged(MouseEvent e) {
         Graphics g = getGraphics();
         g.drawLine(x0, y0, e.getX(), e.getY());
         t.append("\n("+x0+","+y0+")-
>("+e.getX()+","+e.getY()+")");
         x0 = e.getX(); y0 = e.getY();
      }
   }

   class CSouris extends MouseAdapter {
      public void mousePressed(MouseEvent e) {
         x0 = e.getX(); y0 = e.getY();
      }
   }

   public void init() {
      t = new TextArea("(x, y)", 5, 20);
      this.add(t);
      addMouseListener(new CSouris());
      addMouseMotionListener(new MSouris());
   }
}
```

Nous constatons que les coordonnées s'inscrivent bien dans le champ de texte. Malheureusement, celui-ci occupe une partie importante de notre espace de travail (par ailleurs limité), nous privant ainsi d'une large zone de dessin. Que faire dans ce cas? C'est ce que nous allons voir.



Applet 41 : Dessiner et afficher les coordonnées dans un champ de texte

15.2 Ouvrir une autre fenêtre

Pour conserver à la fois l'ensemble des fonctionnalités de l'exemple précédent et un espace complet pour dessiner, il est nécessaire de créer une fenêtre séparée destinée à l'affichage des coordonnées (voir applet 42).

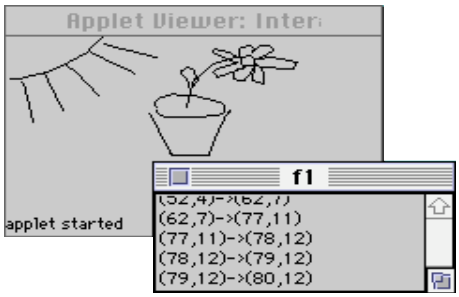
À cet effet, nous créons une nouvelle classe *Fenetre* étendant *Frame*, qui possédera un champ de texte et une méthode *afficher()* pour ajouter du texte dans ce champ. Le constructeur de *Fenetre* crée le champ texte et l'ajoute à la fenêtre, il invoque ensuite de la méthode *pack()* qui positionne correctement les composants de la fenêtre en fonction de leur taille, puis rend la fenêtre visible avec *show()*.

L'applet est modifiée de manière à déclarer une fenêtre *f1*. À chaque nouveau segment, nous ajoutons les coordonnées de celui-ci dans la fenêtre *f1* à l'aide de la méthode *afficher()*.

On notera également la définition de la méthode *stop()* dans l'applet, qui demande à la fenêtre qu'elle a créée de disparaître (*dispose()*).

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

class Fenetre extends Frame {
   TextArea t;

   Fenetre(String titre) {
      super(titre); // constructeur de Frame
      t = new TextArea("(x, y)", 5, 20);
      add("Center", t);
      pack();
```

```
      show();
   }
   public void afficher(String s) {
      t.append(s);
   }
}
public class Interaction extends Applet {
   int x0 = 0, y0 = 0;
   Fenetre f1;

   class MSouris extends MouseMotionAdapter {
      public void mouseDragged(MouseEvent e) {
         Graphics g = getGraphics();
         g.drawLine(x0, y0, e.getX(), e.getY());
         f1.afficher("\n("+x0+","+y0+")-
>("+e.getX()+","+e.getY()+")");
         x0 = e.getX(); y0 = e.getY();
      }
   }
   class CSouris extends MouseAdapter {
      public void mousePressed(MouseEvent e) {
         x0 = e.getX(); y0 = e.getY();
      }
   }
   public void init() {
      f1 = new Fenetre("f1");
      addMouseListener(new CSouris());
      addMouseMotionListener(new MSouris());
   }
   public void stop() {
      f1.dispose();
   }
}
```



Applet 42 : Gestion d'un champ de texte dans une fenêtre séparée

15.3 Invocation d'une applet depuis une application

Depuis le début du livre nous avons toujours fait la différence entre une *application* Java (méthode *main()*) et une *applet* Java (invoquée depuis une page HTML).

Dans l'exemple ci-dessous, nous montrons comment créer une application qui démarre une ou plusieurs applet(s). Dans notre cas, chaque applet est un panneau qui demande un cadre afin de pouvoir s'afficher; nous proposons d'utiliser un *Frame* à cet effet.

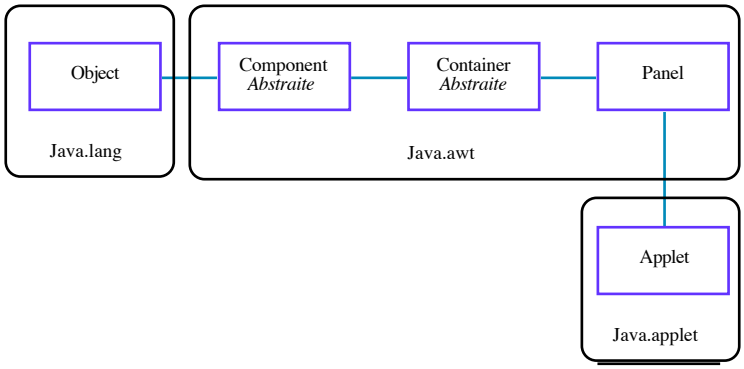


Figure 15.2 La classe Applet hérite de Container

Nous déclarons une nouvelle classe *Fenetre* étendant la classe *Frame*. Nous déclarons ensuite une variable *appletCourante* permettant de référencer l'applet à exécuter.

La classe interne *WAction* sert à définir un objet qui écoute l'événement de fermeture de la fenêtre (méthode *windowClosing()*). Au moment de la fermeture, l'applet sera stoppée et supprimée (*destroy()*) puis la fenêtre sera supprimée (*dispose()*).

Après avoir initialisé la fenêtre et son écouteur, le constructeur de fenêtre ajoute l'applet au centre, l'initialise (*init()*) puis rend la fenêtre visible (*show()*), et finalement démarre l'applet (*start()*). Notons encore que la taille de la fenêtre est fixée avec *setSize()* et non calculée par *pack()* comme dans l'exemple précédent.

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

class Fenetre extends Frame   {

   private Applet appletCourante;

   class WAction extends WindowAdapter {
      public void windowClosing(WindowEvent e) {
         appletCourante.stop();
         appletCourante.destroy();
         ((Window)e.getSource()).dispose();
      }
   }
   Fenetre(Applet a,String titre, int l, int h) {
      super(titre);
```

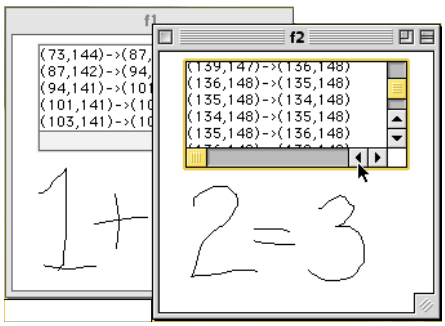
```
        setSize(l, h);
        addWindowListener(new WAction());
        appletCourante = a;
        add(a, BorderLayout.CENTER);
        a.init(); // !! initialiser avant de montrer
        show();
        a.start();
    }
}
```

Nous réutilisons l'applet 41 afin d'illustrer l'invocation depuis une application. À noter que l'applet ne doit pas utiliser de paramètres (*getParameter()*).

```
class Dessin extends java.applet.Applet {
        ...
        // comme dans l'exemple ''mon applet est trop petite''
        ...
}
```

Le programme se résume à l'association de l'applet à sa fenêtre d'exécution. Nous en profitons (applet 43) pour lancer deux exécutions simultanées de la même applet :

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

public class Interaction{
   public static void main(String args[]){
      Fenetre f1=new Fenetre(new Dessin(),"f1",100,200);
      Fenetre f2=new Fenetre(new Dessin(),"f2",200,250);
   }
}
```



Applet 43 : Applet lancée deux fois par une application Java

15.4 Ajouter des menus aux fenêtres

La classe *Frame* implante l'interface *MenuContainer*. On peut ainsi ajouter des menus aux fenêtres. Les menus sont principalement gérés par trois classes :

- *MenuBar* : gère la barre de menus;

- *MenuItem* : gère les entités de chaque menu de la barre;
- *CheckboxMenuItem* : gère les entités « cochables » des menus.

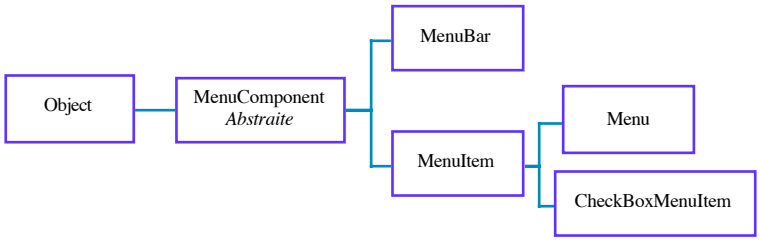


Figure 15.3 Hiérarchie des menus

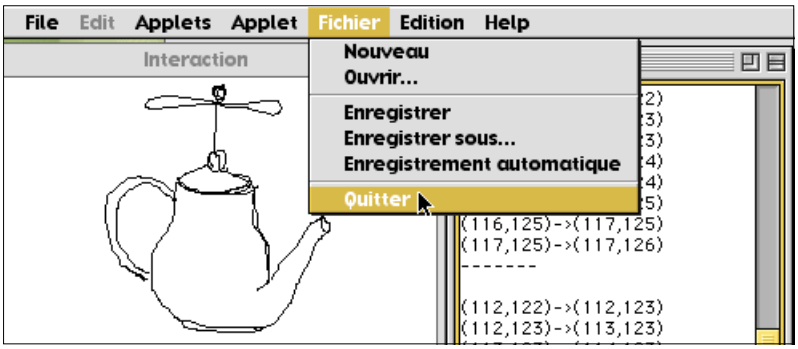
Reprenons l'exemple de l'applet de dessin munie d'une fenêtre d'affichage des coordonnées (applet 42) et voyons comment ajouter des menus à cette fenêtre (applet 44). Nous créons une nouvelle barre de menus à laquelle nous rattachons des menus *Fichier*, *Edition*, *Aide* (ajoutés au fur et à mesure avec *add()*). Ensuite, nous associons à chaque menu ses articles (également avec *add()*, sans oublier d'associer un écouteur d'actions à ceux que nous voulons rendre actifs(*addActionListener()*). Nous insérons également des séparateurs entre les groupes d'articles (*addSeparator()*). Finalement, nous demandons de rendre la barre de menus active (*setMenuBar()*).

```
Fenetre(String titre, int l, int h) {
   super(titre);
   setSize(l, h);
   t = new TextArea("(x, y)", 5, 20);
   add(t, BorderLayout.CENTER);

   MenuBar barreDeMenu = new MenuBar();
   Menu m1 = new Menu("Fichier");
   barreDeMenu.add(m1);
   Menu m2 = new Menu("Edition");
   barreDeMenu.add(m2);
   Menu m3 = new Menu("Aide");
   barreDeMenu.add(m3);
   barreDeMenu.setHelpMenu(m3);

   m1.add(new MenuItem ("Nouveau"));
   m1.add(new MenuItem("Ouvrir..."));
   m1.addSeparator();
   m1.add(new MenuItem ("Enregistrer"));
   m1.add(new MenuItem ("Enregistrer sous..."));
   m1.add(new MenuItem ("Enregistrement automatique"));
   m1.addSeparator();
   m1.add(m1Quitter = new MenuItem ("Quitter"));
   m1Quitter.addActionListener(this);

   m2.add(m2Effacer= new MenuItem ("Effacer"));
```

```
        m2Effacer.addActionListener(this);
        m2.add(m2Separ= new MenuItem ("Separer"));
        m2Separ.addActionListener(this);

        setMenuBar(barreDeMenu);
        show();
    }
```

La gestion de l'événement généré par la sélection d'un article de menu est similaire à celle d'un bouton. La méthode *actionPerformed()* de l'écouteur est appelée à chaque sélection. Cette dernière agit en fonction de la commande transmise (par défaut, c'est le nom de l'article sélectionné).

```
    public void actionPerformed(ActionEvent e) {
        if (e.getActionCommand().equals("Quitter")) this.dispose();
        if (e.getActionCommand().equals("Effacer")) t.setText("");
        if (e.getActionCommand().equals("Separer"))
            afficher("\n-------\n");
    }
}
```



Applet 44 : Création et utilisation de menus

15.5 Un gestionnaire d'alertes

L'exemple suivant (applet 45) montre comment étendre la classe *Dialog* afin d'obtenir une classe *Alerte* qui affiche un message puis attend une réponse de l'utilisateur (clic dans le bouton *OK*) et finalement disparaît. Il faut noter qu'une fenêtre *Dialog* doit toujours être rattachée à un *Frame* ou à un *Dialog* parent que l'on passe comme paramètre du constructeur.

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

class Alerte extends Dialog implements ActionListener{

    Button boutonOK = new Button("OK");
    Fenetre messages;
```

```
    Alerte (Fenetre f, String alerteTexte) {
       super(f, "Alerte", true);
       messages = f;
       add(new Label(alerteTexte), BorderLayout.CENTER);
       add(boutonOK, BorderLayout.SOUTH);
       boutonOK.addActionListener(this);
       pack();
       show();
    }
    public void actionPerformed(ActionEvent e) {
       messages.afficher("--OK--\n");
       dispose();
    }
}
```

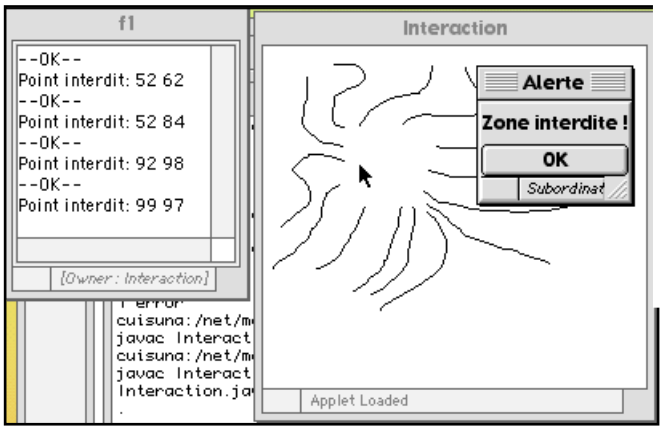
L'utilisation de cette classe se résume à ajouter une ligne du type :

```
Alerte x=new Alerte("<message>",f1);
```

chaque fois que l'on désire solliciter l'attention de l'utilisateur. L'applet ci-dessous dessine en suivant les mouvements de la souris, mais interdit tout dessin dans la zone $50 < x < 100$ et $50 < y < 100$. La variable booléenne *bloque* signale qu'on a pénétré dans la zone interdite. Elle est nécessaire car on peut encore recevoir des événements souris alors que l'alerte est déjà affichée et il faut éviter d'ouvrir plusieurs alertes superposées.

```
public class Interaction extends Applet {
   int x0 = 0, y0 = 0;
   Fenetre f1; Alerte a1;
   boolean bloque = false;

   class MSouris extends MouseMotionAdapter {

      public void mouseDragged(MouseEvent e) {
         Graphics g = getGraphics();
         int x = e.getX(), y = e.getY();
         if (!bloque)
            if (50 < x && x < 100 & 50 < y && y < 100) {
               bloque = true;
               f1.afficher("Point interdit: "+x+" "+y+"\n");
               Alerte a = new Alerte (f1,"Zone interdite !");
            }
            else {
               g.drawLine(x0, y0, x, y);
               x0 = x; y0 = y;
            }
      }
   }
   class CSouris extends MouseAdapter {
      public void mousePressed(MouseEvent e) {
         x0 = e.getX(); y0 = e.getY();
         bloque = false;
```

```
        }
    }
    public void init() {
        // on utilise la classe Fenêtre définie dans l'applet 42
        f1 = new Fenetre("f1", 200, 500);
        addMouseListener(new CSouris());
        addMouseMotionListener(new MSouris());
    }
    public void stop() {
        f1.dispose();
    }
}
```



Applet 45 : Une alerte gérée dans une fenêtre séparée

15.6 Invitation à explorer

Essayez d'ouvrir une fenêtre de dialogue affichant les répertoires de fichiers à l'aide d'une application.

Définissez les composants graphiques d'un mini-éditeur de textes que l'on complétera dans la partie suivante.

Chapitre 16

Protocoles de mise en pages

Nous venons d'examiner les composants graphiques et les conteneurs. Il nous faut maintenant aborder les protocoles de mise en pages associés aux conteneurs. Un protocole définit la méthode de placement des composants d'un conteneur les uns par rapport aux autres.

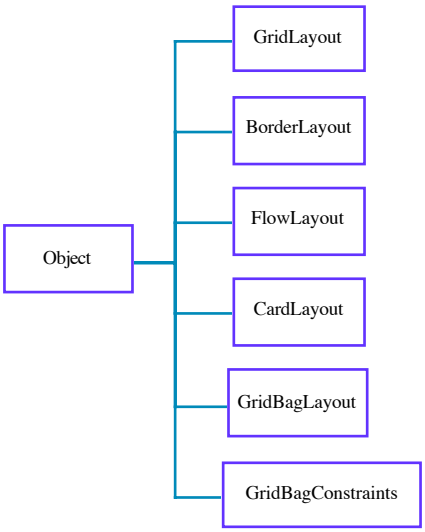


Figure 16.1 Hiérarchie des protocoles de mise en pages

Il existe plusieurs protocoles de mise en pages :

- la mise en pages « glissante » : classe *FlowLayout*;

- la mise en pages par spécification des bords : classe *BorderLayout*;
- la mise en pages avec une grille : classe *GridLayout*;
- la mise en pages sous forme de cartes : classe *CardLayout*;
- la mise en pages avec une quadrillage et des contraintes : classe *GridBagLayout* et *GridBagConstraints*.

Chaque conteneur est associé à un seul protocole de mise en pages. Tous ses composants seront soumis à ce protocole. Par contre, il est possible que parmi les composants, il se trouve des conteneurs qui eux sont soumis à d'autres protocoles de mise en pages.

L'utilisation de protocoles de mise en pages permet de rester indépendant de la plate-forme physique. On spécifie les contraintes de la mise en pages plutôt que des pixels sur la surface de travail.

Tous les protocoles de mise en pages implantent l'interface *LayoutManager*, dont le tableau 16.1 mentionne les principales méthodes.

Méthode	Description
`addLayoutComponent(String n, Component c)`	Ajoute le composant c et l'associe au nom n.
`layoutContainer(Container c)`	Exécute la mise en pages pour le conteneur c.
`minimumLayoutSize(Container p)`	Calcule la taille minimum (Dimension) pour le panneau p pour le parent le contenant.
`preferredLayoutSize(Container)`	Calcule la dimension préférée (Dimension) pour le panneau p pour le parent le contenant.
`removeLayoutComponent(Component c)`	Supprime le composant c.

Tableau 16.1 Méthodes de l'interface LayoutManager

16.1 Mise en pages « glissante »

La mise en pages « glissante » spécifie le protocole suivant : les composants graphiques sont ajoutés l'un après l'autre de gauche à droite, avec un saut à la ligne dès qu'il ne reste plus d'espace suffisant à droite.

Les constantes *FlowLayout.LEFT*, *FlowLayout.CENTER* et *FlowLayout.RIGHT* représentent l'alignement dans une ligne (par défaut, centré).

L'exemple suivant montre l'utilisation de ce protocole :

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

```
public class Composant extends java.applet.Applet {

   public void init() {
      setLayout (new FlowLayout(FlowLayout.LEFT,8,10));
      add(new Button ("b1"));
      add(new Button ("b2"));
      add(new Button ("b3"));
      add(new Button ("b10"));
      add(new Button ("b20"));
      add(new Button ("b30"));
      add(new Button ("b100"));
      add(new Button ("b200"));
      add(new Button ("b300"));
      add(new Button ("b1000"));
      add(new Button ("b2000"));
      add(new Button ("b3000"));
      add(new Label ("ici le centre"));
      add(new TextArea ("entrez vos ...",4,30));
   }
}
```

L'utilisation de protocoles de mise en pages permet d'adapter celle-ci dynamiquement lors du changement de la taille du conteneur. Les applets 46 et 47 illustrent bien ce phénomène.

Applet 46 : Mise en pages « glissante »

Applet 47 : Mise en pages « glissante » s'adaptant à la taille du conteneur

Les méthodes des classes gérant les protocoles de mise en pages ne doivent pas être invoquées directement. Elles sont appelées à travers les conteneurs qui demandent à respecter le protocole. Nous allons nous contenter de décrire les constructeurs, renvoyant le lecteur à la description de l'API pour plus de détails.

Constructeur	Description
`FlowLayout();`	Crée un protocole.
`FlowLayout (int alignement);`	Crée un protocole en spécifiant l'alignement des lignes.
`FlowLayout (int alignement, int xBord, int yBord);`	Crée un protocole en spécifiant l'alignement des lignes et l'espace entre les composants en x et en y.

Tableau 16.2 Constructeurs de la classe FlowLayout

16.2 Mise en pages par spécification des bords

La mise en pages «glissante» laisse trop de degrés de liberté aux composants. Les protocoles que nous allons examiner maintenant augmentent les contraintes entre les composants.

La mise en pages par spécification des bords (classe *BorderLayout*, tableau 16.3) décompose le conteneur en cinq zones *(North, South, East, West, Center)* qui pourront recevoir chacune au plus un composant.

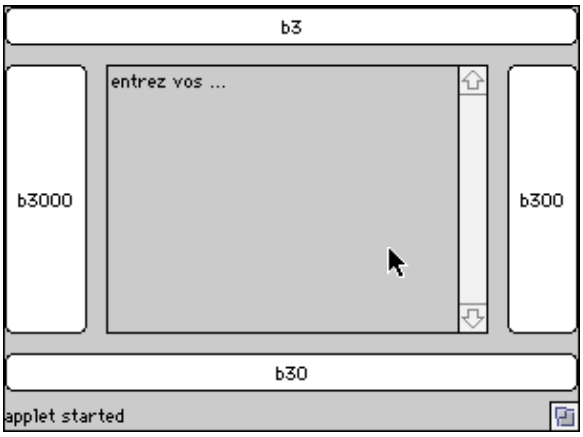
L'exemple suivant montre l'utilisation de ce protocole :

```
import java.awt.*;

public class Composant extends java.applet.Applet {

   public void init() {
      setLayout (new BorderLayout(10,10));
      add("North",new Button ("b3"));
      add("South",new Button ("b30"));
      add("East",new Button ("b300"));
      add("West",new Button ("b3000"));
      add("Center",new TextArea ("entrez vos ...",4,30));
   }
}
```

On peut ainsi observer (applet 48) qu'en modifiant la dimension du conteneur, la dimension des objets peut varier, mais que leur position relative reste par contre inchangée.

Constructeur	Description
`BorderLayout();`	Crée un protocole.
`BorderLayout(int xBord, yBord);`	Crée un protocole en spécifiant l'espace entre les composants en x et en y.

Tableau 16.3 Constructeurs de la classe BorderLayout



Applet 48 : Mise en pages par spécification des bords

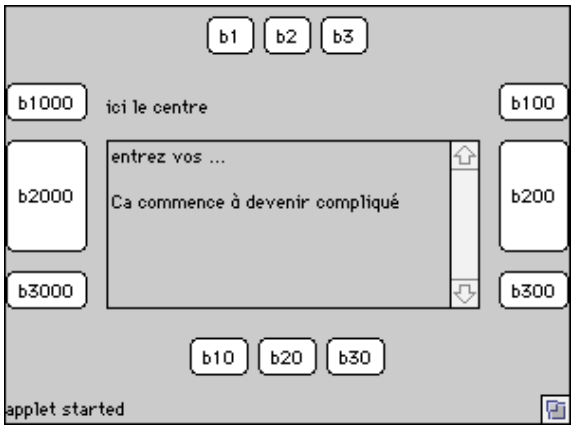
16.3 Récursivité de la mise en pages

Un conteneur contient des composants graphiques, mais un conteneur est lui aussi un composant graphique. Il est donc possible de construire des mises en pages imbriquées, pour lesquelles le panneau *(Panel)* est le mieux adapté.

Dans l'exemple suivant (applet 49), nous avons créé cinq panneaux, placés dans un même panneau. A chacun des cinq panneaux, nous associons un protocole de mise en pages et les composants graphiques qu'il doit contenir.

```
import java.awt.*;

public class Composant extends java.applet.Applet {
   public void init() {
      setLayout (new BorderLayout(10,10));

      Panel panneauAuNord = new Panel();
      add("North",panneauAuNord);
      Panel panneauAuSud = new Panel();
      add("South",panneauAuSud);
      Panel panneauAlEst = new Panel();
      add("East",panneauAlEst);
      Panel panneauAlOuest = new Panel();
      add("West",panneauAlOuest);
      Panel panneauAuCentre = new Panel();
      add("Center",panneauAuCentre);

      panneauAuNord.add(new Button ("b1"));
      panneauAuNord.add(new Button ("b2"));
      panneauAuNord.add(new Button ("b3"));
      panneauAuSud.add(new Button ("b10"));
      panneauAuSud.add(new Button ("b20"));
      panneauAuSud.add(new Button ("b30"));
```

```
        panneauAlEst.setLayout (new BorderLayout(10,10));

        panneauAlEst.add("North",new Button ("b100"));
        panneauAlEst.add("Center",new Button ("b200"));
        panneauAlEst.add("South",new Button ("b300"));

        panneauAlOuest.setLayout (new BorderLayout(10,10));

        panneauAlOuest.add("North",new Button ("b1000"));
        panneauAlOuest.add("Center",new Button ("b2000"));
        panneauAlOuest.add("South",new Button ("b3000"));

        panneauAuCentre.setLayout (new BorderLayout(10,10));

        panneauAuCentre.add("North",new Label ("ici le centre"));
        panneauAuCentre.add("Center",new TextArea
                                    ("entrez vos ...",4,30));
    }
}
```



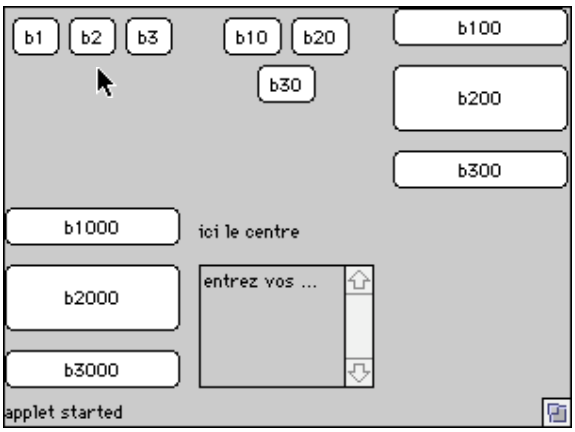
Applet 49 : Mise en pages imbriquée

16.4 Mise en pages avec une grille

Le protocole défini par la classe *GridLayout* spécifie une grille dont chaque cellule peut contenir un composant graphique. Les composants sont ajoutés de gauche à droite, ligne par ligne (voir tableau 16.4). De nouveau, il est possible d'imbriquer les conteneurs comme le montre l'applet 50 ci-dessous :

```
import java.awt.*;

public class Composant extends java.applet.Applet {

   public void init() {
      setLayout (new GridLayout(2,3,10,10));
```

```
        Panel panneau1 = new Panel();
        add(panneau1);
        Panel panneau2 = new Panel();
        add(panneau2);
        Panel panneau3 = new Panel();
        add(panneau3);
        Panel panneau4 = new Panel();
        add(panneau4);
        Panel panneauAuCentre = new Panel();
        add(panneauAuCentre);

        panneau1.add(new Button ("b1"));
        panneau1.add(new Button ("b2"));
        panneau1.add(new Button ("b3"));

        panneau2.add(new Button ("b10"));
        panneau2.add(new Button ("b20"));
        panneau2.add(new Button ("b30"));
        panneau3.setLayout (new BorderLayout(10,10));
        panneau3.add("North",new Button ("b100"));
        panneau3.add("Center",new Button ("b200"));
        panneau3.add("South",new Button ("b300"));

        panneau4.setLayout (new BorderLayout(10,10));
        panneau4.add("North",new Button ("b1000"));
        panneau4.add("Center",new Button ("b2000"));
        panneau4.add("South",new Button ("b3000"));

        panneau5.setLayout (new BorderLayout(10,10));
        panneau5.add("North",new Label ("ici le centre"));
        panneau5.add("Center",new TextArea ("entrez vos ...",4,30));
    }
}
```



Applet 50 : Mise en pages à l'aide d'une grille

Constructeur	Description
`GridLayout (int n, int m);`	Crée un protocole de grille à n lignes et m colonnes.
`GridLayout (int n, int m , int xBord, yBord);`	Crée un protocole de grille à n lignes et m colonnes en spécifiant l'espace entre les composants en x et en y.

Tableau 16.4 Constructeurs de la classe GridLayout

16.5 Mise en pages sous forme de cartes

Le conteneur soumis à ce protocole n'affiche qu'un seul composant à la fois. On utilise généralement des panneaux comme composants, chaque panneau pouvant posséder ses propres composants.

La classe *CardLayout* propose les méthodes *first(), last(), next(), previous()* et *show()* pour naviguer entre les cartes. Celles-ci sont associées à des boutons de navigation généralement affichés sur la carte elle-même. Ce protocole permet de réaliser des interfaces ressemblant à celles d'HyperCard.

Constructeur	Description
`CardLayout();`	Crée un protocole.
`CardLayout(int xBord, yBord);`	Crée un protocole en spécifiant l'espace entre les composants en *x* et en *y*.

Tableau 16.5 Constructeurs de la classe CardLayout

16.6 Mise en pages avec un quadrillage et des contraintes

La mise en pages avec un quadrillage et des contraintes est la plus sophistiquée. Chaque composant graphique est inséré dans le conteneur et on lui associe une contrainte qui décrit son emplacement sur une grille, sa taille en termes de cellules de la grille, son ancrage, la taille des marges, etc.

La contrainte est spécifiée par la classe *GridBagConstraints* qui implante l'interface *Cloneable*. En effet, lors de l'association du composant graphique et de la contrainte, le gestionnaire de mise en pages va effectuer une copie de la contrainte. Celle-ci pourra donc être manipulée ultérieurement.

La contrainte est exprimée par l'initialisation des variables d'instance présentées dans le tableau 16.6 :

Variable d'instance	Définition
`public int gridx`	Position en *x* sur la grille du composant. La constante *RELATIVE* indique un déplacement relatif par rapport au dernier composant.
`public int gridy`	Idem mais pour la position en *y*.
`public int gridwidth`	Indique combien de cellules en *x* sont utilisées sur la grille pour ce composant. La constante *REMAINDER* indique que ce composant est le dernier et qu'il doit occuper l'ensemble des cellules restantes.
`public int gridheight`	Idem mais pour la taille en *y*.
`public double weightx`	Permet de pondérer l'espace disponible entre les différents composants dans la dimension *x*. la valeur *0* indique que le composant ne doit pas participer à cette distribution. Cette variable permet de définir plus finement le comportement lors du redimensionnement d'un conteneur.
`public double weighty`	Idem mais pour la pondération en *y*.
`public int anchor`	Permet d'indiquer à quel coin ou bord de la cellule le composant est attaché s'il est plus petit que la cellule. Les constantes suivantes sont valides : *CENTER*, *EAST*, *NORTH*, *NORTHEAST*, *NORTHWEST*, *SOUTH*, *SOUTHEAST*, *SOUTHWEST*, *WEST*.
`public int fill`	Permet d'indiquer comment remplir la cellule si elle est plus grande que le composant. Les constantes suivantes sont valides : *NONE*, *BOTH*, *HORIZONTAL*, *VERTICAL*.
`public Insets insets`	Spécifie les marges autour d'un composant (haut, bas, gauche, droite).
`public int ipadx`	Permet de définir la taille interne à ajouter en *x* pour chaque composant.
`public int ipady`	Idem en *y*.

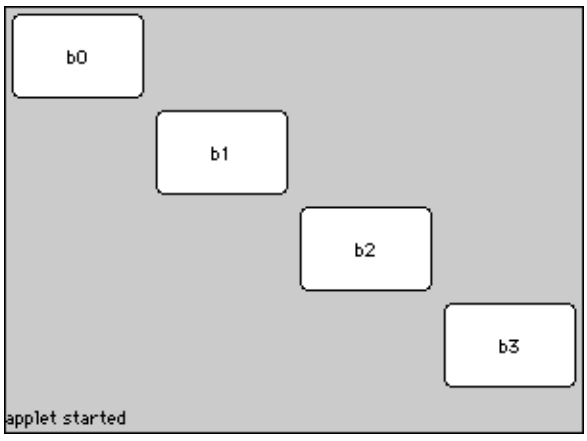
Tableau 16.6 Variables d'instance de la classe GridBagConstraints

Pour faciliter la spécification des contraintes, il est recommandé de construire une méthode représentant une catégorie de contraintes. En effet, le nombre des paramètres peut rapidement rendre la spécification de la contrainte incompréhensible. Dans l'exemple suivant, nous avons défini une méthode *xyPosition()* qui déclare la contrainte, l'associe au composant graphique et finalement ajoute le composant dans le conteneur. Cette méthode n'utilise que les coordonnées

du composant à placer sur la grille, les autres éléments de la contrainte étant définis par défaut dans la méthode.

L'applet 51 ci-dessous place les boutons sur la diagonale de la grille.

```
import java.awt.*;

public class Composant extends java.applet.Applet {
   GridBagLayout g=new GridBagLayout();

   public void init() {
      this.setLayout(g); // le Panel de l'applet

      xyPosition(this,new Button ("b0"),0,0);
      xyPosition(this,new Button ("b1"),1,1);
      xyPosition(this,new Button ("b2"),2,2);
      xyPosition(this,new Button ("b3"),3,3);
   }

   public void xyPosition(Container conteneur, Component element,
                          int x, int y)
   {
      GridBagConstraints c= new GridBagConstraints();
      c.gridx=x; c.gridy=x;
      c.gridwidth=1; c.gridheight=1;
      c.fill=GridBagConstraints.BOTH;
      c.weightx=1;c.weighty=1;
      c.anchor=GridBagConstraints.CENTER;
      c.insets=new Insets(3,3,3,3);
      ((GridBagLayout)conteneur.getLayout()).setConstraints
                                                (element, c);
      conteneur.add(element);
   }
}
```



Applet 51 : Contrainte de mise en pages définie à l'aide d'une méthode

L'utilisation efficace de ce protocole demande donc la création d'un ensemble de méthodes de placement. Sa souplesse en fait pour le programmeur une sorte de générateur de protocoles de mise en pages.

Nous pensons qu'il ne faut pas trop investir dans une connaissance approfondie de ces protocoles car des générateurs graphiques et interactifs d'interfaces devraient rapidement voir le jour. Ces générateurs vous permettront de placer des éléments sur la surface de travail et généreront ensuite le code Java nécessaire.

16.7 Invitation à explorer

Examinez les méthodes de la classe *GridBagLayout*.

Écrivez une interface pour l'applet de dessin (voir point 13.4, p. 169) qui permette de choisir la couleur du trait (rouge, vert, bleu, jaune).

Complétez l'éditeur de texte sans vous préoccuper de la sauvegarde.

Chapitre 17

Manipulation d'images et de sons

Les images constituent une classe d'objets du package *java.awt*. Cependant, les méthodes pour récupérer les images à partir de la définition d'un URL sont spécifiques à la classe *Applet*. La notion de son est actuellement définie et gérée dans la classe *Applet*.

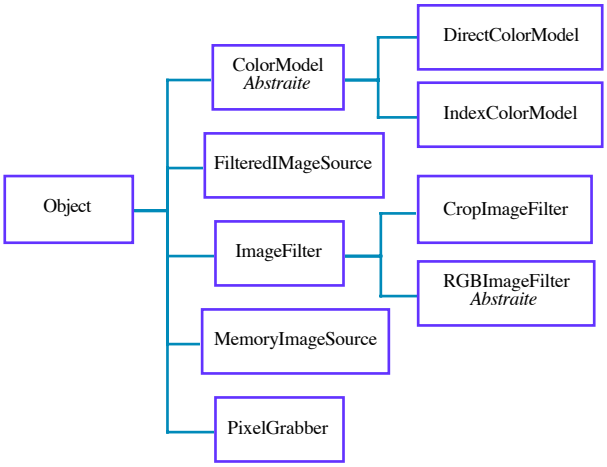


Figure 17.1 Hiérarchie de gestion des images

17.1 Charger des images

Les images sont gérées à l'aide d'objets du type *Image*. Elles peuvent être chargées en utilisant les méthodes indiquées dans le tableau 17.1 :

Méthode	Description
`getImage(URL url)`	Retourne une image se trouvant à l'adresse indiquée par *url*.
`getImage(URL url, String n)`	Retourne une image se trouvant à l'adresse indiquée par *url* en lui concaténant la chaîne *n*.
`getCodeBase()`	Retourne l'*url* de l'applet.

Tableau 17.1 Méthodes de chargement d'une image (classe Applet)

L'image doit correspondre au type d'extension signalé dans l'URL. Les méthodes suivantes de la classe *Graphics* permettent de dessiner une image :

- *drawImage(Image img, int x, int y,ImageObserver o);*
- *drawImage(Image img, int x, int y, Color f, ImageObserver o);*
- *drawImage(Image img, int x, int y, int l, int h, ImageObserver o);*
- *drawImage(Image img,int x,int y,int l,int h,Color f,ImageObserver o).*

Ces méthodes retournent toujours un booléen, en particulier *false* si l'image n'est pas complètement chargée au moment de la demande d'affichage. Dans ce cas, le gestionnaire de l'image (*ImageObserver*) émettra des messages de mise à jour provoquant des affichages répétitifs de l'image incomplète. Ceux-ci risquent alors de pénaliser le chargement, augmentant le temps nécessaire à l'affichage complet.

Les différents paramètres des méthodes d'affichage ont le sens suivant :

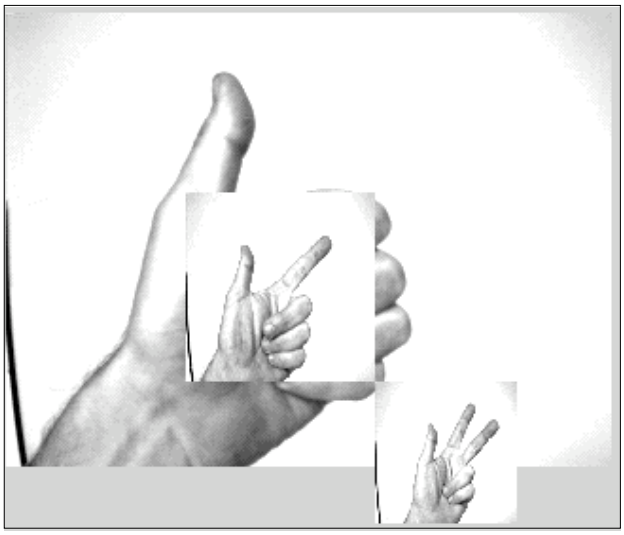
- *img* : image à afficher;
- *x* : coordonnée horizontale de l'image (coin supérieur gauche);
- *y* : coordonnée verticale de l'image (coin supérieur gauche);
- *l* : largeur de l'image en pixels;
- *h* : hauteur de l'image en pixels;
- *f* : couleur du fond;
- *o* : gestionnaire de l'image (celui de l'*applet*).

Pour les méthodes *drawImage()* n'incluant pas les paramètres *l* et *h*, l'image s'affiche en taille réelle. En modifiant les paramètres *l* et *h*, il est possible de créer des effets de « zoom » ou de réduction.

L'applet 52 charge trois images se trouvant dans un répertoire (*Desimages*) situé dans le même dossier que l'applet. On remarquera l'utilisation de la méthode *getCodeBase()* qui permet de rendre relatif le positionnement des fichiers contenant les images. Initialement, les images ont toutes la même taille (320x240 pixels), l'utilisation des paramètres *l* et *h* ayant pour effet de réduire leur taille.

```
import java.awt.Graphics;
```

```
import java.awt.Image;

public class UneImage extends java.applet.Applet {

    Image img1, img2, img3 ;

    public void init(){
        img1=getImage(getCodeBase(),"DesImages/un.gif");
        img2=getImage(getCodeBase(),"DesImages/deux.gif");
        img3=getImage(getCodeBase(),"DesImages/trois.gif");
    }

    public void paint(Graphics g) {
      g.drawImage(img1,5,5,this);
      g.drawImage(img2,100,100,100,100,this);
      g.drawImage(img3,200,200,75,75,this);

      }
}
```



Applet 52 : Chargement de trois images

17.2 Filtrer des images

Sans vouloir donner un cours sur le traitement des images et sur l'animation, on peut signaler quelques points utiles. La classe *MediaTracker* (gestionnaire de médias) permet de synchroniser le chargement des images tandis que le package *java.awt.image* supporte leur traitement. Ce package permet de manipuler les couleurs, d'extraire des zones de l'image, de manipuler les pixels, de construire des filtres, etc.

Nous avons construit un exemple (voir code de l'applet 53 ci-dessous) illustrant

ces possibilités. Cette applet charge une image, lance une activité d'animation qui attend la fin du chargement de l'image, puis affiche la même image mais réduite à 25% et effectue ensuite un «travelling» sur une zone de l'image, de la gauche vers la droite.

Le gestionnaire de médias est utilisé de la manière suivante :

- création de l'objet;
- ajout d'objets (images) à surveiller avec un identifiant;
- test ou mise en attente jusqu'à ce que l'objet soit chargé.

Tout cela est réalisé par le programme suivant :

```
    MediaTracker traceur;
...
    public void init(){
       traceur = new MediaTracker(this);
       this.showStatus("Chargement de l'image");
       img1=getImage(getCodeBase(),"DesImages/imgsmall.gif");
       // img1 est identifié par le numéro 0
       traceur.addImage(img1,0);
...
    public void run() {
       // le processus attend le chargement de id 0)
       try {traceur.waitForID(0); }
          catch(InterruptedException e){}
       // test du chargement
       this.showStatus("Chargement de l'image OK");
       if (traceur.isErrorID(0)) {
          this.showStatus("erreur durant le chargement");
       }
```

Le principe du traitement d'une image (voir figure 17.2) est assez élégant :

- on crée un filtre (*ImageFilter*) qui est à la fois un consommateur et un producteur d'images (deux sous-classes d'*ImageFilter* implantent un traitement de l'image : *CropImageFilter* et *RGBImageFilter)*;
- on crée un producteur d'images (*FilteredImageSource*, qui implante l'interface *ImageProducer*) à qui on fournit l'image source (obtenue à l'aide de la méthode *getSource()*) et le filtre;
- on demande au producteur d'images de créer l'image (*createImage()*).

Chacun de ces éléments se comporte comme un « consommateur-producteur », ce qui permet de les relier (à la manière des *pipes*) afin de produire de véritables chaînes de traitement d'images.

Ces différentes étapes se traduisent de la manière suivante :

```
    Image img1, crop;
    ...
            ImageFilter filtre= new CropImageFilter(i,50,120,120);
            ImageProducer producteur=new
```

```
        FilteredImageSource(img1.getSource(),filtre);
    crop=this.createImage(producteur);
```

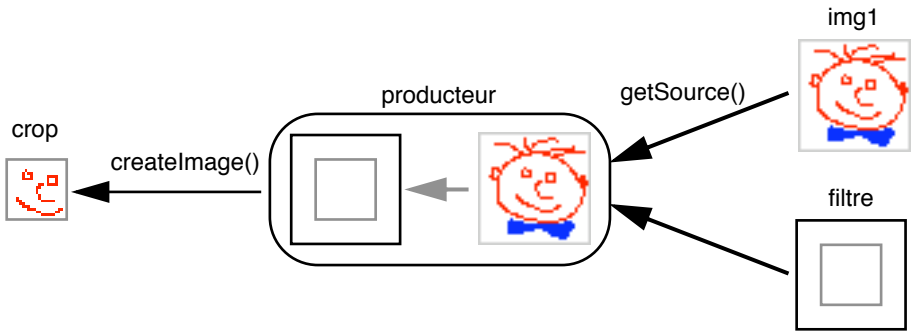
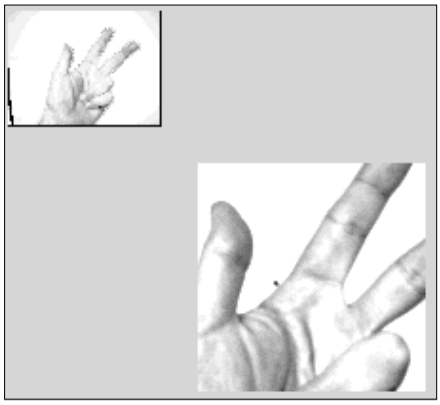


Figure 17.2 Traitement d'une image



Applet 53 : «Travelling» sur une image

Programme complet :

```
import java.awt.Graphics;
import java.awt.Image;
import java.awt.MediaTracker;
import java.awt.image.*;


public class UneImage extends java.applet.Applet implements Runnable
{
   boolean stop=false;
   Image img1, crop;
   MediaTracker traceur;
   Thread actif;

   public void init(){
      traceur = new MediaTracker(this);
      this.showStatus("Chargement de l'image");
```

```
      img1=getImage(getCodeBase(),"DesImages/imgsmall.gif");
      traceur.addImage(img1,0);
   }

   public void start() {
      if (actif==null); {
         actif = new Thread(this);
         actif.start();
      }
   }

   public void stop() {
      if (actif!=null); {
         stop = true;
         actif = null;
      }
   }

   public void run() {
      try {traceur.waitForID(0); }
         catch(InterruptedException e){}
      this.showStatus("Chargement de l'image OK");
      if (traceur.isErrorID(0)) {
         this.showStatus("erreur durant le chargement");
      }
      while (!stop) {
         for (int i=1; i<200;++i){
            ImageFilter filtre= new CropImageFilter(i,50,120,120);
            ImageProducer producteur=new
              FilteredImageSource(img1.getSource(),filtre);
            crop=this.createImage(producteur);
            repaint();
            try {Thread.sleep(500);}
               catch(InterruptedException signal) {}
         }
      }
   }

   public void paint(Graphics g) {
      g.drawImage(img1,0,0,img1.getWidth(this)/
4,img1.getHeight(this)/4,this);
      g.drawRect(0,0,img1.getWidth(this)/4,img1.getHeight(this)/4);
      g.drawRect(0,0,img1.getWidth(this)/4,img1.getHeight(this)/4);
   }

   public void update(Graphics g) {
      g.drawImage(crop,20+img1.getWidth(this)/4,
                  20+img1.getHeight(this)/4,120,120,this);
      paint(g);
   }
}
```

17.3 Animation avec double tampon

Nos animations étaient jusqu'ici relativement simples (l'horloge, par exemple) et l'effet de scintillement dû à la régénération de l'image n'était pas trop perceptible. Dans le cas où la régénération de l'image est plus longue, l'effet de scintillement peu devenir très gênant.

Revenons sur le déroulement exact de la régénération :

- le processus d'animation invoque *repaint();*
- la méthode *repaint()* invoque *update()*;
- *update()* efface le composant graphique, peint le composant avec la couleur de fond, initialise la couleur de dessin et invoque la méthode *paint()* afin qu'elle redessine complètement le composant graphique.

Le scintillement provient donc du délai de latence s'écoulant entre l'effacement complet de la surface graphique de l'applet et sa reconstruction. Les méthodes d'animation cherchant à diminuer le scintillement (et à améliorer la performance de l'animation) tenteront donc d'éviter d'effacer complètement l'applet et de ne redessiner que les parties qui sont modifiées, diminuant ainsi le délai de latence.

Afin d'éviter de devoir effacer complètement la surface de l'applet, il suffit de redéfinir la méthode *update()*, dont le code « classique » est :

```
public void update(Graphics g) {
   paint(g);
}
```

La méthode *update()* « intelligente » se chargera alors de minimiser l'effacement. Pour ne redessiner qu'une partie de l'applet, on peut utiliser la méthode *clipRect()* qui sélectionne une zone rectangulaire, puis invoquer *paint()* qui ne redessinera alors que cette zone. Les coordonnées de la zone à redessiner sont mises à jour par le processus de l'animation. Exemple :

```
public void update(Graphics g) {
   g.clipRect(x1,y1,x2,y2)
   paint(g);
}
```

Pour diminuer le délai de latence, on peut également utiliser une méthode dite « à double tampon ». Elle consiste à préparer la nouvelle image dans un tampon (zone en mémoire) sans l'afficher, puis à afficher l'image lorsqu'elle est prête. Pour réaliser ce principe en Java, on procédera de la manière suivante.

On déclare une image et une surface de dessin (*Graphics*) :

```
Image imgTampon;
Graphics gTampon;
```

On crée une image de la taille de l'applet et on lie l'image à la surface de dessin :

```
imgTampon=createImage(getSize().width,getSize().height);
gTampon=imgTampon.getGraphics();
```

On prépare ensuite une image à l'aide des méthodes de *Graphics* appliquées au tampon (*gTampon*). Lorsque l'image est prête, on invoque *repaint()*.

Dans la méthode *paint()*, on dessine l'image préparée :

```
g.drawImage(imgTampon,0,0,this);
```

Pour des cas plus complexes, on imagine bien qu'il est nécessaire de combiner l'ensemble de ces principes. L'applet 54 utilise le double tampon pour dessiner un disque qui s'insère dans un carré.

```
import java.awt.Graphics;
import java.awt.Image;
import java.awt.image.*;

public class UneImage extends java.applet.Applet implements Runnable
{

   boolean stop=false;
   Image imgTampon;
   Graphics gTampon;
   Thread actif;

   public void init(){
      imgTampon=createImage(getSize().width,getSize().height);
      gTampon=imgTampon.getGraphics();
   }

public void start() {
      if (actif==null); {
         actif = new Thread(this);
         actif.start();
      }
   }
   public void stop() {
      if (actif!=null); {
         stop=true;
         actif = null;
      }
   }
   public void run() {
      gTampon.clearRect(0,0,getSize().width,getSize().height);
      while (!stop) {
         for (int i=1; i<100;++i){
            gTampon.clearRect(100,100,i,i);
            gTampon.fillOval(100,100,i,i);
            repaint();
            try {Thread.sleep(100);}
               catch(InterruptedException signal) {}
         }
```

```
        }
    }
    public void paint(Graphics g) {
        g.drawImage(imgTampon,0,0,this);
    }
}
```

Applet 54 : Affichage à l'aide d'un double tampon

17.4 Ajouter le son

Actuellement, le son est un peu le parent pauvre de Java. Il n'existe que le strict minimum pour gérer des données sonores : les séquences doivent être stockées dans des fichiers dont l'extension est *.au* (format μ-law). Ce format est très compressé, ce qui diminue le temps de transfert et l'espace de stockage, au détriment de la qualité du son.

Supposons que nous ayons un fichier sonore *dingdong.au* dans un répertoire *audio*. Si nous ne devons jouer qu'une seule fois le son, le plus simple est d'invoquer la méthode *play()* de la classe *Applet* (voir tableau 17.2) :

```
play(getCodeBase(),"audio/dingdong.au");
```

Le son est chargé en mémoire (au plus vite) puis joué. À noter que le son est joué en parallèle à l'exécution des autres processus. Ainsi, plusieurs sons peuvent être joués simultanément.

Méthode	Description
`play()`	Joue le son. Si le son n'est pas terminé et qu'on le rejoue, la séquence sonore redémarre au début.
`loop()`	Joue le son et le répète indéfiniment.
`stop()`	Arrête de jouer le son. Doit impérativement être invoquée si l'applet se termine, car le son ne dépend pas directement du processus de l'applet.

Tableau 17.2 Méthodes de la classe Applet pour jouer un son

Si le son doit être joué plusieurs fois, il est intéressant de le stocker dans une variable de type *AudioClip*. Le son est alors chargé en utilisant la méthode *getAu-*

dioClip() de l'interface *AudioClip*. Cette interface définit les méthodes de gestion des sons (voir tableau 17.3).

Dans le code ci-dessous, nous avons repris l'applet précédente et nous lui avons ajouté les instructions nécessaires pour jouer le son *dingdong* à chaque fois que l'animation recommence son cycle (vos voisins apprécieront!).

```
public class UneImage extends Applet implements Runnable {
   ...
   AudioClip dingdong;

   public void init(){
   ...
      dingdong=getAudioClip(getCodeBase(),"audio/dingdong.au");
   }
   ...
      public void stop() {
      ...
      dingdong.stop();
   }

   public void run() {
      while (!stop) {
         if (dingdong!=null) dingdong.play();
            for (int i=1; i<100;i+=3){
            ...
   }
   ...
}
```

Méthode	Description
`play(URL url)`	Joue un son se trouvant à l'adresse indiquée par *url*.
`play(URL url, String n)`	Joue un son se trouvant à l'adresse indiquée par la concaténation de *url* (nom de dossier) et *n* (nom de fichier).
`getAudioClip(URL url)`	Charge un son se trouvant à l'adresse indiquée par *url*.
`getAudioClip(URL url, String n)`	Charge un son se trouvant à l'adresse indiquée par la concaténation de *url* et *n*.

Tableau 17.3 Méthodes de l'interface AudioClip

17.5 Invitation à explorer

Reprenez l'applet du travelling et intégrez la capture des événements de la souris sur l'image réduite de façon à diriger l'image affichée (une sorte de zoom).

Utilisez le son pour signaler le choc de deux boules rebondissant dans un cadre.

Chapitre 18

Les composants Swing

Comme son nom l'indique, l'*Abstract Window Toolkit* (AWT) a été conçu comme un système abstrait de gestion de l'interface utilisateur, donc indépendant de la plate-forme d'exécution. Si cette indépendance est effectivement réalisée au niveau de la programmation, elle n'est pas complète au niveau de l'apparence et du comportement (*look-and-feel*) de l'interface. En effet, les composants de l'AWT tels que les boutons, menus, libellés, etc., sont implantés par des objets de l'environnement d'exécution. Ce qui signifie, par exemple, qu'un objet *Button* aura l'apparence et le comportement d'un bouton Windows sous Windows, d'un bouton MacOS sous MacOS, etc.

18.1 Les limites de l'AWT et Swing

Swing a pour objectif d'offrir des composants d'interfaces totalement indépendants de la plate-forme d'exécution car eux-mêmes écrits en Java. Ces composants, appelés composants légers, ont la même apparence et le même comportement sur toutes les plate-formes. De plus, il est possible de changer dynamiquement l'apparence et le comportement de tous les composants d'une interface par la technique dite du *plugable look-and-fee*l. Ainsi, une application fonctionnant sous Solaris pourrait prendre au choix un look-and-feel Java standard, Solaris, Mac, ou autre.

Comme tous les composants sont écrits en Java, toute amélioration ou adjonction de nouveaux composants sera automatiquement disponible sur toutes les plates-formes. Swing est donc ouvert et de nouveaux composants pourront s'y adjoindre. Ce qui n'est pas le cas de l'AWT car celui-ci dépend de l'existence d'une implantation pour chaque composant sur chaque plate-forme. Swing propose, par exemple, un composant *JTable* qui permet de présenter des informa-

tions sous forme de table à deux dimensions (lignes/colonnes). Un tel composant n'existe en général pas tel quel sur les plates-formes habituelles.

18.2 Présentation de Swing

Tous les composants Swing ne sont pas légers car il faut bien en fin de compte communiquer avec l'interface de fenêtrage de la plate-forme sur laquelle s'exécute l'application. Il y a donc quatre composants « lourds », qui sont les conteneurs de base *JFrame*, *JDialog*, *JWindow* et *JApplet*. Hormis ceux-ci, tous les autres composants sont des descendants de la classe *JComponent*.

Les conteneurs de base

L'une des principales différences de la programmation avec Swing par rapport à l'AWT vient du fait que les conteneurs de base *JFrame*, *JDialog*, *JWindow* et *JApplet* contiennent plusieurs surfaces d'affichage (*panes*) :

- la *contentPane* contient tous les composants habituels de l'interface (boutons, champs de texte, etc.);
- la *glassPane* est une surface « transparente » au-dessus des autres surfaces, qui permet d'afficher temporairement des informations (par exemple, des aides contextuelles), par défaut la *glassPane* n'est pas visible;
- la *menuBar* est une surface réservée à une barre de menu, qui prend place en haut de la totale visible;
- la *layeredPane* est un ensemble de surfaces superposées, qui contient au départ la *contentPane* et la *menuBar* mais à laquellec on peut ajouter d'autres surfaces (par exemple, pour créer ses propres menus pop-up).

L'organisation des surfaces est illustrée sur la figure 18.1.

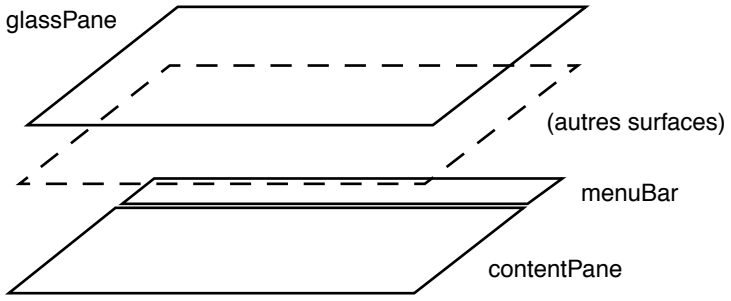


Figure 18.1 Organisation des surfaces d'affichage d'un conteneur de base

La présence de ces différentes surfaces ne permet plus l'ajout simple et direct de composants à un conteneur de base. Par exemple, pour ajouter une étiquette

(*JLabel*) *e* dans un *JFrame f* on ne fait plus :

```
f.add(e)
```

mais :

```
f.getContentPane().add(e) // on ajoute e sur la surface de contenu.
```

Les JComponents

Tous les composants Swing sont des descendants de *JComponent*. Il faut remarquer que *JComponent* est une sous-classe de *Container,* ce qui signifie que tout composant Swing peut contenir d'autres composants, contrairement aux composants AWT. La classe *JComponent* est pour le moins complexe puisqu'elle possède 105 méthodes propres, en hérite 39 de j*ava.awt.Container* et 106 de j*ava.awt.Component*! Pour y voir un peu plus clair, on peut regrouper ces méthodes selon quelques grandes fonctions.

Look-and-feel : associer au composant un objet responsable de son apparence et de son comportement;

Dessin : dessiner et redessiner le composant, sa bordure et ses sous-composants;

Focus : déterminer quand le composant devient actif ou inactif par rapport aux entrées du clavier;

Taille et position : définir et contrôler la taille et le positionnement du composant (taille minimum, maximum, préférée, etc.);

Clavier : déterminer les réactions aux frappes clavier, les raccourcis clavier, etc.;

Aide immédiate : fixer le texte d'aide à afficher quand la souris passe sur le composant;

Accessibilité : déterminer la manière dont le composant doit apparaître à des personnes souffrant de handicaps (affichage de très grosse taille, remplacement de l'image par le son, le braille, etc.).

18.3 Composants Swing et AWT

Pour faciliter la conversion aisée des programmes basés sur AWT vers Swing, la plupart des composants AWT ont un équivalent Swing offrant les fonctions supplémentaires liées à Swing. Le tableau suivant donne ces correspondances en indiquant les principales différences.

AWT	Swing	Commentaires
Button	*AbstractButton* *JButton*	On peut associer une icône plutôt qu'un texte à un *JButton.*

Tableau 18.1 Différences principales en AWT et Swing

AWT	Swing	Commentaires
Canvas		Pas d'équivalent.
Checkbox	*JToggleButton* *JCheckBox* *JRadioButton*	On fait la distinction entre les *JCheckBox* destinées aux sélections multiples et les *JRadioButton* pour les sélections uniques dans un groupe.
CheckboxGroup	*ButtonGroup*	Un groupe de tout ce qui est bouton (checkbox, boutons, etc.).
CheckboxMenuItem	*JCheckboxMenuItem*	Un article de menu qui peut porter une marque de sélection.
Choice	*JPopupMenu* *JComboBox*	Le *JComboBox* permet non seulement de choisir dans une liste mais aussi de saisir un texte dans un champ.
Component	*JComponent*	1. Un *JComponent* comprend un look-and-feel modifiable, des aides immédiates, des composants d'accessibilité, etc. (voir ci-dessus). 2. Un *JComponent* est un conteneur, tout composant Swing peut donc en contenir d'autres.
Dialog	*JDialog*	*JDialog* est un conteneur de base qui contient un *RootPane*.
FileDialog	*JFileChooser*	Ajout de nombreuses méthodes de configuration et de contrôle du dialogue de choix d'un fichier.
Frame	*JFrame*	Extension de *Frame* qui permet la gestion de surfaces de dessin et d'interactions superposées (*glassPane*, *LayeredPane*, *rootPane*) et des barres de menus.
Label	*JLabel*	Peut également contenir une image.
List	*JList*	Une *JList* n'a pas de barre de défilement, il faut l'inclure dans le *viewPort* d'un *JScrollPane*.
Menu	*JMenu*	Un *JMenu* est essentiellement un bouton associé à un *JPopupMenu*.
MenuBar	*JMenuBar*	Une barre formée de *JMenus*.
MenuItem	*JMenuItem*	Un *JMenuItem* est en fait un bouton placé dans une liste. Il peut contenir un texte, une icône ou les deux.

Tableau 18.1 Différences principales en AWT et Swing

AWT	Swing	Commentaires
MenuShortcut		Les équivalents clavier sont gérés globalement pour tous les composants Swing.
Panel	*JPanel*	Le composant qui sert à grouper d'autres composants.
PopupMenu	*JPopupMenu*	Dans AWT le menu pop-up est une sorte de menu, dans Swing c'est le menu qui est composé d'un bouton et d'un menu pop-up.
Scrollbar	*JScrollBar*	En général associé à un *JViewport* qui montre une partie d'un texte ou d'une image.
ScrollPane	*JScrollPane*	Un *JSrollPane* est composé de barres de défilement et d'un objet *JViewport* qui gère l'affichage d'un objet.
TextArea	*JTextArea*	Le *JTextArea* ne possède pas de barre de défilement mais il implémente l'interface *Scrollable*, ce qui permet de le placer dans un *JScrollPane*.
TextComponent	*JTextComponent*	Le contenu (modèle) d'un *JTextComponent* n'est pas un simple *String* mais un *Document*.
TextField	*JTextField* *JPasswordField*	La gestion des champs de type mots de passe doit se faire dans un *JPasswordField*.
Window	*JWindow*	Une fenêtre simple, sans titre ni bouton de manipulation.

Tableau 18.1 Différences principales en AWT et Swing

Swing propose également de nouveaux composants. Ils sont décrits dans le tableau suivant.

Classe	Description
Box	Un conteneur qui utilise la mise en pages *Box*.
JColorChooser	Un panneau pour sélectionner une couleur.
JDesktopPane	Un conteneur pour créer des bureaux (*desktop*) virtuels contenant plusieurs fenêtres (*InternalFrames*).
JEditorPane	Composant texte qui permet d'éditer différents types de contenus : texte simple, HTML, RTF, etc.
JInternalFrame	Comme un *JFrame* mais léger et fait pour vivre à l'intérieur d'un *DesktopPane*.

Tableau 18.2 Nouveaux composants introduits dans Swing

Classe	Description
JLayeredPane	Panneaux superposés.
JOptionPane	Panneau d'information ou d'erreur avec boutons (OK, Cancel, etc.).
JProgressBar	Un composant montrant l'évolution d'un processus sous forme d'une barre qui grandit.
JRootPane	Le composant de départ dans une hiérarchie de composants. Contient d'autres panneaux : contenu, transparent, barre de menu, panneaux superposés.
JSlider	Un règle graduée avec un curseur pour choisir une valeur.
JSplitPane	Juxtaposition de deux composants gauche/droite ou haut/bas, avec possibilité de déplacer la limite.
JTabbedPane	Panneaux superposés avec des onglets à sélectionner au choix.
JTable	Présentation d'information sous forme de table à deux dimensions (lignes/colonnes).
JTextPane	Présentation de texte contenant des indications de style.
JToolBar	Barre d'outils.
JToolTip	Contient un texte d'aide associé à un composant qui apparaît et disparaît en fonction des mouvements de la souris.
JTree	Présentation d'informations arborescentes sous forme de listes imbriquées (exemple : une arborescence de répertoires).
JViewport	La partie visible d'un composant se trouvant dans un *JScrollPane*.

Tableau 18.2 Nouveaux composants introduits dans Swing

On notera que tous ces composants correspondent à ceux que l'on a l'habitude de trouver dans les interfaces de type X-Windows, Windows et MacOS. Swing ne propose à ce jour aucun composant sortant de l'ordinaire.

18.4 L'architecture modèle-vue-contrôleur et Swing

L'architecture modèle-vue-contrôleur (MVC) sert à concevoir des interfaces graphiques modulaires en séparant clairement les trois composantes d'un élément d'interface.

Le **modèle** est une représentation de l'état d'un objet par un ensemble de données (exemple : l'état d'un bouton peut être représenté avec une variable booléenne (enfoncé = oui/non); l'état d'un champ de texte est plus complexe

puisqu'il comprend la liste de caractères formant le texte à afficher, plus d'éventuelles indications de formatage).

La **vue** est la manière dont l'état de l'objet (défini par le modèle) est représenté sur l'écran (ou sur un autre dispositif). Par exemple, un bouton peut être représenté comme un rectangle avec un texte à l'intérieur, lorsque le bouton est enfoncé, on dessine une « ombre » sous le rectangle.

Le **contrôleur** définit la réaction de l'objet aux actions de l'utilisateur. Que se passe-t-il si on clique sur le bouton, si on double-clique, si on tape sur une touche du clavier, etc.?

Dans Swing, l'architecture MVC est utilisée pour rendre le système d'interface très ouvert et reconfigurable dynamiquement. Chaque objet d'interface Swing est associé à deux objets : un objet modèle, qui stocke l'état du composant et un objet d'interface utilisateur (appelé délégué), qui gère à la fois la vue (*look*) et le contrôle (*feel*) du composant. Dans le cas d'un simple bouton, l'objet modèle représente simplement l'état (enfoncé/non enfoncé et le nom du bouton); l'objet d'interface dessine le bouton en fonction de son état et interprète les clics souris sur le dessin du bouton pour modifier l'état (le modèle) en conséquence.

18.5 Le look-and-feel modifiable

Étant donné l'architecture présentée ci-dessus, pour changer le look-and-feel d'un composant il suffit de lui associer un autre objet d'interface. Pour changer le look-and-feel de toute une application, il faut changer le look-and-feel de chaque composant comme illustré sur la figure 18.2.

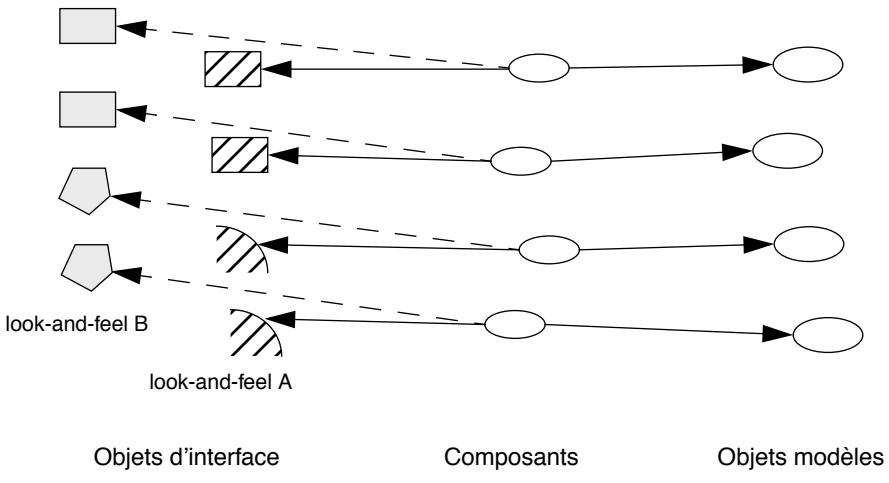


Figure 18.2 Associations entre les composants et leurs objets d'interface

La définition d'un nouveau look-and-feel complet est une tâche d'envergure puisqu'il faut redéfinir toutes les classes d'interface utilisateur de tous les types de composants (*ButtonUI*, *ColorChooserUI*, ..., *TreeUI*, *ViewPortUI*). Heureusement, il existe déjà un certain nombre de look-and-feel prêts à l'emploi : *Metal*, *Windows*, *Motif*, *MacOS*.

Pour utiliser un look-and-feel particulier, ici *Metal*, il faut écrire les instructions suivantes :

```
UIManager.setLookAndFeel(
   "javax.swing.plaf.metal.MetalLookAndFeel");
SwingUtilities.updateComponentTreeUI(topComponent);
```

L'exemple qui suit montre :

- la création d'une interface avec des composants Swing (curseur, panneau, libellés, boutons);
- la création et l'ajout de bords à certains composants;
- l'utilisation d'un nouveau gestionnaire de mise en pages : *BoxLayout*;
- l'obtention de la liste des look-and-feel disponibles;
- le changement dynamique de look-and-feel (chaque fois qu'on clique sur le bouton).

```
import java.awt.*; import java.awt.event.*;
import javax.swing.event.*; import javax.swing.border.*;
import javax.swing.*;

class SliderF extends JFrame implements ChangeListener,
ActionListener {

   JLabel l1; JSlider s; JPanel p, pi;
   DisquePanel dp; JButton changeLAF;

   int diametre = 50;
   Color clr = Color.red;

   UIManager.LookAndFeelInfo laf[];
   int lafNo = 0;

   // un panneau avec un disque de taille variable au centre

   class DisquePanel extends JPanel {
      DisquePanel() {
         setPreferredSize(new Dimension(130,130));
      }
      public void paint(Graphics g) {
         g.setColor(clr);
         g.fillOval(65-diametre/2,65-diametre/2,diametre,
diametre);
```

```
    }
}

public void init() {

    // liste des LAF disponibles
    laf = UIManager.getInstalledLookAndFeels();

    // on crée l'interface dans le contentPane
    getContentPane().add(p = new JPanel());

    BoxLayout bxl = new BoxLayout(p, BoxLayout.Y_AXIS);
    p.setLayout(bxl);

    p.add(l1 = new JLabel("LAF: " +
       UIManager.getLookAndFeel().getName() + " (initial)"));
    l1.setBorder(BorderFactory
       .createMatteBorder(5, 8, 5, 8, Color.orange));

    // un curseur avec divisions et subdivisions
    p.add(s = new JSlider(0,120,40));
    s.setMajorTickSpacing(30);
    s.setMinorTickSpacing(5);
    s.setPaintTicks(true);
    s.addChangeListener(this);

    p.add(pi = new JPanel());
    pi.add(dp = new DisquePanel(), BorderLayout.CENTER);
    pi.setBorder(BorderFactory.createEtchedBorder());

    p.add(changeLAF = new JButton("Change"));
    changeLAF.addActionListener(this);

    setSize(300,250);
    setVisible(true);
}

public void stateChanged(ChangeEvent e) {
    // lit la valeur du curseur
    if (e.getSource() == s) diametre = s.getValue();
    repaint();
}
public void actionPerformed(ActionEvent e) {
    // change de LAF - on prend le suivant dans la liste
    lafNo = (lafNo + 1) % laf.length;
    try {
       UIManager.setLookAndFeel(laf[lafNo].getClassName());
       SwingUtilities.updateComponentTreeUI(this);
    }
    catch (Exception exc) {
    }
    l1.setText("LAF: " + UIManager.getLookAndFeel().getName()
```

```
            + "(" + lafNo + ")");
         repaint();
      }
   }

   // la classe de l'application
   public class SF {

      public static void main(String[] sss) {
         SliderF sf = new SliderF();
         sf.init();
      }
   }
```

Application 1 : Composants Swing et look-and-feel modifiable

La figure 18.4 montre deux copies d'écran effectuées pendant l'exécution de cette application sous MacOS. On remarquera que la fenêtre englobante (*JFrame*) est gérée par la plate-forme d'exécution (MacOS), alors qu'à l'intérieur Swing gère aussi bien un look-and-feel *Metal* qu'un look-and-feel *CDE/Motif*.

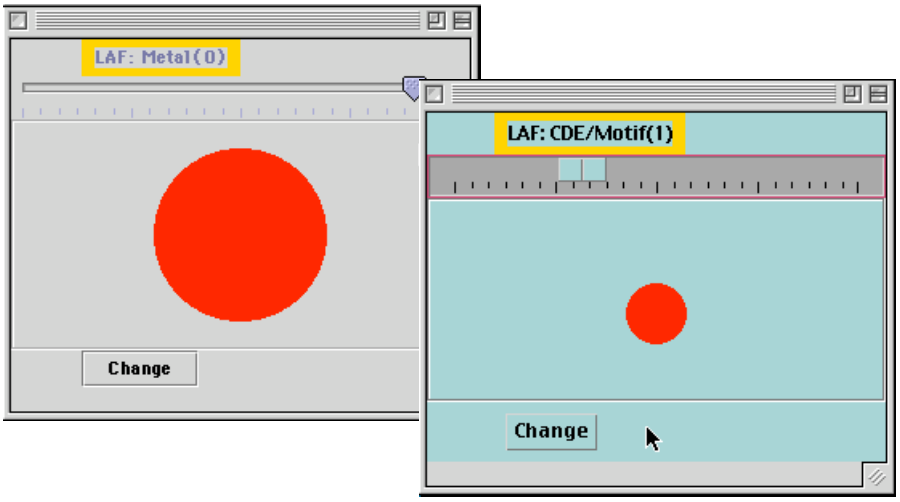


Figure 18.4 Exécution

Pour terminer, signalons qu'il vaut mieux éviter de mélanger les composants AWT et Swing car il peut en résulter des conflits d'affichage. Il faut également être prudent dans la redéfinition de la méthode *paint()* qui risque d'empêcher le dessin d'éventuels sous-composants.

Chapitre 19

Entrées-sorties et persistance des objets

Ce chapitre présente le package *java.io* gérant les entrées-sorties sur les périphériques clavier, écran et unités de mémoire principales et secondaires. Ce package contient également des classes permettant la sérialisation des objets et par conséquent leur persistance. Pour étudier ce package, nous allons considérer successivement les classes manipulant des flux de caractères, puis celles qui manipulent des flux d'octets, et enfin celles qui manipulent des flux d'objets et par conséquent rendent possible la persistance des objets.

Après un examen des principales classes, interfaces et exceptions du package, nous décrirons les entrées-sorties clavier/écran.

Nous construirons ensuite six classes qui seront réutilisées dans les autres exemples de ce chapitre. Ces classes sont : *Fichier*, *FichierLecture*, *FichierEcriture*, *Repertoire*, *RepertoireLecture* et *RepertoireEcriture*.

Les classes du package seront ensuite présentées à l'aide d'exemples (moins animés que ceux du chapitre précédent).

19.1 Description du package

Les classes du package *io* sont presque toutes des sous-classes de la classe *Object*. La plupart d'entre elles sont illustrées dans les exemples du chapitre. Le concept principal sur lequel reposent ces classes est celui de flux. Un flux est un canal dans lequel on lit et/ou écrit des caractères (16 bits) ou des octets (8 bits). Les flux utilisant des octets sont utilisés pour tout ce qui est image ou son. Tou-

tes les entrées-sorties utilisent donc la notion de flux pour la circulation des informations.

Classes

Les classes permettant la lecture et l'écriture de caractères dans des flux sont présentées respectivement sur les figures 19.1 et 19.2. Elles héritent des classes abstraites *Reader* ou *Writer* dont elles utilisent le nom comme suffixe. La plupart de ces classes ont leur équivalent pour les octets (voir figures 19.3 et 19.4), elles héritent alors de la classe abstraite servant de suffixe à leur nom, *InputStream* ou *OutputStream*.

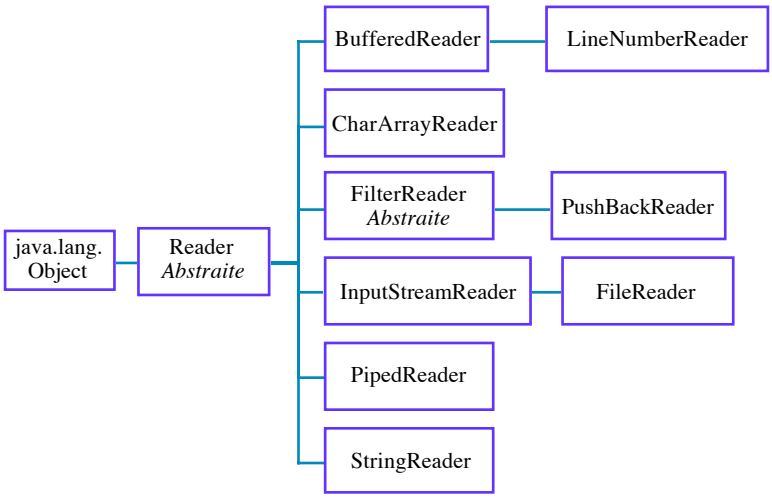


Figure 19.1 Classes pour la lecture de caractères dans des flux

Les classes *BufferedReader* et *BufferedWriter* (*BufferedInputStream* et *BufferedOutputStream* pour les octets) lisent ou écrivent sur un canal en plaçant les caractères (ou les octets) dans un tampon (*buffer*), ce qui diminue le nombre d'accès au flux. Ces opérations sont évidemment beaucoup plus performantes que celles impliquant des flux sans tampon.

Les classes *LineNumberReader* et *LineNumberInputStream* conservent le numéro de ligne courant durant toute la lecture du flux.

Les classes *CharArrayReader*, *CharArrayWriter*, *ByteArrayReader* et *ByteArrayWriter* lisent ou écrivent des tableaux de données en mémoire principale.

Les classes abstraites *FilterReader*, *FilterWriter*, *FilterInputStream* et *FilterOutputStream* effectuent des lectures et des écritures filtrantes. Les classes filtrantes *PushBackReader* et *PushBackInputStream* sont des sous-classes de leur « lecteur » respectif, elles autorisent des retours en arrière sur des données déjà lues, la longueur du retour en arrière est fixée à la création de l'instance.

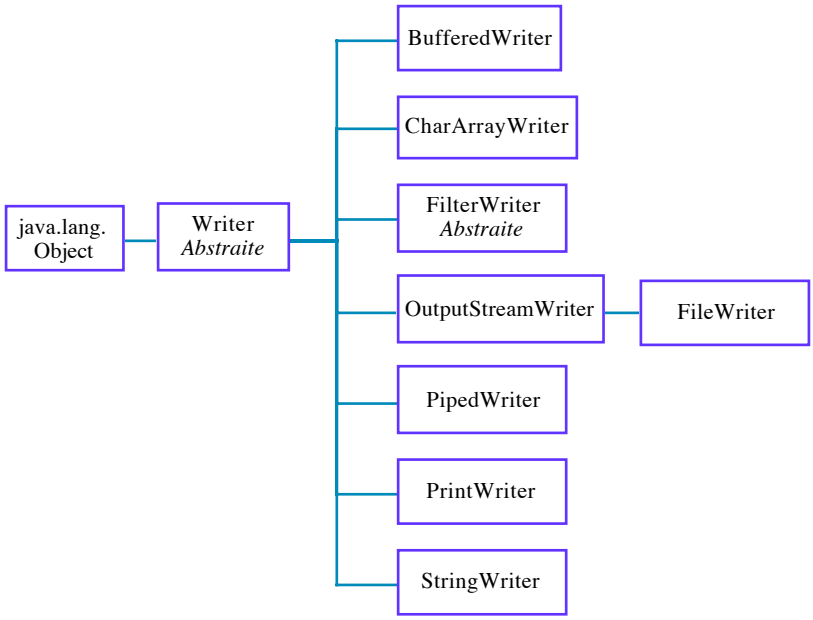


Figure 19.2 Classes pour l'écriture de caractères dans des flux

Les classes *InputStreamReader* et *OutputStreamWriter* convertissent respectivement les octets en caractères et les caractères en octets, et ce, en fonction d'un encodage spécifié à la création de l'instance.

Les classes *FileReader*, *FileWriter*, *FileInputStream* et *FileOutputStream* lisent ou écrivent des fichiers de données en mémoire secondaire. On accède aux fichiers directement à partir de leur nom.

Les classes *PipedReader*, *PipedWriter*, *PipedInputStream* et *PipedOutputStream* lisent ou écrivent des données dans des pipelines (*pipes*) qui permettent, par exemple, la communication entre processus.

La classe *StringReader* lit les caractères d'un String. La classe *StringWriter* écrit dans un *StringBuffer* qui peut être converti en *String*. La classe *StringBufferInputStream*, qui est dépréciée en Java 2.0, convertit de manière incorrecte en octets des caractères lus à partir d'un *String*.

Les classes *PrintWriter* et *PrintStream* sont des classes qui facilitent l'écriture de différents types de données dans des flux de caractères. Elles sont par exemple très utiles pour imprimer la valeur d'un réel ou d'un entier.

Les classes *DataInputStream* et *DataOutputStream* lisent et écrivent des types de données prédéfinis dans Java, indépendamment du système d'exploitation.

La classe *SequenceInputStream* concatène logiquement plusieurs fichiers d'entrée en un seul, la lecture s'enchaînant d'un fichier à l'autre de manière transparente.

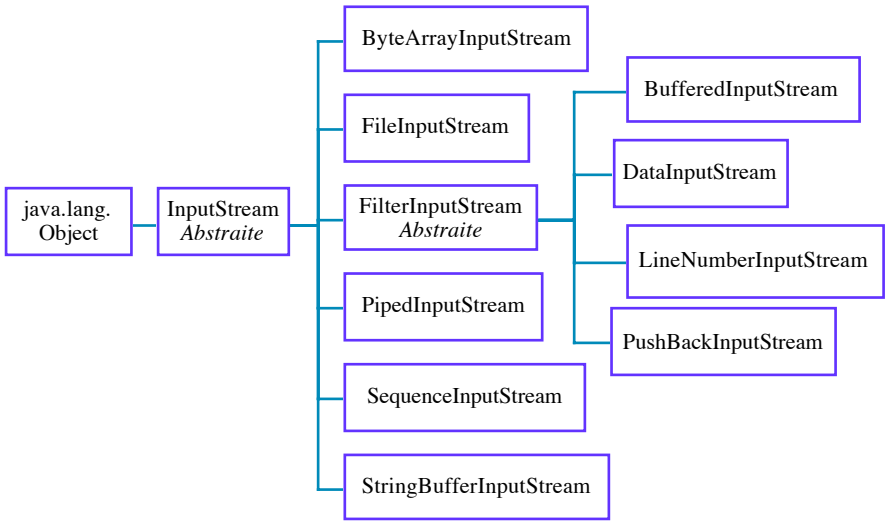


Figure 19.3 Classes pour la lecture d'octets dans des flux

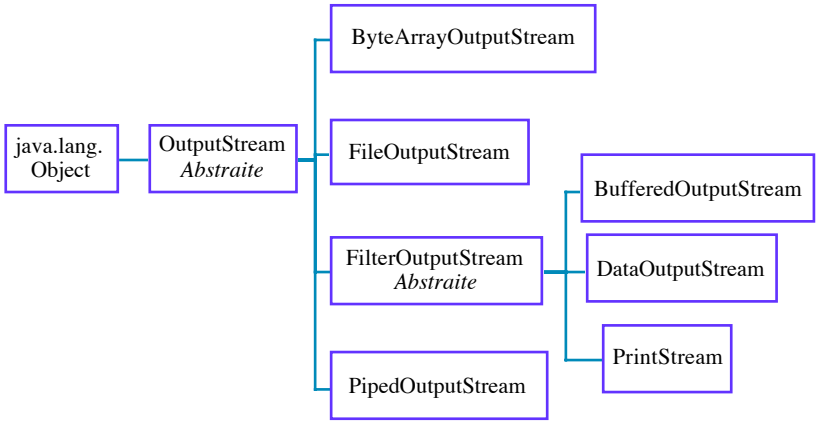


Figure 19.4 Classes pour l'écriture d'octets dans des flux

Outre la persistance et les flux d'entrées-sorties, le package *java.io* contient quelques autres classes utiles illustrées figure 19.5.

La classe *File* permet de construire des noms de fichiers et de répertoires. Elle permet ainsi de représenter les fichiers du système d'exploitation sous-jacent. La classe *FileDescriptor* permet d'obtenir des poignées (*handles*) de bas niveau sur des fichiers ou des sockets ouverts. La classe *RandomAccessFile* est utilisée pour la manipulation de fichiers à accès direct. La classe *StreamTokenizer* sert à faire de l'analyse lexicale en lisant le flux d'entrée, unité lexicale par unité lexicale.

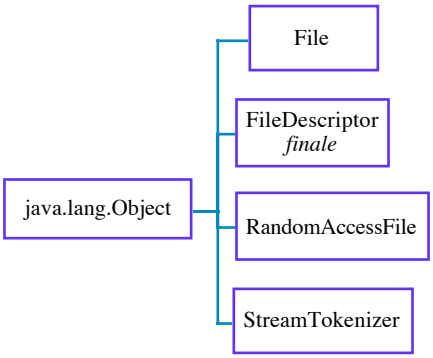


Figure 19.5 Classes diverses

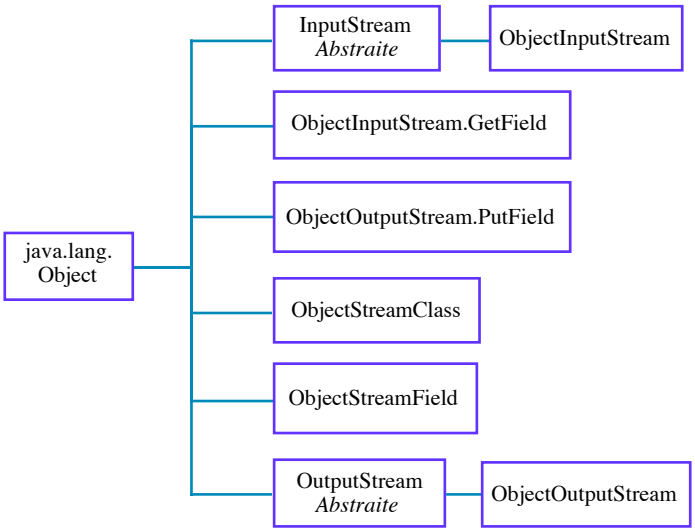


Figure 19.6 Classes pour la sérialisation des objets

La sérialisation consiste à lire et à écrire un objet dans un flux. L'objet est transformé en une séquence d'octets, qui est lue ou écrite en une seule fois. Les classes qui permettent la sérialisation d'objets sont présentées à la figure 19.6

Les classes *ObjectInputStream* et *ObjectOutputStream* permettent la lecture et l'écriture d'objets sérialisables dans des flux (éventuellement des fichiers). Elles comportent respectivement une méthode de lecture et d'écriture qui effectue l'opération souhaitée en une seule fois, les objets qui composent l'objet considéré subissant ipso facto la même opération de manière transparente.

Un objet est sérialisable si sa classe implante l'interface *Serializable*. Les deux classes internes *ObjectInputStream.GetField* et *ObjectOutputStream.PutField* permettent l'accès aux champs persistants d'un objet. Les classes *ObjectStream-*

Class et *ObjectStreamField* donnent accès aux informations décrivant la sérialisabilité d'une classe quelconque. Elles ne sont utiles que pour la gestion de problèmes de version entre la définition de la classe et celle de l'objet persistant ou sérialisé.

Interfaces

L'interface *Serializable* ne contient ni champ ni méthode, il suffit de la mentionner dans la déclaration de la classe des objets à sérialiser.

Une autre interface très utile de ce package est *FilenameFilter*. Elle définit une seule méthode, *accept()*, permettant la sélection des entrées d'un répertoire. Cette interface est référencée par une méthode *list()* de la classe *File* et par la classe *FileDialog* du package AWT.

Exceptions

Les exceptions les plus utilisées sont les suivantes :

- la classe *EOFException* sert à indiquer que la fin de fichier a été atteinte lors de la lecture. La plupart des classes, hormis *DataInputStream*, retournent une valeur spéciale en fin de fichier;
- la classe *FileNotFoundException* indique qu'un fichier n'a pu être trouvé;
- la classe *IOException* signale qu'un problème d'entrée-sortie est survenu. Beaucoup de méthodes du package la génère;
- la classe *InterruptedIOException* signale qu'une entrée-sortie a été interrompue.

19.2 Constructeurs des fichiers d'entrée et de sortie

Seules les classes *FileReader* et *FileInputStream* permettent de construire une instance à partir d'un nom de fichier ou d'une instance de *File*. Les instances des autres classes ne peuvent être élaborées qu'à partir d'un *Reader* ou d'un *InputStream*.

Il est cependant possible de construire une instance de n'importe quelle sous-classe de *InputStream* ou d'un *Reader* à partir d'un nom de fichier, comme le montre l'exemple suivant :

```
BufferedReader b = new BufferedReader(new FileReader("toto.txt"));
```

En effet, toute instance d'une sous-classe peut être considérée comme une instance d'une de ses super-classes. Une instance de *FileReader* peut être considérée comme une instance de *Reader*, qui est la classe du paramètre du constructeur de *BufferedReader*.

Pour les fichiers de sortie, les classes qui permettent la création d'une instance à partir d'un nom de fichier sont *FileWriter* et *FileOutputStream*.

Les fichiers à accès direct (classe *RandomAccessFile*) peuvent eux aussi être créés directement avec un nom de fichier ou un *File*.

19.3 Entrées-sorties clavier/écran

La classe *System* possède trois canaux d'entrées-sorties permettant la saisie de caractères (*in*), l'affichage sur écran (*out*) et l'affichage des erreurs (*err*). Ces canaux sont des variables de classe.

```
public static PrintStream err;
public static InputStream in;
public static PrintStream out;
```

La classe *InputStream* est abstraite, ainsi la classe effective du canal *in* est une sous-classe de *InputStream*. Le programme suivant calcule et affiche cette classe, à savoir *java.io.BufferedInputStream*.

```
import java.io.*;
public class classSystemIn {
  static public void main(String[] args)
   {System.out.println(System.in.getClass().getName());}}
```

Exemple : Affiche le nom de la classe de in

Ainsi, toute saisie au clavier est insérée dans un tampon (*buffer*). La méthode *available()* renvoie le nombre d'octets en attente dans le tampon. Selon votre système d'exploitation, ce nombre peut varier en fonction du caractère spécifiant la fin de la saisie, Control-D (Unix), Control-Z (MSDOS) ou retour chariot (Entrée).

19.4 Variables d'environnement

Outre les canaux permettant les entrées-sorties standard, la classe *System* permet d'accéder aux variables d'environnement. L'une d'entre elles conserve le nom du répertoire courant que nous utiliserons dans un des programmes ci-dessous.

Le programme suivant affiche l'ensemble des variables d'environnement; ce sont des objets de la classe *Properties*. La méthode *list()* de cette classe admet comme paramètre un objet de la classe *OutputStream*. On peut donc lui fournir comme paramètre effectif n'importe quel objet d'une des sous-classes de *OutputStream*. Dans notre programme, nous lui fournissons *System.out* qui est un objet de la classe *PrintStream*.

```
import java.io.*;
import java.util.Properties;
public class DemoProperties {
   static public void main(String[] args)
      {Properties p=System.getProperties();
       p.list(System.out);}}
```

19.5 Accès au répertoire de travail

Le programme suivant affiche le répertoire courant à partir de la variable d'environnement adéquate (*user.dir*).

```
import java.io.*;
public class getWorkingDir {
  static public void main(String[] args)
   {System.out.println(System.getProperty("user.dir"));}
 }
```

19.6 Classe File

De manière indépendante du système d'exploitation, la classe *File* permet :

- de référencer des fichiers et des répertoires à partir de leur nom;
- de tester leur existence, de connaître leurs droits d'accès;
- d'accéder à leurs propriétés (date de modification, longueur);
- de créer des fichiers (temporaires ou non) et des répertoires;
- de détruire ou renommer des répertoires et des fichiers;
- de lister (éventuellement avec un filtre) les entrées d'un répertoire.

Selon les systèmes d'exploitation, les caractères pour séparer les noms de répertoires sont des / ou des \, ou encore des deux-points (:). Pour la construction de noms de répertoire ou de fichier indépendante du système d'exploitation, la classe *File* définit des variables de classes constantes désignant ces séparateurs, les valeurs de ces constantes étant dépendantes du système utilisé.

19.7 Quelques sous-classes de File utiles

Nous avons développé les classes *Fichier*, *FichierLecture*, *FichierEcriture*, *Repertoire*, *RepertoireLecture*, *RepertoireEcriture* qui sont souvent utilisées dans ce chapitre. Leur utilité tient à ce qu'elles n'autorisent la création d'objets que si les droits de lecture ou d'écriture requis sont respectés. La figure 19.7 ci-dessous montre la hiérarchie de ces différentes classes.

En utilisant ces classes, on tente de créer une instance qui a les droits désirés. En cas d'échec de la création, une exception est générée. La réutilisation de ces classes favorise l'écriture de programmes plus simples à développer et à relire ensuite.

La classe *Fichier*

La classe abstraite *Fichier* est une sous-classe de *File*. Son intérêt est de disposer d'une méthode retournant le répertoire dans lequel se trouve le fichier correspondant. Ce sont ses deux sous-classes (*FichierLecture* et *FichierEcriture*) qui testeront les droits d'accès lors de la création d'instances.

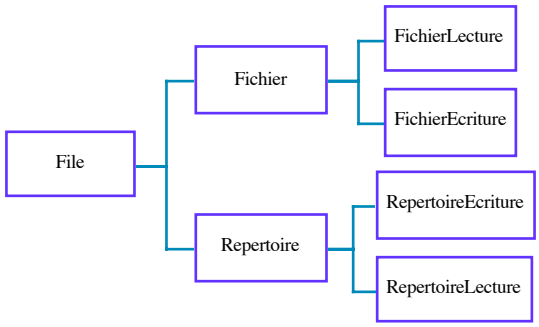


Figure 19.7 Quelques classes utiles, sous-classes de File

```
import java.io.*;
public abstract class Fichier extends File{
  public Fichier(String s) throws NullPointerException    {
  super(s);}
  protected String repertoireDe()   {
    String NomComplet=this.getAbsolutePath();
    return NomComplet.substring(0,
                   NomComplet.lastIndexOf(File.separator));
  }
}
```

La classe Fichier

La classe *FichierLecture*

Elle étend la classe *Fichier* en n'autorisant que la création de références à des fichiers pour lesquels on dispose d'un droit de lecture.

```
import java.io.*;
public class FichierLecture extends Fichier{
   public FichierLecture(String s) throws IOException,
                                          NullPointerException   {
    super(s);
    if (!this.exists() || !this.isFile())
      throw new IOException( this.getName() + ": fichier
inexistant.");
    if (!this.canRead())
      throw new IOException( this.getName() + ": fichier non
lisible.");
   }
}
```

La classe FichierLecture

La classe *FichierEcriture*

Elle étend la classe *Fichier* en n'autorisant que la référence à des fichiers pour lesquels on dispose d'un droit d'écriture (sur le fichier ou sur le répertoire de sa référence, selon l'existence ou non du fichier considéré).

```
import java.io.*;
public class FichierEcriture extends Fichier{
   public FichierEcriture(String s) throws IOException {
    super(s);
    if (this.exists()){
      if (this.isFile()){
           if (!this.canWrite())
              throw new IOException( this.getName() +
                  ": n'est pas modifiable.");
      }
      else
           throw new IOException( this.getName() +
                  ": n'est pas un fichier.");
    }
    else // le fichier n'existe pas
       { // il faut tester le droit en ecriture dans le repertoire
         File Dir=new File(this.repertoireDe());
         if (!Dir.exists())
             throw new IOException( Dir.getName() +
                    ": n'est pas un repertoire");
         if (!Dir.canWrite())
             throw new IOException( Dir.getName() +
                    ": n'est pas modifiable.");
       }
  }
}
```

La classe FichierEcriture

La classe *Repertoire*

C'est une sous-classe abstraite de *File* permettant de référencer un répertoire et d'afficher des informations sur toutes les entrées de celui-ci. Cette classe ne teste aucun droit d'accès sur les répertoires. Son rôle est d'offrir des méthodes d'accès au contenu du répertoire pour ses deux sous-classes : *RepertoireLecture* et *RepertoireEcriture*.

```
import java.io.*;
public abstract class Repertoire extends File {
  private static final int lim=20;

   public Repertoire(String s) throws IOException{
      super(s);}

   private void afficheUnFichier(String ref){
      File unElement= new File(this,ref);
      System.out.print((unElement.canRead())? "L":" ");
      System.out.print((unElement.canWrite())? "E":" ");
      System.out.print(" "+ref);
      if (unElement.isDirectory())
         System.out.println("\t"+"repertoire");
      else  if (unElement.canRead())
```

```
                System.out.println("\t"+unElement.length());
            else System.err.println(" ????");
    }
    public void afficheListe(String[] fichiers){
       System.out.println(this.getName());
      if (fichiers.length==0){
         System.out.println("aucun fichier");
         return;}
      for(int i=0;i<fichiers.length;i++) {
         if ((i>0) && ((i % lim)==0)){
            System.out.println("<retour> pour continuer");
            try {System.in.read();}
            catch (IOException e){}}
         this.afficheUnFichier(fichiers[i]); }; //for
      System.out.println(" "+fichiers.length+ " fichiers");
    }
}
```

La classe Repertoire

La classe *RepertoireLecture*

Son instanciation n'est autorisée que pour les répertoires existants pour lesquels on dispose du droit de lecture.

```
import java.io.*;
public class RepertoireLecture extends Repertoire {
public RepertoireLecture(String s) throws IOException   {
   super(s);
   if (!this.exists() ||  !this.isDirectory())
     throw new IOException( this.getName() + ": repertoire
inexistant.");
   if (!this.canRead())
      throw new IOException( this.getName() +
                           ": repertoire non lisible.");
   }
}
```

La classe RepertoireLecture

La classe *RepertoireEcriture*

Son instanciation n'est autorisée que pour les répertoires existants pour lesquels on dispose du droit d'écriture.

```
import java.io.*;
public class RepertoireEcriture extends Repertoire{
   public RepertoireEcriture(String s) throws IOException   {
    super(s);
    if (!this.exists() ||  !this.isDirectory())
     throw new IOException( this.getName() + ": repertoire
inexistant.");
    if (!this.canWrite())
      throw new IOException( this.getName() +
```

```
                    ": repertoire non modifiable.");
    }
}
```

La classe RepertoireEcriture

Les exemples suivants illustrent les différentes classes de ce package et réutilisent les classes ci-dessus.

19.8 Affichage du contenu d'un répertoire avec filtrage

La classe *AfficheRepertoire* (voir l'application 2 ci-dessous) qui utilise la classe *RepertoireLecture*, affiche la liste des entrées du répertoire[1]. Chaque entrée est affichée par la méthode *Repertoire.afficheListe*. La liste affichée ne contient pas nécessairement toutes les entrées du répertoire : un filtre permet de ne retenir qu'une sélection de ces entrées. C'est le nombre d'arguments passés en paramètres qui détermine s'il y a filtrage ou non.

```
import java.io.*;
public class AfficheRepertoire{
   public static void main(String args[]){
      Repertoire rep;
      String[] fichiers;
     try {
       rep= new RepertoireLecture(args[0]);
       switch (args.length)
        {case 1:
           fichiers=rep.list();
           break;
         case 2:
           FilenameFilter ff;
           ff= new FiltreSuffixe(args[1]);
           fichiers=rep.list(ff);
           break;
         default:
           System.out.println(
                 "Usage: java AfficheRepertoire <nom> [filtre]");
           return;} // switch
     rep.afficheListe(fichiers);} // try
     catch (IOException e)
       {System.err.println(e.getMessage());}
    }
}
```

Application 2 : AfficheRépertoire

Filtrage

La définition d'un filtre s'effectue en implantant l'interface *FileNameFilter*. Le

1. Comme la commande *dir* de MSDos ou *ls* de Unix.

filtrage de la liste proprement dit se fait grâce à la méthode *list()* de la classe *File* qui admet comme paramètre l'interface *FileNameFilter*. Le filtre implanté dans la classe ci-dessous permet la sélection des entrées du répertoire (fichiers ou sous-répertoires) se terminant par un même suffixe.

```
import java.io.*;
class FiltreSuffixe implements FilenameFilter{
   private String suffixe;
   public FiltreSuffixe(String suffixe)
     {this.suffixe=suffixe;}
   public boolean accept(File dir, String nom)
     {return  nom.endsWith(suffixe);}
}
```

La classe FiltreSuffixe

19.9 Lecture d'un fichier

La classe suivante *(Afficheur)* implante un ensemble de méthodes permettant d'afficher le contenu d'un fichier ou d'un URL dans une fenêtre graphique. Cette classe utilise des composants graphiques du package *awt* : champs de texte, menus, fontes. Elle utilise également la classe *FichierLecture*.

```
import java.awt.*;
import java.awt.event.*;
import java.io.*;
import java.net.*;
public class Afficheur extends Frame implements ActionListener {
  MenuItem Quitter;
  private void RemplitFenetre(InputStream dis,int taille)
  throws IOException
  {TextArea text;
   // lecture du fichier dans un tableau
   byte[] contenu= new byte[taille];
   dis.read(contenu,0,taille);
   dis.close();
   // definition et remplissage de la zone de texte
   text= new TextArea(new String(contenu),24,40);
   text.setFont(new Font("Courier",Font.PLAIN,10));
   text.setEditable(false);
   this.add("Center",text);
   // creation de la barre de menus
   MenuBar barreMenu= new MenuBar();
   this.setMenuBar(barreMenu);
   Menu menuFichier= new Menu("Fichier");
   Quitter = new MenuItem("Quitter");
   Quitter.addActionListener(this);
   menuFichier.add(Quitter);
   barreMenu.add(menuFichier);
   // positionnement des composants graphiques
   this.pack();
   // affichage de la fenetre
```

```
     this.setVisible(true);
    }
    public Afficheur(FichierLecture fichier) throws IOException {
      super(fichier.getName());
      FileInputStream canal=new FileInputStream(fichier);
      this.RemplitFenetre(canal,(int) fichier.length());
    } // Afficheur

    public Afficheur(URLConnection urlc) throws IOException {
      super(urlc.getURL().toExternalForm());
      DataInputStream canal=new
DataInputStream(urlc.getInputStream());
      this.RemplitFenetre(canal,(int) urlc.getContentLength());
    } // Afficheur

    public void actionPerformed (ActionEvent e) {
      if (e.getSource()==Quitter)
        {this.setVisible(false);
         this.dispose();
         System.exit(0);}
    } //actionPerformed
}
```

La classe Afficheur

L'affichage du contenu d'un URL (voir point 20.3) est rendu possible car une connexion élaborée à partir d'un URL ouvre un flux de lecture.

Utilisation

Le programme suivant, dont un exemple d'exécution est présenté dans la figure 19.8, utilise la classe *Afficheur*. Le nom du fichier à afficher est passé comme argument sur la ligne de commande. On tente de créer une instance de la classe *FichierLecture* qui servira de paramètre au constructeur de *Afficheur*.

```
import java.io.*;
public class AfficheUnFichier {
   static public void main(String[] args){
      if (args.length!=1) {
         System.out.println
            ("Usage: java AfficheUnFichier <nom de fichier>");
         System.exit(0);}
      try { FichierLecture fichier= new FichierLecture(args[0]);
            Afficheur f= new Afficheur(fichier);}
     catch (IOException e)
       {System.out.println(e);}
    } //main
}
```

Application 3 : AfficheUnFichier

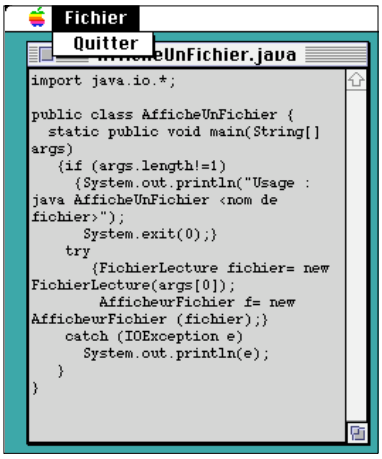


Figure 19.8 Exemple d'exécution de AfficheUnFichier

19.10 Copie d'un fichier

La copie de fichier réalisée par l'application 4 ci-après requiert de tester les droits d'accès suivants :

- en lecture du fichier source;
- en écriture du fichier destination (s'il existe);
- en écriture du répertoire du fichier destination (si le fichier est créé par l'opération de copie).

Le programme suivant utilise donc les classes *FichierLecture* et *FichierEcriture* (qui utilise *RepertoireEcriture*). Ce programme montre l'intérêt des sous-classes que nous avons définies puisqu'il permet de créer directement des objets ayant des propriétés désirées, plutôt que d'en manipuler de plus génériques (en l'occurrence des objets de la classe *File*).

```
import java.io.*;
public class CopieurFichier {
   public static void copie(String src,String dest){
    FileInputStream s=null;
    FileOutputStream d=null;
    try {
      FichierLecture Source= new FichierLecture(src);
      FichierEcriture Destination=new FichierEcriture(dest);
      s=new FileInputStream(Source);
      d=new FileOutputStream(Destination);
      byte[] tampon=new byte[1024];
      int lu=0;
      do {  lu=s.read(tampon);
            if (lu!=-1) d.write(tampon,0,lu);}
      while (lu!=-1);
```

```
      s.close();
      d.close();
    } // try
    catch (IOException e)
        {System.err.println(e.getMessage());}
} // copie

   public static void main(String args[]){
   if (args.length!=2){
      System.err.println("Usage: java CopieurFichier <src> <dest>");
      return;}
   copie(args[0],args[1]);
   }
}
```

Application 4 : Copie de fichiers

19.11 Lecture filtrante d'un fichier

L'exemple suivant (classe *Ponctuation*, application 5) illustre le filtrage des caractères durant la lecture. La classe *Ponctuation* rétablit, s'il n'y sont pas, les espaces typographiques nécessaires autour des virgules, deux-points et points-virgules. Le filtrage consiste à analyser chaque caractère lu et si celui-ci correspond à l'un des trois signes de ponctuation ci-dessus, de rajouter ou de supprimer des espaces si nécessaire.

Définition du filtre

La classe *Ponctuation* est une sous-classe de la classe *PushbackReader* qui permet de réinsérer des caractères déjà lus dans le flux de ceux qui restent à lire (méthode *unread*). Nous avons redéfini les deux méthodes *read* de la classe *PushbackReader* dans la sous-classe afin de réaliser le filtrage qui consiste à ponctuer correctement un texte.

```
import java.io.*;
public class Ponctuation extends PushbackReader {
  private int dernierLu=-1;
  public Ponctuation (Reader fichier) {
    super(fichier,1);
  }
  public int read() throws IOException {
    int c=super.read();//appel à read de PushbackReader
    if ((",;:".indexOf(dernierLu)>=0) && (c!=' ')) {
      unread(c);
      c=' ';
    }
    if (c==' ') {
      int d=super.read();
      if (d!=',') unread(d);
      else c=',';
```

```
    }
    if (((c==';') || (c==':')) && (dernierLu!=' ')){
      unread(c);
      c=' ';
    }
    dernierLu=c;
    return c;
  }
  public int read(char[] cbuf , int off,int len) throws IOException {
    int c=0;
    for (int i=0;(i<len) && (c!=-1);i++) {
       c=read();
       cbuf[off++]=(char) c;
    }
    if (c!=-1) return len;
    else return -1;
  } }
```

La classe Ponctuation

Utilisation du filtre

```
import java.io.*;
public class Ponctue {
    public static void main(String args[]) {
      char[] buf= new char[20];
      int c;
      if (args.length!=1) {
        System.err.println("java Ponctue <nom de fichier>");
        return;
      }
      try {
         FichierLecture f=new FichierLecture(args[0]);
         Ponctuation fp=new Ponctuation (new FileReader(f));
         do {
            c=fp.read(buf,0,20);
            if (c!=-1) System.out.print(buf);
         } while (c!=-1);
        fp.close();}
      catch (IOException e) {System.err.println(e.getMessage());}
    } //main
} //class
```

Application 5 : Corriger la ponctuation d'un fichier texte.

19.12 Fichier à accès direct

La classe *RandomAccessFile* permet simultanément la lecture et l'écriture dans un fichier. Le *byte* est l'unité de mesure des positions et des déplacements. Cette classe comporte plus d'une trentaine de méthodes et manipule tous les types de base du langage Java, en lecture comme en écriture.

Comment faire une classe implantant la gestion de fichiers à accès direct d'entiers?

La gestion des fichiers à accès direct d'entiers ne requiert (outre un constructeur au moins pour la création d'instance) que les méthodes suivantes, l'unité de positionnement étant alors l'entier :

```
void close();
int read();
void write(int  v);
void seek(long  pos);
public long getFilePointer();
public long length();
```

Nous ne pouvons pas implanter la classe désirée par extension de la classe *RandomAccessFile* puisque nous ne pouvons pas interdire l'héritage de certaines méthodes. Il nous faut alors composer ou, sans faire de jeu de mots, construire la classe désirée par composition.

Fichier d'entiers à accès direct

La classe *FichierEntierAccesdirect* est ainsi élaborée par composition d'une variable d'instance de la classe *RandomAccessFile* et d'une constante permettant de définir l'unité de positionnement en nombre de bytes (en l'occurrence 4).

```
import java.io.*;
public class FichierEntierAccesDirect  {
  private static final int taille=4; //taille d'un entier
  protected RandomAccessFile f;
 // constructeurs
  public FichierEntierAccesDirect(File  file, String  mode)
  throws IOException
    {f=new RandomAccessFile(file, mode);}
  public FichierEntierAccesDirect(String  name, String  mode)
  throws IOException
    {f=new RandomAccessFile(name,mode);}

  public void close() throws IOException
      {f.close();}
  public int read() throws IOException
      {return f.readInt(); // si read(), il y a des erreurs
      }
  void write(int  v) throws IOException
      {f.writeInt(v); // si write(int v), il y a des erreurs
      }
  void seek(long  pos) throws IOException
      {f.seek(taille*pos);};
  public long getFilePointer() throws IOException
      { return f.getFilePointer() / taille;}
  public long length() throws IOException
      { return f.length() / taille;}
} // FichierEntierAccesDirect
```

La classe FichierEntierAccesDirect

Utilisation des fichiers d'entiers à accès direct

Le programme suivant (application 6) utilise la classe *FichierEntierAccesDirect*. On crée un fichier d'entiers avec des valeurs choisies au hasard puis on le ferme. On l'ouvre ensuite pour le lire. En fin de lecture, on positionne le curseur sur le 5e entier du fichier.

```
import java.util.Random;
import java.io.IOException;

public class DemoIntFichier  {
  static public void main(String[] args) {
   FichierEntierAccesDirect f;
   Random v=new Random();
   int x;
   try {
      f=new FichierEntierAccesDirect("Monfichier", "rw");
      for (int i=0;i<10;i++)
          {x=v.nextInt();
           System.out.println(i+":" +x);
           f.write(x);}
      f.close();}
   catch (IOException e) {System.exit(1);}
// lecture du fichier que nous venons de créer
   try {
      f=new FichierEntierAccesDirect("Monfichier", "r");
      System.out.println("taille du fichier: "+f.length());
      long y;
      for (int i=0;i<f.length();i++)
        {y=f.getFilePointer();
         x=f.read();
         System.out.println(y+":"+x);}

      // positionnement sur le 5eme entier du fichier
       f.seek(5);
      System.out.println("le 5 eme element est:" + f.read());
      f.close();}
   catch (IOException e) {System.out.println(e);}} // main
}
```

Application 6 : Utilisation des fichiers d'entiers à accès direct

19.13 Analyse lexicale

L'application 7 illustre l'utilisation de la classe *StreamTokenizer* qui permet de lire des unités lexicales (*tokens*) et de connaître leur type. Cette classe est très utile pour faire de l'analyse lexicale et existe aussi en version *StringTokenizer*, la lecture s'effectuant alors à partir d'une chaîne de caractères et non à partir d'un fichier.

La variable d'instance *ttype* indique le type de l'unité lexicale lue. Les valeurs des types sont définies par des variables de classes constantes (*TT_WORD*,

TT_NUMBER, etc.). Si l'unité lexicale est de type nombre, sa valeur est stockée dans la variable d'instance *nval*, si c'est un mot dans *sval*.

La méthode *nextToken()* permet de lire les unités lexicales les unes après les autres.

Le programme suivant utilise la classe *StreamTokenizer* pour compter les mots et les nombres d'un fichier.

```
import java.io.*;
public class Compteur {
   public static void main(String args[]){
     int nNombres=0;
     int nMots=0;
     Reader r=null;
     StreamTokenizer st=null;
     try {
         r = new BufferedReader(new FileReader(args[0]));
         st = new StreamTokenizer(r);}
     catch (IOException e)
        {System.out.println("usage : java Compteur <filename>");
         try {r.close();}  catch (IOException e1) {};
         System.exit(1);}
     st.eolIsSignificant(true);
     st.wordChars((int) '_',(int)'_');
     st.ordinaryChar((int) '.');
     try
       {while (st.nextToken()!=st.TT_EOF)
         {if (st.ttype== st.TT_WORD)
            {System.out.println(st.sval);
             nMots++;}
          if (st.ttype== st.TT_NUMBER)
            {System.out.println(st.nval);
             nNombres++;}
         } }
     catch (IOException e)
       {System.out.println("error when reading.");}
     finally
       {try {r.close();} catch (IOException e1) {};}
     System.out.println("mots : "+ nMots);
     System.out.println("nombres : "+ nNombres);
    } //main
}
```

Application 7 : Compteur de mots et de nombres d'un fichier

19.14 Dialogues pour l'ouverture et la fermeture d'un fichier

La classe *FileDialog* du package *awt* permet la création d'une fenêtre de dialogue pour l'ouverture ou la sauvegarde d'un fichier. Cette fenêtre correspond à la

fenêtre standard d'ouverture et de fermeture de fichiers de l'interface graphique du système d'exploitation utilisé.

Le constructeur de cette classe admet un paramètre indiquant le type de fenêtre souhaité : pour l'ouverture d'un fichier, sa valeur est *FileDialog.LOAD*, pour la sauvegarde d'un fichier, sa valeur est *FileDialog.SAVE*.

Cette classe a des méthodes permettant de récupérer et de spécifier le répertoire et le fichier. Ces méthodes sont :

- *getDirectory();*
- *getFile();*
- *setDirectory(String dir);*
- *setFile().*

La méthode *setFileNameFilter(FileNameFilter f)* permet le filtrage des noms d'entrées affichés dans la fenêtre.

19.15 Sérialisation et persistance des objets

La sérialisation est une opération qui consiste à lire et écrire un objet dans un flux, l'objet étant transformé en une séquence d'octets lue ou écrite en une seule fois. Cette opération fonctionne également pour des objets complexes, c'est-à-dire des objets dont les valeurs des champs sont elles-mêmes des objets. Le graphe d'objets est alors lu ou écrit intégralement en une seule opération. La sérialisation a deux usages essentiels :

- faire circuler (via des flux) des objets entre des applications distribuées, sans encodage ni décodage spécifiques à programmer;
- rendre persistant des objets, toujours sans aucune programmation.

Un objet est dit persistant si sa durée de vie est supérieure à celle du programme qui l'a créé. La persistance des objets est implantée par leur sérialisation dans des fichiers.

L'exemple ci-dessous montre l'intérêt de la sérialisation pour la persistance des objets. On crée dynamiquement un arbre de chaîne de caractères que l'on sauve dans un fichier pour être relu ultérieurement. La classe *Arbre* implante la notion d'arbre binaire de recherche, et déclare l'interface *Serializable* (qui ne contient, on le rappelle, ni champ, ni méthode).

```
import java.io.Serializable;
class Arbre implements Serializable {
   Arbre sag, sad; // sous-arbres gauche et droit
   String Label;
   public Arbre (String l) {
     Label=l;
     sag=null;
     sad=null;
   }
```

```
    void inserer(String l) {
      if (Label==null) Label=l;
      else if (l.compareTo(Label)<0) {
             if (sag==null) sag=new Arbre(l);
             else sag.inserer(l);
           }
      else if (l.compareTo(Label)>0) {
             if (sad==null) sad=new Arbre(l);
             else sad.inserer(l);
           }
    }
    void parcours() {
      if (sag!=null) sag.parcours();
      System.out.print(Label+" ");
      if (sad!=null) sad.parcours();
    }
}
```

La classe Arbre

Le programme ci-dessous crée un arbre binaire de recherche à partir d'une phrase passée en paramètre sur la ligne de commande, puis appelle la méthode *parcours* qui imprime les nœuds de l'arbre dans l'ordre alphabétique. L'arbre est rendu persistant par l'écriture de l'objet (méthode *writeObject*) dans un *ObjectOutputStream*.

```
import java.io.*;
class DemoArbre {
    static public void main(String[] args) {
      Arbre a=new Arbre(null);
      if (args.length==0) {
         System.out.println("Usage : java DemoArbre <X1> .. <Xn>");
         System.exit(0);
      }
      for (int i=0; (i<args.length); i++) {
         a.inserer(args[i]);
      }
      a.parcours();
      // rendre l'arbre persistant
      try {
          FileOutputStream fos = new FileOutputStream("arbre.tmp");
          ObjectOutputStream oos = new ObjectOutputStream(fos);
          oos.writeObject(a);
          oos.flush();
          oos.close();
      } catch (Exception e) {
           e.printStackTrace();
           System.exit(1);
      }
    } // main
}
```

Application 8 : Rendre un arbre persistant

Le programme ci-dessous lit cet objet persistant (méthode *readObject* de la classe *ObjectInputStream*).

```
import java.io.*;
class DemoArbreLecteur {
   static public void main(String[] args) {
     Arbre b=null;
     try {
         FileInputStream fis = new FileInputStream("arbre.tmp");
         ObjectInputStream ois = new ObjectInputStream(fis);
         b = (Arbre) ois.readObject();
         ois.close();
     } catch (Exception e) {
          e.printStackTrace();
          System.exit(1);
     }
     b.parcours();
   } // main
}
```

Application 9 : Lecture d'un objet arbre persistant

La sérialisation de la racine *a* entraîne bien celle de tous les autres nœuds de l'arbre. La persistance des objets est donc réalisée avec la lecture ou l'écriture d'un objet en une seule opération. Les difficultés apparaissent seulement si des versions différentes des classes d'objets sérialisés sont incompatibles. Il est préférable de ne pas modifier la définition d'une classe tant qu'il en existe des instances persistantes. Certaines modifications peuvent en effet poser des problèmes d'incompatibilité comme, par exemple, la destruction d'un champ.

La sérialisation permet également, avec une grande facilité, la circulation des objets aussi complexes soient-ils, entre des applications distribuées.

19.16 Invitation à explorer

Complétez l'éditeur de texte pour ajouter les fonctions d'ouverture et de sauvegarde de fichiers.

Chapitre 20

Les accès au réseau (*java.net*)

Ce package fournit un ensemble de classes pour communiquer sur le réseau Internet, télécharger des URL, définir des gestionnaires de type MIME et de protocoles. Dans ce chapitre, nous développerons des applications client-serveur à partir d'exemples complets.

Après avoir décrit les composants principaux de ce package, nous en présentons des exemples d'utilisation illustrant l'identification du contenu d'un URL et l'affichage de son contenu dans une fenêtre graphique.

Un autre exemple présente la vérification de tous les URL contenus dans une page de type *text/html* téléchargée à partir de son URL.

Les deux derniers exemples de ce chapitre sont basés sur le principe du client-serveur. Le premier montre la réalisation d'un service simple et définit une architecture de base réutilisable. Le second, une relation client-serveur plus élaborée, la télédiscussion. Ces exemples peuvent être étendus à la réalisation de vos propres serveurs.

20.1 Description du package

Les classes

Deux types de classes constituent ce package :

- les classes de bas niveau permettent de définir des gestionnaires de protocoles et de types MIME[1];

- les classes de haut niveau permettent de se connecter sur un URL, d'en télécharger le contenu et de faire de la communication interprocessus sur le réseau Internet.

Les classes de bas niveau

Ces classes sont très utiles si l'on souhaite gérer de nouveaux types MIME absents dans l'environnement Java ou des protocoles.

Les classes permettant de définir des types MIME sont (figure 20.1) :

- *ContentHandler* : cette classe abstraite définit ce qu'est un gestionnaire de type MIME. Un tel gestionnaire télécharge un contenu à partir d'une connexion sur un URL et transforme ce contenu en une instance d'une classe d'objets, classe prédéfinie dans Java ou implantée par vous-même. La méthode *getContent()* réalise cette transformation. Par exemple, Java incorpore un gestionnaire pour les images au format *gif*. Ce gestionnaire d'images est une sous-classe de la classe *ContentHandler*;
- *URLStreamHandler* : cette classe abstraite définit ce qu'est un protocole pour la connexion sur un URL. Par exemple, Java implante un protocole permettant la connexion sur des serveurs *HTTP*. Ce gestionnaire de protocole est une sous-classe de la classe *URLStreamHandler*.

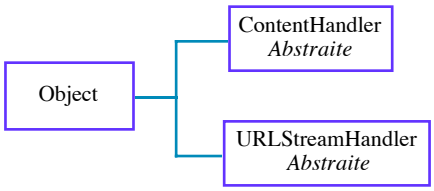


Figure 20.1 Classes pour implanter des gestionnaires

Les classes permettant l'envoi et la réception de paquets sur le réseau sont (figure 20.2) :

- *DatagramPacket* : cette classe représente un paquet à envoyer ou à recevoir à travers le réseau. Un paquet à envoyer est composé d'un tableau d'octets et d'une adresse Internet (un objet de la classe *InetAddress*). Un paquet à recevoir est composé d'un tableau d'octets;
- *DatagramSocket* : cette classe représente un émetteur-récepteur de paquets (méthodes *send()* et *receive()*) mais sans aucune garantie concernant l'ordre d'arrivée des paquets ni leur arrivée elle-même. Il est

1. MIME est un acronyme de *Multipurpose Internet Mail Extensions*. L'URL http://www.imc.org/rfcs.html#mime décrit complètement cette spécification pour la mise en forme de messages Internet.

préférable d'utiliser des objets des classes *Socket* et *ServerSocket* pour communiquer de manière plus sûre sur le réseau;

- la classe *MulticastSocket* permet l'envoi et la réception de paquets à un groupe de destinataires et plus à un seul, comme la classe précédente;
- *SocketImpl* : cette classe définit un ensemble de méthodes nécessaires à l'implantation de la communication à travers des sockets.

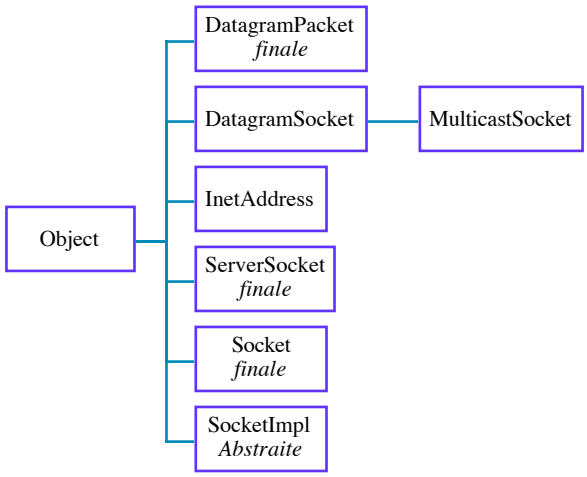


Figure 20.2 Classes pour la communication de paquets par des sockets

Les classes de haut niveau

La classe *InetAddress* représente des adresses Internet. Elle est utilisée par l'un des constructeurs de la classe *Socket* et par celui de la classe *DatagramSocket* pour l'envoi de paquets.

La classe *ServerSocket* permet la construction de serveurs : un serveur attend des demandes de connexion émanant de clients sur un port particulier. La méthode *accept()* renvoie un socket lorsqu'une demande est reçue.

La classe *Socket* permet la communication interprocessus à travers le réseau. En spécifiant une machine (ou une adresse Internet) et un numéro de port, un socket peut être construit. Le protocole par défaut est celui des flots (*streams*). Les méthodes *getInputStream()* et *getOutputStream()* permettent la récupération des canaux de lecture et d'écriture.

La classe URL (*Uniform Resource Locator*) permet le téléchargement des ressources référencées par un URL. Elle offre trois possibilités de téléchargement :

- directement par la méthode *getContent()*. Cette méthode renvoie un objet dont la classe correspond à un type MIME pour lequel on dispose d'un gestionnaire (un *ContentHandler*);

- indirectement par une *URLConnection* obtenue par la méthode *openConnection()*, puis ensuite par la méthode *getContent()* qui est similaire à la précédente;
- par un canal de type *InputStream* obtenu par la méthode *openStream()*. Cette voie est la seule possibilité si l'URL contient des informations qui ne sont pas d'un type MIME pour lequel on dispose d'un gestionnaire.

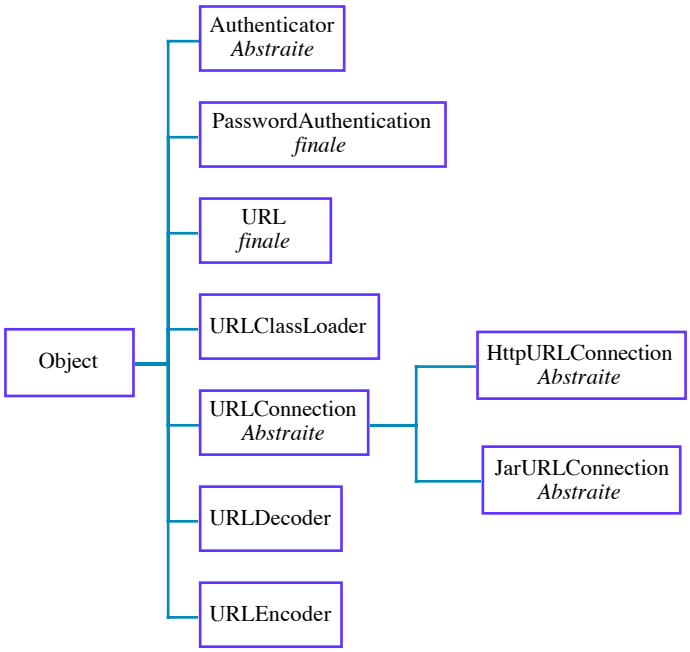


Figure 20.3 Classes pour utiliser les URL

Les classes *Authenticator* et *PasswordAuthentication* permettent l'accès par des programmes Java à des ressources désignées par un URL et protégées par mot de passe, à condition évidemment d'être un utilisateur autorisé de ces ressources.

La classe abstraite *URLConnection* a pour rôle d'offrir un meilleur contrôle sur le téléchargement de ressources référencées par un URL. Cette classe dispose de méthodes permettant de connaître le type, l'en-tête, la date de dernière modification d'un contenu. Cette classe permet le téléchargement des données brutes de ce contenu.

La classe abstraite *HttpURLConnection* établit des connexions vers des URL en utilisant les spécificités du protocole HTTP; il est par exemple possible de savoir si la connexion passe au travers d'un proxy ou non.

La classe abstraite *JarURLConnection* établit des connexions vers des archives Java ou des entrées d'une archive Java à partir d'un URL afin de les télécharger.

Les classes *URLDecoder et URLEncoder* offrent chacune une seule méthode de

classe, dont le but est de coder/décoder un URL en une chaîne de caractères plus portable et inversement. Les espaces et les caractères non alphanumériques sont codés différemment.

Interfaces

Les interfaces de ce package fournissent les signatures des méthodes pour les classes de bas niveau. Ces interfaces sont :

- *ContentHandlerFactory* : cette interface ne définit que la signature d'une seule méthode, *ContentHandler createContentHandler(String mimetype)*. Cette méthode définit l'association entre un type MIME et le gestionnaire correspondant;
- *FileNameMap* : cette interface sert à déterminer, à partir du nom d'un fichier, le nom du type MIME qui va en permettre la lecture;
- *URLStreamHandlerFactory* : cette interface joue le même rôle pour les protocoles que l'interface *ContentHandlerFactory* pour les gestionnaires de type MIME;
- *SocketImplFactory* : cette interface est utilisée par les classes *Socket* et *ServerSocket* pour réaliser les implantations effectives des sockets;
- *SocketOptions* : cette interface est utilisée pour lire et écrire des options liées à une nouvelle implantation de sockets.

Exceptions

- *BindException* : cette classe d'exception indique un problème d'établissement de liaison entre un socket, une adresse et un port locaux;
- *ConnectException* et *NoRouteToHostException* : ces deux classes d'exception indiquent un problème d'établissement de la connexion entre un socket, une adresse et un port distants;
- *MalformedURLException* : cette classe d'exception permet de signaler la non-reconnaissance d'un protocole connu dans la chaîne de caractères spécifiant un URL, ou l'échec de l'analyse lexicale de cette chaîne;
- *ProtocolException* : cette classe d'exception signale les erreurs dans un protocole, par exemple une erreur TCP;
- *SocketException* : cette classe d'exception signale une mauvaise utilisation des sockets;
- *UnknownHostException* : cette classe d'exception signale que l'adresse IP d'un hôte n'a pu être déterminée;
- *UnknownServiceException* : cette classe d'exception signale la non-reconnaissance d'un type MIME ou une tentative d'écriture sur un URL avec accès en lecture seulement.

De nombreuses classes de ce package ont des méthodes signalant des exceptions de la classe *IOException*.

Les exemples suivants utilisent les classes *URL*, *URLConnection*, *Socket*, *ServerSocket, Authenticator* et *PasswordAuthentication*.

20.2 Identification du contenu d'un URL

L'application 10 utilise les classes *URL* et *URLConnection* pour afficher le type MIME de la ressource référencée par un *URL* passé sur la ligne de commande. La méthode *openConnection()* de la classe *URL* peut signaler une exception de la classe *IOException;* cette classe est donc importée dans notre application.

```
import java.io.IOException;
import java.net.*;
public class ContenuURL {
  public static void afficheInfos(URLConnection u){
  // affiche l'URL et les infos s'y rattachant
     System.out.println(u.getURL().toExternalForm() + ":");
     System.out.println("Contenu :"+ u.getContentType());
     System.out.println("Longueur :"+ u.getContentLength());
      try {
        Reader r=new InputStreamReader(u.getInputStream());
        BufferedReader br=new BufferedReader (r);
        for (int i=0;i<3;i++) {
           System.out.println(br.readLine());
        }
        System.out.println("  ...");
        br.close();
      } catch (IOException e) {System.out.println(e.getMessage());}
  }
  static public void main(String[] args)
       throws MalformedURLException, IOException {
     if (args.length==0) return;
     URL url=new URL(args[0]);
     URLConnection connection=url.openConnection();
     afficheInfos(connection);
  }
}
```

Application 10 : Identification du contenu d'un URL

Dans l'exemple d'exécution suivant, le contenu est du type *text/html*.

```
>java ContenuURL http://java.sun.com/
http://java.sun.com/:
Contenu:text/html
Longueur:3495
.....+ trois 1ères lignes HTML du fichier
```

20.3 Accès à un URL protégé par un mot de passe

L'exemple précédent a illustré l'identification du contenu d'un URL. Un programme Java peut accéder à des contenus protégés d'un URL, à la condition évidente d'en posséder les droits d'accès. Un droit d'accès est défini par un nom d'utilisateur et un mot de passe.

Tout d'abord, définissons une classe dont le rôle est d'acquérir une demande d'accès et de fabriquer une demande d'identification qui sera envoyée de manière transparente au serveur HTTP qui héberge la ressource protégée.

```
import java.net.*;
import java.io.*;
class Identification extends Authenticator {
  protected PasswordAuthentication getPasswordAuthentication() {
   PasswordAuthentication ident=null;
   System.out.print("username :");System.out.flush();
   try {
      BufferedReader r= new BufferedReader(
                        new InputStreamReader(System.in));
      String u=r.readLine();
      System.out.print("password :");
      String p=r.readLine();
      ident =new PasswordAuthentication(u,p.toCharArray());
    } catch (IOException e) {System.out.println(e.getMessage());}
   return ident;
  }
}
```

La Classe Identification

La seconde étape consiste à installer une instance de cette classe dans le programme d'accès aux URL. La ligne de code ci-dessous peut se rajouter à l'application 10 dans la méthode *main* avant l'instruction qui demande la connexion.

```
Authenticator.setDefault(new Identification());
```

Lorsque l'application accède à un URL protégé, la méthode *getPasswordAuthentication* de la classe *Identification* est appelée afin d'obtenir un compte utilisateur et un mot de passe.

20.4 Affichage du contenu d'un URL dans une fenêtre

Les exemples ci-dessus nous ont montré comment vérifier un URL et accéder à son contenu que celui-ci soit protégé ou non. Ce nouvel exemple montre le téléchargement du contenu des URL. Dans le chapitre sur les entrées-sorties, nous avons défini la classe *Afficheur* permettant d'afficher le contenu d'un fichier ou d'un URL. Nous réutilisons donc cette classe dans l'application 11 :

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

```
public class AfficheUnURL {
  static public void main(String[] args){
    if (args.length!=1) {
      System.out.println("Usage : java AfficheUnURL <URL>");
      System.exit(0);}
    try {
      URL url= new URL(args[0]);
       Afficheur f= new Afficheur(url.openConnection());}
    catch (IOException e)
      {System.out.println(e);}
    }
 }
```

Application 11 : Affichage du contenu d'un URL

20.5 Vérificateur d'URL

L'application suivante télécharge le contenu d'un URL et vérifie tous les URL qui le composent. Cette application utilise la classe *ValiditeURLConnexion* dotée de deux méthodes de classe chargées de vérifier la validité d'un URL et de lever une exception si nécessaire.

```
import java.io.*;
import java.net.*;
public class ValiditeURLConnexion {
   public static void test(URL url) throws IOException {
    URLConnection c=url.openConnection();
    DataInputStream dis =new DataInputStream(c.getInputStream());
    dis.close();
   }

   public static void test(String s)
      throws MalformedURLException,IOException{
    test(new URL(s));
   }
 }
```

La classe ValiditeURLConnexion

Ce vérificateur d'URL est très simple puisqu'il suppose que :

- les balises autour d'un URL sont spécifiées ainsi : *<A HREF="un url">texte affiché</A>*, les guillemets délimitant la chaîne de caractères spécifiant l'URL à vérifier;
- les URL sont des références absolues et non relatives à un URL courant;

Le principe de cette vérification est le suivant :

1. À partir d'une *URLConnexion* obtenue par la création d'un URL passé dans la ligne de commandes, on teste son type MIME. S'il s'agit d'un *text/html*, son contenu est vérifié. Ce contenu est lu, unité lexicale par unité lexicale.
2. Pour chaque ligne, on cherche le motif *<A*, puis on saute deux unités lexicales correspondant à *HREF* et à =. L'URL à analyser est contenu

entre deux guillemets. On appelle ensuite la méthode *test()* de la classe *ValiditeURLConnexion* pour vérifier la validité de cet URL.

Les réglages de l'extracteur d'unités lexicales (*StreamTokenizer*) sont importants puisqu'ils assemblent en une seule unité lexicale, d'une part, le symbole < avec les mots (voir la méthode *wordChars*) et, d'autre part, tous les caractères compris entre deux guillemets en une chaîne de caractères (voir la méthode *quoteChar*).

```
import java.io.*;
import java.net.*;
public class VerifieURL {
 public static void verifie(URLConnection u) throws IOException {
if (!u.getContentType().equals("text/html")) {
   System.out.println("seuls les text/html sont verifies.");
   System.exit(0);};
   Reader r=new InputStreamReader(u.getInputStream());
   StreamTokenizer st=new StreamTokenizer (r);
   st.wordChars('<','<');
   st.quoteChar('"');
   while (st.nextToken()!=st.TT_EOF) {
    if ((st.ttype== st.TT_WORD) &&
       (st.sval.compareToIgnoreCase("<a")==0)) {
      st.nextToken(); // saute href
      st.nextToken(); // saute =
      st.nextToken(); // extrait l'URL
      System.out.println(st.sval);
      try {ValiditeURLConnexion.test(st.sval);}
      catch (IOException e) {
         System.out.println("URL incorrect :"+st.sval);}
    } // if
   } // while
     r.close();
    }
public static void main(String args[])
  throws MalformedURLException, IOException {
   if (args.length!=1)
      {System.out.println("Usage : java verifieURL <URL>");
       System.exit(0);}
   URLConnection c=null;
   try {
      URL url =new URL(args[0]);
      ValiditeURLConnexion.test(url);
       c= url.openConnection();
       System.out.println(c.getClass().getName());}
   catch (IOException e)
         {System.out.println("URL incorrect : "+args[0]);
          System.exit(0);}
   verifie(c);
   }}
```

Application 12 : Vérificateur d'URL

20.6 Un exemple client-serveur

La communication entre les clients et le serveur s'effectue à travers le réseau Internet. Dans ce premier exemple de client-serveur, les clients ne communiquent pas entre eux. Lorsqu'une application demande une connexion, le serveur lui en construit une. Le service que rend cette connexion est de renvoyer en majuscule tous les messages que le client lui envoie. Un client se déconnecte du serveur par l'envoi du message *fin*.

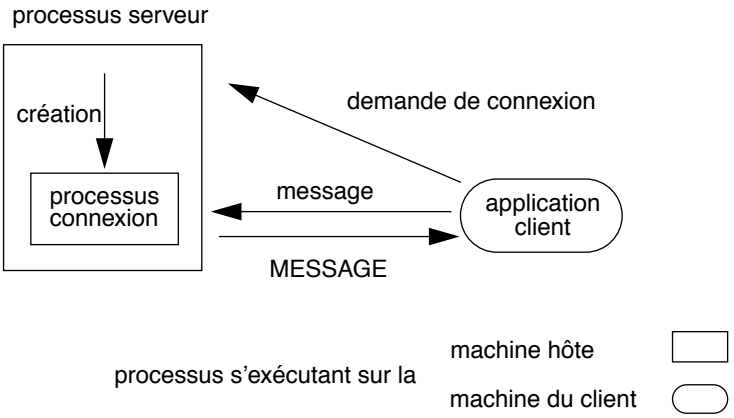


Figure 20.4 Schéma de communication entre processus

La demande de connexion par le client est la création d'une instance de la classe *Socket* indiquant la machine hôte et un numéro de port.

Le lancement du serveur sur l'hôte de nom *mycpu* s'effectue ainsi :

```
>java Serveur
Serveur en ecoute sur le port: 45678
```

Le serveur est en attente de connexions de la part des clients. Un client demande une connexion ainsi :

```
>java Client mycpu
Connexion: mycpu/189.154.12.87 port: 45678
?toto
!TOTO
?lulu
!LULU
?fin
Connexion stoppee par le serveur
```

Le point d'interrogation indique que le client attend un message de l'utilisateur. Le point d'exclamation préfixe le service rendu par la connexion.

Serveur

Le serveur est un processus dont l'une des variables d'instance est une instance de *ServerSocket* initialisée à la création du serveur. La méthode *run()* de ce processus est une boucle interruptible par un signal de la classe *IOException*. Cette boucle effectue le cycle suivant : attente d'une demande de connexion (méthode *accept()*) puis création d'une instance de la classe *Connexion* à partir du *Socket* que lui a renvoyé la méthode *accept()*. C'est la classe *Connexion* qui réalise le service de transformation des caractères en majuscules.

```
import java.io.*;
import java.net.*;
public class Serveur extends Thread {
    protected static final int PORT=45678;
    protected ServerSocket ecoute;
  public Serveur () {
   try {
     ecoute=new ServerSocket(PORT);}
   catch (IOException e) {
     System.err.println(e.getMessage());
     System.exit(1);
   }
   System.out.println("Serveur en ecoute sur le port :"+PORT);
   this.start();
  }
  public void run() {
   try {
     while (true) {
         Socket client=ecoute.accept();
         Connexion c= new Connexion(client);
   }}
   catch (IOException e) {
     System.err.println(e.getMessage());
     System.exit(1);
   }
  }
  public static void main(String[] args) {
   new Serveur();
  }
}
```

La classe Serveur

Connexion

Une instance de la classe *Connexion* est créée par le serveur. Cette *Connexion* réalise la transformation des caractères en majuscules. Il y a un processus de cette classe pour chaque client. Le canal de communication entre le client et une connexion est le socket que le serveur a fourni à la suite d'une demande de connexion d'un client.

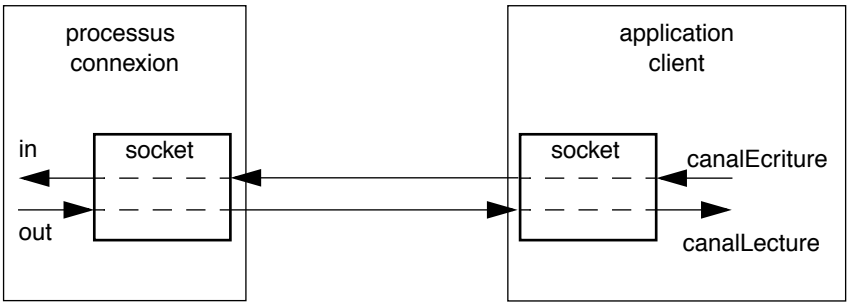


Figure 20.5 Les canaux de lecture, d'écriture et les sockets

Les canaux de lecture et d'écriture sur les sockets sont récupérés par les méthodes *getInputStream()* et *getOutputStream()* de la classe *Socket*.

```
import java.io.*;
import java.net.*;
class Connexion extends Thread{
  protected Socket client;
  protected BufferedReader in;
  protected PrintStream out;
  public Connexion(Socket client_soc) {
   client=client_soc;
   try {in =new BufferedReader(new
                 InputStreamReader(client.getInputStream()));
        out =new PrintStream(client.getOutputStream());}
   catch (IOException e) {
      try {client.close();} catch (IOException e1) {};
      System.err.println(e.getMessage());
      return;
   }
   this.start();
  }

  public void run() {
   String ligne;
   try {
     while(true) {
       ligne=in.readLine(); // lecture avec attente message client
       // test de la fin de connexion demandée par le client
       if (ligne.toUpperCase().compareTo("FIN")==0) break;
       out.println(ligne.toUpperCase()); //réalisation du service
     }}
    catch (IOException e)
       {System.out.println("connexion :"+e.toString());}
    finally
       {try {client.close();} catch (IOException e){};}
  }
}
```

La classe Connexion

Client du serveur

Un client est une application s'exécutant sur une autre[1] machine. La connexion se fait par la demande de création d'un socket sur un hôte et sur un numéro de port. Ce numéro de port doit être identique à celui du serveur.

```
import java.io.*;
import java.net.*;
public class Client {
  protected static final int PORT=45678;

  public static void erreur() {
    System.err.println("Usage: java Client <hostname>");
    System.exit(1);
  }
  public static void main(String[] args) {
   Socket s=null;
   if (args.length!=1) erreur();
   try {
     s=new Socket(args[0],PORT);
     BufferedReader canalLecture=new BufferedReader(new
                     InputStreamReader(s.getInputStream()));
     BufferedReader console=new BufferedReader(new
                     InputStreamReader(System.in));
     PrintStream canalEcriture=
                     new PrintStream(s.getOutputStream());
     System.out.println("Connexion : "+ s.getInetAddress()+
           " port : "+s.getPort());
     String ligne;
     while (true) {
       System.out.print("?");
       System.out.flush();
       ligne=console.readLine(); // saisie message utilisateur
       canalEcriture.println(ligne); //demande du service
       ligne=canalLecture.readLine();//lecture service réalisé
       // test de fin de connexion
       if (ligne==null){
         System.out.println("Connexion stoppee par le serveur");
         break;}
         // le service rendu est affiché à l'écran
        System.out.println("!"+ ligne);
      }
    } // try
    catch (IOException e) {System.err.println(e);}
    finally
      {try {if (s!=null) s.close();}
       catch (IOException e2){}}
   } // main
}
```

Application 13 : Client

1. L'application fonctionne également si le client est sur la même machine que le serveur.

Le client indique la fin d'une connexion par l'envoi du message *fin*. Lorsque ce message est reçu par l'instance de la classe *Connexion*, celle-ci ferme le socket. Le processus termine alors son exécution.

Le client identifie la fin effective de la connexion en détectant la fin du fichier sur le canal de lecture. En effet, la méthode *readLine()* renvoie la chaîne *null* en fin de fichier.

20.7 Télédiscussion

Cet exemple est basé sur une relation client-serveur avec des clients communiquant par l'intermédiaire du serveur, celui-ci maintenant une liste des connexions.

Le fonctionnement de ce serveur est simple. Il envoie à tous les clients le message qu'il a reçu d'un client. Un tel service constitue une télédiscussion entre plusieurs personnes communiquant par l'intermédiaire de leur clavier.

Tous les messages envoyés sont préfixés par le nom du client afin d'identifier les propos des différents interlocuteurs.

La télédiscussion à *n* personnes nécessite le maintien d'une liste des connexions courantes. Cette liste doit être mise à jour à chaque nouvelle connexion d'un client et à chaque départ d'un client de la discussion. Dès qu'une connexion reçoit un message de son client, elle doit accéder à la liste des connexions pour envoyer ce message à tous les clients.

Plusieurs processus accèdent en lecture et en écriture à cette liste des connexions, qui doit être protégée (voir p. 107). Le mot clé *synchronized* garantit cette protection.

Identification et rôle des processus nécessaires à la télédiscussion

La figure 20.6 illustre les processus nécessaires :

- un serveur en attente de connexions : lorsqu'une nouvelle demande arrive, le serveur crée une connexion et l'insère dans la liste, puis attend une nouvelle demande;
- une connexion pour chaque client : cette connexion attend un message de son client puis demande un accès à la liste des connexions afin de leur envoyer le message pour qu'elles le transmettent à leur client respectif. Lorsqu'un client indique une fin de connexion, sa connexion avertit un processus nettoyeur chargé de la retirer de la liste des connexions;
- le processus nettoyeur : il s'exécute sur la machine hôte du serveur comme toutes les connexions. Ce processus se réveille périodiquement

ou est réveillé par une connexion. Son rôle est de tester l'activité d'une connexion. Si celle-ci est inactive, elle est retirée de la liste. Une connexion est inactive si sa méthode *run()* a terminé son exécution;

- l'application client : son rôle est de permettre la saisie des messages de l'utilisateur, et d'afficher les messages de sa connexion (venant en fait d'autres clients via leur connexion respective). Contrairement à l'exemple précédent, où l'émission et la réception de messages étaient synchronisées, ici un client peut rester à « écouter » la discussion, en intervenant de temps en temps; l'écoute et la réception sont donc asynchrones. Un second processus, l'*écouteur* est donc nécessaire;
- le processus écouteur : il s'exécute sur la même machine que le client. Son rôle est d'afficher tous les messages qu'il reçoit de la connexion. Il fonctionne indépendamment de l'utilisateur;

Tous les processus s'exécutant sur la machine hôte accèdent de manière exclusive à la liste des connexions :

- le serveur pour ajouter des connexions;
- le nettoyeur pour en enlever;
- chaque connexion pour répercuter les messages sur toutes les connexions (y compris elle-même) afin que la discussion soit partagée par tous.

Le serveur

Le constructeur de ce processus crée une instance de *ServerSocket* pour gérer les demandes de connexions, une instance de *Vector* pour stocker la liste des connexions et une instance du processus *Nettoyeur*.

```
import java.io.*;
import java.net.*;
import java.util.*;
public class ServeurTD extends Thread {
   protected static final int PORT=45678;
   protected ServerSocket ecoute;
   protected Vector connexions;
   protected Nettoyeur nettoyeur;

  public ServeurTD ()
   { try
       {ecoute=new ServerSocket(PORT);}
     catch (IOException e)
       {System.err.println(e.getMessage());
        System.exit(1);
       }
     System.out.println("Serveur en ecoute sur le port :"+PORT);
     connexions=new Vector();
     nettoyeur=new Nettoyeur(this);
```

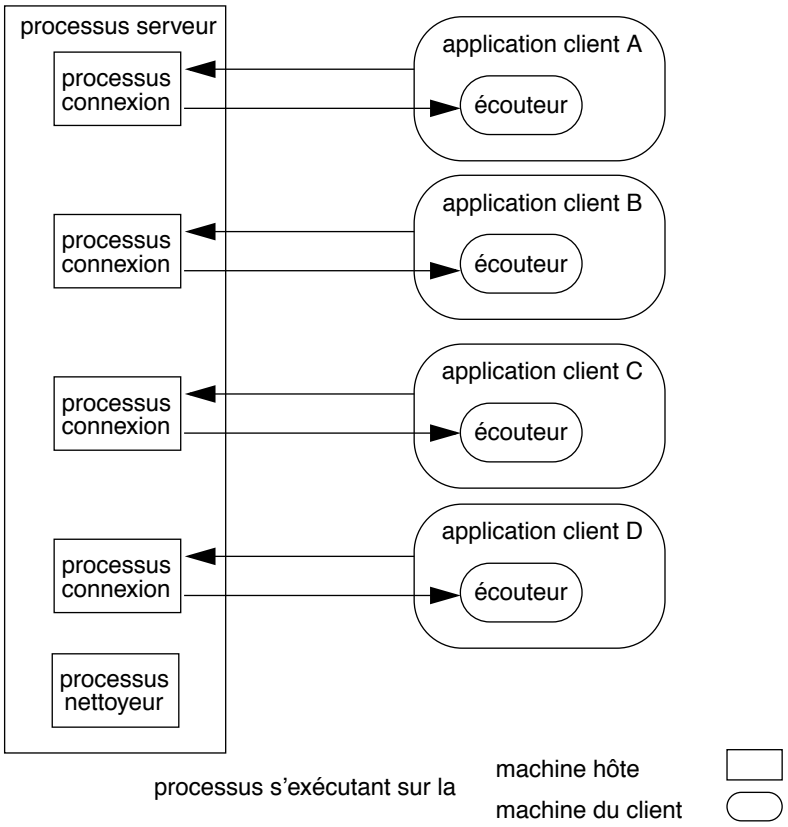


Figure 20.6 Les processus de la télédiscussion

```
      this.start();
   }
   public void run(){
   try{
      while (true){
         Socket client=ecoute.accept();
         System.out.println("Demande de connexion...");
         ConnexionTD c= new ConnexionTD(client,nettoyeur,this);
         synchronized (connexions) {connexions.addElement(c);}
   }}
   catch (IOException e)
       {System.err.println(e.getMessage());
        System.exit(1);
       }
   }
  public static void main(String[] args)
  {new ServeurTD(); }
}
```

La classe ServeurTD

La création d'une nouvelle connexion entraîne son ajout dans la liste *connexions*, cet ajout étant protégé de l'exécution simultanée d'autres processus.

Une connexion

Les instances de cette classe sont également des processus. Le constructeur de cette classe reçoit plusieurs références en paramètres. Le socket est la liaison entre le client et la connexion. La référence au serveur permet l'accès à la liste des connexions; celle du nettoyeur permet son réveil par la connexion.

La méthode *run()* de ce processus attend un message du client sur son socket. Lorsqu'elle en reçoit un, elle demande l'accès exclusif à la liste des connexions, puis écrit sur le socket de chaque connexion de cette liste le message qu'elle a reçu. Pendant cette écriture, il est à noter que toutes les connexions de la liste (si elles sont actives) sont en attente soit d'un message de leur client, soit de l'accès à la liste des connexions (c'est-à-dire qu'elles ont reçu un message, mais ne l'ont pas encore traité). Une connexion non active peut figurer dans la liste car le nettoyeur n'est pas prioritaire pour l'accès à la liste; cela ne produira pas d'erreur car la méthode *println()* de la classe *PrintStream* ne signale pas d'exception en cas de problème.

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

import java.io.*;
import java.net.*;
import java.util.*;

class ConnexionTD extends Thread{
  protected Socket client;
  protected ServeurTD serveur;
  protected BufferedReader in;
  protected PrintStream out;
  protected Nettoyeur nettoyeur;
  public ConnexionTD(Socket client_soc, Nettoyeur n, ServeurTD s)
    {client=client_soc;
     nettoyeur=n;
     serveur=s;
      try{
         in =new BufferedReader(new
               InputStreamReader(client.getInputStream()));
         out =new PrintStream(client.getOutputStream());}
      catch (IOException e){
         try {client.close();} catch (IOException e1) {};
         System.err.println(e.getMessage());
         return;
        }
      this.start();
     }
```

```
public void run()
{ String ligne;
  ConnexionTD c;
  try
    {while(true)
     {ligne=in.readLine();
     // envoyer a tous les clients la ligne
       synchronized(serveur.connexions)
      {for (int i=0;i<serveur.connexions.size();i++)
       {c=(ConnexionTD) serveur.connexions.elementAt(i);
        c.out.println(ligne);}}
      if (ligne.endsWith("FIN")) break;
     }}
  catch (IOException e){}
  finally{
     try {client.close();} catch (IOException e){};
     System.out.println("Fin de connexion...");
     synchronized(nettoyeur) {nettoyeur.notify();}
  }
}}
```

La classe ConnexionTD

Le nettoyeur

Le nettoyeur est un processus dont le constructeur requiert une référence au serveur pour l'accès à la liste des connexions. La méthode *run()* de ce processus est une boucle sans fin (l'arrêt du serveur entraînera l'arrêt du nettoyeur). Cette boucle fait attendre le nettoyeur cinq secondes puis demande l'accès exclusif à la liste des connexions pour en vérifier l'activité. Une connexion inactive est retirée de la liste. Le parcours de cette liste se fait de la fin au début car la méthode de retrait de la liste compacte les éléments restant dans la liste[1].

La veille du nettoyeur peut être interrompue par une notification provenant d'une connexion.

```
import java.io.*;
import java.net.*;
import java.util.*;
class Nettoyeur extends Thread{
  protected ServeurTD serveur;

  protected Nettoyeur(ServeurTD serveur)
   {this.serveur=serveur;
    this.start();
   }
```

1. Le faire dans l'autre sens impliquerait, en cas de retrait, de ne pas incrémenter l'indice de la boucle. Dans les boucles *for*, cet indice est mis à jour automatiquement et une bonne pratique de programmation consiste à ne pas le modifier dans le corps de la boucle. Un second intérêt de ce parcours à l'envers réside dans le temps passé à conserver exclusivement une ressource partagée : l'évaluation de la taille de la liste n'a lieu qu'une seule fois pour initialiser l'indice, et non à chaque itération dans un parcours du début à la fin. Le parallélisme est donc augmenté.

```
  public synchronized void run()
  { while(true)
      {try {this.wait(5000);}
       catch (InterruptedException e){};
       synchronized(serveur.connexions)
        {for (int i=serveur.connexions.size()-1;i>=0;i--)
         {ConnexionTD c= (ConnexionTD)
serveur.connexions.elementAt(i);
          if (!c.isAlive())
            {serveur.connexions.removeElementAt(i);
             System.out.println("Fin de connexion : OK");
            }
         }} // for
      } // while
  }
 }
```

La Classe Nettoyeur

L'application client

L'interface de cette application (application 14, figures 20.7 et 20.8) est une fenêtre composée de deux champs de texte : un en bas pour la saisie et un en haut pour l'affichage des messages. Cette fenêtre contient également deux boutons *envoi* et *stop*.

La classe *AppliClient* est une sous-classe de la classe *Frame*. Le constructeur de l'application client appelle le constructeur de la classe *Frame*, puis demande la création d'un socket sur la machine sur laquelle fonctionne le serveur. Le processus *écouteur* est créé pour permettre la lecture sur le socket de manière asynchrone avec la saisie des messages de l'utilisateur.

La méthode *actionPerformed()* traite les messages de fin de ligne dans la zone de saisie et l'appui sur le bouton *envoi*. Le traitement de ce message consiste à écrire sur le socket le texte que l'utilisateur a tapé. L'appui sur *stop* signale la demande de fin de connexion de la part d'un client puis arrête l'application client.

```
import java.io.*;
import java.net.*;
import java.awt.*;
import java.awt.event.*;
public class AppliClient extends Frame implements ActionListener {
    public static final int PORT=45678;
    Socket s;
    PrintStream canalEcriture;
    TextField entree;
    TextArea  visu;
    Button envoi,stop;
    Panel boutons;
    String Nom;

  public AppliClient(String n, String host) {
```

```
      super("client"+ " "+n);
      try {
        Nom=n;
        s=new Socket(host,PORT);
        canalEcriture=new PrintStream(s.getOutputStream());

        // construction de l'interface graphique
        entree=new TextField();
        visu=new TextArea();
        visu.setEditable(false);
        this.setLayout(new BorderLayout());
        this.add("North",visu);
        this.add("Center",entree);
        entree.addActionListener(this);
        boutons=new Panel();
        envoi=new Button("envoi");
        stop =new Button("stop");
        boutons.add(envoi);
        envoi.addActionListener(this);
        boutons.add(stop);
        stop.addActionListener(this);
        this.add("South",boutons);
        this.pack();
        this.show();
        // la connexion est etablie :
        visu.setText("Connexion : "+ s.getInetAddress()+
            " port : "+s.getPort());

        // lancement du processus accedant en lecture au socket
        Ecouteur ecoute=new Ecouteur(s,visu);
      } catch  (IOException e) {visu.setText(e.toString());}
    } // constructeur AppliClient

  public void actionPerformed(ActionEvent e) {
     if ((e.getSource()==envoi) || (e.getSource()==entree)) {
        canalEcriture.println(Nom+">"+entree.getText());
        entree.setText("");
      }
     if (e.getSource()==stop)  {
        canalEcriture.println(Nom+">FIN");
        System.exit(0);
      }
    } // actionPerformed

    public static void main(String[] args) {
     Frame f= new AppliClient(args[0],args[1]);
    } //main
  }
```

Application 14 : Client de la télédiscussion

Le client prend un paramètre sur la ligne de commande, le nom de l'utilisateur, afin de préfixer chaque message par son auteur.

Le processus écouteur

Le processus écouteur est assez simple. Il attend les messages provenant du socket et les affiche au fur et à mesure dans le champ de texte adéquat de la fenêtre graphique.

```
import java.io.*;
import java.net.*;
import java.awt.*;
class Ecouteur extends Thread{
 BufferedReader canalLecture;
 TextArea  visu;

 public Ecouteur(Socket s,TextArea  out) throws IOException
 { canalLecture= new BufferedReader (new
                   InputStreamReader(s.getInputStream()));
   visu=out;
   this.start();
  }

 public void run()
 { String ligne;
   try
      {while (true)
       {ligne=canalLecture.readLine();
        if (ligne==null) break;
        visu.append("\n"+ligne);
       }}
    catch (IOException e) {visu.setText(e.toString());}
    finally {visu.setText("connexion interrompue par le serveur");}
   }
}
```

Le processus Ecouteur

Les copies d'écrans ci-dessous montrent une télédiscussion entre Gilles et André. Les deux clients fonctionnent respectivement sur Windows 95 et sur Sun.

Figure 20.7 Télédiscussion : client sous Sun/solaris

Figure 20.8 Télédiscussion : client sous Windows 95

20.8 Programmation distribuée sur Internet

Ces exemples vous ont montré l'accès à des ressources Internet et à des applications client-serveur distribuées sur ce même réseau. Un premier type d'accès illustré dans ces exemples nécessitait un gestionnaire de contenu chargé de donner du sens au nœud d'information. Un autre type d'accès consistait à interpréter soi-même le contenu comme, par exemple, le programme de vérification des URL contenus dans un nœud d'information.

Les exemples client-serveur nous ont montré la facilité d'écriture de clients qui ne sont pas des butineurs, et de serveurs qui ne sont pas seulement des serveurs de documents.

La relation client-serveur fournit la base de la programmation distribuée sur Internet. La distribution de services sur Internet passe par la distribution des clients. Les nœuds d'information dynamiques (c'est-à-dire un nœud d'information statique associé à une applet) favorisent la diffusion d'applets clients. Les applets sont les meilleures candidates pour cet objectif car leur fonctionnement ne peut perturber les ressources du système sur lequel elles s'exécutent. L'un des intérêts majeurs de Java est la facilité de programmation d'applications client-serveur qui ne se résument pas à butiner des URL ici ou là.

Nous aborderons, dans d'autres chapitres, la distribution d'objets et l'invocation de méthodes à distance.

D'ici là, à vous d'imaginer de nouveaux services...

Partie IV

La méthode : développer en Java

> Je m'étais fourré dans la tête qu'envol et locomotion ne pouvaient être réalisés que dans un processus en deux temps. D'abord monter aussi haut que je le pouvais, ensuite me propulser de l'avant. Je m'étais entraîné à faire la première de ces choses, et je croyais pouvoir accomplir la seconde à partir de la première. Mais en réalité, la seconde annulait ce qui la précédait.
>
> Paul Auster, *Mr Vertigo*.

Chapitre 21

Conception des classes

Dans les chapitres précédents, nous avons abondamment utilisé les classes de l'API Java, et créé des classes pour réaliser des applets et de petites applications. L'objectif de cette partie est de présenter des éléments méthodologiques de programmation orientée objet pour permettre le développement de projets plus ambitieux. Nous passerons en revue les principaux concepts de l'approche objet tels qu'ils apparaissent dans Java et nous montrerons comment les employer pour développer un logiciel de qualité. Le présent chapitre revient en détail sur la notion de classe et ses différents aspects, nous étudierons ensuite la notion de réutilisation, puis l'architecture du logiciel.

21.1 Les origines

Pour bien comprendre à quoi sert une classe dans un système logiciel, il nous semble important de remonter à des notions qui sont à l'origine de l'« invention » des classes : la modularité, l'encapsulation et les types abstraits.

Modularité et encapsulation

La modularité est une technique de développement, que l'on retrouve du reste dans la plupart des disciplines d'ingénierie, qui consiste à voir un système complexe comme un assemblage de parties plus simples, appelées modules. Pour que la technique soit efficace, il est primordial que la construction de chaque module soit aussi indépendante que possible de celle des autres modules.

L'encapsulation est une technique pour réaliser cette indépendance entre modules ; elle est basée sur une nette séparation entre les parties publique et privée de chaque module. La partie publique d'un module, souvent appelée inter-

face, définit les services que ce module offre aux autres. La partie privée, appelée implantation, définit la façon dont le module réalise les services offerts; cette partie contient des structures de données et les instructions exécutables nécessaires.

Lorsqu'un module veut obtenir un service d'un autre module, il s'adresse toujours et uniquement à l'interface publique de ce dernier, il n'a aucun accès à sa partie privée. La figure 21.1 illustre ce principe.

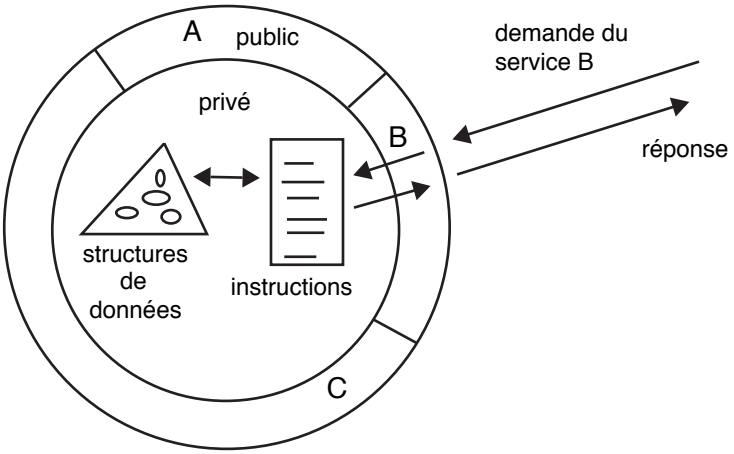


Figure 21.1 Demande d'un service à un module

L'intérêt de cette séparation public-privé réside dans le fait que la partie privée peut être réalisée et modifiée sans que les clients du module ne soient perturbés, puisqu'ils voient toujours la même interface. Un logiciel modulaire avec encapsulation sera donc plus facile à développer et surtout à maintenir qu'un logiciel monolithique dont toutes les parties sont interdépendantes.

Les types abstraits

Le principe de base des types abstraits est de décrire un type de données uniquement par les opérations qu'on peut lui appliquer, sans s'occuper de l'ensemble de ses valeurs. Ainsi, lorsqu'un programmeur utilise des nombres entiers dans un programme, il ne s'intéresse généralement qu'aux opérations possibles sur les entiers et peut complètement ignorer la manière dont ils sont représentés sous forme de chaînes de bits. La même observation peut s'appliquer pour n'importe quel type de données. Peu importe qu'un rectangle soit représenté par les coordonnées du coin supérieur gauche et du coin inférieur droit ou encore par les coordonnées du coin supérieur droit, la largeur et la hauteur; ce qui compte, c'est de pouvoir définir un rectangle, le translater, l'agrandir, tester s'il contient un point donné, etc.

Il s'ensuit une manière de définir les types de données en deux phases : premièrement, on s'intéresse aux opérations et à leur signification, c'est la partie abstraite de la définition; deuxièmement, on choisit une représentation pour ce type de données et on écrit le corps (les instructions) des opérations, c'est la concrétisation (aussi appelée réification). Une même définition abstraite peut donner lieu à plusieurs réalisations concrètes suivant les moyens à disposition, les contraintes, les normes de programmation, etc.

Si l'on reprend l'exemple précédent du type *rectangle*, on commencera par définir un ensemble d'opérations telles que :

- *rect : (entier, entier, entier, entier) → rectangle;*
- *translation : (rectangle, entier, entier) → rectangle;*
- *agrandissement : (rectangle, entier, entier) → rectangle;*

qui servent à produire des rectangles. L'opération *rect(X, Y, L, H)* fournit le rectangle dont le coin supérieur gauche est (*X*, *Y*), la largeur *L* et la hauteur *H*. L'opération *translation(R, DX, DY)* fournit le rectangle obtenu par translation du rectangle *R* d'une distance horizontale *DX* et verticale *DY*. L'opération *agrandissement(R, DL, DH)* fournit le rectangle situé au même endroit que *R* mais dont la largeur et la hauteur sont celles de *R* agrandies de *DL* et *DH* respectivement.

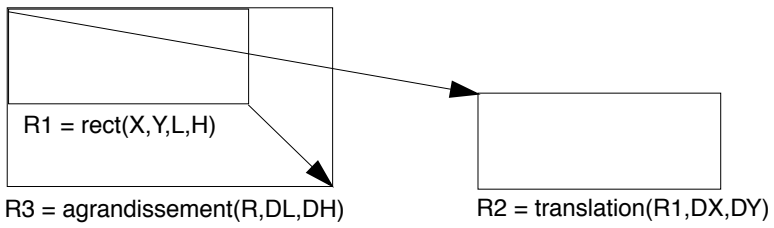


Figure 21.2 Méthodes agissant sur des rectangles

Les opérations suivantes sont des opérations d'accès qui fournissent des propriétés d'une valeur de type rectangle.

- *gauche : (rectangle) → entier;*
- *haut : (rectangle) → entier;*
- *bas : (rectangle) → entier;*
- *droite : (rectangle) → entier;*
- *largeur : (rectangle) → entier;*
- *hauteur : (rectangle) → entier;*

Pour compléter la description d'un type abstrait, il faut expliquer la signification des opérations. On peut bien sûr le faire à l'aide d'un texte en français mais si

l'on veut une explication absolument non ambiguë, il est préférable de recourir à un ensemble d'équivalences telles que :

1. *Gauche(rect(X, Y, L, H))* ↔ *X.*
2. *Largeur(rect(X, Y, L, H))* ↔*L.*
3. *Gauche(translation(R, DX, DY))* ↔ *gauche(R) + DX.*
4. *Gauche(agrandissement(R, DL, DH))* ↔ *gauche(R).*
5. *Largeur(agrandissement(R, DL, DH))* ↔ *largeur(R) + DL.*
6. *Droite(R)* ↔ *gauche(R) + largeur(R).*

Ces équivalences définissent formellement la signification des opérations, de sorte qu'un utilisateur du type abstrait peut comprendre précisément ce qui va se passer lorsqu'il utilisera telle ou telle opération. Par exemple, l'équivalence 5 exprime le fait que le rectangle obtenu par agrandissement est plus large de DL que le rectangle original.

La définition complète d'un type abstrait peut s'avérer assez fastidieuse, car il faut évidemment énoncer toutes les propriétés des opérations. Le lecteur intéressé trouvera de nombreux ouvrages de génie logiciel traitant de ce sujet, comme par exemple [12] ou [1].

21.2 Objets et classes

Les langages à objets ont intégré les concepts de modularité, d'encapsulation et de type abstrait dans leurs notions d'objet et de classe.

Un objet est un module

Chaque objet est un module qui contient des données « entourées » par des opérations appelées méthodes qui permettent d'accéder à ces données et de les modifier. L'accès aux données d'un objet ne peut se faire qu'au travers des méthodes, comme le veut le principe d'encapsulation. Les noms et paramètres des méthodes forment la partie publique de l'objet alors que les structures de données internes et le corps des méthodes en forment la partie privée.

Exemple : un objet représentant un rectangle

La structure de données interne est formée de quatre variables entières représentant les coordonnées du coin supérieur gauche (haut, gauche), la largeur et la hauteur du rectangle. L'interface publique de l'objet est formée des méthodes qui lui sont applicables (*deplacer(...)*, *positionner(...)*, ...).

Une classe est un schéma et un générateur d'objets

Une classe est un schéma d'objet qui en décrit la partie privée et la partie publique. Par instanciation, on peut fabriquer des objets obéissant tous à ce schéma. En Java, ce sont l'opération *new* et les constructeurs qui réalisent l'instanciation.

```
class Rectangle {
   /* schéma de la partie privée */
   private int gauche, haut, largeur, hauteur;

   /* schéma de la partie publique */
   public void positioner(int posX, int posY)
         { haut = posX; gauche = posY; }
   public void deplacer(int dv,int dh)
         { haut += dv; gauche += dh; }
   // etc.
   }
...
/* instanciation */
Rectangle r = new Rectangle();
```

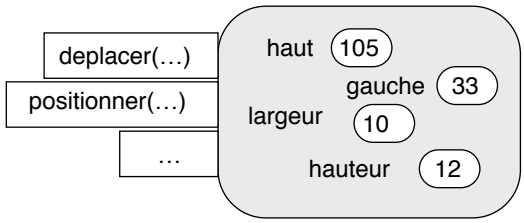


Figure 21.3 Structure d'un objet

Une classe est elle-même un objet

Une classe n'est pas seulement un élément syntaxique utilisé pour définir en un seul endroit la structure de tous les objets de même nature, c'est également un objet qui peut posséder ses propres variables internes et ses méthodes. En fait, chaque classe est une instance de la classe *java.lang.Class*.

Le modificateur *static* introduit les variables et méthodes de classe dans une définition de classe. Par exemple :

```
class Rectangle {

/* variables et méthodes de classe */
static double echelle       // variable communes à tous les
rectangles
public static void defEchelle(int e) {
   echelle = e; }

/* variables et méthodes d'instance */
public double surface() {
      return hauteur * largeur * echelle; // utilise echelle
   }
...
}
```

La variable *echelle* appartient à la classe elle-même, elle est partagée par toutes les instances. Dans l'exemple ci-dessus, la méthode *surface()* utilise la variable

de classe *echelle*. Les variables de classe se rapprochent des variables globales qui existent dans les langages procéduraux comme Pascal ou C.

Pour exécuter une méthode de classe, il faut la préfixer du nom de sa classe. Par exemple :

```
Rectangle.defEchelle(4.25);
```

En fait, tout se passe comme si *defEchelle()* était une méthode d'instance d'un objet appelé *Rectangle*.

Une classe spécifie un type abstrait

L'ensemble des méthodes de la partie publique d'une classe est en fait la spécification d'un type abstrait puisqu'il détermine toutes les opérations possibles sur les objets de la classe. La définition de *Rectangle* ci-dessus définit un type abstrait *Rectangle* dont la spécification est composée des opérations :

- *positionner : (Rectangle, int, int) → Rectangle;*
- *deplacer : (Rectangle, int, int) → Rectangle;*
- *etc.*

Cette spécification est bien sûr partielle car elle ne contient pas les équations qui permettraient de donner un sens précis à chaque opération. Elle est cependant suffisante pour utiliser correctement ce type dans un programme (à condition qu'une description précise des méthodes soit disponible, par exemple sous forme de commentaires).

Une déclaration de la forme :

```
Rectangle r;
```

qui apparaît dans un programme indique que l'on peut utiliser les instructions (après la création de l'objet *r*) :

```
r.positionner(x, y)
r.deplacer(dx, dy)
etc.
```

La réalisation est découplée de la spécification

Nous avons déjà dit qu'un même type abstrait, par exemple *Rectangle*, peut être représenté de différentes manières. Selon la représentation (soit *haut-gauche*, *largeur*, *hauteur*, soit *haut-gauche*, *bas-droite*, soit une autre), on obtiendra différentes versions de la classe *Rectangle*. Cependant, chacune d'entre elles doit offrir les méthodes prévues par le type abstrait; les parties publiques seront donc toutes les mêmes alors que les parties privées présenteront des variantes.

Exemple de deux classes *Rectangle* (en gras : l'interface abstraite commune),

```
/*  variante 1.  */
```

```
class Rectangle {
   private int haut, bas, gauche, droite;

public void positionner(int x, int y)
{ int l = droite - gauche;
  int h = bas - haut;
   haut = x; gauche = y;
   bas = haut + h;
   droite = gauche + l;
}
public int bas()
{ return bas; }

public defBas(int b)
{ bas = b; }

public int surface()
{ return(bas-haut)*(droite-gauche);
}

/* variante 2.   */

class Rectangle {
   private int haut, gauche, hauteur, largeur;

public void positionner(int x, int y)
{ haut = x; gauche = y; }

public int bas()
{ return haut + hauteur; } // les coordonnées croissent vers le bas !

public defBas(int b)
{ hauteur = b - haut; }

public int surface()
{ return (hauteur * largeur); }
```

21.3 Interfaces

La notion d'interface en Java correspond à une spécification de type abstrait sans aucune implantation, c'est-à-dire à une spécification de type décrite uniquement par la liste de ses opérations. Une interface est composée d'un ensemble d'en-têtes de méthodes et éventuellement de constantes. Par exemple :

```
interface Vendable {
   public int prix();
   public int rabais(int quantite);
   public void changerPrix(int prix);
}
```

La réalisation concrète d'une interface est déléguée à une classe devant posséder (ou hériter) une méthode concrète pour chaque méthode abstraite de l'interface. Par exemple :

```
class Moto implements Vendable {
   private String marque, modele;
   private int prix, cylindree, poids, puissance;
   // méthodes servant à impanter Vendable
   public int prix() { return prix; }
   public int rabais(int quantite) {
      if(quantite > 5) return prix/10; else return 0;}
   public void changerPrix(int prix) {
      this.prix = prix; }
   // autres méthodes
   Moto(String m, int p) {marque = m; prix = p; }
   void imprime() {System.out.println(marque + " prix: " + prix);}
}
```

La notion d'interface pousse à l'extrême le découplage entre la spécification et l'implantation d'un type.

Une interface définit un type de données qui peut servir à typer des variables ou des paramètres de méthodes. On peut affecter à une variable déclarée d'un type interface I tout objet d'une classe qui implante I. Par exemple :

```
Vendable v;
v = new Moto("Honda", 16678);
```

De même, partout où un paramètre de type I est attendu, on peut passer un objet d'une classe qui implante I.

21.4 Spécification d'une classe

Lorsqu'on utilise une classe, on doit savoir non seulement quelles sont les méthodes à appeler et leurs paramètres mais aussi ce que font précisément ces méthodes. Or, la partie publique d'une classe ne constitue pas une spécification suffisante du comportement des objets. On peut bien sûr imaginer qu'une méthode *retrait()* d'une classe *CompteEnBanque* ne va pas ajouter de l'argent sur le compte mais plutôt en soustraire. Cependant, le seul nom d'une méthode est bien insuffisant en tant que description de son fonctionnement.

Ainsi, l'opération *positionner* pour un rectangle a-t-elle pour effet de changer la forme du rectangle ? Les deux nombres donnés en paramètre définissent-ils la nouvelle position du coin supérieur gauche, du coin inférieur droit ou du centre ?

Description textuelle

Une manière simple de décrire l'effet d'une méthode consiste à écrire un texte en français. Un tel texte présente les avantages et inconvénients de la langue

naturelle : d'un côté, il est facilement lisible par un être humain et peut s'appuyer sur toutes sortes de connaissances implicites; d'un autre côté, il peut présenter des ambiguïtés telles que deux lecteurs pourront l'interpréter différemment. La langue naturelle peut également s'avérer trop lourde pour décrire des faits qui s'expriment très simplement dans un langage plus formel comme les mathématiques.

C'est pourquoi nous proposons d'ajouter une description plus formelle de chaque méthode et de chaque classe. Il existe pour cela de nombreux formalismes plus ou moins abstraits et plus ou moins simples à manipuler; nous examinerons la technique des pré- et postconditions et des invariants.

Les préconditions

Une méthode travaille à partir d'un objet *(this)* et des paramètres qui lui sont fournis. Cependant, certaines valeurs de paramètres ou variables d'instance de l'objet peuvent ne pas avoir de sens pour la méthode. Par exemple, la méthode qui modifie la largeur d'un rectangle ne peut rien faire d'un paramètre négatif, de même une méthode calculant la moyenne des nombres contenus dans une liste n'a pas de sens pour une liste vide. Les préconditions fixent donc les conditions dans lesquelles une méthode peut s'exécuter correctement. Elles dépendent de l'état de l'objet *this* (état défini par ses variables d'instance) et des paramètres.

Exemples :

```
public void defLargeur(int lrg) {
   // PRE-CONDITION: lrg > 0
...
}
public void deplace(int dh, int dv) {
   // PRE-CONDITION: this.gauche() + dh > 0
   //            et  this.haut() + dv > 0
   ...
}
```

La condition ci-dessus garantit qu'on ne tente pas un déplacement qui amènerait le rectangle hors du système de coordonnées (qui commence à (0,0)).

On remarque sur ce dernier cas, que la précondition est exprimée à l'aide des méthodes publiques *gauche()* et *haut()* et non des variables privées *gauche* et *haut*. Cette expression est donc publique, ce qui signifie qu'on peut la tester depuis l'extérieur de la classe *Rectangle*, par exemple avant d'appeler la méthode.

```
Rectangle r;
...
r = new Rectangle;
...
if (r.gauche() + deplH > 0 & r.haut() + deplV > 0)
   { r.deplace(deplH, deplV); }
else ...
```

Alternativement, si la précondition est testée dans la méthode et déclenche une exception, la méthode appelante saura, si elle capte l'exception, que l'expression de précondition est fausse.

Une telle précondition est indépendante de la représentation interne de l'objet, elle est donc compatible avec le principe d'encapsulation et fournit une information sur le comportement d'une méthode de la classe compréhensible indépendamment des détails internes de la classe.

Les post-conditions

Si la précondition garde la porte d'entrée d'une méthode, la postcondition indique ce qu'est devenu l'objet après l'exécution de la méthode, s'il a été transformé ou non, et quel est l'éventuel résultat produit.

Nous utiliserons deux pseudovariables : *resultat* qui représente le résultat retourné par la méthode, s'il y en a un, et *this_post* qui représente la valeur de l'objet *this* après l'exécution de la méthode.

Considérons tout d'abord le cas d'une méthode qui ne modifie pas l'objet mais fournit un résultat de calcul.

```
public int surface() {
// PRE-CONDITION: aucune
// POST-CONDITION: resultat = this.hauteur() * this.largeur()
   ...
}
```

La postcondition détermine la valeur qui sera retournée comme résultat en fonction de l'état de l'objet (ici sa largeur et sa hauteur). Il faut bien voir que la postcondition n'est pas là pour dire comment le calcul est fait; le calcul dépend de la représentation interne de l'objet. Comme dans le cas des préconditions, on exprime les postconditions à l'aide des paramètres, de l'objet *this* et des méthodes publiques de manière à rendre cette condition « lisible » de l'extérieur de la classe.

Lorsque la méthode modifie l'objet, on peut exprimer ce changement soit par son effet sur les variables d'instance de l'objet, soit en disant ce qui se passe lorsqu'on applique des méthodes d'accès sur l'objet transformé. La deuxième technique est la meilleure car elle ne met pas en jeu la représentation interne de l'objet.

Complétons ainsi nos définitions des méthodes *defLargeur* et *deplacer* de la classe *Rectangle*.

```
public void defLargeur(int l) {
   // PRE-CONDITION: l > 0
   // POST-CONDITION: this_post.largeur() = l
   //    (le nouveua rectangle a une largeur l)
```

```
   ...
}
public void deplace(int dh, int dv) {
   // PRE-CONDITION: this.gauche() + dh > 0
   //               et  this.haut() + dv > 0
   // POST-CONDITION: this.gauche() = this_pre.gauche() + dh
   //                et this.haut() = this_pre.haut() + dv
   ...
}
```

Les invariants

Les invariants sont des expressions logiques qui doivent rester vraies durant toute la vie d'un objet. Un invariant ne décrit pas ce que fait une méthode mais ce qu'est et doit rester un objet de cette classe. Dans notre classe *Rectangle*, on pourrait ajouter l'invariant :

```
pour tout Rectangle r: r. hauteur() >= 0 et r.largeur() >= 0
```

pour préciser qu'en toute circonstance ni la largeur ni la hauteur d'un rectangle ne peuvent devenir négatives. Dans une classe *RectanglePlat* destinée à représenter des rectangles dont la largeur est supérieure à la hauteur on aura l'invariant :

```
pour tout RectanglePlat c: c.hauteur() <= c.largeur()
```

Les invariants induisent bien évidemment l'obligation pour toutes les méthodes de la classe de les maintenir à « vrai ». Par exemple, dans le cas de notre classe *RectanglePlat,* la méthode *defLargeur(int l)* devra d'une manière ou d'une autre garantir que la largeur reste supérieure à la hauteur.

21.5 Exceptions

Le mécanisme d'exceptions offre un moyen élégant de structurer la gestion des erreurs ou des cas exceptionnels qui empêchent le bon déroulement des opérations. D'un point de vue purement syntaxique, le mécanisme d'exceptions simplifie l'écriture du code en remplaçant :

- des cascades de *if* imbriqués;
- la transmission d'indicateurs de succès ou d'échec des méthodes.

Dans le premier cas, un fragment de code de la forme :

```
if (erreur) traiterErreur();
else {
   ...
   if (erreur) traiterErreur();
   else {
      ...
      if (erreur) traiterErreur();
```

```
        else {
           ... etc.
        }
     }
  }
```

devient :

```
  try {
     if (erreur) throw new MonException();
     ...
     if (erreur) throw new MonException();
     ...
     if (erreur) throw new MonException();
     ... etc.
     }
  catch (MonException me) traiterErreur();
```

Il n'y a plus d'imbrication de *if* et le traitement de l'erreur n'est spécifié qu'une seule fois, ce qui garantit qu'il est uniforme pour toutes les erreurs d'un type donné.

La propagation des exceptions aux méthodes appelantes remplace les codes de retour que la méthode appelante devrait tester pour s'assurer que tout s'est bien passé :

```
     if (canal.lire(desBytes) != 0) traiterErreur();
     else {  ...
             if canal2.ecrire(desBytes) != 0) traiterErreur();
             else { ... suite
             }
     }
```

devient :

```
  try {
     canal.lire(desBytes); // il n'y a plus de code à tester
     ...
     canal2.ecrire(desBytes);
     ...}
  catch (ExceptionCanal ec) traiterErreur();
```

Malgré ces possibilités attrayantes, il ne faut pas abuser de l'utilisation des exceptions. Le traitement d'exceptions doit rester limité aux cas... exceptionnels.

La propagation et la capture des exceptions sont relativement coûteuses en temps de calcul. Le *try... catch* ne doit donc pas servir de nouvelle structure de contrôle, les *if*, *switch*, *while* et *for* étant amplement suffisants pour exprimer

les algorithmes et s'exécutant beaucoup plus rapidement. Il ne faut pas non plus en faire un « *goto* » déguisé.

D'autre part, certains types d'exceptions ne devraient pas être capturés car ils représentent des erreurs profondes, souvent irréparables. Les capturer reviendrait à poursuivre l'exécution du programme dans un état instable qui ne conduirait qu'à de nouvelles erreurs. La hiérarchie des exceptions prédéfinies en Java est ainsi subdivisée en deux parties :

- les erreurs qui sont des sous-classes de *Error*;
- les exceptions qui sont des sous-classes de *Exception*.

21.6 Modéliser avec les classes

Tout composant logiciel (application, applet, package, etc.) est construit à partir de classes qui déterminent tous les types d'objets qui vont être utilisés par le composant. Ces classes sont soit définies par le développeur, soit préexistantes et choisies dans une bibliothèque de composants. L'identification des différents types d'objets et donc des classes qui vont intervenir dans la fabrication d'un composant logiciel est une tâche primordiale de conception qui détermine l'architecture du logiciel.

Des classes pour les concepts

Une manière d'aborder le problème du choix ou de la conception des classes consiste à considérer la structure de classes comme un modèle de la réalité extérieure au programme. Chaque classe représente alors une catégorie d'objets concrets ou abstraits (un concept) du domaine d'application. Dans un programme qui traite de facturation on pourra, par exemple, trouver des classes *Commande*, *Facture*, *Rappel*, *Client*, *Marchandise*, etc.; dans un système de gestion d'interface graphique, comme *java.awt*, on trouvera des classes *Fenetre*, *Bouton*, *Panneau*, *Menu*, etc., qui correspondent aux divers éléments d'interface utilisateur.

Empressons-nous de préciser que tout concept du domaine d'application ne doit pas forcément donner lieu à une classe et que, réciproquement, toute classe ne correspond pas nécessairement à un concept du domaine.

Les relations entre classes

Les concepts d'un domaine d'application ne sont en général pas isolés car il existe toutes sortes de relations entre eux. Par exemple, une facture est adressée à un client, elle correspond à un ensemble de commandes. Une commande est passée par un client, elle porte sur une marchandise pour une quantité et un prix donnés. Si nous considérons toujours qu'une structure de classe est un modèle de la réalité, ce modèle doit prendre en compte les relations entre concepts.

Il est commode de représenter graphiquement les relations entre classes. De nombreux formalismes ont été développés à cet effet, du modèle Entités-associations des années 70 au modèle unifié UML [http://www.rational.com/uml] qui tend à devenir le standard actuel. Dans ce formalisme, on représentera les relations entre commandes, clients, marchandises et factures que nous avons énoncées plus haut de la manière suivante :

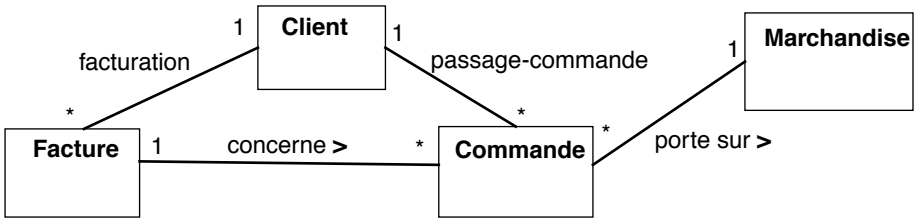


Figure 21.4 Diagramme des relations entre classes

Les relations sont représentées par des lignes étiquetées par un nom. Les symboles « 1 » et « * » aux extrémités indiquent les cardinalités uniques ou multiples des relations. Par exemple, la relation passage-commande peut associer un client à une ou plusieurs commande(s) mais chaque commande à au plus un client. Le symbole « **>** » suivant un nom de relation indique simplement un sens de lecture (« une facture concerne une commande » et non « une commande concerne une facture »)

Implantation des relations

La manière la plus directe de réaliser une relation entre objets consiste à utiliser des variables d'instance. La structure de classe esquissée ci-dessous implante les relations décrites précédemment.

```
class Client {
   String nom, adresse;
   ...
}
class Facture {
   Client client;      // relation "facturation" avec le client
   Commande[] commandes; // relation "concerne" avec les commandes
   double montant();
   ...
}
class Commande {
   Date date;
   Client client;      // relation "passage-commande"
   Marchandise marchandise;
   int quantite;
   double prix;
   ...
}
```

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

Remarquons toutefois que la représentation par variables d'instance est à sens unique. Dans notre exemple, on peut atteindre un client depuis une facture, grâce à la variable *client*, par contre on ne peut pas trouver à partir d'un client toutes les factures le concernant. Il faudrait pour cela ajouter une variable *Facture [] factures* dans la classe *Client* et s'assurer qu'en tout temps, il y a cohérence entre *client* et *factures*. Sur un diagramme, on peut indiquer le ou les sens de parcours autorisés en ajoutant des pointes de flèches aux extrémités des lignes figurant une relation.

Il est également possible de représenter certaines relations à l'aide de méthodes. Par exemple, la relation *facturation* de *Facture* à *Client* peut être calculée par une méthode, en se basant sur l'hypothèse que le client à facturer est celui qui a passé la première commande faisant l'objet de la facture.

```
class Facture {
   Client client() {
      return commandes[0].client; }
   ...
```

Nous verrons dans le chapitre consacré aux structures de données qu'il existe d'autres possibilités de réaliser des relations en utilisant les classes collections.

Parmi les relations entre concepts, la relation générique-spécifique ou « est un » (une chaise est un meuble, un ordinateur est une machine, etc.) joue un rôle particulier que nous étudierons dans le cadre de l'héritage entre classes. En règle générale, on évitera de représenter cette relation par une variable ou une méthode.

Un exemple complet : le jeu du serpent

Nous présentons ici un exemple complet d'applet où interviennent plusieurs classes. Il s'agit de créer une applet permettant de jouer au jeu du serpent (voir figure 21.5). L'espace de l'applet sera divisé en une grille sur laquelle évolue le serpent. Cette grille est bordée par un cadre. Le serpent se déplace dans les quatre directions (haut, bas, droite, gauche). La direction du déplacement est indiquée à l'aide du clavier au moyen des flèches. Le serpent a initialement la taille d'une case.

Sur le terrain se trouvent des appâts que le joueur doit faire manger au serpent, en passant dessus. Le serpent grandit lorsqu'il a mangé un appât. Il grandit en laissant immobile l'extrémité de sa queue. En temps normal, le serpent se déplace de manière continue (en conservant sa longueur).

Le but du jeu est de rendre le serpent le plus long possible. Le jeu se termine lorsque le serpent heurte le cadre ou lorsqu'il se mord lui-même.

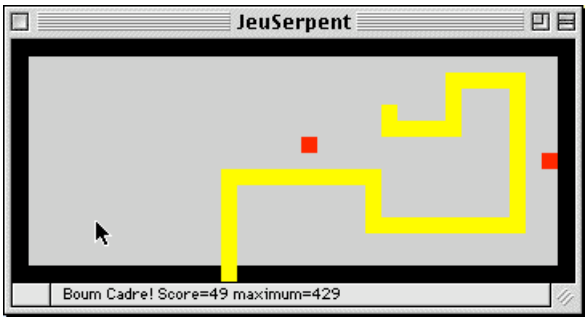


Figure 21.5 Le jeu du serpent

Cette solution est basée principalement sur quatre classes d'objets : *Terrain*, *PointMobile*, *Serpent* et *Appat*. Le diagramme ci-dessous décrit les relations qui nous intéressent entre ces classes.

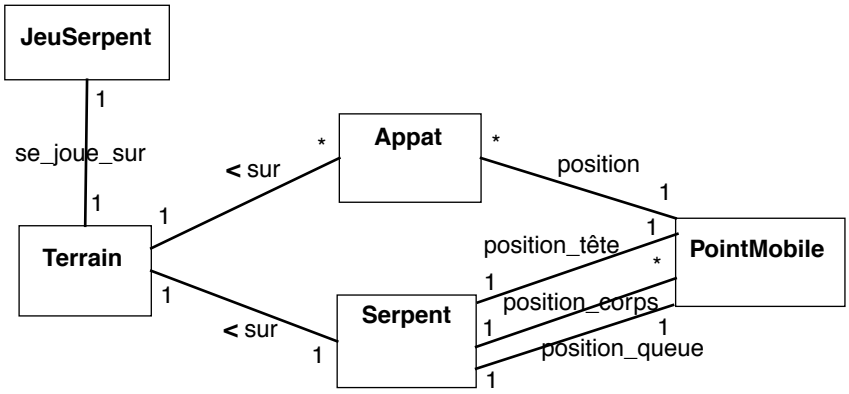


Figure 21.6 Structure des classes de l'applet JeuSerpent

Un terrain est le lieu où se déroule le jeu, il est composé d'une surface de jeu dont les cases sont arrangées en lignes et colonnes. Chaque case du terrain peut contenir rien, une partie du cadre, une partie du serpent ou un appât.

Les opérations sur un terrain consistent essentiellement à lire ou modifier le contenu d'une case, et à dessiner le terrain en entier ou juste une case. La classe définit un ensemble de constantes qui permettent d'écrire du code plus lisible.

La position du corps du serpent est représentée par les cases du terrain qui contiennent l'une des constantes *DROITE*, *GAUCHE*, *BAS* ou *HAUT*. Ces constantes indiquent la direction à suivre pour arriver à la case suivante du serpent. On peut donc dire que ces cases matérialisent la relation *position_corps*.

```
class Terrain {

// constantes
 static final Color
```

```
    COULEUR_VIDE = Color.lightGray, COULEUR_CADRE = Color.black,
    COULEUR_APPAT = Color.red,      COULEUR_SERPENT = Color.yellow;
  static final char CADRE='C', VIDE ='V', APPAT='A', DROITE='D',
                    GAUCHE='G', HAUT='H', BAS='B';

// representation du terrain par une matrice de caracteres

  private char t[][];
  public int  largeur,hauteur;
  public int  tailleCase; // taille d'une case en pixel

  private Graphics g; // la feuille de desin de l'applet

    Terrain(int l, int h, int tc, Graphics appletG){
       g = appletG;
       largeur = l;hauteur = h;tailleCase = tc;
       t = new char [l+1][h+1];
       // initialisation du terrain: cadre au bord et vide a
l'interieur
       for (int i = 0; i <= largeur; i++)
          for (int j = 0; j <= hauteur ; j++) {
             if ((i==0) | (i==largeur) | (j==0) | (j==hauteur))
                t[i][j]= CADRE;
             else t[i][j]=VIDE;
          }
    }
    public int taille() { return (largeur+hauteur)/2;}

    public char contenuCase (PointMobile p) {return t[p.x][p.y];}

    public void modifCase (PointMobile p, char v) {t[p.x][p.y]=v;}

    public void redessine(){
       PointMobile p = new PointMobile(0,0);
       for (int i = 0; i <= largeur; i++)
          for (int j = 0; j <= hauteur ; j++){
             p.deplacer(i,j);
             dessineCase(p);
          }
    }
    // Modifie le contenu d'une case et la redessine
    public void majCase(PointMobile p, char v) {
       modifCase(p, v);
       dessineCase(p);
    }
    // Dessine une case, la couleur indique le contenu
    public void dessineCase (PointMobile p) {
       switch (t[p.x][p.y]) {
          case APPAT: g.setColor(COULEUR_APPAT); break;
          case CADRE: g.setColor(COULEUR_CADRE); break;
          case VIDE: g.setColor(COULEUR_VIDE); break;
          default: g.setColor(COULEUR_SERPENT);
       }
```

```
        g.fillRect( p.x*tailleCase, p.y*tailleCase,
                    tailleCase , tailleCase);
    }
```

Un *pointMobile* représente une position sur le terrain, il peut se déplacer dans les quatre directions droite, gauche, haut, bas.

```
class PointMobile {
   public int x, y;

   PointMobile(int initx, int inity){
     x=initx;y=inity;
   }
   public void deplacer(int cx, int cy){
     x=cx;y=cy;
   }
   public void avance(char direction){
      switch (direction) {
         case Terrain.DROITE: x++;break;
         case Terrain.GAUCHE: x--;break;
         case Terrain.BAS: y++;break;
         case Terrain.HAUT: y--;break;
      }
   }
}
```

Un serpent évolue dans un terrain, on peut définir la direction dans laquelle le serpent doit avancer (*defDirection*) et faire avancer le serpent (*avance*). Le serpent fournit aussi un message qui indique son état.

La représentation interne du serpent est composée des deux variables de type *PointMobile* : *pTete* et *pQueue*, qui implantent les relations *position_tête* et *position_queue* du diagramme de classes. La variable *terrain* implante la relation *sur* entre *Serpent* et *Terrain*.

```
class Serpent {
   String       msg;
   PointMobile  pTete = new PointMobile(3,3); // position tete
   PointMobile  pQueue = new PointMobile(3,3); // position queue
   int          longueur = 1;
   Terrain      terrain;
   char         directionTete = Terrain.BAS;
   char         directionQueue;
   int          grandir = 0;

   Serpent (Terrain t){
      terrain=t;
      terrain.majCase(pTete,directionTete);
   }

   // Avance la tete du serpent dans la direction 'directionTete'
```

```
    public boolean avance(){
       boolean fini=false;
       terrain.modifCase(pTete,directionTete);
       pTete.avance(directionTete);
       switch (terrain.contenuCase(pTete)) {
          case Terrain.CADRE:
             msg = " Boum Cadre!";
             fini=true; break;
          case Terrain.HAUT: case Terrain.BAS:
          case Terrain.DROITE: case Terrain.GAUCHE:
             msg = " Queue mordue!";
             fini=true; break;
          case Terrain.APPAT: grandir+=terrain.taille();
       }
       terrain.majCase(pTete,directionTete);

       // Avance la queue du serpent si necessaire
       if  (grandir==0) {// la queue doit avancer
          directionQueue = terrain.contenuCase(pQueue);
          terrain.majCase(pQueue,Terrain.VIDE);
          pQueue.avance(directionQueue);
       } else { // la queue doit rester fixe
          grandir--; longueur++;
       }
       return fini;
    }

    public void defDirection(char d){
       directionTete=d;
    }
}
```

Un appât occupe une case d'un terrain. Un appât disparaît au bout d'un laps de temps pour réapparaître ailleurs. La variable *terrain* implante la relation *sur* entre *Serpent* et *Terrain*.

```
class Appat{
   private PointMobile position = new PointMobile(0,0);
   private int vieAppat = 0;
   private Terrain terrain;

   Appat (Terrain t){
      terrain = t;
   }
   public void vivre(){
      if  (vieAppat > 0) vieAppat--;
      if  (vieAppat==0) {
         if (terrain.contenuCase(position)==Terrain.APPAT)
            // effacement d'un appat non consomme
            terrain.majCase(position,Terrain.VIDE);
         // l'appat mort reapparait sur une autre case vide
         do {
```

```
            position.x=
               (int)Math.floor(Math.random()*(terrain.largeur));
            position.y=
               (int)Math.floor(Math.random()*(terrain.hauteur));
         } while (terrain.contenuCase(position) != Terrain.VIDE);
         vieAppat=terrain.taille()*4;
         terrain.majCase(position, Terrain.APPAT);
      }
   }
}
```

Un jeu est composé d'un terrain, d'un serpent, de deux appâts et d'un processus d'animation (*runner*) qui met à jour régulièrement la position du serpent et des appâts ainsi que le score. La variable *terrain* implante la relation *se_joue_sur* entre *JeuSerpent* et *Terrain*. Quant aux variables *s*, *a1* et *a2*, elles permettent d'accéder directement au serpent et aux appâts qui sont sur le terrain de jeu.

C'est l'applet qui par son processus *runner* anime le jeu. La méthode *run()* met à jour le serpent et les appâts puis s'endort un moment et recommence.

```
public class JeuSerpent extends java.applet.Applet
      implements Runnable, KeyListener, MouseListener{

    Thread runner;
    Terrain t;
    Serpent s;
    Appat a1,a2;

    int tailleCase = 10; // taille en pixel d'une case
    int vitesse = 50;// nb. de millisecondes entre chaque mouvement
du serpent
    int hauteur = 100,largeur = 200;
    public void init() {
      String parametre;

      this.addKeyListener(this);
      this.addMouseListener(this);

      // lecture et initialisation parametres

      parametre = getParameter("vitesse");
      if (parametre != null)
         vitesse = 1000 - Integer.parseInt(parametre);

      parametre = getParameter("hauteur");
      if (parametre != null) hauteur = Integer.parseInt(parametre);

      parametre = getParameter("largeur");
      if (parametre != null) largeur = Integer.parseInt(parametre);

      parametre = getParameter("grain");
      if (parametre != null) tailleCase =
```

```
Integer.parseInt(parametre);

      resize(largeur, hauteur);

      t = new Terrain((int)(this.getSize().width / tailleCase )-1,
                 (int)(this.getSize(). height / tailleCase)-1,
                 tailleCase,this.getGraphics());
      setBackground(Terrain.COULEUR_VIDE);
      s = new Serpent (t);
      a1 = new Appat (t);
      a2 = new Appat (t);
    }

    public void start() {
      if (runner == null); {
         runner = new Thread(this);
         runner.start();
      }
   }
    public void stop() {
        if (runner != null); {
             runner.stop();
             runner = null;
       }
   }
   public void run() {
      int scoreMax = (t.largeur-1)*(t.hauteur-1);
      while (true) {
         if (s.avance()) {
            this.showStatus(s.msg+" Score="+s.longueur+
                         " maximum="+scoreMax);
            stop();
         }
         a1.vivre();
         a2.vivre();
         try { Thread.sleep(vitesse);}
         catch (InterruptedException e) { }
         showStatus("Score="+s.longueur+" maximum="+scoreMax);
      }
   }
   public void paint(Graphics g) {
      t.redessine();
   }

   /// implantation de KeyListener ///

   public void keyPressed(KeyEvent e) {
      int key = e.getKeyCode();
      switch (key) {
         case KeyEvent.VK_DOWN:
                 s.defDirection(Terrain.BAS); break;
         case KeyEvent.VK_UP:
```

```
                s.defDirection(Terrain.HAUT); break;
            case KeyEvent.VK_LEFT:
                s.defDirection(Terrain.GAUCHE); break;
            case KeyEvent.VK_RIGHT:
                s.defDirection(Terrain.DROITE); break;
        }
    }
    public void keyTyped(KeyEvent e) {}
    public void keyReleased(KeyEvent e) {}

    /// implantation de MouseListener ///

    public void mouseClicked(MouseEvent e) {

        int xm = e.getX();
        int ym = e.getY();

        if (Math.abs(xm - s.pTete.x*tailleCase) > Math.abs(ym -
s.pTete.y*tailleCase)) {
            // la distance horizontale (x) entre la tete et le click est
            // superieure a la distance verticale (y)
            if  (xm>s.pTete.x*tailleCase)
s.defDirection(Terrain.DROITE);
            else s.defDirection(Terrain.GAUCHE);
        }
        else { // la distance en y est plus grande
            if (ym>s.pTete.y*tailleCase) s.defDirection(Terrain.BAS);
            else s.defDirection(Terrain.HAUT);
        }
    }
    public void mouseEntered(MouseEvent e) { }
    public void mouseExited(MouseEvent e) { }
    public void mousePressed(MouseEvent e){ }
    public void mouseReleased(MouseEvent e){ }

}
```

Applet 55 : Le jeu du serpent

Le choix des classes retenu pour cette applet n'est bien entendu pas le seul possible; il correspond à un choix de concepts du domaine (le terrain, les points, le serpent, les appâts). Nous aurions pu en faire un autre où apparaîtraient les classes *Bord*, *Trajectoire*, *Score*, etc. Néanmoins, le fait que notre logiciel soit structuré en classes tend à améliorer sa maintenabilité et son évolutivité. Ainsi, vous devriez être capable de créer de nouvelles versions de l'applet qui répondent à de nouvelles spécifications telles que :

- le serpent change de couleur chaque fois qu'il mange un appât;
- par endroits, le serpent passe au-dessus du sol, ce qui l'autorise à se croiser lui-même;

- la surface de jeu est une « sphère ». Si le serpent sort à gauche, il revient par la droite (il n'y a plus de bords);
- les appâts se déplacent et cherchent à fuir le serpent;
- il y a deux serpents symétriques sur la surface de jeu!
- etc.

Chapitre 22

Mécanismes pour la réutilisation

S'il est bon de savoir définir de nouvelles classes, le développement orienté objet est également basé sur la réutilisation de classes existantes. Dans la partie III, nous avons abondamment utilisé les classes de l'environnement Java, ce qui nous a évité de réécrire un gestionnaire d'interfaces, un gestionnaire d'entrées/sorties, un gestionnaire de communications, etc. Dans ce chapitre, nous commencerons par étudier en détail le mécanisme d'héritage qui permet de dériver de nouvelles classes à partir de classes existantes. Nous nous intéresserons également aux notions de polymorphisme et de généricité qui sont d'un grand intérêt pour construire des composants réutilisables.

22.1 Héritage et sous-classes

Les langages à objets possèdent pratiquement tous un mécanisme d'héritage servant à dériver une sous-classe à partir d'une ou plusieurs classes existantes. La sous-classe :

- hérite de toutes les caractéristiques de son ou ses « ancêtre(s) »;
- peut avoir des caractéristiques propres;
- peut redéfinir des caractéristiques héritées.

Si la notion d'héritage est en elle-même fort simple, elle est également très puissante et possède plusieurs usages fort différents, nous y reviendrons plus loin. L'héritage permet de réutiliser une classe existante (en particulier le code déjà écrit pour les méthodes) et de l'étendre à la fois structurellement et comporte-

mentalement. On dispose de ce fait d'une technique de réutilisation très souple. Passons tout d'abord en revue les principales caractéristiques de l'héritage en Java.

Héritage et redéfinition en Java

La définition d'une sous-classe en Java se fait par « extension » d'une classe existante. Une sous-classe **hérite** des variables (la structure) et des méthodes (le comportement) de sa super-classe. La structure de la sous-classe est donc composée de ses propres variables **et** de celles provenant de sa super-classe, de même pour les méthodes.

Par exemple, si les classes *Point* et *PointTopographique* sont définies comme :

```
class Point {
   int x, y;
   Point(int px, int py) {x = px; y = py};
   public double distanceOrigine() {...}
   public double distance(Point pt) {...}
   public void deplacer(int dx, int dy) {...}
}
class PointTopographique extends Point {
   int alt;
   PointTopographique(int px, int py, int a)
      {super(px, py); alt = a;}
   public int altitude() {...}
   public void ajusterAltitude(int a) {...}
}
```

un objet de la classe *PointTopographique* possédera trois variables d'instance *x*, *y* et *alt*, et on pourra lui appliquer les méthodes :

- *distanceOrigine, distance* et *deplacer* (méthodes héritées);
- *altitude, ajusterAltitude* (méthodes propres).

Redéfinition et liaison dynamique

Une méthode héritée peut également être redéfinie dans la sous-classe. Il suffit pour cela de la réécrire en conservant la même signature (même nom et mêmes paramètres).

Par exemple, la classe *PointTopographique* peut redéfinir la méthode *distanceOrigine* de manière à ce que la distance ne soit plus la distance horizontale (calculée à partir des coordonnées *x, y*), mais une distance dans l'espace à trois dimensions tenant compte de l'altitude du point.

La redéfinition atteint toute sa puissance lorsqu'elle est combinée avec la liaison dynamique. Rappelons tout d'abord qu'une variable ou un paramètre déclaré de classe *C* désigne toujours un objet de la classe ou de l'une des sous-classes de *C*. Le fragment de code ci-dessous est donc correct :

```
Point p = new Point(3, 5);
PointTopographique q = new PointTopographique(6, 4, 12);
Point r = q;        // r fait référence à un PointTopographique
```

Le mécanisme de liaison dynamique fait que lorsque l'instruction :

```
r.distanceOrigine()
```

sera exécutée, le système d'exécution choisira la « bonne » méthode *distanceOrigine()*, en fonction de la classe de l'objet désigné par *r*, indépendamment du type déclaré de *r*. En l'occurrence, ce sera la méthode *distanceOrigine()* de la classe *PointTopographique* qui sera choisie.

La déclaration de *r* comme *Point* et le fait qu'on ne puisse affecter à *r* que des objets de sous-classes de *Point* garantit que l'objet désigné par *r* possède forcément une méthode *distanceOrigine*, et qu'il n'y aura pas d'erreur d'exécution du genre « cette méthode n'existe pas pour cet objet ».

Redéfinition et réutilisation de méthodes

De même qu'une classe peut étendre sa super-classe, une méthode peut étendre le comportement d'une méthode héritée plutôt que de la redéfinir complètement. La méthode *m()* définie dans une classe peut invoquer la méthode *m()* de sa super-classe (ou toute autre méthode redéfinie de la super-classe) en employant la notation *super.m()*. Par exemple, si la classe *Point* possède une méthode *dessiner(Graphics g)*, sa sous-classe *PointTopographique* peut étendre cette méthode en définissant :

```
void dessiner(Graphics g) {
   super.dessiner(g); // réutilisation
   g.drawString(...); // extension: affiche l'altitude à côté du
point
}
```

Redéfinition et surcharge

Contrairement à d'autres langages, en Java, la redéfinition d'une méthode doit avoir strictement les mêmes paramètres que l'originale, sinon il n'y a pas redéfinition mais surcharge. Si, dans la classe *PointTopographique* on définit une méthode :

```
public double distance(PointTopographique p) {...}
```

on ne redéfinit pas la méthode *distance()* héritée de *Point* mais on la surcharge, c'est-à-dire que notre classe possède désormais deux méthodes *distance* :

```
public double distance(Point pt) {...} // héritée
public double distance(PointTopographique p) {...} // nouvelle
```

l'une s'appliquant si le paramètre effectif *p* est un *Point* et l'autre si le paramètre effectif est un *PointTopographique*.

La règle consiste à choisir la méthode la plus spécifique, c'est-à-dire celle dont les paramètres sont le plus « bas » dans la hiérarchie des classes.

```
Point p1, p2;
PointTopographique pt1, pt2;
d=p1.distance(p2); // exécute distance de Point car p1 est un Point
d=p1.distance(pt1); // idem, un paramètre d'une sous-classe est
admis
d=pt1.distance(p1); // exécute la méthode distance(Point pt) héritée
                    // car p1 est un Point
d=pt1.distance(pt2);// exécute distance(PointTopographique p) car
                  // pt1 est un PointTopographique (liaison
dynamique)
                  // et pt2 est un PointTopographique
```

Cette règle n'exclut pas les ambiguïtés. Considérons deux méthodes *m(Point p, PointTopographique pt)* et *m(PointTopographique pt, Point p)* et *p1*, *p2* et *p3* qui sont *PointTopographique*. L'instruction *p1.m(p2, p3)* est alors ambiguë car les deux méthodes sont applicables mais aucune n'est plus spécifique que l'autre. En Java, ces ambiguïtés provoquent une erreur à la compilation.

Classes abstraites

Les classes abstraites sont des classes dont certaines méthodes, dites abstraites, sont déclarées (nom et paramètres) mais ne possèdent pas de corps d'instructions. Les différents types de classe et de méthode sont les suivants :

- une méthode est **concrète** si elle comporte un corps d'instructions (éventuellement vide {});
- une méthode est **abstraite** si elle ne possède pas de corps d'instruction. Une telle méthode doit être nécessairement implantée dans une sous-classe désirée concrète;
- une classe est **abstraite** si elle définit une ou plusieurs signatures de méthodes abstraites;
- une classe est **implicitement abstraite** si elle ne définit aucune signature de méthode abstraite et si elle hérite de méthodes abstraites de ses super-classes directes ou non;
- une classe est **concrète** si toutes ces méthodes héritées ou non sont concrètes. Seules les classes concrètes sont instanciables.

Le modificateur *abstract* doit préfixer la définition des classes et méthodes abstraites. Les classes abstraites ne sont pas instanciables, elles peuvent cependant servir à typer des variables (ou des paramètres). Une telle variable fera toujours référence à un objet d'une sous-classe concrète.

22.2 Utilisations de l'héritage

Dans l'un de ses articles [7], B. Meyer, concepteur du langage orienté objet

Eiffel, distingue douze manières réellement différentes d'utiliser l'héritage en programmation orientée objet. Nous ne les expliciterons pas toutes mais mettrons en évidence quelques points qui nous semblent importants. Nous distinguerons essentiellement deux types d'utilisation de l'héritage : l'héritage d'implantation et l'héritage de modélisation.

Héritage d'implantation

L'intérêt de l'héritage tient évidemment à l'économie d'écriture de code. On ne réinvente pas ce qui existe et qui nous satisfait déjà partiellement. De plus, on peut compléter ce qui est hérité, voire le redéfinir, de manière à réaliser le comportement voulu. L'extension est donc une technique de réutilisation relativement souple qui se distingue du « tout ou rien ».

Suppression du traitement des cas

Du point de vue de l'architecture du logiciel, l'héritage combiné à la liaison dynamique peuvent nettement simplifier l'écriture des méthodes en évitant les traitements de cas particuliers. Considérons une classe *Piece* qui représente des pièces fabriquées par une usine et supposons que ces pièces peuvent être soit simples, soit composées d'autres pièces. Si l'on ne définit qu'une seule classe pour représenter toutes les pièces, on obtient une définition de la forme :

```
class Piece {
   String nom;
   boolean estComposee; // la pièce est-elle composée ?
   int poids;
   int nbComposantes;
   Piece [] composantes;
   ...
   int poidsTotal() {
        if (estComposee) {
           int pt = 0;
           for (int i = 0; i < nbComposantes; i++)
              pt += composantes[i].poidsTotal();
           return pt;
        } else return poids;
   }
   ...
```

Cette classe présente deux problèmes : premièrement, certaines variables d'instance sont utilisées seulement si la pièce est simple (*poids*) et d'autres seulement si elle est composée (*nbComposantes* et *composantes*); deuxièmement, la méthode *poidsTotal* doit tester le genre de la pièce pour calculer son poids. Il en est de même pour toutes les méthodes qui doivent se comporter différemment suivant le genre de pièce. S'il y avait encore d'autres types de pièces, on verrait proliférer les instructions *switch* dans le corps des méthodes.

Pour obtenir du code plus simple, et donc moins sujet à erreur, introduisons une petite hiérarchie de classes : la classe *Piece* contient tout ce qui est commun à

tous les genres de pièce. Chaque sous-classe définit la structure et les traitements particuliers à un type de pièce. *Piece* est abstraite car les pièces sont forcément soit simples soit composées, donc il n'y a pas d'instance de *Piece*.

```
abstract class Piece {
   String nom;
   abstract int poidsTotal();
}
class PieceSimple extends Piece {
   int poids;
   int poidsTotal() {return poids}
}
class PieceComposee extends Piece {
    int nbComposantes;
    Piece [] composantes;
    int poidsTotal() {
             int pt = 0;
             for (int i = 0; i < nbComposantes; i++)
                pt += composantes[i].poidsTotal();
             return pt;
    }
}
```

Nous obtenons un code beaucoup plus clair, qui ne comporte plus ni variables inutilisées ni tests du genre de pièce dans les méthodes. En fait, c'est le mécanisme de liaison dynamique qui fait le travail de sélection à notre place. Une instruction :

```
p.poidsTotal()
```

appellera soit la méthode *poidsTotal()* de *PieceSimple* soit celle de *PieceComposee,* en fonction de la classe de l'objet désigné par *p*.

Factorisation

L'exemple ci-dessus illustre également le processus de factorisation des propriétés communes à plusieurs classes. La variable *nom* existe pour toute pièce, qu'elle soit simple ou composée, c'est pourquoi elle est définie dans *Piece*. On peut également factoriser des méthodes ou des parties de méthodes. Par exemple, la méthode *affiche* de *Piece* :

```
void affiche() {
   System.out.println("nom:" + nom +
                      " poids: " + this.poidsTotal());
```

et la méthode *affiche* de *PieceSimple* :

```
void affiche() {
   System.out.println("Pièce simple ");
   super.affiche() // utilise la partie commune
}
```

L'intérêt de la factorisation est de définir les parties communes à plusieurs classes en un seul endroit. Ceci garantit automatiquement la cohérence des caracté-

ristiques communes car il n'y a pas à vérifier qu'elles sont bien définies de la même manière pour tous les types de pièces.

Héritage de modélisation

Être ou avoir?

Si nous reprenons l'optique selon laquelle les classes forment un modèle du domaine d'application, il est alors intéressant de considérer la relation entre une sous-classe et une classe comme un lien « est un » entre des concepts du domaine. Par exemple, un technicien **est un** employé d'une organisation, autrement dit, l'ensemble des techniciens est inclus dans l'ensemble des employés. Ainsi la classe *Technicien* devrait être une sous-classe de *Employe*. Par contre, un employé **fait partie** d'une organisation mais n'est pas une organisation, on peut aussi dire qu'une organisation **a** des employés. Dès lors *Employe* ne devrait pas être une sous-classe de *Organisation* et la relation entre ces classes (voir figure 22.1) correspond à celle que nous avons vue au point 21.6.

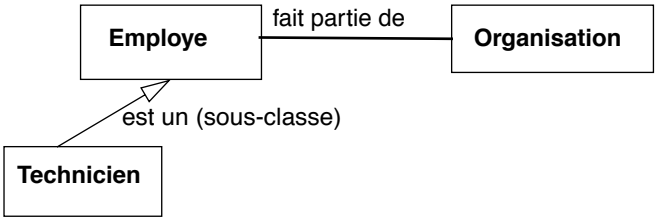


Figure 22.1 Deux natures de liens entre classes

Cette vision de l'héritage clarifie l'architecture du logiciel, car la hiérarchie des classes correspond assez naturellement à celle des concepts du domaine. Du reste, dans certaines disciplines scientifiques comme la botanique ou la zoologie, il existe déjà des hiérarchies complètes de classification des objets (appelées taxonomies).

Modéliser ou implanter?

Il arrive parfois que la vision taxonomique s'oppose à l'héritage d'implantation qui, comme nous l'avons vu, a pour objectif de réutiliser un maximum de code déjà écrit. Prenons l'exemple des deux classes *Carre* et *Rectangle*. Si l'on ne pense qu'en termes d'implantation, on définira tout d'abord *Carre* avec trois variables : *x*, *y* (coordonnées du coin supérieur gauche) et *cote* (longueur des côtés). Puis on dira que *Rectangle* hérite de *Carre* et on ajoutera une variable *coteHorizontal* pour représenter la longueur du côté horizontal alors que *cote* servira pour la longueur du côté vertical.

```
class Rectangle extends Carre {
   int coteHorizontal;
   ... }
```

La hiérarchie ainsi créée est exactement à l'opposé de ce que nous connaissons en géométrie où un carré est en fait un rectangle particulier dont les deux côtés sont de même longueur. Une solution plus proche du domaine de la géométrie consisterait à définir *Carre* comme une sous-classe de *Rectangle*. Dans ce cas, la sous-classe n'ajoutera aucune propriété à celles héritées mais définira la contrainte *hauteur = largeur*.

Négocier

Il est trop strict d'imposer le principe de l'héritage basé sur le lien « est un », car il existe bien des cas où ce lien n'est pas évident. Dans notre exemple du point 22.1, la classe *PointTopographique* est une sous-classe de *Point*. Or, d'un point de vue géométrique, on ne peut pas dire que l'ensemble des points topographiques tel que nous l'avons défini (deux coordonnées plus une altitude) est un sous-ensemble de l'ensemble des points du plan (deux coordonnées). Malgré cela, on peut se dire qu'un point topographique est bien une sorte de point.

En résumé, il est préférable de rester le plus proche possible d'une hiérarchie correspondant au lien « est un » sans en faire un dogme absolu.

Héritage multiple

Java ne permet pas l'héritage multiple dans la hiérarchie des classes, par contre une classe peut à la fois hériter d'une autre et implanter une ou plusieurs interfaces. Cette situation correspond à un cas fréquent d'utilisation de l'héritage multiple dans les langages qui l'autorisent. Une classe *C* hérite de *A* et de *B* mais n'utilise que l'implantation des méthodes de *A* et redéfinit celles provenant de *B*. *B* n'est là que pour apporter son interface. Par exemple, une classe *PileParTableau* qui hérite de *Pile* et de *Tableau*, utilisera l'implantation de *Tableau* pour gérer sa structure de données internes, et l'interface de *Pile* dont elle redéfinira les méthodes *empiler*, *depiler*, etc. Ce genre d'héritage asymétrique correspond en Java à une classe *C* qui hérite de *A* et implante l'interface *B*.

La situation que Java ne couvre pas est celle où l'implantation des deux superclasses *A* et *B* serait réellement utilisée.

22.3 Utilisation des classes abstraites

Comme leur nom l'indique, ces classes représentent des abstractions, donc des concepts plus généraux que les classes concrètes. On les utilisera principalement pour former le sommet d'une hiérarchie de classes qui représentent différentes variantes du concept le plus général. Dans notre exemple des pièces du point 22.2, la classe abstraite *Piece* possède deux sous-classes : *PieceSimple* et *PieceComposee*. Ceci représente bien le fait qu'une pièce concrète est soit simple soit composée, et ne peut pas être d'une autre nature. La classe abstraite est ici le

sommet d'une taxonomie stricte où toute pièce doit forcément être classée dans l'une des sous-classes.

La classe *Dictionary* du package *java.util* nous offre un autre cas d'utilisation d'une classe abstraite. *Dictionary* définit ce qu'est un dictionnaire (une structure où l'on peut stocker des paires clé-valeur) mais n'impose aucune réalisation. La classe *Hashtable* est une sous-classe concrète de *Dictionary* qui, elle, définit une structure de stockage (une table de hachage) et redéfinit concrètement les méthodes de stockage et de recherche de *Dictionary*. L'idée est de fournir un cadre général et de laisser les développeurs réaliser des classes concrètes qui satisfont des besoins particuliers, par exemple implanter les classes *ArbreBinaire* ou *B_Arbre* comme autres sous-classes de *Dictionary*.

22.4 Polymorphisme

Héritage et polymorphisme

D'après les règles de typage de Java, si une variable est déclarée de type C, on peut lui affecter un objet de C ou de n'importe quelle sous-classe de C. *A fortiori*, si la variable est de type *Object*, on peut lui affecter n'importe quel objet, car toute classe est sous-classe de *Object*. Cette propriété permet de créer des structures polymorphes composées d'objets provenant de diverses classes. Par exemple, la déclaration :

```
Piece[] composantes;
```

indique que le tableau *tabPieces* contiendra des objets de *PieceSimple* ou de *PieceComposee*. Par contre, il sera interdit d'y mettre des objets de *Point* ou de *Rectangle,* qui ne sont pas des sous-classes de *Piece*. Le polymorphisme est ici contrôlé (limité) par la classe *Piece*. Le polymorphisme nous évite la création d'autant de structures qu'il y a de types d'objets à stocker ou à traiter. Sans polymorphisme, nous aurions dû définir un tableau pour les pièces simples et un autre pour les pièces composées.

Les deux classes *Vector* et *Hashtable* permettent de créer des structures de données polymorphes sans restriction car leurs méthodes acceptent des paramètres de type *Object*. On peut, par exemple, mettre dans un même *Vector* : un *Rectangle*, une *Applet* et... un *Vector*.

Interfaces et polymorphisme

Les interfaces offrent une seconde manière de créer des structures polymorphes contrôlées indépendamment de la hiérarchie d'héritage. Si une variable est déclarée d'un type interface I, on peut lui affecter un objet de n'importe quelle classe qui implante I.

Si les classes *Moto*, *Montre* et *Casserole* implantent toutes l'interface *Vendable*, un tableau de type *Vendable[6]* pourra contenir deux motos, trois montres et une casserole, bien que leur classe respective n'ait aucune super-classe commune, mis à part *Object*.

22.5 Généricité et interface

Un composant est dit générique s'il peut travailler sur différents types de données, par exemple une méthode de tri pourrait aussi bien trier des entiers que des chaînes de caractères ou d'autres types encore. La méthode ne peut cependant pas trier des objets pour lesquels aucune opération de comparaison n'est disponible.

Dans cet exemple, nous allons définir une méthode de tri qui s'applique à n'importe quel tableau d'objets, pour autant que ceux-ci possèdent une méthode *plusPetitOuEgal()*. Nous définissons donc une interface :

```
interface Comparable {
   public boolean plusPetitOuEgal(Object c);
}
```

Définissons ensuite une classe *Trieuse* qui possède une méthode de classe *tri()* :

```
class Trieuse {
   public static void tri(Comparable[] x) {
      Comparable tmp;
      /* un simple tri par échange */
      for (int i = 0; i<x.length - 1; i++)
         for (int j = i+1; j<x.length; j++)
            if (x[j].plusPetitOuEgal(x[i])) {
               tmp = x[i];   x[i] = x[j];  x[j] = tmp; }
   }
}
```

Nous voulons trier des motos selon leur prix, pour cela nous définissons une sous-classe de *Moto* qui implante l'interface *Comparable* :

```
class MotoTriable extends Moto implements Comparable {
   public boolean plusPetitOuEgal(Object c) {
      return this.prix() <= ((Moto)c).prix();
   }
   MotoTriable(String m, int p) {super(m,p); }
}
```

Finalement, nous pouvons utiliser, par exemple dans un programme principal, notre méthode de tri :

```
   public static void main(String[] args) {
         MotoTriable mt[] = {new MotoTriable("Honda", 16678),
         new MotoTriable("Kawasaki", 15099),
         new MotoTriable("Puch", 16000)};
      Trieuse.tri(mt);
```

```
        for (int i = 0; i<mt.length; i++) mt[i].imprime();
    }
```

Cet exemple montre une limite de la généricité en Java. Dans l'interface *Comparable*, nous avons défini la méthode *plusPetitOuEgal(Object c)* dont le paramètre est de type *Object*, ce qui signifie qu'un objet est considéré comme comparable seulement s'il est comparable à n'importe quel autre objet. En fait, il nous suffirait que l'objet soit comparable à un autre objet de la même classe. Or ceci n'est pas exprimable en Java. De plus, dans la méthode *tri(Comparable[] x),* il n'est pas possible de spécifier que le paramètre *x* doit contenir des objets qui sont tous des instances de la même classe. On acceptera donc de trier des tableaux contenant à la fois des motos, des vélos et des montgolfières.

À l'heure actuelle, Java n'offre donc pas de véritable mécanisme de généricité indépendant du polymorphisme. De tels mécanismes existent depuis longtemps dans des langages comme Ada (packages, procédures, fonction génériques), C++ (templates) ou Eiffel (classes génériques).

Chapitre 23

Structures de données

Un objet est un module qui contient des données et des opérations d'accès et de modification de celles-ci. Jusqu'à présent, nous avons essentiellement rencontré des objets dont les données étaient simples : soit des variables d'un type de base; soit des références à d'autres objets; ou encore des tableaux. Suivant le genre de problème que l'on attaque, nous aurons besoin de données de structure beaucoup plus complexes, pensons par exemple à un objet représentant en détail l'ensemble de toutes les lignes ferroviaires d'un pays. L'étude des structures de données sert précisément à réaliser de tels objets complexes de manière systématique et efficace. Nous verrons dans ce chapitre que l'API Java peut venir à notre secours dans ce domaine.

Avant d'examiner les structures de données proprement dites, il est bon de se remémorer quelques notions relatives à l'identité et l'égalité des objets. L'une des difficultés de la programmation par objets vient en particulier du fait que la distinction entre un objet et sa valeur reste souvent implicite. Pour les types de données primitifs, cette distinction n'a pas de raison d'être, on peut dire que l'objet « 4 » et la valeur 4 sont la même entité. Par contre, pour les objets plus élaborés, il n'en va pas de même. Prenons par exemple trois rectangles construits de la manière suivante :

```
r1 = new Rectangle(10, 10, 8, 4);
r2 = new Rectangle(10, 15, 7, 3);
r3 = new Rectangle(10, 10, 8, 4);
```

Il s'agit bien de trois objets différents (les tests *r1 == r2*, *r1 == r3* et *r2 == r3* donneraient à chaque fois la valeur *false*), cependant *r1* et *r3* ont les mêmes valeurs (si on les dessinait, ils seraient parfaitement superposés). C'est pourquoi il existe une méthode *equals()* qui teste non pas l'identité des objets, mais l'égalité de

leur valeur. Ce test est basé sur le contenu des variables d'instance des objets. Dans l'exemple ci-dessus, on aurait :

r1.equals(r2) vaut *false* mais *r1.equals(r3)* vaut *true*.

23.1 Les chaînes de caractères

La chaîne de caractères est l'une des structures de données les plus simples et les plus utilisées. Néanmoins, la manipulation de chaînes requiert des précautions.

Comparaisons

Bien que les chaînes de caractères possèdent en Java une représentation littérale sous forme d'un texte entre guillemets, il est essentiel de se souvenir qu'il s'agit d'objets et non de valeurs simples comme les booléens, entiers, flottants ou caractères. Cela signifie en particulier qu'il faut bien prendre garde à distinguer l'égalité et l'identité entre deux chaînes. Par exemple, si l'on écrit :

```
String s1 = "Panoramix";
String s2 = "Pano" + "ramix";
```

rien ne garantit que *(s1 == s2)* donne comme résultat *true*. En effet, *s1* et *s2* peuvent être représentés par deux objets différents qui ont tous deux la valeur *"Panoramix"*, cela dépend de l'implantation de la classe *String*. Par contre, il est certain que *(s1.equals(s2))* donne *true* car *equals()* compare les deux objets caractère par caractère.

Représentation canonique

L'utilisation de *equals()* peut s'avérer coûteuse en temps de calcul si l'on doit comparer un grand nombre de fois des chaînes de grande taille. Pour remédier à cela, la classe *String* offre une représentation canonique des chaînes grâce à la méthode *intern()*. L'exécution de *s.intern()* place la chaîne *s* dans un pool de chaînes internes à la classe *String*. Si le pool contient déjà une chaîne *t* égale à *s*, *s.intern()* retourne *t* comme résultat, sinon *s*. Donc le pool ne contient jamais deux chaînes de même valeur, ce qui fait que si *s1.equals(s2)* vaut *true* alors *s1.intern() == s2.intern()* vaut aussi *true*, et réciproquement.

Lorsqu'on travaille sur des chaînes immuables que l'on doit comparer très souvent, cela vaut donc la peine de les remplacer par leur représentation canonique obtenue par *intern()*. On peut alors utiliser sans problème ==, qui est beaucoup plus rapide que *equals()* puisqu'il compare les identités d'objets et non les contenus.

Performances et StringBuffer

La représentation des chaînes comme objets (instances de la classe *String*) peut également avoir de fâcheuses conséquences sur les performances des program-

mes si l'on n'y prend garde. Ceci est dû au fait que les objets *String* sont immuables, ce qui signifie que toute modification d'une variable *String* implique la création d'un nouvel objet *String*. Si l'on exécute le code suivant :

```
String allStars = "*";
for (int i = 1; i<10000; i++) allStars = allStars + "*";
```

chaque tour dans la boucle créera un nouvel objet. On se retrouvera à la fin avec un objet *String* contenant la chaîne souhaitée et 9999 objets inutiles contenant les valeurs "*", "**", "***", etc. Si l'on tient compte des temps d'allocation et de désallocation de l'espace mémoire occupé par ces objets, la performance du programme sera nettement affaiblie par cet usage des Strings.

Pour des programmes faisant un usage intensif de chaînes de caractères dont la valeur change, il est fortement conseillé d'utiliser la classe *StringBuffer*. Cette dernière offre des chaînes modifiables sans création de nouvel objet et donc de bien meilleures performances. On écrira par exemple :

```
StringBuffer allStarsB = new StringBuffer("*");
for (int i = 1; i<10000; i++) allStarsB.append("*");
String = allStarsB.toString();
```

23.2 Les collections

La théorie élémentaire des ensembles fournit des structures fondamentales telles que les ensembles, séquences, multi-ensembles, fonctions, etc., qui forment des briques de base pour la construction du logiciel. Par rapport aux structures de données fournies par les langages (tableaux et classes en Java), ces structures permettent souvent d'exprimer les algorithmes d'une manière plus claire et plus naturelle. On peut également s'en servir, comme le proposent des méthodes telles que Z [10] ou VDM [5], pour modéliser un domaine d'application et obtenir une spécification formelle.

Il existe en général, dans tous les environnements de développement, des classes qui implantent ces structures directement ou non, mais sous des formes parfois difficiles à reconnaître (pour une étude détaillée de ces structures, voir par exemple [2]).

À partir de la version 1.2 de l'API, le package *java.util* offre des interfaces et classes correspondant aux collections de données fondamentales. Selon le bon principe de l'abstraction, il y a pour chaque type de collection une interface qui définit les opérations, et une ou plusieurs classes qui implante(nt) concrètement ce type. Le tableau ci-dessous résume les différentes interfaces et implantations fournies dans le package java.util.

Il existe plusieurs implantations car on ne connaît pas de technique idéale pour représenter les collections. Chacune possède des avantages et inconvénients en terme de vitesse d'exécution et d'espace mémoire utilisé. De plus, une implanta-

tion peut interdire certaines opérations. Par exemple, on pourrait créer une implantation *ListeCroissante* de *List* qui permet d'ajouter des éléments mais interdit d'en retirer.

Les interfaces *Set*, *List* et *Map* héritent de *Collection*. Dans chacun de ces trois types de collection, on retrouve des méthodes pour :

- ajouter un ou des éléments;
- retirer un ou des éléments;
- tester l'appartenance d'éléments de la collection;
- obtenir des caractéristiques de la collection (taille, vide/non vide, comparaison);
- effectuer des opérations propres à ce type.

C'est la sémantique de ces opérations qui distingue ces types de collections entre eux. Examinons plus en détail chaque type.

D'un point de vue pratique, l'utilisation d'une collection consiste à choisir le type de collection adapté au problème à traiter, puis à choisir l'implantation qui convient le mieux en termes de performances. Dans un programme Java, on aura en général :

```
<InterfaceCollection> maCollection;
maCollection = new <classe implantant InterfaceCollection>(...);
```

Par exemple :

```
Set motsCles;
motsCles = new TreeSet();
...etc...
```

Implantation / **Interface**	par table de hachage	tableau dynamique	arbre équilibré	cellules liées
Set	*HashSet*		*TreeSet*	
List		*ArrayList* *Vector*		*LinkedList*
Map	*HashMap* *Hashtable*		*TreeMap*	
SortedSet			*TreeSet*	
SortedMap			*TreeMap*	

Tableau 23.1 Interfaces et implantations des collections

Ensembles (*Set*)

Un ensemble est une collection, éventuellement vide, d'objets qui ont tous des valeurs différentes. Donc si deux objets *a* et *b* sont dans le même ensemble, *a.equals(b)* est forcément faux. De même, le pseudo-objet *null* peut se trouver au plus un fois dans un ensemble. Il n'y a pas d'ordre des éléments dans un ensemble, on ne peut donc pas parler du premier ou du quatorzième élément d'un ensemble.

Les principales méthodes de l'interface *Set* sont les suivantes.

Méthode	Définition
`add(Object e)`	Ajoute l'élément *e* à l'ensemble. Retourne vrai si *e* a effectivement été ajouté, faux sinon (par exemple, s'il y avait déjà un élément de même valeur).
`remove(Object e)`	Retire *e* de l'ensemble, c'est-à-dire retire tout élément égal à *e*. Retourne vrai si l'ensemble a effectivement été modifié, faux sinon.
`contains(Object e)`	Retourne vrai si l'ensemble contient un élément égal à *e*, faux sinon.
`isEmpty()`	Vrai si l'ensemble est vide.
`size()`	Nombre d'éléments de l'ensemble.
`equals(Object o)`	Retourne vrai seulement si *o* est un ensemble et tous les éléments de *o* sont contenus dans cet ensemble et tous les éléments de cet ensemble sont contenus dans *o*.
`addAll(Collection c)`	Ajoute tous les éléments de *c* à cet ensemble.
`removeAll(Collection c)`	Retire de cet ensemble tous les éléments qui sont aussi dans *c*.
`retainAll(Collection c)`	Retire tous les éléments sauf ceux qui sont aussi dans *c*.
`containsAll(Collection c)`	Vrai si l'ensemble contient tous les éléments de *c*.

Tableau 23.2 Méthodes de l'interface Set

Il ne faut pas négliger les quatre dernières méthodes de la liste ci-dessus car elles permettent d'effectuer les opérations ensemblistes classiques :

- *s1.addAll(s2) revient à faire l'union s1 = s1 ∪ s2;*
- *s1.retainAll(s2) revient à faire l'intersection s1 = s1 ∩ s2;*
- *s1.removeAll(s2) revient à faire la différence s1 = s1 \ s2;*

- *b = s1.containsAll(s2) teste l'inclusion s1 = s2 ⊆ s1;*

Remarquons qu'un ensemble ne stocke que des objets (*Objects*). Pour y ranger des valeurs de types simples (*char*, *int*, *double*, etc.), il est nécessaire de les envelopper dans l'une des classes prévues à cet effet : *Character*, *Integer*, *Double*, etc. (voir point 6.10, p. 95).

Les deux implantations des ensembles fournies par le package *java.util* diffèrent par leurs performances et l'ordre de stockage des éléments.

HashSet : une implantation par table de hachage[1], l'ajout et le test d'appartenance sont très rapides (temps quasi constant).

TreeSet : les éléments sont placés dans un arbre de recherche binaire équilibré, l'ajout et le test d'appartenance prennent un temps proportionnel au logarithme du nombre d'éléments (c'est encore rapide); de manière interne, les éléments sont maintenus triés selon leur ordre naturel (défini par la méthode *compare()*), ce qui permettra, par exemple, de parcourir facilement les éléments par ordre croissant (voir point 23.3 «Itérateurs»).

Un exemple : le compteur de mots

La programme suivant compte le nombre de mots différents contenus dans un fichier de texte. La méthode est simple, il suffit d'ajouter chaque mot lu à un ensemble. Si le mot y est déjà, il n'est pas ajouté une seconde fois. À la fin, la taille de l'ensemble donne le nombre de mots différents.

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

class CompteDiff {

public static void main(String[] args) {
   try {
      // initialisation du lecteur de mots (voir chap. Entrees/
Sorties)
      Reader r=new FileReader(args[0]);
      StreamTokenizer st=new StreamTokenizer (r);
      st.whitespaceChars('.','.') ; // le '.' est un séparateur

      Set mots = new HashSet();

      while (st.nextToken()!=st.TT_EOF) {
         if (st.ttype== st.TT_WORD) {
            mots.add(st.sval); // st.sval contient le mot lu
```

1. Dans une table de hachage, un objet est stocké à une position donnée par une fonction de hachage appliquée à la valeur de l'objet et qui fournit un entier compris entre 0 et la taille de la table - 1. On utilise la même fonction pour retrouver la place d'une paire stockée. Si deux objets doivent occuper la même position, l'un des deux est déplacé selon une règle de « résolution de collision ». Tant qu'il y a peu de collisions, l'accès aux données est extrêmement rapide.

```
            }
         }
         r.close();
         System.out.println("Nb. de mots fifferents: "+mots.size());
      }
      catch (FileNotFoundException e)
         {System.out.println("Fichier non trouve");}
      catch (IOException e)
         {System.out.println("Erreur de lecture");}
   }}
```

Séquences (*List*)

Une séquence est une collection ordonnée d'objets. Chaque objet possède une position dans la séquence et plusieurs objets de mêmes valeurs peuvent apparaître dans une séquence, On voit dans la table ci-dessous que les méthodes d'ajout, de suppression, etc., demandent comme paramètre la position à laquelle doit s'effectuer l'opération.

Méthode	Description
`add(int i, Object e)`	Ajoute *e* à la position *i*.
`add(Object e)`	Ajoute *e* à la fin de la séquence.
`remove(int i)`	Retire l'élément se trouvant à la position *i*.
`remove(Object e)`	Retire la première occurrence de *e* dans la séquence.
`clear()`	Vide la séquence de tous ses éléments.
`set(int i, Object e)`	Remplace l'élément à la position *i* par *e*.
`contains(Object e)`	Vrai si cette séquence contient *e*.
`indexOf(Object e)`	Retourne la position (*int*) de la première occurrence de *e* dans la séquence ou -1 s'il n'y est pas.
`lastIndexOf(Object e)`	Position de la dernière occurrence de *e*.
`get(int i)`	Retourne l'objet placé à la position *i*.
`size()`	Nombre d'éléments dans la séquence.
`isEmpty()`	Vrai si la séquence est vide.
`equals(Object o)`	Vrai si *o* est une liste qui contient des éléments égaux à des positions identiques.
`addAll(Collection c)`	Ajoute *c* à la fin de la séquence (concaténation).

Tableau 23.3 Opérations de l'interface List

Méthode	Description
`addAll(int i, Collection c)`	Insère *c* à la position *i*.
`containsAll(Collection c)`	Vrai si la séquence contient tous les éléments de *c*.
`removeAll(Collection c)`	Retire de la séquence les éléments qui sont aussi dans *c*.
`retainAll(Collection c)`	Ne retient que les éléments qui sont aussi dans *c*.
`subList(int debut, int fin)`	Crée une liste qui est une vue de celle-ci entre *debut* et *fin*. La séquence ainsi créée partage ses éléments avec cette séquence!

Tableau 23.3 Opérations de l'interface List

Les trois implantations offertes pour les listes se distinguent par leurs performances.

ArrayList : la liste est maintenue dans un tableau, l'accès à un élément par sa position est très rapide, par contre l'insertion au début prend un temps proportionnel à la taille de la liste. C'est en règle générale un bon choix.

LinkedList : la liste est formée de cellules liées par des références, l'insertion est rapide, par contre l'accès à un élément par sa position prend un temps proportionnel à la position. Intéressante essentiellement si l'on fait de fréquentes insertions au début et de nombreux parcours avec suppressions.

Vector : c'est la classe originellement fournie avec les premières versions de l'API Java, elle est similaire à *ArrayList*. La liste est stockée dans un tableau dont on peut contrôler les paramètres d'allocation. Cette classe est employée dans de nombreux programmes Java développés pour l'API 1.0 et 1.1. Les méthodes sont synchronisées : deux processus concurrents ne peuvent accéder simultanément au même *Vector*.

Fonctions (*Map*)

Une fonction d'un ensemble S vers un ensemble D est un ensemble de liens $s \rightarrow d$ où s est un élément de S et d un élément de D. À un objet de la source S peut correspondre au plus un seul objet de la destination D. Par contre, un même objet peut être la destination de plusieurs liens. On peut donc avoir les liens $s_1 \rightarrow d$ et $s_2 \rightarrow d$ dans la même fonction, mais pas $s \rightarrow d_1$ et $s \rightarrow d_2$.

Dans un lien $s \rightarrow d$, on appelle s la clé et d la valeur et on parle donc de paire clé-valeur.

Comme dans le cas des ensembles, on a une implantation par hachage (*HashMap*) et une implantation par arbre (*TreeMap*). Pour pouvoir parcourir les clés en ordre croissant, il faut choisir le *TreeMap*. Le *HashMap* est plus rapide.

Méthode	Description
`put(Object cle, Object val)`	Ajoute à la fonction le lien *cle* → *val*. Retourne l'ancienne valeur associée à *cle* ou *null* si *cle* n'était pas associée.
`remove(Object c)`	Retire le lien de clé *c*, s'il existe. Retourne l'ancienne valeur associée à *cle*.
`clear()`	Vide la fonction.
`containsKey(Object c)`	Vrai si la fonction associe la clé *c* à une valeur.
`containsValue(Object v)`	Vrai s'il existe une ou plusieurs clé(s) associée(s) à la valeur *v*.
`get(Object c)`	Retourne la valeur (*Object*) liée à la clé *c* dans cette fonction, ou *null* s'il n'y en a pas.
`equals(Object o)`	Teste l'égalité entre deux fonctions (mêmes paires clé-valeur).
`isEmpty()`	Vrai si la fonction est vide.
`putAll(Map t)`	Ajoute toutes les paires clé-valeur de *t* à cette fonction.
`size()`	Retourne le nombre de paires clé-valeur de cette fonction.
`keySet()`	Fournit un *Set* contenant l'ensemble des clés de cette fonction. Attention, cet ensemble partage ses éléments avec la fonction!
`values()`	Fournit une collection contenant les valeurs de cette fonction.

Tableau 23.4 Opérations du type Map

À la fin du paragraphe 23.3 sur les itérateurs, nous présenterons un exemple complet utilisant les fonctions.

Une remarque de modélisation : fonctions et encapsulation des liens

Les fonctions permettent d'établir des liens entre objets sans briser l'encapsulation, c'est-à-dire sans toucher à la structure interne des classes concernées.

Supposons que l'on désire établir des liens entre des objets de la classe *Avion* et de la classe *Aéroport* pour indiquer l'aéroport où s'effectue l'entretien de chaque avion.

La première manière de procéder, que nous avons vue au point 21.6, consiste à ajouter une variable d'instance *entretien* de type *Aeroport* dans la classe *Avion*. Ce qui nous oblige à modifier la classe *Avion*.

Les fonctions nous offrent une autre solution qui consiste à créer une fonction *entretien* sous forme d'une *Map* pour stocker des liens (avion → aéroport). La classe *Avion* reste alors indépendante de la fonction *entretien*.

```
Map entretien = new HashMap();
Avion hb56 = ... ;
Aeroport gva = ...;
entretien.put(hb56, gva); // associe hb56 à gva.
```

Fonctions et multiensembles

Un multiensemble correspond à un ensemble où un même élément pourrait apparaître plusieurs fois; on parle alors du nombre d'occurrences d'un élément. En fait, un multiensemble est une fonction qui associe à chaque clé un nombre entier qui représente son nombre d'occurrences. On peut donc facilement réaliser une classe *MultiEnsemble* à l'aide de *Map*.

23.3 Itérateurs (Iterator)

Les itérateurs sont des objets permettant de parcourir les éléments d'une collection. On peut voir un itérateur comme une sorte de curseur qui se déplace le long d'une collection en visitant successivement tous les éléments. Les itérateurs sont nécessaires pour le parcours de collections dont les éléments ne sont pas indicés ou lorsque les valeurs des indices (clés) sont inconnues.

Un itérateur établit (et c'est là son intérêt majeur) une totale indépendance entre la structure de la collection et le code nécessaire pour la parcourir.

En général, on crée un itérateur en invoquant la méthode *iterator()* de la collection à parcourir. On effectue le parcours des éléments de la collection à l'aide des deux méthodes :

- *hasNext() :* retourne vrai s'il y a encore des éléments à parcourir;
- *next() :* retourne l'objet suivant selon l'ordre de parcours fixé par l'itérateur.

On peut également retirer de la collection l'élément sur lequel est positionné l'itérateur avec la méthode *remove()* de l'itérateur. Il faut absolument éviter de retirer des éléments pendant un parcours en utilisant la méthode *remove()* de la collection, car cela peut perturber le fonctionnement de l'itérateur.

Le schéma classique d'utilisation des itérateurs est :

```
Iterator i = maCollection.iterator();
while (i.hasNext()) {
   objetCourant = i.next();
   ... <traitement de objetCourant>
   ... <éventuellement> i.remove(); // jamais
maCollection.remove(...)
}
```

Remarque

Jusqu'à la version 1.1, l'API Java offrait la classe *Enumeration*, munie des méthodes *hasMoreElements()* et *nextElement()* comme mécanismes d'itération. Il est désormais préférable d'utiliser *Iterator* qui permet de retirer des éléments en cours de route.

Un exemple : l'indexation d'un texte

Ce petit programme lit un texte et crée un index qui contient, pour chaque mot, la liste des numéros de lignes où il apparaît.

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

class Indexeur {

public static void main(String[] args) {
   try {
      Reader r=new FileReader(args[0]);
      StreamTokenizer st=new StreamTokenizer (r);
      st.whitespaceChars('.','.') ;
      st.eolIsSignificant(true); // reconnait les fins de ligne

      Map index = new TreeMap(); // fonctions avec clés ordonnées
      int noLigne = 1;

      while (st.nextToken()!=st.TT_EOF) {
         if (st.ttype == st.TT_EOL) noLigne++; // fin de ligne
         if (st.ttype== st.TT_WORD) { // mot
            List lignes = (List)index.get(st.sval);
            if (lignes == null) { // premiere fois qu'on trouve ce mot
               lignes = new ArrayList(); // une liste vide
               index.put(st.sval, lignes); // lui est associee
            }
            lignes.add(new Integer(noLigne));
         }
      } // while
      r.close();

      // Parcours de toutes les cles de la fonction
      Iterator i = index.keySet().iterator();
      while (i.hasNext()) {
         Object mot = i.next();
         System.out.print((String)mot);
         List lignes = (List)index.get(mot);

         // Parcours de la liste des nos de lignes
         Iterator j = lignes.iterator();
         while (j.hasNext()) {
            int ligne = ( (Integer)j.next() ).intValue();
            System.out.print(" " + ligne);
         }
```

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

      }
      catch (FileNotFoundException e) {System.out.println("Fichier non
trouve");}
      catch (IOException e) {System.out.println("Erreur de lecture");}
}
}
```

Dans ce programme, on utilise une fonction (index) qui associe à chaque mot lu une liste contenant les numéros des lignes où apparaît le mot. C'est donc une structure de collection à deux niveaux.

Pour imprimer l'index, on parcourt avec l'itérateur *i* l'ensemble des clés de la fonction (obtenu par *keySet()*) et, pour chaque clé, on parcourt avec l'itérateur *j* la liste qui lui est associée.

L'exécution de ce programme avec son propre texte source comme fichier de données produit :

```
ArrayList 21
EOF 16
EOL 17
FileNotFoundException 46
FileReader 8
IOException 47
Indexeur 4
Integer 24 39
Iterator 30 37
List 19 19 34 34
Map 13
etc.
```

23.4 Copies et égalités de surface et profondes

Nous avons déjà mentionné la distinction entre égalité et identité de deux objets, et la nécessité d'avoir une méthode qui teste l'égalité de deux objets en se basant sur leur contenu. Dans le cas d'une classe *Rectangle* simple, le test d'égalité (*equals*) revient à tester l'égalité des quatre nombres entiers qui forment la valeur d'un objet *Rectangle* (coordonnée x, coordonnée y, largeur et hauteur). Mais ce n'est pas toujours aussi simple. Considérons une classe représentant des fractions de la forme (*p/q*) :

```
class Fraction {
   private int numerateur, denominateur;
```

Deux fractions sont égales si leur numérateur et leur dénominateur le sont, mais ce n'est pas tout. Tout le monde sait que 1/2 = 2/4 = 3/6, etc. Deux fractions sont égales si les multiplications croisées {numérateur de *a* * dénominateur de

b} et {numérateur de *b* * dénominateur de *a*} donnent le même résultat. Il faut donc écrire une méthode *equals()* spécifique pour les fractions :

```
public boolean equals(Fraction f) {
   return (this.numerateur * f.denominateur ==
        this.denominateur * f.numerateur); }
```

Dans le cas où les variables d'instance des objets à comparer ne contiennent pas des valeurs primitives (*int*, *long*, *float*, etc.), la méthode *equals* doit en tenir compte et décider d'appliquer soit le test d'identité (==), soit le test d'égalité (*equals*) sur celles-ci. Prenons par exemple une classe *ListeRect* dont chaque objet a pour but de gérer un ensemble de rectangles stockés dans un tableau :

```
class ListeRect {
   private Rectangle[] lesRectangles;
   private int nbRectangles;
```

Egalité de surface et égalité profonde

On peut définir deux notions d'égalité pour la classe *ListeRect*. Dans le premier cas, on dira que deux *ListeRect a* et *b* sont égales si elles contiennent **les mêmes** rectangles, c'est-à-dire que :

a.lesRectangles[0] == b.lesRectangles[0] **et**

a.lesRectangles[1] == b.lesRectangles[1] **et**

etc.

Ce type d'égalité est souvent appelé égalité de surface.

Dans le second cas, on dira que les deux *ListeRect* sont égales si elles contiennent des rectangles égaux, c'est-à-dire que :

a.lesRectangles[0].equals (b.lesRectangles[0]) **et**

a.lesRectangles[1].equals (b.lesRectangles[1]) **et**

etc.

On parle dans ce cas d' égalité profonde car le test *equals* se propage en profondeur dans les composantes des objets à tester.

Ni l'une ni l'autre de ces deux notions n'est la plus juste dans l'absolu, cela dépend essentiellement de l'usage qu'on fait de cette classe. Il est par contre essentiel de bien décrire la notion d'égalité utilisée, afin d'éviter de mauvaises utilisations de la classe. De plus, l'opération *equals()* se doit d'être :

- *symétrique : a.equals(b)* et *b.equals(a)* doivent toujours donner le même résultat;
- *réflexive : a.equals(a)* doit toujours donner *true*;

- *transitive : si a.equals(b)* et *b.equals(c)* donnent *true,* alors *a.equals(c)* doit aussi donner *true.*

Copie d'objets

La copie d'objets soulève le même genre de questions que l'égalité. En effet, l'opérateur « = » n'effectue aucune copie d'objet. Par exemple, si l'on a :

```
r1 = new Rectangle(0, 0, 10, 15);
```

l'opération :

```
r2 = r1;
```

ne crée pas une nouvelle instance de *Rectangle* mais place dans *r2* une référence à l'objet référencé par *r1*. Après cette opération, *r1 == r2* vaut *true*.

Le but d'une opération de copie est de créer un **nouvel objet** qui soit égal (au sens de *equals()*) à l'objet de départ. Dans le jargon Java, on parle d'opération de *clonage*. On veut donc une opération *clone()* telle qu'après l'instruction :

```
r2 = r1.clone()
```

on ait *r2 != r1* mais *r2.equals(r1)*.

De même que pour l'égalité, on peut définir des opérations de clonage de surface ou de clonage profond. Si l'on reprend la classe *ListeRect*, une copie de surface de la *ListeRect a* consiste à créer un nouvel objet *b* qui contient son propre tableau *lesRectangles*, et à faire :

```
for (i=0; i<nbRectangles; i++)
{b.lesRectangles[i] = a.lesRectangles[i]; }
```

On ne crée pas de nouveaux rectangles, la nouvelle liste fait référence aux rectangles de la précédente liste.

Pour une copie en profondeur, on créera au contraire un nouvel exemplaire de chaque rectangle :

```
for (i=0; i<nbRectangles; i++)
{b.lesRectangles[i] = a.lesRectangles[i].clone(); }
```

L'opération de clonage doit être très clairement décrite pour éviter toute mauvaise interprétation par les futurs utilisateurs de la classe (ou par l'auteur qui ne sait plus très bien ce qu'il a fait il y a six mois). On notera en particulier que le clonage de surface introduit un partage d'objets, c'est-à-dire que deux objets *ListeRect* font référence aux mêmes objets *Rectangle*. Donc, si l'on modifie l'un de ces *Rectangle,* la modification apparaît dans les deux *ListeRect*; à nouveau, cela peut être considéré comme un avantage ou un inconvénient mais il faut en être conscient.

23.5 Invitation à explorer

Les interfaces *Comparable* et *java.util.Comparator* permettent de définir un ordre sur un type d'objets et de trier des listes ou de maintenir des ensembles triés.

Explorez la classe *java.util.Arrays* qui offre toute sortes d'opérations de manipulation de tableaux, dont la transformation d'un tableau en liste. *Set* et *List* ont aussi des méthodes pour produire des tableaux.

Pour les séquences, il existe un type d'itérateur *ListIterator* qui permet d'avancer et de reculer, de modifier la liste en cours de route, etc.

Lisez l'excellent tutoriel sur les collections de Joshua Bloch :

http://java.sun.com/docs/books/tutorial/collections/index.html.

Chapitre 24

Composition du logiciel

Bien que la notion de classe permette de construire une application de manière modulaire, elle n'est plus suffisante lorsqu'on se trouve face à des applications de grande taille (ce que les anglophones nomment *programming in the large*). C'est pour cette raison que certains langages et méthodes orientés objet proposent de regrouper les classes dans des structures de plus haut niveau. Le langage Java offre dans ce but la notion de *package*. Nous verrons également dans ce chapitre comment définir l'accessibilité aux objets dans des projets impliquant plusieurs packages.

Si les packages constituent un outil de structuration du logiciel, les Java Beans ouvrent, quant à eux, la possibilité de construire du logiciel par composition d'objets. Nous verrons pour terminer le mécanisme d'introspection nécessaire au développement des Java Beans.

24.1 Les packages

Un package Java est simplement un groupe de classes. Pour déclarer qu'une classe appartient à un package, il faut placer, au début du fichier source qui contient la définition de la classe, l'énoncé :

```
package <nom-du-package>;
```

Une classe ne peut appartenir qu'à un seul package.

Certains environnements de développement Java exigent que tous les fichiers de code source et de byte-code des classes d'un package se trouvent dans le même

répertoire du système de fichier, ce répertoire portant nécessairement le même nom que le package.

Les packages peuvent eux-mêmes être organisés hiérarchiquement, il faut alors utiliser une notation pointée pour nommer chaque package. Un nom de la forme :

```
packA.packB
```

spécifie un sous-package de *packA* qui se nomme *packA.packB*. Les classes de ce package doivent alors être placées dans un sous-répertoire de *packA* nommé *packB*.

Si aucun package n'est déclaré en début de fichier, les classes définies dans ce fichier appartiennent à un package anonyme. Ces classes ne sont importables dans aucun autre package. Il s'agit d'une facilité de test pour de petits programmes, qui doit être abandonnée dès qu'il faut aborder un développement conséquent.

Constitution des packages

Le design des packages, comme celui des classes, n'est pas chose facile. Comment décider quelle classe ira dans quel package? En principe, un package devrait regrouper des classes partageant des caractéristiques communes. Une bonne pratique consiste à fixer, pour chaque package, un critère qui permet de décider si une classe doit appartenir ou non au package. Ce critère peut être de différente nature, par exemple :

- critère fonctionnel : les classes du package concourent à réaliser une fonction du système, par exemple la gestion de l'interaction avec l'utilisateur, la gestion du réseau de communications, etc;
- critère de domaine : les classes du package représentent des objets d'un sous-domaine de l'application comme, par exemple, des objets géométriques (carrés, droites, cercles, points, surface, etc.), des unités de mesure (poids, surfaces, dates, etc.);
- variantes d'implantation : les classes du package implantent le même type abstrait de manières différentes. On pourrait avoir un package fonction contenant des classes *fonctionSequence, fonctionArbre, fonctionHash*, etc., qui sont autant de manières différentes d'implanter le type fonction.

Classes publiques et privées d'un package

Chaque classe d'un package peut être déclarée publique (*public*), auquel cas elle sera utilisable par des classes d'autres packages, sinon son utilisation sera réservée aux autres classes de son package.

Une classe non publique est donc encapsulée dans son package, ce qui permet de ne montrer à l'extérieur du package que les classes que l'on désire mettre à disposition des autres. Les classes non publiques sont en général des classes auxiliaires qui ne présentent pas d'intérêt pour l'utilisateur du package, mais qui sont techniquement nécessaires à sa réalisation.

Le mot réservé *public*, placé devant la définition d'une classe, indique que celle-ci est publique, sinon elle est à usage interne du package.

Pour utiliser une classe *C* d'un package *p* depuis un autre package, il faut :

- soit préfixer son nom par celui du package *(p.C)* à chaque référence;
- soit importer la classe *(import p.C),* ce qui permet de n'utiliser que le nom C par la suite.

Notons que l'on peut importer d'un coup toutes les classes d'un package *p* en faisant *import p.**

24.2 Règles d'accessibilité en Java

Dans la notion d'encapsulation présentée au point 21.1, toute la partie structurelle d'une classe (les variables) était encapsulée alors que toute la partie dynamique (les méthodes) était visible de l'extérieur.

Dans un langage comme Java, la règle ne peut être aussi simple car il faut compter avec les notions de sous-classe et de package pour définir l'accessibilité (également appelée visibilité). Le principe est de distinguer différents degrés d'accessibilité entre classes suivant que l'une est sous-classe de l'autre ou que l'une appartient au même package que l'autre.

Un membre d'une classe (variable ou méthode) est accessible depuis une autre classe dans les conditions suivantes : si le membre est déclaré public (modificateur **public** placé avant la déclaration), il est accessible depuis n'importe quelle autre classe qui peut accéder à la classe de déclaration; s'il est déclaré privé (**private**), il n'est accessible que depuis le code des méthodes de sa classe :

- s'il n'y a pas de déclaration particulière, il est accessible depuis les classes du même package;
- s'il est déclaré protégé *(**protected**),* il est accessible depuis les sous-classes de sa classe de déclaration, qu'elles soient ou non dans le même package.

La condition d'accessibilité des membres protégés possède quelques raffinements supplémentaires, dont la principale conséquence est la suivante : si *m* est une variable ou méthode protégée d'une classe *C*, une expression de la forme *e.m* qui apparaît dans une sous-classe *S* de *C* n'est correcte que si le type de *e* est

S ou une sous-classe de *S* ou si *e* est la pseudovariable *super*. En d'autres termes, *S* ne peut accéder au membre *m* d'objets qui sont de type *C* mais pas de type *S*.

Nous présentons ci-dessous quelques cas d'utilisation des modificateurs d'accessibilité.

Transparence des objets

Un membre privé n'est visible qu'à l'intérieur de sa classe. Cela correspond à l'encapsulation la plus forte en Java. Notons cependant que les objets d'une même classe sont complètement transparents les uns par rapport aux autres. Une méthode peut donc accéder aux membres privés de *this* mais aussi aux membres privés de tout objet de la même classe que *this* (cette possibilité surprendra les habitués de Smalltalk) :

```
class Piece {
   private String type;

   boolean memeType(Piece p) {
      // accès à une variable privée d'un autre objet Piece
      return p.type == this.type;
      }
   ...
}
```

Héritage des variables privées

Le modificateur *private*, placé devant une variable, rend celle-ci inaccessible dans les sous-classes, bien qu'elle soit héritée. Ainsi, lorsqu'une méthode d'une sous-classe veut modifier une variable héritée mais inaccessible, elle doit forcément faire appel à une méthode de modification de la super-classe, comme dans l'exemple ci-dessous :

```
package p1;
class Piece {
   private String type;
   int    noSerie;

   void defType(String s) {type = s;}
   void defSerie(int i) {noSerie = i;}
}

class PieceSimple extends Piece{
   void uneMethode() {
      // acces a la var. heritee à travers une méthode de Piece:
      // this.type = "vis" causerait une erreur !
      this.defType("vis");

      // accès à une variable héritée visible
      noSerie = 455848;
   }
}
```

Modificateur *protected*

Cet exemple illustre la règle particulière d'emploi des variables protégées :

```
package p1;

public class Piece {
   protected int cout;
   ...
}

package p2;
import p1.Piece;

class PieceSimple extends Piece{
   ...
   void uneMethode(Piece p) {
      PieceSimple ps = new PieceSimple();

      this.cout = 345; // accès autorisé
      // p.cout = 456; // accès interdit car p est de type Piece qui
n'est
                       // pas une sous-classe de PieceSimple
      ps.cout = 765; // accès autorisé
   }
}
```

Accessibilité et redéfinition

Si une classe qui hérite d'une méthode *m()* la redéfinit, en donnant donc le même nom et les mêmes types de paramètres à une nouvelle méthode, la méthode redéfinie doit être au moins aussi accessible que la méthode héritée.

Cette règle garantit qu'un objet d'une sous-classe sait faire tout ce que sait faire un objet de sa super-classe.

Si ce n'était pas le cas, on ne pourrait plus considérer qu'une sous-classe étend sa super-classe; on perdrait alors la propriété selon laquelle partout où un objet d'une classe est requis, on peut le remplacer par un objet d'une de ses sous-classes. Le compilateur ne pourrait plus faire de contrôle statique des types des expressions, et des erreurs d'exécution pourraient s'ensuivre.

Quelques conseils de design

Définir un niveau de visibilité est évidemment une question de design qui ne possède pas de réponse toute faite. Le principe général reste d'accorder une large visibilité aux méthodes et de restreindre la visibilité des variables.

Rappelons quelques points qui peuvent aider à choisir le niveau de visibilité adéquat des variables :

- plus les variables d'une classe sont cachées, plus celle-ci est indépendante des autres et donc facile à maintenir;
- plus le projet sur lequel on travaille est grand, plus les choix de visibilité portent à conséquence. L'API Java constitue un exemple extrême car pratiquement tout programme Java écrit quelque part dans le monde utilise les classes de l'API;
- l'extension de la visibilité d'une variable peut parfois éviter d'écrire plusieurs méthodes qui ne servent qu'à accéder aux variables des objets;
- si la visibilité est limitée au package, on n'a pas à craindre qu'un grand nombre de classes se mette à accéder à la variable;
- il n'est pas toujours vrai que l'accès direct aux variables est plus rapide que le passage par une méthode d'accès, car les compilateurs modernes peuvent très bien optimiser ces méthodes (par exemple, par expansion directe du code). De plus, les compilateurs peuvent tenir compte du modificateur final qui empêche la redéfinition d'une méthode, ce qui évite de passer par le mécanisme de liaison dynamique lors des appels.

En ce qui concerne les méthodes, on peut noter deux cas où il paraît judicieux de limiter la visibilité :

- il peut être utile de cacher des méthodes auxiliaires qui ont un usage interne important mais n'offrent pas de service intéressant aux autres classes. De cette manière, on allège l'interface de la classe et on en facilite la réutilisation par d'autres concepteurs;
- dans le même ordre d'idée, on évitera également de rendre publiques des méthodes dont l'effet peut s'avérer désastreux si elles ne sont pas utilisées selon des règles bien précises, en général connues du développeur seul.

24.3 Les Java Beans

Java Beans est une architecture logicielle conçue pour favoriser la création d'applications par assemblage de composants. Il existe déjà un certain nombre d'architectures allant dans ce sens. Parmi les plus intéressantes, on peut citer :

Les *plug-ins*

Il s'agit de composants additionnels qui enrichissent une application existante (butineur Web, traitement de texte, traitement d'images, etc.) de nouvelles fonctions.

Les applets Java

C'est une architecture basée sur la mobilité du code stocké sur un serveur et exécuté sur n'importe quelle machine cliente munie d'une machine

virtuelle Java. Les applets sont fournies à l'utilisateur en fonction des tâches qu'il veut effectuer.

DCOM

L'architecture de composants distribués définie par MicroSoft permet d'écrire des composants logiciels capables de communiquer avec d'autres composants DCOM pour s'intégrer dans un système distribué.

AppleScript

Ce langage de haut niveau permet d'invoquer des services de différentes applications (tableurs, traitements de texte, bases de données, etc.) pour écrire une « macro-application ». Des ensembles de services standards sont répertoriés dans des dictionnaires.

Anatomie et vie d'un Java Bean

Techniquement, un Java Bean n'est rien d'autre qu'un objet Java dont la classe remplit certains critères de dénomination de ses méthodes qui peut être sérialisé.

Conceptuellement, un Bean est composé :

- de propriétés (persistantes);
- d'événements reconnus;
- de méthodes de traitement des événements.

Un Bean possède deux vies. Il vit la première à l'intérieur d'un outil de génie logiciel qui sert à développer des applications par composition de Beans. La vie du Bean commence au moment où l'outil l'instancie (*new*). L'outil sert ensuite à configurer les propriétés du Bean, par exemple définir la taille, la couleur et le texte d'un Bean de type bouton. Il sert également à définir le comportement de la future application en configurant l'envoi d'événements entre Beans. On spécifie, par exemple, que le bouton « efface » envoie un événement à un champ de texte qui devra alors exécuter la méthode *effacement()*. En général, l'outil de développement présentera les Beans et leurs interactions sous forme graphique, comme esquissé sur la figure 24.1. Une fois tous les Beans configurés, ils sont sérialisés, c'est-à-dire sauvegardés dans des fichiers archives (de type JAR) dans l'état où ils se trouvent.

Notons encore qu'un Bean peut définir sa propre interface graphique de configuration.

La seconde vie du Bean commence lorsque l'application dont il fait partie démarre. Une fois désérialisé, il se retrouve tel qu'on l'a configuré et prêt à exécuter les services (méthodes) qu'on lui demande au sein de l'application. On peut représenter le cycle de vie d'un Bean par le diagramme d'états/transitions de la figure 24.2.

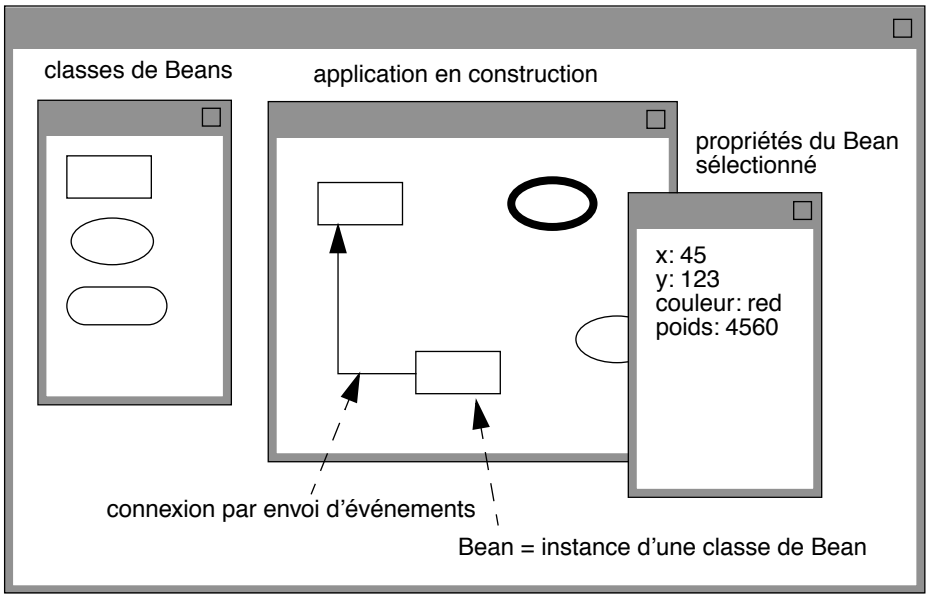


Figure 24.1 Outil de développement interactif avec des Beans

Les événements

Le modèle d'événements décrit au chapitre 13 est une composante essentielle de l'architecture Java Beans, car c'est le support de la communication entre Beans. Un Bean communique avec d'autres Beans, qui se sont inscrits comme écouteurs, en leur envoyant des événements.

Rappelons brièvement comment fonctionne ce modèle.

Pour qu'un Bean puisse produire des événements de type *T* il faut :

- définir une classe *TEvent* qui contiendra la mémoire de l'événement, c'est-à-dire les données à transmettre aux écouteurs;
- définir une interface d'écoute *TListener* qui spécifie quelles méthodes les écouteurs doivent implanter. Ces méthodes ont en général un seul paramètre de type *TEvent;*
- définir dans la classe du *Bean* une méthode d'inscription et une méthode de retrait d'écouteurs *:*

  public void addTListener(TListener ecouteur)

  public void removeTListener(TListener ecouteur);
- éventuellement définir une classe d'adaptateurs *TAdapter* qui implante *TListener.*

Pour transmettre un événement, le Bean source parcourt la liste de tous les écouteurs inscrits et invoque pour chacun d'eux l'une des méthodes de l'interface *TListener*.

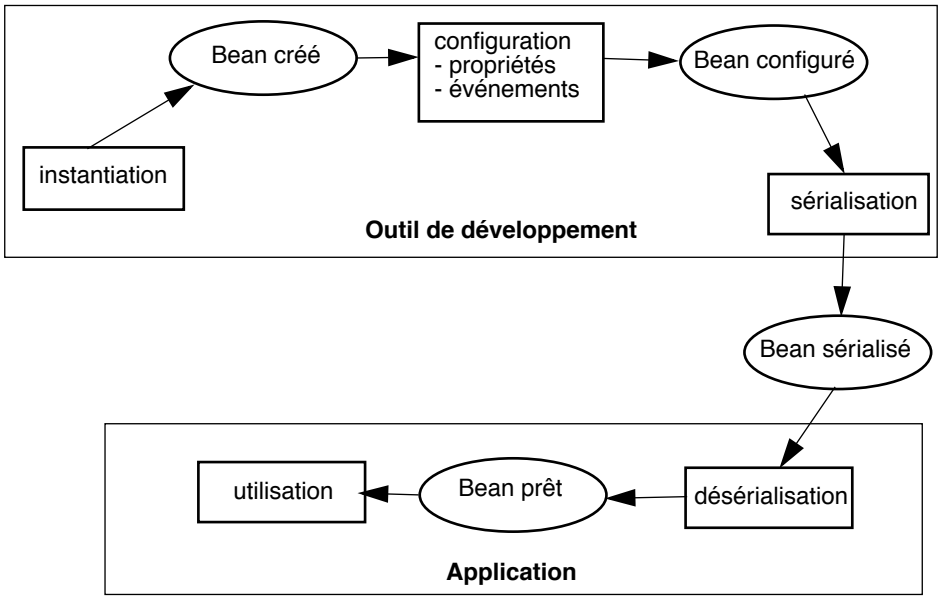


Figure 24.2 Cycle de vie d'un Bean

Les propriétés

Une propriété est un attribut d'un Bean qui est identifiée par un nom et possède une valeur d'un type quelconque. Par exemple, un Bean représentant un vélo pourra avoir une propriété *poids* de type *double*. Pour qu'une propriété de nom *P* soit reconnue comme telle, le Bean doit posséder une méthode d'accès *getP()* et une méthode d'affectation *setP()*. Notre vélo aura donc une méthode *getPoids()* qui retourne un *double* et une méthode *setPoids(double valeur)*.

Une propriété peut être indexée, dans ce cas sa valeur est un tableau de valeurs. Les méthodes *getP()* et *setP()* ont alors un paramètre supplémentaire qui est l'indice dans le tableau. Par exemple, si la propriété couleur est indexée on écrira :

```
Couleur couleurPrincipale = monVelo.getCouleur(0);
monVelo.setCouleur(1, couleurSecondaire);
```

Un Bean peut également avoir des propriétés liées, ce qui signifie que tout changement de valeur doit être signalé à d'autres objets. Lors d'un changement de valeur, le Bean doit créer un événement de type *PropertyChangeEvent* et le transmettre à tous les objets inscrits en appelant leur méthode *property-*

Change(). La classe *PropertyChangeSupport* aide à gérer les inscriptions et transmissions de tels événements.

Une propriété peut être liée encore plus fortement à d'autres objets, on dit alors que la propriété est contrainte. Dans ce cas, le Bean signale aux abonnés qu'il veut changer la valeur de la propriété et leur permet de s'opposer au changement. En cas d'opposition, un abonné lance une exception de type *PropertyVetoException* lorsque le bean appelle sa méthode *vetoableChange()*. Si le Bean reçoit une telle exception, il doit renvoyer un événement pour défaire le changement, sinon il envoie un événement de changement normal pour confirmer le changement à tous les abonnés. Il s'agit donc d'un protocole à deux phases.

Les conventions de dénomination stricte des méthodes liées aux propriétés et aux événement sont là pour permettre aux outils de développement de reconnaître les propriétés et événements d'un Bean.

On remarquera que les classes de l'AWT, de même que celles de Swing respectent toutes les conventions d'écriture des Beans. Les objets de ces classes sont donc des Beans, ce qui permet de les utiliser dans les outils de développement d'interfaces graphiques.

24.4 Introspection

Mais comment un outil de développement peut-il connaître les méthodes offertes par une classe sans lire le code source de celle-ci? C'est précisément pour répondre à cette question qu'a été introduite l'introspection en Java. L'introspection permet à un système de se regarder lui-même pendant qu'il fonctionne. En Java, les objets de la classe *Class* représentent les classes du système et les objets de la classe *Method* (dans le package *java.lang.reflect*) représentent les méthodes.

Voici quelques exemples d'introspection :

- rechercher une classe d'après son nom

```
Class classeBouton = Class.forName("Bouton");
```

- créer une nouvelle instance de cette classe

```
Object monBouton = classeBouton.newInstance();
```

- récupérer toutes les méthodes de cette classe

```
Method [] methodesBouton = classeBouton.getDeclaredMethods();
```

S'il existe dans la classe *Bouton* une méthode *efface* avec un paramètre de type *Rectangle*, on l'invoque sur l'objet *monBouton*.

```
// crée le tableau des types de paramètres
Class [] typesParam = { Class.forName("Rectangle") }
```

```
// crée le tableau des types de paramètres
Class [] typesParam = { Class.forName("java.awt.Rectangle") };

Method efface = classeBouton.getDeclaredMethod("efface",
                typesParam);

// prépare les parametres effectifs
Object[] valParam = {new Rectangle(1,1,100,191)};

efface.invoke(monBouton, valParam);
```

Ces exemples montrent qu'on peut non seulement connaître les méthodes d'une classe et leurs paramètres, mais que l'introspection nous autorise à créer des objets (*newInstance()*) et à invoquer des méthodes (*invoke()*). C'est exactement ce que fait un outil de développement interactif avec les Beans.

Chapitre 25

Processus et concurrence

La machine virtuelle Java permet l'exécution concurrente de plusieurs processus dans un programme. Un processus Java est un objet auquel est associé un fil d'exécution (*thread*) qui possède :

- un état actif ou inactif, l'état actif étant subdivisé en : prêt à l'exécution, en exécution, et en attente;
- un nom (une chaîne de caractères pas forcément unique);
- un contexte d'exécution (instruction courante, pile de mémorisation des appels de méthodes, etc.).

Les deux possibilités consistant à créer des processus sont soit définir une sous-classe de *Thread* qui redéfinit la méthode *run()*, soit créer un objet de *Thread* en donnant comme paramètre un objet de type *Runnable* (qui possède une méthode *run()*).

La méthode *start()* initialise le processus et passe le contrôle à la méthode *run()* du processus ou de l'objet passé comme argument au constructeur du processus. Le processus s'arrête lorsqu'il sort de la méthode *run()*.

Différentes méthodes permettent de contrôler et de synchroniser les processus : *stop(), suspend(), resume(), sleep(), yield()* (passe le contrôle à un autre processus en attente), *join()* (attend la fin d'un processus).

Par défaut, une application Java ne contient qu'un seul processus qui est implicitement mis en marche au lancement de l'application et qui appelle la méthode *main()*.

Une applet simple est contrôlée par un processus du butineur qui appelle les méthodes *init()*, *start()*, *stop()*, *repaint()*, etc., suivant les besoins.

Pourquoi utiliser plusieurs processus?

- pour utiliser au mieux une machine multiprocesseur, un cas encore peu répandu;
- pour effectuer des tâches qui doivent se répéter à intervalles réguliers et indépendamment des autres, par exemple redessiner une image animée toutes les cinquante millisecondes;
- pour simplifier le design des systèmes qui fonctionnent en temps réel et ont des interactions asynchrones avec l'extérieur (dans l'exemple de l'application de télédiscussion, un processus gère un client).

25.1 Problème de la concurrence d'accès

En principe, tout processus peut accéder à tout objet de l'application dont il fait partie. Les processus n'ont pas de mémoire privée, tout le monde travaille dans un espace commun. L'avantage de cette conception est qu'il n'y a pas besoin de créer des canaux de communication entre processus; les échanges entre processus peuvent passer par l'intermédiaire d'objets. Par contre, l'accès simultané au même objet par deux processus peut causer d'importants problèmes de cohérence. Par exemple, si un processus *A* exécute l'instruction :

```
if (a[1] != 0) { ...; c = b/a[1]; ... };
```

et qu'un processus *B* exécute :

```
a[1] = 0;
```

après que *A* ait testé *a[1] != 0* mais avant qu'il ait eu le temps de faire *b/a[1]*, alors une erreur de division par 0 se produira dans *A*, malgré le test qui précède. Une manière d'éviter ce type de problème consiste à garantir l'accès exclusif à un objet pendant l'exécution d'une ou plusieurs instructions.

25.2 La synchronisation en Java

La synchronisation en Java est basée sur la technique des moniteurs de C.A.R. Hoare qui garantit l'accès exclusif à une ressource (en l'occurrence un objet ou un tableau) pendant l'exécution d'une portion de code appelée **section critique**.

Chaque objet possède un moniteur d'accès pour gérer une file d'attente des processus qui veulent avoir un accès exclusif à l'objet. Une section critique est soit une méthode dont la déclaration est précédée du modificateur *synchronized*, soit une instruction précédée de *synchronized(expression)*.

Pour que les instructions de l'exemple précédent fonctionnent correctement, il aurait fallu écrire :

```
synchronized (a) if (a[1] != 0) { ...; c = b/a[1]; ... }
```

dans le processus *A* et :

```
synchronized (a) a[1] = 0;
```

dans *B*.

La technique des moniteurs introduit un raffinement supplémentaire dans la synchronisation. Lorsqu'un processus possède l'accès à un objet, il peut appeler la méthode *wait()* de l'objet pour se mettre en attente et libérer l'objet. Pour le sortir de cette attente, il faut qu'un autre processus exécute un *notify()* sur cet objet et que lui-même obtienne à nouveau l'accès exclusif à l'objet.

Les méthodes *wait()* et *notify()* existent pour tout objet (elles sont héritées de *Object*); elles ne peuvent être appelées que lorsque le processus possède un accès exclusif à l'objet. À noter que l'exécution de *notify()* ne libère pas l'objet.

L'exemple suivant illustre leur utilisation; il s'agit d'un cas classique de producteur-consommateur. Un processus producteur produit des données (de simples entiers dans notre cas) et les place dans une file d'attente. Un consommateur prend ce qu'il trouve dans la file. La file est ici limitée à un élément.

Chaque objet de la classe *Produit* est une file d'attente qui, pour simplifier, peut contenir 0 ou 1 élément entier (*contenu*). On peut ajouter ou retirer un élément et tester si la file est vide.

```
class Produit {
   int contenu;
   boolean vide;
   Produit() {vide = true;}
   void ajoute(int x) {contenu = x; vide = false;}
   int retire() {vide = true; return contenu;}
   boolean estVide() {return vide;}
}
```

Lorsque la file est pleine, le processus producteur doit attendre que le processus consommateur la vide. Quand il a mis un nombre dans la file, il le signale afin de redémarrer le consommateur qui serait en attente.

```
class Producteur extends Thread {
Produit produit;
Producteur(Produit p) {this.produit = p;}
public void run() {
   int ip;
   while (true) {
      ...
      synchronized (produit) {
         if (!produit.estVide()) {
            try produit.wait();
```

```
            catch (InterruptedException e);
         }
         ip =  (int)(Math.random() * 1000); // produit un nb.
aléatoire
         produit.ajoute(ip);
         // signale qu'il y a qqch à prendre
         produit.notify();
      }
   }
}}
```

Un consommateur prend ce qu'il y a dans la file de produits et l'affiche; lorsqu'il n'y a rien, il se met en attente. Lorsqu'il a vidé la file, il le signale afin de débloquer le processus producteur s'il était en attente.

```
class Consommateur extends Thread {
Produit produit;
Consommateur (Produit p) {this.produit = p;}
public void run() {
   int jp;
   while (true) {
      synchronized (produit) {
      if (produit.estVide()) {
         try produit.wait();
         catch (InterruptedException e);
      }
      jp =  produit.retire();
      produit.notify();
      System.out.println(jp);
      }
   }
}}
```

Le programme principal crée les deux processus et les démarre :

```
public class ProdCons {
public static void main(String[] argv) {
  Produit pp = new Produit();
  Producteur pr = new Producteur(pp);
  Consommateur co = new Consommateur(pp);
  co.start(); pr.start();
}}
```

La figure 25.1 ci-après illustre le fonctionnement de *synchronized*, *wait()* et *notify()* dans le cas où le consommateur trouve la file vide.

Cette solution fonctionne mais n'est pas entièrement satisfaisante du point de vue du design de l'application. En effet, rien n'oblige à passer par une instruction *synchronized* pour accéder à *produit*. Un processus sauvage dont le développeur ne serait pas bien informé de la nature partagée de *produit* pourrait y accéder sans contrôle et semer le trouble. Pour obtenir un mécanisme nettement plus sûr, il faut se souvenir des principes de la programmation orientée objet et

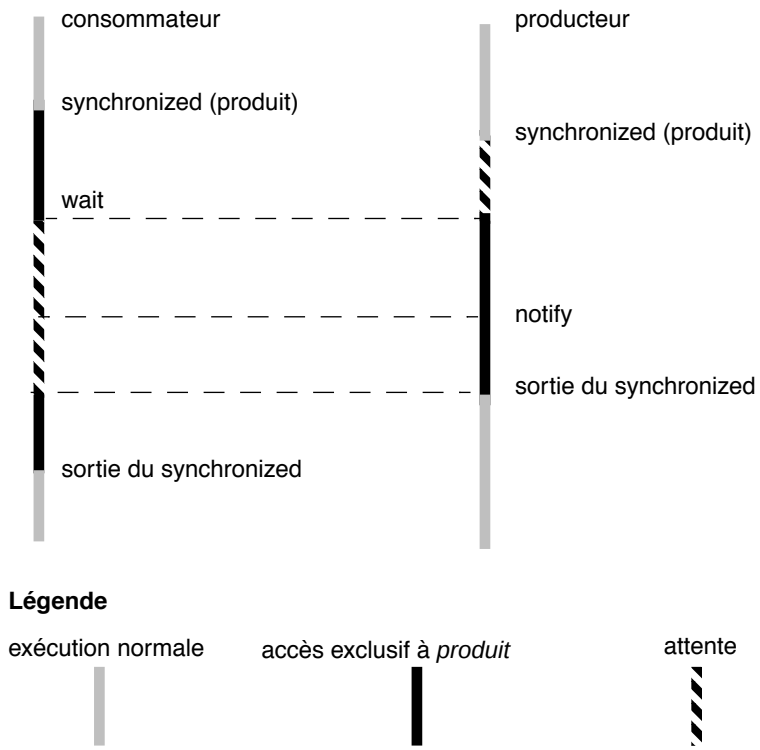


Figure 25.1 Fonctionnement de la synchronisation

appliquer l'encapsulation. Redéfinissons ainsi notre classe *Produit* pour la rendre sûre par rapport aux processus :

```
class Produit {
   int contenu;
   boolean vide;

   Produit() {vide = true;}

   synchronized void ajoute(int x) {
      while (!vide) {
         try wait(); catch (InterruptedException e);}
      contenu = x; vide = false; notify();
   }

   synchronized int retire() {
      while (vide) {
         try wait(); catch (InterruptedException e);}
      vide = true;  notify(); return contenu;
   }

   synchronized boolean estVide() {return vide;}
}
```

Désormais, l'accès à un objet produit de la classe *Produit* passe obligatoirement par des méthodes synchronisées, les méthodes utilisatrices de *Produit* n'ont plus à se soucier de faire des *synchronized*. On peut réécrire le producteur beaucoup plus simplement :

```
class Producteur extends Thread {

Produit produit;

Producteur(Produit p) {this.produit = p;}

public void run() {
  int ip;
  while (true) {
    try {sleep((int)(Math.random() * 1000));}
    catch (InterruptedException e) {};
    ip =  (int)(Math.random() * 1000);
    produit.ajoute(ip);
  }
}}
```

25.3 .Interblocages

L'accès exclusif aux objets offre des garanties quant à l'intégrité des données et à la consistance des traitements. Par contre, il introduit des dépendances entre processus du type : un processus *A* attend que *B* ait libéré un objet *O*. Lorsque la chaîne des dépendances devient cyclique, il y a interblocage des processus. Par exemple, si *A* a obtenu l'accès à *o1* et attend l'accès à *o2* et *B* a obtenu l'accès à *o2* et attend l'accès à *o1*. Les deux processus sont en attente : A attend sur *B* et *B* attend sur *A*, il y a interblocage.

Seule une étude rigoureuse de la dynamique des processus, et en particulier des dépendances possibles, peut prévenir ce genre de problème. Il existe pour cela des techniques formelles comme la modélisation par réseaux de Petri.

Chapitre 26

Intégration des applets

Nous avons examiné l'utilisation de Java pour développer des applications et des applets. Les applets que nous avons montrées étaient en fait de petits programmes qui s'exécutaient sous la direction d'un butineur ou d'un visualiseur d'applet. Nous pouvons maintenant passer à une autre vision des applets, en lien direct avec la notion de document et de document dynamique.

26.1 Documents et types de contenus

Les langages de balisage comme HTML, ou mieux SGML, ont pour fonction première d'indiquer le rôle de chaque composante d'un document. Dans HTML, les balises <Hn> et </Hn> délimitent une partie du contenu de type entête de niveau n, les balises <UL>, <OL>, <DL> et </UL>, </OL> </DL> indiquent un contenu de type liste (ordonnée, non ordonnée, de définitions, etc.). On trouve également des balises pour le type image, le type table, le type formulaire de saisie, et ainsi de suite. Cependant, les types de contenu admis par HTML sont et resteront certainement limités à des types très généraux que l'on retrouve dans la plupart des documents[1]. Cet état de fait est parfois gênant pour des utilisateurs d'un domaine particulier qui aimeraient bien avoir des types de contenus propres à leur domaine.

1. Par contre, le nombre de balises destinées uniquement à régler l'apparence des documents (fontes, centrage, bordures, etc.) va sans doute augmenter sous la pression des utilisateurs commerciaux du World-Wide Web.

26.2 Traiter de nouveaux types de contenu

Prenons le cas d'un spécialiste du jeu d'échecs qui souhaite rédiger des documents pédagogiques à propos de ce jeu. Ses documents seront sans doute remplis de figures décrivant des situations de jeu ou des enchaînements de coups. Comme il n'existe pas de balise <ECHECS> en HTML, la seule solution consiste à créer des images avec un logiciel de dessin et à les incorporer au document à l'aide d'une balise <IMG>. Une autre solution consiste à utiliser les applets!

Supposons que notre spécialiste découvre qu'il existe sur le Web une applet *Echec.class* qui, à partir d'une situation donnée par les valeurs des paramètres *BLANCS* et *NOIRS*, dessine un échiquier et les pièces correspondant à la situation. Il pourra alors insérer dans son document des balises de la forme :

```
<APPLET CODE="Echec.class" SOURCE="http://www.mon-club-echec.com/"
   WIDTH=500 HEIGHT=500>
<PARAMETER NAME=BLANCS VALUE="Re1,Dd1,e2">
<PARAMETER NAME=NOIRS VALUE="Rb8,Ta7,b2">
</APPLET>
```

et le lecteur verra apparaître la situation sous forme de dessin. Mieux encore, il pourrait exister une applet *AnimePartie.class* capable de visualiser une partie coup par coup, d'avancer, de reculer, etc. Cette fois, le paramètre serait une chaîne de caractères décrivant une partie selon l'une des notations standard des échecs. L'utilisation de cette applet ferait du document un document dynamique et interactif.

Les applets Java nous permettent donc de gérer des types de contenus qui ne sont pas prévus dans HTML (parties d'échecs, molécules chimiques, pièces mécaniques, etc.).

26.3 Définir une applet comme gestionnaire de contenu

Il y a deux points essentiels à satisfaire pour qu'une applet devienne un gestionnaire de contenu valable :

- définir clairement le type de contenu que l'applet peut gérer;
- définir en détail la syntaxe des paramètres.

Dans le cas de notre applet hypothétique *AnimePartie*, le type de contenu géré est n'importe quelle partie d'échec et la syntaxe du paramètre est une partie décrite selon la notation algébrique. Le document HTML devra contenir :

```
<H2>Gambit de la dame</H2>
<APPLET  CODE="AnimePartie.class"
         SOURCE="http://www.mon-club-echec.com/"
         WIDTH=500 HEIGHT=500>
<PARAMETER NAME=PARTIE
```

```
   VALUE="1.d2-d4,d7-d5 2.c2-c4,e7-e6 3.Cb1-e3,Cg8-f6 4.Fc1-g5,Ff8-
e7">
</APPLET>
```

Parmi les applets de démonstration fournies par Sun dans le JDK, se trouve une applet de dessin dynamique de graphe. Un graphe est décrit par un paramètre *edges*, séquence d'arêtes de la forme *nomNoeud1-nomNoeud2* séparées par des virgules. Ci-dessous se trouve l'exemple d'inclusion d'un tel graphe dans un document :

```
<applet code="Graph.class" width=400 height=400>
<param name=edges value="a1-a2,a2-a3,a3-a4,a4-a5,a5-a6,b1-b2,b2-
b3,b3-b4,b4-b5,b5-b6,c1-c2,c2-c3,c3-c4,c4-c5,c5-c6,x-a1,x-b1,x-
c1,x-a6,x-b6,x-c6">
</applet>
```

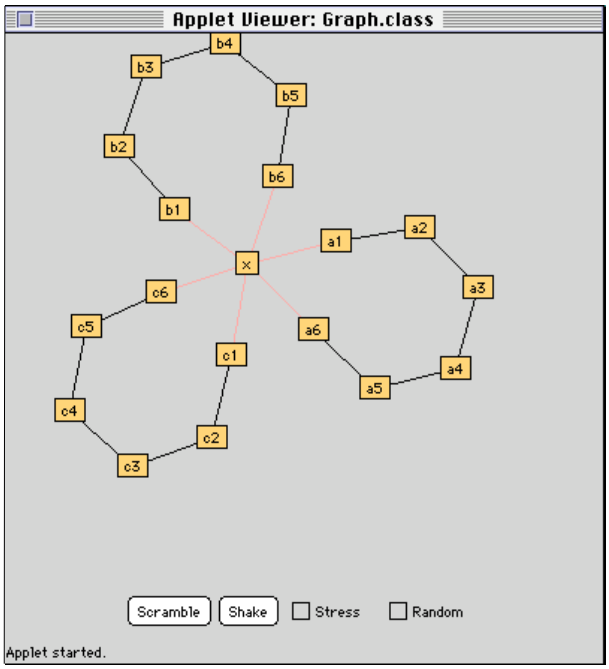


Figure 26.1 Applet paramétrable de graphe (Sun Microsystems)

N.B. Le livre que vous êtes en train de lire n'étant pas dynamique, nous avons dû procéder à une copie d'écran.

Les outils fournis par Java pour traiter les paramètres

Nous avons déjà vu à plusieurs reprises l'utilisation de la méthode *getParameter()* de la classe *Applet* (voir l'applet du serpent, p. 303) qui fournit le *String* donné comme valeur d'un paramètre dans une balise <*APPLET*>.

D'autres méthodes et classes sont également de première utilité :

- les méthodes de conversion de la classe *String* vers les types numériques que l'on trouve dans les classes *Float, Integer, Long* et *Double;*
- la classe *StringTokenizer* qui analyse un *String*, unité syntaxique *(token)* par unité syntaxique.

Cette dernière classe définit un itérateur qui parcourt la chaîne en fournissant un mot à chaque appel à la méthode *nextToken()*.

```
String partie = getParameter("PARTIE");
StringTokenizer st = new StringTokenizer(partie, " ");
while (st.hasMoreTokens()) {
   String unCoup = st.nextToken();
   ...
}
```

Le second paramètre du constructeur de *StringTokenizer* est une chaîne qui détermine la liste des caractères utilisés comme séparateurs de mots, dans notre cas l'espace.

Chapitre 27

Sécurité en Java

Sun Microsystems a conçu Java en y incorporant dès le départ des critères de sécurité. La sécurité est un concept qui doit être traité *a priori* et non *a posteriori*. Cette prise en compte de la sécurité dès la conception est perceptible dans les différents choix qui ont été faits (absence de pointeur, vérification de types, génération de byte-code, etc.). Ces choix amènent d'autres aspects positifs : robustesse et portabilité des programmes, mais ils sont surtout nécessaires pour implanter le niveau de sécurité visé.

Nous avons déjà expliqué pourquoi un tel niveau de sécurité est requis. Rappelons qu'avec la généralisation des butineurs ayant la capacité d'exécuter du Java, il est possible de télécharger des programmes depuis n'importe quel point du réseau Internet. Vous avez constaté que la programmation en Java n'est pas si compliquée et nous espérons que vous avez déjà programmé quelques applets, peut-être même en avez-vous ajouté une à votre page personnelle.

Si la sécurité de Java n'existait pas, il vous serait possible d'écrire une applet qui établit une liste de tous les fichiers *.EXE* d'un disque et qui la poste à *x@y.com*, ou bien encore de vous procurer un virus connu ou inconnu pour l'installer sur un disque, etc. Sans la sécurité de Java, personne ne voudrait prendre le risque de télécharger et d'exécuter des applets.

Nous développons ce chapitre sur la sécurité pour contribuer, nous l'espérons, à une meilleure compréhension des mécanismes sous-jacents. Ceux-ci doivent être publiés (et compréhensibles), d'une part pour qu'ils soient crédibles et d'autre part pour qu'ils soient éprouvés : s'ils sont vraiment sûrs, alors leurs concepteurs peuvent les publier sans crainte. Cette démarche, utilisée pour Java, a

déjà permis de découvrir quelques faiblesses, non pas dans les principes mais dans le code de la machine virtuelle Java.

L'architecture Java comporte quatre niveaux de sécurité :

1. Le langage et le compilateur.
2. Le vérificateur de byte-code.
3. Le chargeur de classes.
4. La protection des fichiers et des accès au réseau.

27.1 Le langage et le compilateur

Les risques majeurs lors de l'exécution de code sont la manipulation involontaire de la mémoire abritant le code et les données (il s'agit alors d'une erreur), et la manipulation malintentionnée de celle-ci en vue de :

- détruire/modifier du matériel, du logiciel ou des informations sur la machine du client;
- publier des données considérées comme confidentielles par le client[1];
- rendre la machine du client inutilisable en accaparant ses ressources.

Les deux premiers points sont pris en charge par les concepts du langage et par le compilateur (le programme qui génère du byte-code).

Les points suivants éliminent des risques de manipulation de la mémoire :

- l'absence de pointeurs : le programmeur ne peut pas créer de fausses références à la mémoire, ni détourner des mécanismes d'assignation en vue de modifier le code du programme. Il ne peut pas explorer des zones de mémoire qui lui sont étrangères (les variables *private* d'autres objets);
- la vérification des bornes des vecteurs : le programmeur ne peut pas utiliser des indices en dehors des bornes comme moyen d'explorer ou de modifier la mémoire;
- la désallocation automatique de la mémoire interdit, d'une façon subtile, qu'on désalloue un objet tout en gardant une référence sur lui et que l'on attribue cet emplacement à un autre objet (dont le programmeur malintentionné aurait le contrôle);

1. Ce dernier point est assez difficile à éliminer. En effet, la nuisance est dans le comportement du programme, qui peut calculer sans interruption, être bruyant, ouvrir des fenêtres, lancer des processus. Il existe à la fois des programmes utiles et des programmes hostiles ayant un tel comportement. Néanmoins, de tels programmes hostiles ne peuvent que nuire temporairement, leurs auteurs risquant d'être rapidement ensevelis sous une avalanche de courrier électronique.

- la vérification stricte des types n'autorise pas la conversion implicite des types. Les droits d'accès aux méthodes et aux variables sont vérifiés à la compilation pour déterminer s'ils sont en accord avec les spécifications de visibilité (*protected, private,* etc.). Les entiers ne peuvent pas être convertis en objets et réciproquement.

Le code Java de l'interpréteur et du compilateur sont disponibles, ainsi que la spécification du langage. Ainsi, chacun peut théoriquement vérifier que le code est bien conforme à la spécification.

27.2 Le vérificateur de byte-code

Le passage à l'étape suivante repose sur le compilateur et sur la définition du byte-code. La machine virtuelle Java a été conçue afin de rendre le code portable sur des plates-formes physiquement différentes. De plus, le byte-code a été enrichi d'informations et d'instructions qui ne sont pas fonctionnellement utiles à l'exécution du code mais qui ont des visées de sécurité. Le compilateur génère ainsi des éléments de code qui sont structurellement vérifiables et qui possèdent de bonnes propriétés.

L'interpréteur de byte-code ne consommant pas directement le code généré par le compilateur, est obligé de refaire la preuve des bonnes propriétés du code qu'il doit exécuter. En effet, on pourrait imaginer que quelqu'un établisse un compilateur hostile ou modifie manuellement le fichier compilé d'une classe.

La vérification est dépendante de la machine virtuelle, caractérisée par :

- des types primitifs (six) : entiers 32 bits, entiers longs 64 bits, flottants 32 bits, doubles 64 bits, références à des objets, adresses de retour;
- des structures de vecteurs basées sur les types primitifs et étendues aux octets, aux entiers courts *(short)* et aux caractères *Unicode.* Ces structures contiennent des informations supplémentaires telles que le type d'objets acceptés par le vecteur;
- une structure de pile, avec des registres de manipulation de la pile. Chaque méthode reçoit une pile pour l'évaluation et un ensemble de registres;
- le jeu d'instructions de la machine; il est partitionné en catégories :
  - empiler des constantes;
  - manipuler la valeur d'un registre;
  - accéder aux valeurs d'un vecteur;
  - manipuler la pile (empiler, dépiler, échanger, etc.);
  - instructions arithmétiques, logiques, de conversion;
  - instructions de transfert du contrôle;

- fonction de retour;
- manipulation des champs d'un objet;
- invocation de méthode;
- création d'un objet;
- conversion de type (casting).

Les instructions sont éventuellement suivies par des arguments sur leurs opérandes.

L'espace mémoire de la machine est réparti en différentes zones :

- une zone mémoire où seront créés les objets;
- une zone de constantes hétérogènes *(pool)*;
- une zone de description pour le byte-code;
- une zone d'activation pour chaque méthode comprenant une pile et des registres locaux.

Tester le format du fichier .class

La première tâche du vérificateur est de tester le format du fichier de byte-code qu'il est en train de charger. Le fichier de byte-code d'une classe comprend :

- une constante magique;
- des informations à propos des versions;
- des informations sur la classe (nom, super-classe, interface, etc.);
- des informations sur tous les champs et les méthodes de cette classe;
- des informations de mise au point *(debugging)*;
- un raton laveur (mais oui, voyons);
- la description de la zone de constantes contenant :
  - les valeurs des *String*;
  - les chaînes de caractères Unicode;
  - les références aux champs et aux méthodes;
  - les noms des classes et des interfaces;
- pour chaque méthode, nous avons :
  - le nom et la signature de la méthode;
  - la taille maximum utilisée sur la pile;
  - le nombre maximum de registres locaux;
  - une table des exceptions;
  - les bytes-codes de la méthode.

Le fichier lu, on vérifie son nombre magique (pour s'assurer que la transmission est correcte), ensuite on vérifie le format du fichier (attribut et longueur des attributs). Le fichier doit être complet, sans ajout ni élimination. La structure de la zone de constantes doit être correcte.

Tester la vraisemblance

La deuxième étape va tester la vraisemblance des structures et des références dans la zone de constantes. Les vérifications suivantes sont effectuées :

- la structure des classes finales n'est pas étendue (idem pour les méthodes finales);
- toutes les classes ont une super-classe;
- toutes les références à des champs et à des méthodes ont des noms bien formés et des signatures légales dans la zone de constantes.

Dans cette étape, on ne cherche pas à savoir si les noms existent vraiment, mais seulement si la forme des noms est correcte.

Prouver la description

Cette étape est décisive. Il s'agit de vérifier pour chaque méthode que son modèle d'exécution est correct. Ce modèle est le suivant :

- à chaque point du programme, on peut associer un état de la pile (longueur et type des éléments); cet état est indépendant des possibilités d'exécution qui précèdent ce point;
- les registres locaux ne sont pas accédés avant d'avoir reçu une valeur (de type compatible);
- les méthodes sont invoquées avec des arguments corrects;
- les champs des objets sont modifiés avec des valeurs spécifiées par leur type;
- les *bytes-codes* ont des arguments corrects (par rapport aux registres locaux, à la pile et à la zone de constantes).

Concentrons-nous sur la première contrainte : elle nécessite de construire un modèle d'exécution de la classe simulant les différentes possibilités, afin de vérifier que chaque chemin mènera bien au même état. On va ainsi transformer les bytes-codes en flux de données.

Les bytes-codes vont être regroupés en séquences d'instructions. Chaque instruction doit comporter un début et une fin; quitter l'instruction au milieu est interdit, se brancher au milieu également. Chaque instruction va être vérifiée (accès aux registres locaux, références à la zone des constantes).

À chaque instruction va être associée une précondition qui mémorise l'état de la pile et l'utilisation des registres locaux en termes de types.

Les bytes-codes de la machine virtuelle sont conçus de manière à ce que l'état de la pile soit entièrement déterminé par le type des opérandes et le code à exécuter. L'état de la pile n'est donc pas conditionné par les valeurs (mais seulement par les types).

L'analyse du flux de données ne prendra en compte que les types des objets.

La première instruction de la méthode est initialisée avec une pile vide, les registres locaux contenant les types indiqués par la signature de la méthode. Cette instruction est marquée à vérifier.

L'algorithme de l'analyseur est conceptuellement le suivant :

1. Chercher une instruction marquée « à vérifier ». S'il n'y en a plus, la méthode est vérifiée.
2. Simuler l'effet de l'instruction sur la pile et sur les registres :
   a) en vérifiant qu'il n'y a pas de dépassement des dimensions de la pile (*underflow* et *overflow*) par rapport à la taille maximum indiquée pour la méthode;
   b) en vérifiant la conformité des types utilisés sur la pile;
   c) en vérifiant que les registres utilisés soient initialisés et accessibles à la méthode (nombre maximum de registres).

   À la fin du point 2, on a déterminé un nouvel état de la pile et des registres.
3. Déterminer la prochaine instruction à exécuter, qui peut être soit :
   a) la prochaine instruction;
   b) les branches d'une instruction conditionnelle;
   c) un retour ou une émission d'exception.
4. Déterminer l'état de la pile pour les instructions suivantes :
   a) si la nouvelle instruction est visitée pour la première fois, on initialise cet état avec celui obtenu en 2);
   b) s'il existe déjà un état, il faut fusionner les deux états. Les piles doivent être de taille égale. Les types doivent être égaux. Dans le cas de conflit entre deux classes, on choisit l'ancêtre commun le plus proche. Pour fusionner les registres, on vérifie la compatibilité des types; en cas d'incompatibilité, le registre prend l'état indéfini.

Si la précondition de l'instruction suivante est modifiée, elle est marquée comme à vérifier; la marque de l'instruction examinée est effacée, on recommence au point 1. Toute anomalie survenant pendant la vérification provoque l'abandon de la vérification avec un statut négatif.

Nous présentons des exemples qui montrent bien le principe de cette vérification (nous restons ici au niveau du compilateur qui utilise des mécanismes similaires).

Soit deux classes *O1* et *O2* :

```
class O1 {
 int f1() {return 1;}
}
class O2 {
 int f2() {return 2;}
}
```

Le programme suivant (*Securite1*) tente de construire un état ambigu à l'issue de la première condition : soit *o1* est créé, soit *o2* est créé. La deuxième condition risque d'utiliser une variable non initialisée. Le compilateur ne peut accepter cette ambiguïté et la signale.

```
// Classe refusée à la compilation:

class Securite1 {
   public static void main (String args[]) {
    int i=1;
    O1 o1; O2 o2;
    if (i==1) o1=new O1();
      else o2=new O2();
    if (i==1) System.out.println(o1.f1());
// variable o1 peut ne pas être initialisée
      else System.out.println(o2.f2());
// variable o2 peut ne pas être initialisée
   }
}
```

On peut réécrire ce programme d'une autre manière (*Securite2*). Le compilateur l'accepte, car la vérification des conversions de type allant du général au particulier est repoussée à l'exécution. Ces conversions doivent donc être utilisées avec prudence car elles peuvent être source d'erreurs à l'exécution.

```
// Classe acceptée à la compilation pouvant générer une erreur // à
l'exécution

class Securite2 {
   public static void main (String args[]) {
    int i=1;
    Object o;
    if (i==1) o=new O1();
    else o=new O2();
    if (i==1) System.out.println(((O1)o).f1());
    else System.out.println(((O2)o).f2());
   }
}
```

Pour ceux qui croient encore que l'on peut s'en passer, voici un dernier exemple de programme montrant que les vérifications statique et dynamique des types sont une affaire d'ordinateurs.

```
class O1 {
   int f1() {return 1;}
   O1 f(O2 o) {return new O1();}
   O2 f(O1 o) {return new O2();}
}
class O2 {
   int f2() {return 2;}
   O1 f(O2 o) {return new O1();}
   O2 f(O1 o) {return new O2();}
}

class Securite {
   public static void main (String args[]) {
      int i=1;
      O2 o2=new O2();
//oui l'expression est bien un O1!
      O1 o1=o2.f(o2).f(o2.f(o2)).f(o2).f(o2.f(o2).f(o2.f(o2)).
         f(o2)).f(o2.f(o2).f(o2.f(o2)).f(o2)).f(o2).f(o2.f(o2).
         f(o2.f(o2)).f(o2).f(o2.f(o2).f(o2.f(o2)).f(o2)).
         f(o2.f(o2).f(o2.f(o2)).f(o2)).f(o2)).f(o2.f(o2).
         f(o2.f(o2)).f(o2).f(o2.f(o2).f(o2.f(o2)).f(o2)).
         f(o2.f(o2).f(o2.f(o2)).f(o2)).f(o2)).f(o2);
      System.out.println(o1.f1());
   }
}
```

Optimiser l'exécution

En cas de succès, le vérificateur effectue un dernier passage sur le code pour remplacer certaines instructions par leur version rapide (*_quick*). Par exemple, les codes *new*, *invokestatic*, *anewarray*, etc., sont remplacés par *new_quick*, *invokestatic_quick*, *anewarray_quick*, etc.

À partir de ce moment, pour exécuter le byte-code, on peut utiliser :

- un simple interpréteur de byte-code : les instructions sont interprétées à chaque exécution;
- un interpréteur associé à un compilateur à la volée : les instructions sont compilées (code natif de la machine cliente) lors de leur première exécution et ce code est exécuté lors des exécutions ultérieures;
- un compilateur de *byte-code* créant du code natif;
- et bientôt sans doute un processeur *Java* (projet de Sun Microsystems) qui exécutera directement le byte-code.

27.3 Le chargeur de classes

Lors de cette dernière étape, qui a lieu pendant l'exécution du code, on charge les classes (non encore chargées) invoquées par l'instruction en cours d'exécution. On vérifie à ce moment que cette instruction est autorisée à référencer la classe chargée.

De même, on vérifie pour chaque accès à un champ ou à une méthode appartenant à une classe chargée dynamiquement que la méthode ou le champ référencé existe, que les signatures sont compatibles et les droits de visibilité respectés.

Chaque classe est chargée dans un espace de noms qui lui est propre. Il est donc impossible qu'une des classes de base (*java.lang*, par exemple) soit remplacée par une classe chargée dynamiquement. Les classes gardent toujours la marque de leur provenance (le nom complet du package, qui indique le nom du serveur).

Ces vérifications peuvent détecter des changements de définitions dans les classes chargées.

À l'issue de la vérification, on a une assurance formelle sur la qualité du byte-code. Cela permettra d'éviter certains tests sur la compatibilité des types, sur l'utilisation de la pile, etc. , car on a déjà l'assurance que ces contraintes sont respectées (l'interpréteur de byte-code sera donc plus performant).

27.4 La protection des fichiers et des accès au réseau

L'utilisateur peut définir des droits par rapport à l'accès aux fichiers locaux et aux ressources de communication. Le droit d'accès aux fichiers locaux défini par défaut est demande l'approbation de l'utilisateur[1].

Les droits de communication sont déterminés par la combinaison entre la source du byte-code et l'emplacement de son exécution.

Emplacement source	Emplacement exécution
Extérieur du mur pare-feu	Intérieur du mur pare-feu
Intérieur du mur pare-feu	Extérieur du mur pare-feu
Intérieur du mur pare-feu	Intérieur du mur pare-feu

Tableau 27.1 Table des droits de communication

1. Le butineur actuel de Netscape quant à lui ne prend aucun risque, il interdit l'écriture et la lecture locales.

On peut ainsi adopter différentes politiques comme, par exemple, ne pas autoriser les applets extérieures à communiquer à l'intérieur du mur pare-feu, ou autoriser l'applet à communiquer uniquement avec sa source (le serveur d'où l'on a chargé la classe), etc. Ajoutons encore que ces mécanismes de sécurité doivent être modifiés avec circonspection.

Finalement, si la sécurité doit être adaptée à une situation particulière, la classe *SecurityManager* peut être étendue, afin de supporter de nouveaux degrés de protection ou de nouveaux critères de protection.

Chapitre 28

Java et les bases de données

JDBC

La nature distribuée de Java, sa portabilité et ses possibilités d'accès à des machines distantes le positionnent comme un langage de choix pour le développement d'applications interagissant avec des systèmes de gestion de bases de données (SGBD). De plus, les mécanismes de sécurité propres à Java limitent les possibilités d'accès à des données locales (fichiers).

Dans ce contexte, les acteurs du monde des bases de données (vendeurs de SGBD, de *middleware*, d'outils CASE) ont développé des solutions propriétaires, c'est-à-dire spécialisées dans l'interaction entre Java et leur propre système.

Ainsi, une certaine pression s'est exercée sur les concepteurs de Java afin qu'ils ajoutent au langage des fonctionnalités d'accès à des bases de données.

La réaction se devait d'être rapide, la multiplication de solutions élaborées sans souci de standardisation étant contraire à l'esprit de Java et risquant de lui faire perdre l'un de ses avantages principaux, à savoir sa portabilité. Dans le cas de l'accès à des bases de données, la portabilité signifie une certaine indépendance vis-à-vis du SGBD et du protocole de connexion utilisés. Une telle indépendance ne peut être réalisée qu'en se basant sur un ensemble de classes et de méthodes communes, regroupées sous le terme d'interface de programmation d'applications (API, en anglais).

La réponse des concepteurs de Java a été la publication d'une proposition d'API définissant l'interaction entre un programme Java et une base de données (relationnelle dans un premier temps). Cette API, nommée JDBC (*Java DataBase Connectivity*), supporte les fonctionnalités de base de SQL (langage d'interrogation des SGBD relationnels). En ce sens, elle peut être considérée comme une API de bas niveau, simple et robuste (à l'image de Java), l'idée étant de construire par la suite des niveaux supérieurs tels que systèmes de transactions, éditeurs de schémas, etc.

JDBC est basée sur les définitions de X/Open SQL CLI (*Call Level Interface*), tout comme ODBC, la solution de Microsoft pour l'accès à des bases de données. JDBC présente ainsi de nombreuses similitudes avec ODBC et définit une interface Java pour les principaux concepts de X/Open CLI.

La spécification complète de JDBC est disponible chez Sun [4].

Nous reviendrons au point 28.3 sur les objectifs de JDBC et sur ses relations avec ODBC, SQL et Java.

28.1 Architecture de JDBC

JDBC est composée de deux ensembles d'interfaces abstraites :

- JDBC API;
- JDBC Driver API.

JDBC API

L'API JDBC est destinée aux développeurs de programmes Java (applets et applications) désirant interagir avec un SGBD relationnel. Elle définit les structures nécessaires à la connexion à une base de données, à l'envoi de requêtes SQL et à la réception de leurs résultats. Les principales interfaces définies dans l'API JDBC se trouve dans le tableau 28.1 .

Interface	Description
`java.sql.CallableStatement`	Gère l'invocation de procédures stockées.
`java.sql.Connection`	Gère les connexions existantes, crée les requêtes, gère les transactions.
`java.sql.Driver`	Gère les accès à un SGBD à travers un protocole.
`java.sql.DriverManager`	Gère le chargement des drivers et la création des connexions TCP.

Tableau 28.1 Interfaces de l'API JDBC

Interface	Description
`java.sql.PreparedStatement`	Représente une requête paramétrée.
`java.sql.ResultSet`	Gère l'accès aux tuples d'un résultat.
`java.sql.Statement`	Gère les requêtes à exécuter, reçoit les résultats.

Tableau 28.1 Interfaces de l'API JDBC

JDBC Driver API

L'interface *java.sql.Driver* est destinée aux développeurs de drivers (pilotes) désirant interfacer un SGBD à Java en utilisant JDBC. La programmation d'un driver JDBC consiste à implanter les éléments définis dans les interfaces abstraites de l'API JDBC.

28.2 Principaux éléments de JDBC

Les points suivants décrivent les principaux éléments (interfaces, classes) composant JDBC. Outre le *DriverManager*, gestionnaire des différents drivers disponibles, les notions de *Connection*, *Statement* et *ResultSet* réalisent l'interaction avec une base de données. Le choix de ces trois notions n'est pas propre à JDBC : en effet, d'autres API basées sur un découpage identique ont été commercialisées pour différents environnements de développement. Parmi celles-ci on peut citer :

- DB Kit de NeXT, pour l'environnement NEXTSTEP (Objective C);
- DataLens de ParcPlace, pour VisualWorks (Smalltalk);
- DBTools.h++ de Rogue Wave (C++).

DriverManager

Le *DriverManager* gère la liste des drivers JDBC chargés dans la machine virtuelle Java. Il sert également à mettre en correspondance les URL utilisés par les connexions avec les drivers à disposition. Ainsi, le *DriverManager* décortique les URL afin d'en extraire le sous-protocole et le service, puis recherche dans sa liste de drivers celui (ou ceux) capable(s) d'établir la connexion.

Dans le cas où plusieurs drivers sont disponibles, le *DriverManager* peut consulter une propriété Java (*sql.drivers*) que l'utilisateur a la possibilité de configurer; cette propriété regroupe une liste de noms de drivers. Le *DriverManager* parcourt alors cette liste dans l'ordre, choisissant le premier driver utilisable. Dans le cas où la propriété *sql.drivers* n'est pas définie, c'est l'ordre dans lequel les drivers ont été chargés qui sera considéré.

Précisons encore que la propriété *sql.drivers* est utilisée par le *DriverManager* au moment de son initialisation : il charge alors l'ensemble des drivers indiqués dans la liste, les plaçant ainsi en tête des drivers disponibles.

Connection

Une *Connection* représente un canal de communication (une connexion) entre un programme Java et une base de données. La création et la gestion des *Connections* sont prises en charge par le *DriverManager*.

Une *Connection* est créée à partir d'un URL décrivant le protocole et le service à utiliser. Lors de la création d'une *Connection*, des arguments tels que le nom de l'utilisateur et son mot de passe peuvent être fournis sous la forme d'objets de la classe *java.util.Properties*.

Une fois créée, une *Connection* va servir à envoyer des requêtes au SGBD et à récupérer les résultats, ces deux tâches étant effectuées à l'aide des interfaces *Statement* (requête) et *ResultSet* (ensemble résultat).

Précisons encore qu'une *Connection* peut servir à de multiples séquences requête/résultat et qu'il n'est donc pas nécessaire de créer systématiquement une nouvelle *Connection* pour chaque requête.

Transactions

L'interface *Connection* est capable de gérer des transactions SQL. Par défaut, une *Connection* est placée en mode de *commit* (confirmation) automatique : chaque requête est exécutée dans une transaction SQL séparée puis confirmée automatiquement. Ce mode peut être désactivé (méthode *setAutoCommit()*) afin de regrouper l'exécution de plusieurs requêtes en une seule transaction. En mode non automatique, la *Connection* maintient ouverte une transaction courante, qui peut être confirmée (*Connection.commit()*) ou annulée (*Connection.rollback()*) au moment voulu. La *Connection* libère alors cette transaction et en crée immédiatement une nouvelle.

Requêtes SQL dans JDBC

Les requêtes SQL sont représentées dans JDBC à l'aide des interfaces *Statement*, *PreparedStatement* et *CallableStatement*. Chacune de ces interfaces est spécialisée dans un type particulier de requêtes, comme le montre le tableau 28.2 .

Interface	Description
`Statement`	Requêtes SQL statiques.

Tableau 28.2 Fonctions des interfaces pour les requêtes SQL

Interface	Description
`PreparedStatement`	Requêtes SQL paramétrées.
`CallableStatement`	Procédures stockées paramétrées.

Tableau 28.2 Fonctions des interfaces pour les requêtes SQL

Avant de décrire ces différentes interfaces, précisons qu'une requête doit être obtenue auprès d'une *Connection* existante avant de pouvoir être préparée et exécutée. Le résultat d'une requête est obtenu auprès de la requête elle-même.

Statement

Statement est destinée aux requêtes SQL simples, c'est-à-dire celles dont l'expression SQL est prête à l'emploi, pouvant être transmises telles quelles au SGBD.

Une fois obtenue auprès d'une *Connection*, une requête peut être exécutée, son résultat prenant la forme d'un *ResultSet* (voir le point ResultSet) dans le cas d'un SELECT. Ce résultat peut alors être vérifié (valeur NULL, ensemble vide, etc.) et parcouru ligne par ligne. Les ordres DELETE, INSERT, UPDATE quant à eux retournent un entier.

PreparedStatement (extension de Statement)

PreparedStatement permet de créer une requête contenant des paramètres. Une telle requête est destinée à être préparée et exécutée plusieurs fois de suite, avec des valeurs de paramètres différentes. Son code SQL contient des caractères '*?*' indiquant l'emplacement des paramètres, qui sont ensuite remplacés par des valeurs claires au moment de l'envoi de la requête à la *Connection* associée.

La mise à jour des paramètres (*binding*) est réalisée en invoquant les méthodes *PreparedStatement.set...()* adaptées aux types des paramètres. Le binding est effectué avant chaque exécution, préparant ainsi la requête. À noter qu'un paramètre, une fois mis à jour, peut servir à plusieurs exécutions successives de la requête, la méthode *clearParameters()* permettant de vider l'ensemble des paramètres.

CallableStatement (extension de Statement)

CallableStatement est destinée à exécuter des procédures stockées dans le SGBD. Les procédures de ce type acceptent généralement des paramètres en entrée (IN) et fournissent d'autres paramètres (OUT) en sortie.

L'assignation des paramètres IN s'effectue à l'aide des méthodes *PreparedStatement.set...()*. Les paramètres OUT doivent être déclarés (*CallableStatement.registerOutParameter()*) avant l'exécution de la procédure : on invoque une fois cette méthode pour chaque paramètre OUT, en indiquant son type SQL.

Après l'exécution, les paramètres OUT peuvent être récupérés à l'aide des méthodes *CallableStatement.get...()* adaptées à leurs types.

ResultSet

Une requête SQL produit différents types de résultats. Dans le cas d'une insertion, d'une mise à jour ou d'une suppression, le résultat retourné par le SGBD est généralement un nombre entier indiquant uniquement le nombre de tuples ayant été touchés par la modification.

Au contraire, dans le cas d'une sélection (ordre SELECT), le résultat, fourni sous la forme d'un ensemble de tuples géré par un *ResultSet*, est beaucoup plus intéressant.

Nous allons donc commencer par décrire le résultat d'une sélection avant de mentionner celui des autres ordres SQL.

Résultat d'un ordre SELECT

La méthode *executeQuery()* exécute une requête et retourne le résultat sous forme d'un *ResultSet*. Un *ResultSet* est un ensemble de lignes (chaque ligne représentant un tuple de la relation résultat); chaque ligne comporte le même nombre de colonnes (chaque colonne représentant un attribut de la relation résultat).

De nombreuses méthodes sont définies dans la classe *ResultSet* afin de pouvoir parcourir le résultat ligne par ligne et d'accéder de différentes manières aux colonnes de la ligne courante. Au niveau du programme Java, une simple boucle *while* combinée à la méthode *ResultSet.next()* permet de récupérer chaque ligne du résultat, comme le montre l'exemple suivant :

```
// admettons qu'une Connection 'conn' soit deja établie
// nous allons sélectionner certains tuples de la
// relation AMI dont les attributs sont:
// prenom (VARCHAR), age (INTEGER), remarque (VARCHAR)

java.sql.Statement requete = conn.createStatement();
ResultSet resultat = requete.executeQuery ("SELECT prenom,
age FROM AMI WHERE age > 14");
// on s'intéresse aux personnes de plus de 14 ans
while (resultat.next()) {
   // on récupère le prénom et l'âge de chaque tuple
   int a = resultat.getInt ("age");// accès par le nom de colonne
   string p = resultat.getString (1);// accès par l'index de colonne
   System.out.println (p + " est age(e) de " + a + "ans");
}
```

Cet exemple illustre les deux manières de référencer une colonne :

- à l'aide de son nom (ex. : "age");
- à l'aide de son index (ex. : *1* pour la première colonne).

Il existe ainsi deux versions de chaque méthode *ResultSet.get...()*, l'une acceptant une chaîne de caractères, l'autre un entier.

La notion de **curseur** est réalisée à l'aide de la méthode *getCursorName()* qui retourne un curseur associé au *ResultSet* courant. JDBC ne fournit pour l'instant qu'un support simpliste de la notion de curseur, qui sera certainement étoffé dans le futur. Une fois encore, l'absence de standard dans ce domaine justifie ce choix minimaliste.

Résultat d'un ordre UPDATE, INSERT ou DELETE

Les ordres de mise à jour ne retournent généralement que des indications concernant le nombre de tuples touchés par l'exécution ou concernant le statut de la requête (exécutée, refusée, etc.).

Le code ci-dessous montre un exemple de mises à jour effectuées à l'aide d'une *PreparedStatement*. Le nombre de tuples touchés par chaque exécution est récupéré dans la variable *nbTuples*.

```
java.sql.PreparedStatement requete = conn.prepareStatement("UPDATE
AMI SET remarque = ? WHERE age = ?");

// le 1er paramètre est fixé une fois pour toutes
requete.setString(1, "fête une dizaine");
for (ans = 10; ans < 120; ans += 10) {
   requete.setInt(2, ans);// 2ème paramètre: varie selon l'exécution
   int nbTuples = requete.executeUpdate(); // nombre de tuples
touchés
}
```

28.3 JDBC et les standards

La démarche des concepteurs de JDBC s'attache à faire accepter cette API par l'ensemble des acteurs concernés : développeurs de SGBD, d'outils CASE, d'environnements de programmation, etc. Leur choix s'est donc porté sur des éléments (syntaxe, formats, structures) connus et déjà utilisés dans le domaine, à savoir ODBC et SQL.

Dès lors, on peut se poser la question suivante : pourquoi ne pas avoir directement intégré ODBC dans Java plutôt que de développer une API spécifique?

Les points suivants tentent d'y répondre en décrivant tour à tour les rapports entre JDBC, ODBC, SQL et... Java.

JDBC et ODBC

La réponse à la question posée ci-dessus, à savoir pourquoi ne pas avoir intégré ODBC dans Java, se résume en quelques mots : ODBC est très (trop) lié au langage C.

Ainsi, ODBC se base sur des éléments propres au C tels que les types, les pointeurs ou l'usage des (*void **). Compte tenu des différences entre Java et C dans ce domaine (voir annexe A), une intégration d'ODBC dans Java n'aurait pu s'effectuer que de manière forcée, même si elle était techniquement réalisable.

D'autre part, ODBC passe pour être un système compliqué, difficile à apprendre et à mettre en œuvre, y compris pour des utilisations simples. Enfin, la paramétrisation d'ODBC s'effectue à l'aide de fichiers de configuration stockés localement, ce qui va à l'encontre des principes de sécurité de Java (du moins en ce qui concerne les applets).

Ces différents aspects montrent que l'intégration Java-ODBC s'écarte de la philosophie Java en matière de simplicité, d'uniformité et de robustesse. Ainsi, sans jamais remettre en question les qualités et les avantages du couple C-ODBC (disponibilité sur de nombreuses plates-formes, large diffusion, performance, etc.), les concepteurs de Java ont choisi de réaliser dans un premier temps l'API JDBC que l'on peut situer au-dessus de la couche ODBC, celle-ci jouant le rôle de sous-protocole et pouvant être utilisée à l'aide de passerelles JDBC-ODBC.

Un exemple complet en fin de chapitre, utilise cette passerelle.

JDBC et SQL

Le but de JDBC est de permettre à un programme Java de transmettre des ordres SQL à une base de données et de récupérer les résultats éventuels dans des structures Java (objets, tableaux, etc.) afin de les traiter.

JDBC supporte les fonctionnalités de base de SQL, telles que définies dans la norme SQL2 Entry Level. Ce choix quelque peu restrictif se justifie par le fait qu'il représente le dénominateur commun en matière de SQL entre les principaux SGBD relationnels. Ici encore, l'idée des concepteurs de JDBC est d'offrir rapidement un noyau de bas niveau commun et standardisé, des fonctionnalités de plus haut niveau pouvant être définies par la suite sous forme de nouvelles API ou d'outils Java invoquant JDBC.

À cet effet, des mécanismes d'extension de JDBC ont été prévus : éléments de syntaxe, conversions de types. Les principes suivants ont été retenus :

- JDBC permet à un programme Java de transmettre n'importe quelle requête à un SGBD, autorisant ainsi l'utilisation des fonctionnalités propres à celui-ci. Cette possibilité risque néanmoins de compromettre sérieusement la portabilité de ce programme sur un autre SGBD;
- un label de conformité JDBC Compliant™ a été défini par Sun. Un driver JDBC désirant obtenir ce label devra se soumettre à des tests montrant qu'il supporte au minimum SQL2 Entry Level. Dès que ce label se sera imposé comme LA référence en matière de driver JDBC,

les programmes Java pourront utiliser cet ensemble d'ordres tout en restant portables.

À plus long terme, le but est de supporter la totalité de la norme SQL2, lorsque celle-ci aura vraiment été adoptée et surtout lorsque les principaux SGBD la supporteront de manière complète.

JDBC et Java

JDBC, c'est du Java! Cette exclamation reflète bien le souci des concepteurs de JDBC de conserver l'**esprit** Java dans leur API. En effet, l'immense succès de ce langage est très lié à sa relative simplicité et à l'élégance de bon nombre de ses concepts. Dès lors, JDBC, comme toutes les (futures) API Java, se doit d'apparaître comme une extension naturelle du langage et non comme un « emplâtre » ajouté de force.

Influence de Java sur JDBC

Les principes de base de Java se retrouvent dans l'ensemble de l'API. Ainsi, l'utilisation d'expressions fortement typées est encouragée, afin de maximiser les contrôles d'erreurs à la compilation (cet aspect du typage fort pose d'ailleurs quelques problèmes).

Néanmoins, la nature dynamique des résultats de certaines requêtes SQL est prise en compte par JDBC. De même, l'API est définie de manière relativement simple et cohérente, n'offrant généralement que peu d'alternatives à la réalisation d'une tâche.

Enfin, JDBC, comme d'autres librairies Java, utilise abondamment les principes de la programmation orientée objet, comme par exemple l'héritage, le polymorphisme ou les interfaces. Un nombre relativement important de méthodes a ainsi été défini, méthodes spécialisées dans une tâche précise et facile à comprendre.

28.4 Utilisation de JDBC

Les distinctions faites en matière de sécurité entre une applet Java et une application Java restent valables dans le contexte de JDBC :

- pour une applet : le stockage local est interdit et les connexions à travers un driver se limitent à la machine d'où provient le driver;
- pour une application : dans ce cas, l'accès aux ressources locales et au réseau peut être considérablement étendu, permettant ainsi le stockage local de données ou l'accès simultané à plusieurs SGBD.

Ces éléments font pencher la balance en faveur des applications Java lorsqu'il s'agit d'accéder à différentes bases de données disséminées sur le réseau. De

même, la possibilité de stocker des données de manière locale peut paraître indispensable, tout au moins dans des cas d'une certaine complexité.

La solution de l'applet reste intéressante dans les cas les plus simples (et ils sont nombreux !), l'intégration de l'applet dans un document HTML restant son principal avantage.

28.5 JDBC et la sécurité

Les problèmes de sécurité sont bien évidemment traités de manière sérieuse dans JDBC, ceci d'autant plus que, par définition, les accès à des bases de données s'effectuent à travers le réseau. À ce niveau, les concepteurs de JDBC distinguent les trois cas suivants :

- sécurité des applets;
- sécurité des applications;
- sécurité des drivers.

Sécurité des applets

Dans le cas des applets contenant des appels à JDBC, des règles ont été établies afin de garantir un niveau de sécurité équivalent à celui d'une applet n'invoquant pas JDBC. Ainsi, un driver JDBC une fois téléchargé ne pourra effectuer de connexions qu'avec des bases de données situées sur la machine d'où il provient. Il servira alors de point de passage commun aux différentes connexions établies par des classes provenant de cette machine.

De même, une applet Java ne pourra établir de connexions qu'avec des bases de données situées sur la machine d'où elle est issue. Enfin, une applet ne devrait pas être autorisée à accéder à d'éventuelles données locales.

Si le conditionnel est parfois utilisé dans ce texte, c'est que certaines règles proposées par les concepteurs de JDBC peuvent ne pas être respectées par certains drivers, d'où l'intérêt du label mentionné au point JDBC et SQL, p. 380.

Sécurité des applications

Une application Java, avec ou sans appels à JDBC, est généralement considérée comme sûre. Dès lors, les restrictions citées ci-dessus ne sont pas applicables. Néanmoins, un cas de figure reste délicat : lorsqu'un driver est chargé depuis une machine, alors ce driver ne devrait être utilisé que par du code provenant de cette même machine.

Sécurité des drivers

Les développeurs de drivers doivent quant à eux observer certaines règles élémentaires de sécurité afin de protéger leur code contre des utilisations abusives,

aussi bien par des applets locales (attaque depuis l'intérieur) que par du code téléchargé (attaque depuis l'extérieur). En effet, le driver est un maillon sensible de la chaîne de connexion entre un client et un serveur.

Les recommandations des concepteurs de JDBC concernent plus particulièrement la phase d'ouverture des connexions réseau (TCP/IP). Dans le cas le plus simple d'une connexion non partagée, les contrôles effectués par le *SecurityManager* suffisent à garantir la sécurité d'une connexion, c'est-à-dire que l'objet Java demandant la connexion est bien autorisé à accéder à la machine en question.

La différence apparaît lorsqu'un driver désire pouvoir partager une connexion TCP entre plusieurs *Connections* JDBC (par exemple, lorsque plusieurs processus interrogent la même base de données en parallèle). Dans ce cas, c'est au driver de vérifier que les différents clients (threads, classes) sont tous bien autorisés à accéder à la base de données définie dans la connexion TCP partagée.

Là encore, l'utilisation du *SecurityManager* est recommandée afin de vérifier les droits d'accès. Le développeur d'un driver doit également veiller à la sécurité des accès aux ressources du système local. Le comportement sera différent selon qu'il s'agit d'accès uniques ou partagés.

Pour conclure, précisons encore que toutes les méthodes d'un driver doivent supporter des accès concurrents (multithread, voir chapitre 25).

28.6 Formats d'URL et JDBC

Le système de nommage utilisé dans JDBC pour décrire une connexion (driver, protocole, machine, etc.) est directement inspiré de la notion d'URL (voir le point URL, p. 11). Contrairement à ODBC, JDBC ne suppose pas l'existence de fichiers de configuration contenant les informations nécessaires à une connexion. Ce principe typiquement Java place la gestion de ces informations au niveau des connexions elles-mêmes.

La syntaxe des URL recommandée est la suivante :

```
jdbc:sous-protocole:service
```

- la chaîne *jdbc* indique le protocole, au même titre que *http;*
- le sous-protocole est le mécanisme de connectivité utilisé (*odbc* par exemple);
- le service (*subname* en anglais) est une chaîne de caractères indiquant la machine, le port et la base de données utilisés. Exemple de service :

  ```
  mycpu.unige.ch:9876:mybase
  ```

Le mécanisme des URL a été retenu comme format générique de description des connexions JDBC, en raison de son statut de standard en matière de nommage des ressources sur le Web. Néanmoins, la syntaxe pourra varier selon les

environnements (drivers, sous-protocoles, SGBD). Dans le cas du sous-protocole *odbc* (en minuscules dans les URL), une syntaxe a déjà été spécifiée afin de pouvoir inclure des valeurs de paramètres dans le nom du service.

Équivalences de types entre Java et SQL

De nombreuses équivalences ont été établies entre les types de données, aussi bien dans le sens Java-SQL (envoi de paramètres) que dans le sens SQL-Java (résultats et paramètres retournés).

Ces équivalences se reflètent également dans les nombreuses méthodes de conversion à disposition du programmeur JDBC.

28.7 Base de données et Java : un exemple complet

L'application ci-dessous illustre la création de table, l'insertion, la mise à jour et la destruction de données. Elle contient également des interrogations ainsi qu'un exemple de traitement d'erreur.

Cette application est bâtie sur deux classes : *SQLService* et *ExampleWithJDBCOracle*. La classe *SQLService* traite les requêtes d'interrogation et de manipulation des données ou du schéma. Elle utilise quatre méthodes :

- *ExecDML* qui effectue les traitements sur les données ou le schéma;
- *ExecSelect* qui interroge puis affiche les résultats de la requête;
- *PrintResult* qui affiche les résultats;
- *PrintSQLError* qui explicite en détail une erreur de traitement sur la base, erreur signalée par une exception (*SQLException*).

```
import java.sql.*;
class SQLService {
// instance variables
public static String query;
public static Connection con;
public static Statement  stmt;
public static ResultSet  results;

public static String ExecDML(String sqlText) {
/* permet d'executer une commande de manipulation des donnees
   - insert, delete, update
   ou de manipulation du schema
   - create, drop, ...*/
  int ResultCode;

  try {stmt = con.createStatement();
       ResultCode = stmt.executeUpdate(sqlText);
       return ("OK ("+ResultCode+"): "+sqlText);
       }
```

```
  catch (Exception ex)
       {return ( "** Erreur dans : "+sqlText);}
}

public static String ExecSelect(String sqlText) {
/* permet d'executer une selection
   - select ... from ... where ...
   et d'afficher le resultat */
  try {stmt = con.createStatement();
       results = stmt.executeQuery( sqlText );
       PrintResults( results );
       return ("OK: "+sqlText);
      }
  catch (Exception ex)
       {return ( "** Erreur dans : "+sqlText);}
}

public static void PrintResults( ResultSet results )
  throws SQLException
     {int     i;
      int     nbLigne=0; // nbre de lignes du resultat
      int     nbCol;     // nbre de colonnes du resultat
      boolean pasfini= results.next (); // encore des données

     // recupere le schema du resultat (nom des colonnes, type, ...)
      ResultSetMetaData resultSchema = results.getMetaData ();
      nbCol= resultSchema.getColumnCount (); //nbre de colonnes

     // affiche les entetes des colonnes
      System.out.println();
      for (i=1; i<=nbCol; i++)
        {System.out.print(resultSchema.getColumnLabel(i));
         System.out.print("|"); // séparateur de colonne
        }
      System.out.println();

      // affiche les données ligne par ligne
      while (pasfini)
        { for (i=1; i<=nbCol; i++) // pour chaque colonne
            { System.out.print(results.getString(i));
              System.out.print("|");
            }
          System.out.println("");
          nbLigne++;
          pasfini = results.next (); //prochaine ligne
        }
     System.out.println(nbLigne+" lignes trouvees par la requete");
}

public static void PrintSQLError(SQLException ex) {
   System.out.println ("**ERREUR SQLException\n");
   while (ex != null)
```

```
        { System.out.println ("Etat   : " +ex.getSQLState ());
          System.out.println ("Message: " +ex.getMessage ());
          System.out.println ("Fournis: " +ex.getErrorCode ());
          ex = ex.getNextException ();
          System.out.println ();
        }
    }
}
```

La classe SQLService

L'application ci-dessous utilise la classe *SQLService* pour effectuer les manipulations suivantes :

- création de la table;
- insertion des données;
- mise à jour des données;
- suppression des données;
- interrogation à chaque manipulation pour vérifier l'effet;
- exemple de traitement d'erreur.

```
class ExampleWithJDBCOracle {
   static String DML="DML";
   static String SEL="SEL";


  public static void doIt(String typereq, String s){
     if (typereq.equals("DML"))
        System.out.println(SQLService.ExecDML(s));
     if (typereq.equals("SEL"))
        System.out.println(SQLService.ExecSelect(s));
  }

  public static void main (String args[]) {
    try { // Charger le driver jdbc-odbc bridge
          Class.forName ("sun.jdbc.odbc.JdbcOdbcDriver");
        // Connexion au driver
        //(fournir la source odbc, utilisateur, mot de passe)
          SQLService.con = DriverManager.getConnection
              ("jdbc:odbc:sgbdOracle", "javauser", "pwd");

          // ici, on est connecté, sinon on a généré une exception

        // Créer la table BANQUE, on commence par détruire la table!
          doIt(DML, "drop table BANQUE");
          doIt(DML, "create table "+
                    "BANQUE(NOM_COMPTE CHAR(20), MONTANT Number)");
          doIt(SEL, "Select * from BANQUE");

          doIt(DML, "insert into BANQUE values ('PourMoi' , 100)");
```

```
            doIt(DML, "insert into BANQUE values ('PourToi' , 100)");
            doIt(DML, "insert into BANQUE values ('PourLui' , 500)");
            doIt(DML, "insert into BANQUE values ('PourElle', 300)");

            doIt(SEL, "Select * from BANQUE");

            doIt(DML, "update BANQUE "+
                      "set MONTANT= 200 where NOM_COMPTE='PourToi'");
            doIt(SEL, "Select * from BANQUE");

            doIt(DML, "delete from BANQUE where MONTANT> 250");
            doIt(SEL, "Select * from BANQUE");

            // Close the statement
            SQLService.stmt.close();

            // Close the connection
            SQLService.con.close();

            // Une nouvelle connection avec une erreur
            SQLService.con = DriverManager.getConnection
                ("jdbc:odbc:sgbdOracle", "javatest", "?");
        }
        catch (SQLException e) {SQLService.PrintSQLError(e);}
        catch (Exception e   ) {e.printStackTrace ();}
    }
}
```

Application 15 : La classe ExampleWithJDBCOracle

L'exécution du programme donne le résultat suivant :

```
OK (-1): drop table BANQUE
OK (-1): create table BANQUE(NOM_COMPTE CHAR(20), MONTANT Number)

NOM_COMPTE|MONTANT|
0 lignes trouvees par la requete
OK: Select * from BANQUE
OK (1): insert into BANQUE values ('PourMoi' , 100)
OK (1): insert into BANQUE values ('PourToi' , 100)
OK (1): insert into BANQUE values ('PourLui' , 500)
OK (1): insert into BANQUE values ('PourElle', 300)

NOM_COMPTE|MONTANT|
PourMoi             |100.|
PourToi             |100.|
PourLui             |500.|
PourElle            |300.|
4 lignes trouvees par la requete
OK: Select * from BANQUE
OK (1): update BANQUE set MONTANT= 200 where NOM_COMPTE='PourToi'

NOM_COMPTE|MONTANT|
```

```
PourMoi             |100.|
PourToi             |200.|
PourLui             |500.|
PourElle            |300.|
4 lignes trouvees par la requete
OK: Select * from BANQUE
OK (2): delete from BANQUE where MONTANT> 250

NOM_COMPTE|MONTANT|
PourMoi             |100.|
PourToi             |200.|
2 lignes trouvees par la requete
OK: Select * from BANQUE
**ERREUR SQLException

Etat   : 28000
Message: [Oracle][ODBC][Ora]ORA-01017: invalid username/password;
logon denied
Fournis: 1017
```

En modifiant une seule ligne de ce programme, il est possible de se connecter à une autre source ODBC, dans notre cas MicroSoft Access (le nom de l'utilisateur et le mot de passe ne sont pas nécessaires) :

```
SQLService.con = DriverManager.getConnection
("jdbc:odbc:sgbdOffice");
```

28.8 Invitation à explorer

Ce chapitre vous a présenté les notions de base des interactions de Java avec les bases de données relationnelles via JDBC. De nombreux points n'ont été abordés que de manière succincte et mériteraient de plus longues explications. De même, certaines fonctionnalités n'ont pas été mentionnées afin de ne pas déborder du cadre de cet ouvrage. Parmi elles :

- les résultats multiples;
- les extensions de SQL : types (DATE), fonctions ODBC;
- la gestion des erreurs;
- les métadonnées.

Les métadonnées décrivent le dictionnaire de données d'un SGBD (catalogue des tables, des colonnes, etc.). Leur support dans JDBC a pour but d'en normaliser l'accès, facilitant ainsi la programmation d'outils de haut niveau (éditeurs de schémas, de requêtes, etc.).

Chapitre 29

Fabriquer des objets répartis avec RMI

Le package RMI (*Remote Method Invocation*[1]) permet l'élaboration de programmes utilisant des objets répartis sur des machines différentes. Les principes de RMI sont très proches de CORBA (*Common Object Request Broker Architecture*) qui définit une norme d'interopérabilité entre des objets.

29.1 Description du package

Le package RMI contient, outre ses propres interfaces, classes et exceptions, quatre sous-packages :

- *java.rmi.dgc* contient une interface et deux classes de gestion du ramasse-miettes distribué. Celui-ci piste les objets lors des transferts entre machines, et détruit les objets lorsqu'ils ne sont plus référencés;
- *java.rmi.registry* contient une classe et deux interfaces pour la gestion du référentiel d'implantation qui met en correspondance un nom avec son serveur d'objet;
- *java.rmi.server* contient les classes et interfaces pour gérer toute la partie serveur des objets distribués;
- *java.rmi.activation* contient les classes, interfaces et exceptions pour activer des objets à partir de leurs références persistantes. Les objets sont alors activés effectivement pour être utilisés.

1. Invocation de méthode à distance.

Dans le package *rmi*, l'interface *Remote* et la classe *Naming* sont essentielles à tout développement et utilisation d'objets distribués.

Les concepts de ce package étant parfois difficiles à comprendre, il nous est apparu plus utile d'en décrire les principes, puis d'en expliquer les composants à travers un exemple complet.

29.2 Principes de fonctionnement

Considérons un programme client (une application ou une applet) qui vous permet d'accéder à votre compte en banque géré par un programme serveur. Toute l'interface utilisateur est programmée dans le client, les traitements effectifs sur votre compte s'exécutent sur le serveur. Pour que cela soit possible, il faut que le client puisse accéder à l'objet compte du serveur et ensuite lui appliquer les méthodes correspondant aux opérations financières classiques sur un compte.

L'applet doit donc :

- savoir comment accéder à un compte particulier géré par un serveur;
- connaître les services (traitements) que peut effectuer un objet compte.

Cette connaissance mutuelle des services repose évidemment sur la notion d'interface Java, puisque celle-ci définit contractuellement les services que doit implanter une classe d'objets si celle-ci déclare implanter telle ou telle interface. Les programmes client et serveur ont donc connaissance de l'interface, le serveur parce qu'il implante les services de l'interface, le client parce qu'il les sollicite. Le courtier d'objet (*request broker*) a pour mission de faire circuler les demandes de service et les résultats de leur exécution entre les programmes client et serveur.

Au niveau du client, la demande de service est un appel habituel de méthode. À la différence près que l'exécution en local de la méthode ne réalise pas le service demandé, mais transmet au courtier d'objet la demande de service qui, à son tour, la transmet au travers du réseau au programme serveur approprié.

Le code qui implante la demande de service relayé par le courtier s'appelle une souche (*stub*). Le code qui reçoit du courtier la demande de service et qui la transmet au serveur s'appelle le squelette (*skeleton*).

Le squelette et la souche font également le transfert des résultats de l'exécution du service du serveur vers le client, qui termine alors l'exécution de la méthode pour ensuite passer à l'instruction suivante.

La seule question encore ouverte est comment disposer d'un mécanisme permettant au client d'identifier un objet réparti et de savoir si celui-ci est disponible ou activé. Pour ça, l'objet serveur s'enregistre sur un référentiel d'implantation que va interroger le courtier pour trouver la machine sur laquelle le serveur est en cours d'exécution.

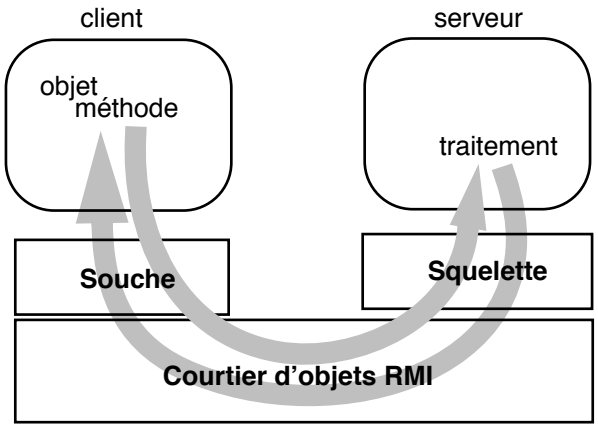


Figure 29.1 Principes de fonctionnement de RMI

29.3 Le développement d'une application distribuée avec RMI pas à pas

Nos allons décrire, pas à pas, l'implantation répartie de la gestion d'un compte en banque telle qu'elle a été décrite ci-dessus.

Définition de l'interface

```
public interface compteDistant extends java.rmi.Remote {
// depot d'argent
float depot(float amount) throws java.rmi.RemoteException;
// retrait d'argent float retrait(float amount) throws
java.rmi.RemoteException;
// lecture du solde
float litSolde() throws java.rmi.RemoteException;
}
```

Implanter l'interface dans un objet serveur

Un objet serveur est une sous-classe de *UnicastRemoteServer*, qui implante l'interface définie ci-dessus.

```
public class CompteServeur extends UnicastRemoteObject implements
compteDistant {
```

Les points clés sont les suivants :

- le constructeur de l'objet serveur doit appeler le constructeur de la classe dont il hérite;

```
public CompteServeur() throws java.rmi.RemoteException { super(); }
```

- l'objet serveur implante toutes les méthodes de l'interface (le code des méthodes ne tient absolument pas compte du fait que les méthodes vont être appelées par un objet distant);

- le lancement du serveur requiert l'installation d'un gestionnaire de sécurité et également l'enregistrement de l'objet serveur sur un référentiel d'implantation.

```
public static void main(String args[]) {
//installation d'un gestionnaire de sécurité
if (System.getSecurityManager()==null) {
System.setSecurityManager(new RMISecurityManager());
}
System.out.println("security manager installé");
try {
   // création d'un objet CompteServeur
CompteServeur name = new CompteServeur();
// enregistrement de l'objet sur le referentiel d'implantation
Naming.rebind("//site:host/gestionCompteDistant", name);
} catch (Exception e) {
   System.out.println("Exception: " + e.getMessage()); }
}
```

L'élaboration des squelette et souche se fait après la compilation du programme serveur avec l'utilitaire *rmic* du JDK :

```
>javac CompteServeur.java
>rmic CompteServeur
```

rmic construit deux fichiers *CompteServeur_stub.class* et *CompteServeur_Skel.class*. Ceux-ci seront téléchargés avec l'applet sur le poste client.

Le code complet de l'objet serveur est le suivant :

```
import java.rmi.* ;
import java.rmi.server.UnicastRemoteObject;

public class CompteServeur extends UnicastRemoteObject
implements compteDistant {
float solde = 0;
// constructeur
public CompteServeur() throws java.rmi.RemoteException {
super();
}
// depot montant et renvoie le nouveau solde
public float depot(float montant) throws java.rmi.RemoteException {
solde += montant;
return solde;
}
// renvoie le solde courant
public float litSolde() throws java.rmi.RemoteException {
return solde;
}
// retrait montant et renvoie le nouveau solde
public float retrait(float montant) throws java.rmi.RemoteException
{
solde -= montant;
```

```
return solde;
}
public static void main(String args[]) {
//installation d'un gestionnaire de sécurité
if (System.getSecurityManager()==null) {
   System.setSecurityManager(new RMISecurityManager());}
try {
   // création d'un objet CompteServeur
   CompteServeur name = new CompteServeur();
   // enregistrement de l'objet sur le referentiel d'implantation
   Naming.rebind("gestionCompteDistant", name);
} catch (Exception e) {
   System.out.println("Exception: " + e.getMessage()); }
} //main
}
```

L'applet déclare un identificateur sur l'interface *CompteDistant*. Elle obtient ensuite une référence sur un objet dont elle connaît le nom et qui implante cette interface; c'est le rôle de la méthode *Naming.lookup*. L'applet peut ensuite appeler les différentes méthodes de cet objet.

```
import java.awt.*;
import java.awt.event.*;
import java.rmi.*;
import java.net.MalformedURLException;
import java.net.UnknownHostException;
import java.applet.Applet;

public class AppletClient extends Applet implements ActionListener {
 // remote interface declaration
 private CompteDistant accesDistant;
 private Button Depot, Retrait;
 private Label Solde;
 private TextField saisieMontant;
 private float MontantSolde;

 //constructeur
 public AppletClient(){
  connexionServeur();
  demandeSolde();
  creeInterfaceGraphique();
 }

 public void connexionServeur(){
 // recherche de nom dans les registres
   try {
     accesDistant=(CompteDistant) Naming.lookup
                 ("//site:port/gestionCompteDistant");
   }
   catch (Exception e) {
     System.out.println("Exeception "+ e.getMessage());
```

```
    }
  }
  public void demandeSolde(){
    try {
      MontantSolde=accesDistant.litSolde();
    }
    catch (RemoteException e) {
      System.out.println("Exception "+ e.getMessage());
    }
  }

  public void creeInterfaceGraphique(){
    setLayout( new BorderLayout());
    Panel p=new Panel();
    p.setLayout( new FlowLayout(FlowLayout.LEFT));

    Solde=new Label(" Solde:  ");
    Solde.setText("  Solde: CHF "+MontantSolde);
    add("North",Solde);
    p.add(new Label("montant : "));
    saisieMontant=new TextField(3);
    p.add(saisieMontant);
    add("Center", p);

    p=new Panel();
    Depot= new Button("Depot");
    p.add(Depot);
    Depot.addActionListener(this);
    Retrait= new Button("Retrait");
    p.add(Retrait);
    Retrait.addActionListener(this);
    add("South",p);
  }

 public void actionPerformed(ActionEvent e) {
    String arg = e.getActionCommand();
    float solde, montant;
    try {
      montant = (new Float(saisieMontant.getText()).floatValue());
      }
    catch (NumberFormatException ex) {
       montant=0;
    }
    if ("Depot".equals(arg)) {
         try {
         solde= accesDistant.depot(montant);
         Solde.setText("  Solde: CHF "+solde);
         }
         catch (RemoteException re) {
              System.out.println("Exception "+ re.getMessage());
         }
    }
```

```
 else if ("Retrait".equals(arg)) {
      try {
         solde= accesDistant.retrait(montant);
         Solde.setText("  Solde: CHF "+solde);
      }
      catch (RemoteException re) {
           System.out.println("Exception "+ re.getMessage());
      }
 }
}

public static void main(String args[]) {
  try {
       Frame f = new Frame("Compte Client");
       AppletClient appletClient = new AppletClient();
       appletClient.init();
       appletClient.start();
       f.add("Center", appletClient);
       f.setSize(300, 300);
       f.show();
     }
   catch (Exception e) {
     System.out.println("Exception "+ e.getMessage());
    }
 }
}
```

Applet 56 : Une applet utilisant RMI

Le déploiement comporte trois phases :

- démarrage du référentiel d'implantation (*rmiregistry*);
- lancement du programme serveur, l'instance de *CompteServeur* va alors se déclarer auprès de ce référentiel;
- téléchargement et exécution de l'applet.

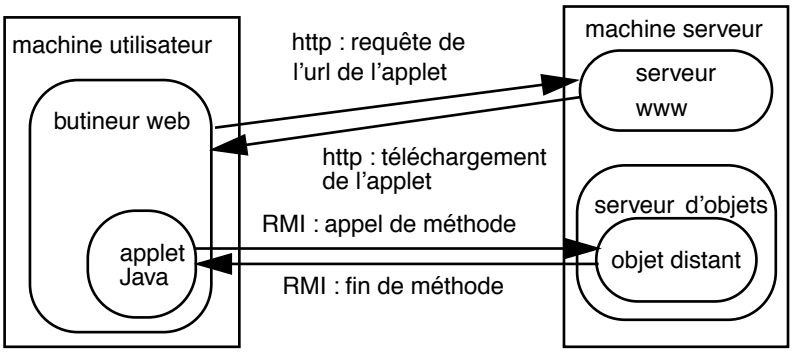


Figure 29.2 Déploiement et fonctionnement de la gestion de compte bancaire distribuée

La figure ci-dessous montre un exemple d'exécution de cette applet.

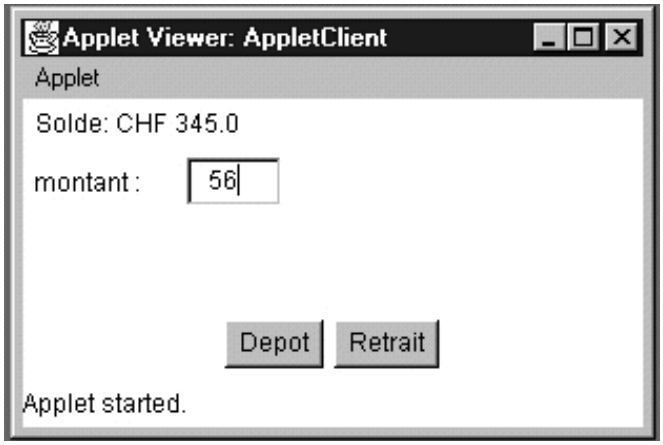


Figure 29.3 L'applet de gestion de compte en banque à distance

L'exemple que nous avons présenté n'illustre qu'une partie des fonctionnalités de RMI. Une méthode distante accepte des paramètres et peut renvoyer des résultats qui sont des objets et pas seulement des types simples. Le protocole de sérialisation (vu au point 19.15, p. 261) est utilisé pour l'envoi et la réception d'objet. On parle alors d'objet mobile, puisqu'il y a déplacement d'objet de machine en machine.

Chapitre 30

Fabriquer des objets répartis avec CORBA

CORBA (*Common Object Request Broker Architecture*) est une proposition commune de courtiers de services objets émanant de **l'OMG** (*Object Management Group*, www.omg.org), correspondant au standard d'un bus logiciel pour des objets répartis. Ce bus définit les interfaces de base pour la communication entre des objets distribués et hétérogènes. La distribution des objets mène directement à une architecture client/serveur ou, dans sa plus grande généralisation, à une architecture à plusieurs rangs (*multi-tiers* en anglais). L'hétérogénéité relève, elle, de la possibilité de faire communiquer des programmes développés dans des langages différents, s'exécutant sur des systèmes d'exploitation différents et sur des réseaux différents. CORBA permet l'intégration et la coopération de systèmes patrimoines (*legacy systems*) entre eux et avec les nouveaux applicatifs développés via des technologies plus récentes. C'est même son intérêt principal.

La version 2 de Java est livrée avec des packages permettant de tâter le développement avec CORBA. Il est possible de construire des objets CORBA en y faisant référence statiquement à travers des souches. Il est également possible de construire un *transient* serveur et de référencer les objets dans un serveur de noms livré par Sun. Un compilateur IDL vers Java est aussi disponible. Sun appelle cet ensemble **Java-IDL**.

Ce marché du *middleware* est en pleine ébullition et nous conseillons de bien observer les batailles qui s'y livrent pour le dominer. Les produits Orbix de IONA technologies et VisiBroker de Inprise (ex-Borland et ex-Visigenic) se dis-

putent actuellement le marché. À partir de la version 4, Netscape Navigator est livré avec l'ORB (*Object Request Broker*) VisiBroker. Et il est donc possible pour une applet d'utiliser les services de cet ORB pour accéder à des services CORBA. Pendant une période, Sun a aussi essayé de développer un produit ORB (sous le nom de NEO). Actuellement, la stratégie de Sun est de pousser sa technologie de *Enterprise Java Bean* chez les développeurs d'ORB.

RMI est l'équivalent de CORBA dans le monde Java. Leurs principes et leur fonctionnement sont similaires. Il est conseillé de lire le chapitre sur RMI avant de poursuivre celui-ci.

30.1 CORBA en quelques mots

L'idée de CORBA est de pouvoir distribuer les services d'objets, sans se préoccuper de connaître le lieu d'exécution ni la nature du code de ces objets. Il s'agit donc de services distribués et hétérogènes. CORBA est une norme qui définit comment publier, accéder, gérer, retrouver ces objets. Cette norme, établie par l'OMG qui regroupe plus de 800 entreprises, est purement définitoire. Le composant principal de son architecture est l'ORB (*Object Request Broker* ou courtier de requêtes objet), qui gère les échanges entre objets (demandes de services, résultats, passage des paramètres, types, etc.). La métaphore de l'ORB est le bus électronique de données que l'on trouve dans les ordinateurs et qui permet par sa définition de spécifier et brancher différentes cartes électroniques, le bus n'ayant pour fonction que la définition statique d'un standard électrique. L'implantation de l'ORB n'est pas définie dans la norme, ce sont les vendeurs d'ORB qui la prennent en charge.

Le bus d'objets répartis CORBA

La norme CORBA a pour objectif principal la spécification de ce bus qui se caractérise par les points suivants :

- **IDL, un langage de définition d'interface** : les services offerts par les objets répartis sont définis avec ce langage. À l'instar des interfaces Java, une interface écrite en IDL définit contractuellement un ensemble de services qui doivent être implantés par un programme. Ce programme peut être écrit dans la plupart des langages existants (C++, Smalltalk, Cobol, Ada, Java, etc.). Si des programmes déjà existants correspondent à ces services, ils pourront être encapsulés pour les rendre interopérables avec d'autres objets CORBA existants ou à créer. Cette encapsulation des services autorise, par exemple, la publication d'applications COBOL sous forme d'objets. Il est aisé de redonner une nouvelle vie à une comptabilité ou une gestion de stocks, écrites dans les années 80 en COBOL avec des fichiers séquentiels indexés, en

publiant certains services sous forme d'objets CORBA et en créant des applets Java accédant à ces services;

- **la transparence des appels** : une requête sur un objet distant (et donc potentiellement implantée dans un langage différent) s'exprime exactement comme une requête sur un objet local. La transmission et les conversions de représentation des paramètres sont entièrement prises en charge par les souches et le courtier d'objets, le développeur d'applications ne s'en occupant pas;
- **des appels statiques ou dynamiques** : l'appel statique, le plus simple à utiliser (celui proposé par le JDK 1.2), consiste à appeler un service d'un objet distant, objet que le programme appelant connaît nommément. L'appel dynamique utilise, quant à lui, un référentiel d'interfaces pour identifier un objet possédant un service que le programme appelant souhaite obtenir sans pour autant connaître *a priori* cet objet.

La norme CORBA définit également comment plusieurs ORB peuvent communiquer entre eux. Pour le protocole de réseau TCP/IP, cette spécification se nomme IIOP *(Internet Inter ORB Protocol)*. Cette norme rend les objets CORBA disponibles sur le réseau Internet, comme le sont des pages définies en HTML.

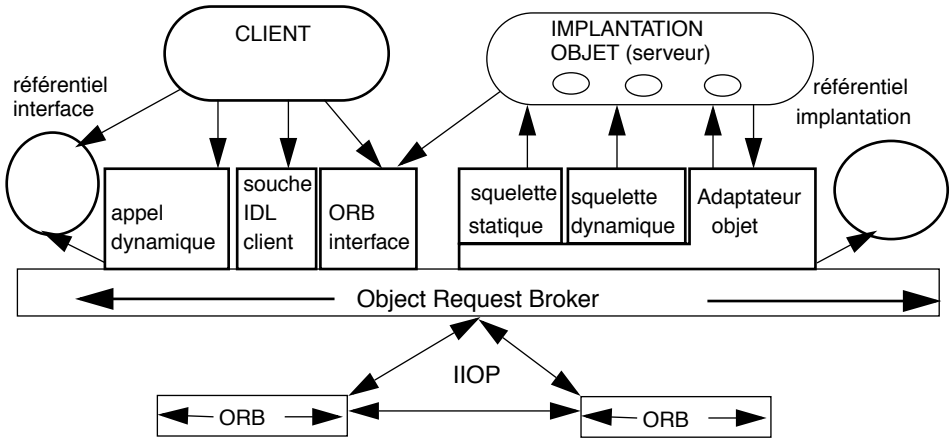


Figure 30.1 La structure de l'ORB

Les services et les commodités de CORBA

IDL est également utilisé dans CORBA pour définir différentes interfaces proposant des services utiles à la gestion d'objets distribués. Nous avons, entre autres, les services suivants :

- service de **nommage** : permet la recherche de la référence d'objets sur le bus à travers des annuaires;

- service de **persistance** : définition d'une interface pour gérer la persistance des objets;
- service de contrôle de la **concurrence** : définition d'une interface pour gérer la concurrence et autoriser le verrouillage des objets par les transactions;
- service **transactionnel** : interface pour définir les mécanismes transactionnels sur les objets;
- etc.

Les commodités de CORBA définissent un ensemble de composants pour la gestion et la création de systèmes d'information. Ils sont bien sûr définis en IDL. Voici la liste des domaines ayant retenu l'attention de l'OMG :

- gestion des documents;
- gestion des informations;
- administration des systèmes;
- gestion des tâches.

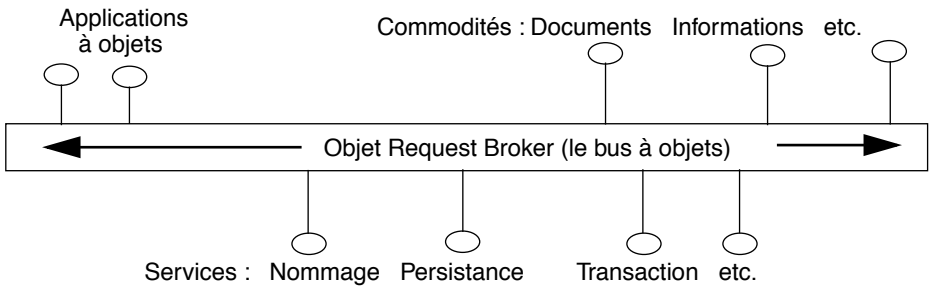


Figure 30.2 OMA (*Object Management Architecture*) de L'OMG

30.2 IDL

IDL (*Interface Definition Language*) est le langage pour définir les services devant être rendus par les objets. Les requêtes à ces objets seront acheminées par l'ORB du client vers le serveur. En retour, les résultats de ces requêtes seront acheminés par l'ORB du serveur vers le client. L'ORB adapte les paramètres et transmet les requêtes, tout en masquant s'il y a lieu l'hétérogénéité des systèmes. Des clients et des serveurs programmés dans des langages différents cohabitent ainsi grâce à CORBA. C'est la dimension interopérable de CORBA.

Les services spécifiés en IDL doivent être implantés dans le langage cible. Il faut disposer d'un traducteur d'IDL vers ce langage cible, qui est un langage de spécification purement déclaratif (il n'est pas possible d'exprimer l'implantation d'une méthode). On trouve des traducteurs pour C, C++, Smalltalk, COBOL, Ada, Java, etc.

La structure IDL est la suivante : un module contient des interfaces et ces dernières, des opérations. Cela correspond aux notions de packages, interfaces et méthodes de Java. En reprenant l'exemple du chapitre sur RMI, on peut définir le comportement d'un compte bancaire avec la déclaration IDL suivante :

```
module LikeCompte
{
  interface Compte
  { float litSolde();
    float retrait (in float x);
    float depot   (in float x);
  };
};
```

Le module IDL de compte en banque

Sans entrer dans les détails de la syntaxe d'IDL, on peut reconnaître les différentes méthodes de l'interface *CompteDistant* du chapitre sur RMI. Idltojava est le compilateur qui permet de transcrire les concepts IDL en Java[1], en utilisant la commande suivante :

```
idltojava -fno-cpp Compte.idl
```

Cette commande va créer un dossier *LikeCompte* dans lequel on trouvera les résultats de la traduction. Voici les fichiers générés :

- *Compte.java* : interface à implanter;
- *_CompteStub.java* : fonctionnalités CORBA pour le client (la souche statique);
- *_CompteImplBase.java* : classe abstraite servant de guide pour l'objet à implanter;
- *CompteHolder.java* et *CompteHelper.java* : classes de services pour la conversion de type (entre l'ORB et Java).

En examinant *Compte.java*, on peut constater la traduction énoncée plus haut.

```
package LikeCompte;
public interface Compte extends org.omg.CORBA.Object {
    float litSolde();
    float retrait(float x);
    float depot(float x);
}
```

Le module IDL de compte en banque traduit en interface Java

1. À partir de cette déclaration IDL, nous aurions pu générer un programme C++ pour illustrer l'hétérogénéité permise par CORBA. En restant dans Java, les différences entre des applications distribuées avec RMI ou CORBA sont plus visibles.

La figure ci-dessous illustre le principe de développement d'objets répartis dans des langages différents. Jusqu'à présent, nous n'avons développé que la partie client (partie gauche de la figure).

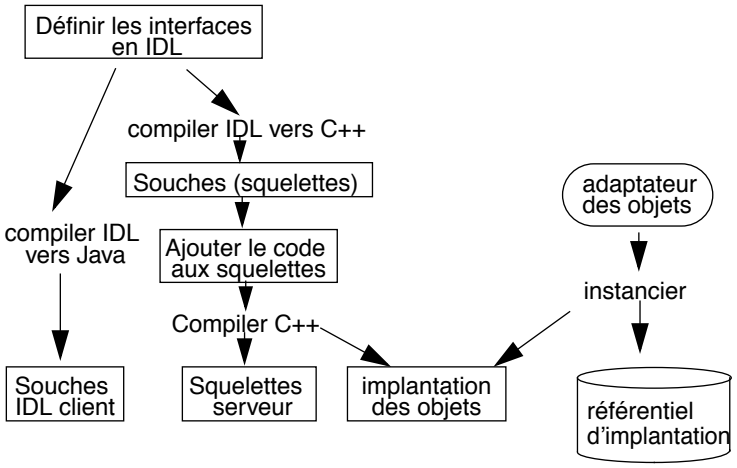


Figure 30.3 Principes de développement d'objets distribués CORBA

30.3 Un objet CORBA

La prochaine étape est donc de développer une classe qui étende le comportement de la classe abstraite *_CompteImplBase.java* et implante l'interface *Compte*. Sans surprise, nous obtenons le code suivant :

```
public class CompteImpl extends LikeCompte._CompteImplBase {
  private float v;
  public CompteImpl(java.lang.String name) {
    super();
    System.out.println("Objet Compte disponible : "+name);
  }
  public float litSolde() {
    return v;
  }
  public float retrait(float x) {
    v-=x; return v;
  }
  public float depot(float x) {
    v+=x; return v;
  }
}
```

La classe CompteImpl

30.4 Le Serveur CORBA

Nous avons donc maintenant une classe *CompteImpl* permettant d'instancier des objets qui seront manipulables à travers un ORB. Il nous reste deux choses à faire. La première est de développer le serveur qui abritera les instances de ces objets et leur donnera des noms. La seconde est d'adapter l'applet du chapitre sur RMI pour qu'elle utilise les objets communiquant via l'ORB. Comme nous allons utiliser les services de nommage, un annuaire qui associe un nom à la référence d'un objet, commençons par décrire la dynamique de la situation :

1. Le serveur localise le service de nommage.
2. Le serveur crée l'objet et enregistre sa référence dans le référentiel d'implantation (qui est en fait un serveur de noms).
3. Le client localise le service de nommage.
4. Le client demande la référence de l'objet qu'il cherche, au référentiel d'implantation.
5. Le client utilise cette référence pour adresser ses requêtes à l'objet (en passant par son serveur).

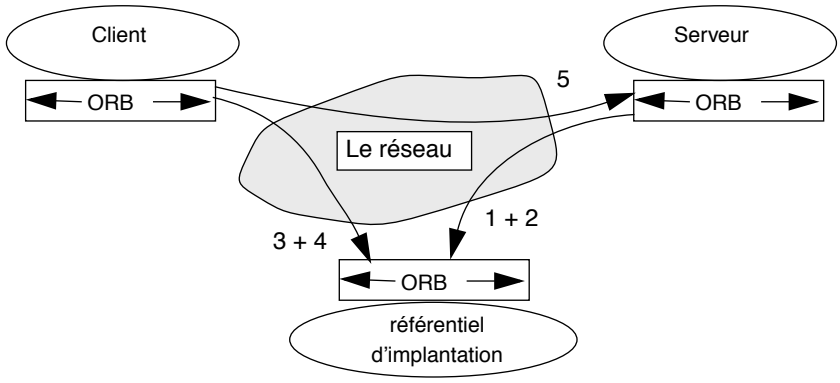


Figure 30.4 Dynamique des appels

Cette médiation du service de nommage permet de rendre complètement indépendants client et serveur. Examinons le code source du programme du serveur.

```
import LikeCompte.*;        // Le package contenant nos stubs
import org.omg.CosNaming.*; // serveur de noms : le ref.
d'implantation
import org.omg.CosNaming.NamingContextPackage.*;
import org.omg.CORBA.*;     // Corba

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

```
    try{ORB orb = ORB.init(args, null);

        CompteImpl CompteRef = new CompteImpl("PourToi");
        orb.connect(CompteRef);

        org.omg.CORBA.Object
            nameS = orb.resolve_initial_references("NameService");
        NamingContext ncRef = NamingContextHelper.narrow(nameS);

        NameComponent nc = new NameComponent("PourToi", " ");
        NameComponent path[] = {nc};
        ncRef.rebind(path, CompteRef);

        java.lang.Object sync = new java.lang.Object();
        synchronized(sync){sync.wait();}
    } catch(Exception e) {
        System.err.println("ERROR: " +
e);e.printStackTrace(System.out);
      }
  }
}
```

La classe SetOfCompteServer

Commentons un peu le programme du serveur. Nous importons d'abord les différents packages nécessaires à l'utilisation de CORBA et du référentiel d'implantation.

```
import LikeCompte.*;        // Le package contenant nos stubs
import org.omg.CosNaming.*; // le serveur de noms
import org.omg.CosNaming.NamingContextPackage.*;
import org.omg.CORBA.*;     // Corba
```

Ensuite, nous initialisons l'ORB et créons une référence orb qui servira à enregistrer nos objets.

```
ORB orb = ORB.init(args, null);
```

Nous créons une instance de compte et l'enregistrons dans l'ORB.

```
CompteImpl CompteRef = new CompteImpl("PourToi");
orb.connect(CompteRef);
```

Les objets que nous publions sont connus des clients par un nom, et associés à une référence. Pour entretenir cette association entre les noms et les références, nous utilisons le service de nommage de CORBA. Les noms sont spécifiés de manière hiérarchique. La première étape consiste à chercher la racine du contexte des noms (*ncRef*). La méthode *narrow* est utilisée en général pour typer la référence CORBA avec le type utilisé dans Java (équivalent d'une conversion de type (*casting*) entre IDL et Java).

```
org.omg.CORBA.Object nameS =
```

```
      orb.resolve_initial_references("NameService");
NamingContext ncRef = NamingContextHelper.narrow(nameS);
```

La deuxième étape consiste à associer un nom à la référence de l'objet et à enregistrer cette association dans le référentiel d'implantation (*PourToi* est le nom du compte, le deuxième paramètre étant sa catégorie).

```
NameComponent nc = new NameComponent("PourToi", " ");
NameComponent path[] = {nc};
ncRef.rebind(path, CompteRef);
```

La fin du programme traite les erreurs et on attend que les clients invoquent les objets activés par le serveur.

Après avoir compilé les différentes classes, nous pouvons démarrer le serveur. Au préalable, il faut démarrer le référentiel d'implantation de CORBA avec la commande suivante :

```
start tnameserv -ORBInitialPort 1050
```

Le référentiel d'implantation publie le contexte initial de la racine de la hiérarchie de nommage (IOR pour *Interoperable Object Reference*). Le service est accessible sur le port 1050.

```
Initial Naming Context:
IOR:000000000000002849444c3a6f6d672e6f72672f436f734e616d696e672f4e616d696e67436f
6e746578743a312e3000000000010000000000000030000100000000000a6e745f66696e5f323200
045900000018afabcafe0000000276b1268a000000080000000000000000
TransientNameServer: setting port for initial object references to: 1050
```

Le serveur doit donc être activé avec le même port :

```
start java SetOfCompteServer -ORBInitialPort 1050
```

Le serveur n'est pas très bavard mais l'objet, au moment de son instanciation, déclare qu'il est actif :

```
    Objet Compte disponible : PourToi
```

30.5 Le client CORBA

Il nous reste à modifier le client de l'exemple RMI pour l'adapter à notre serveur. Nous avons mis en évidence les parties modifiées. On importe les packages nécessaires à l'utilisation de CORBA et du référentiel d'implantation. On crée une variable *Compte* qui permettra de récupérer la référence à notre objet. Pour cela, on procède comme dans le serveur à l'initialisation de l'ORB et on recherche la racine du référentiel d'implantation. Ensuite, il faut construire le chemin du nom pour retrouver notre objet. Et enfin, on demande au référentiel d'implantation de nous donner la référence du compte. Seule cette dernière instruction diffère du programme serveur :

```
accesDistant = CompteHelper.narrow(ncRef.resolve(path));
```

Le programme client est le suivant :

```
import java.awt.*;
import java.awt.event.*;
import java.applet.Applet;
import LikeCompte.*;
import org.omg.CosNaming.*; // serveur de noms : ref d'implantation
import org.omg.CosNaming.NamingContextPackage.*;
import org.omg.CORBA.*; // Corba

public class AppletClient extends Applet
   implements ActionListener {

 private Compte accesDistant; // la référence du compte dans l'ORB
 private Button Depot, Retrait;
 private Label Solde;
 private TextField saisieMontant;
 private float MontantSolde;

 public AppletClient(String args[]){
  connexionServeur(args);
  demandeSolde();
  creeInterfaceGraphique();
 }
 public void connexionServeur(String args[]){
   try {ORB orb = ORB.init(args, null);
        org.omg.CORBA.Object
           nameS = orb.resolve_initial_references("NameService");

        NamingContext ncRef = NamingContextHelper.narrow(nameS);

        NameComponent nc = new NameComponent("PourToi", " ");
        NameComponent path[ ] = {nc};
        accesDistant = CompteHelper.narrow(ncRef.resolve(path));
   }
   catch (Exception e) {System.out.println(e);}
}
public void demandeSolde(){
   // idem exemple RMI
}
public void creeInterfaceGraphique(){
   // idem exemple RMI
}
public void actionPerformed(ActionEvent e) {
   // idem exemple RMI
}
public static void main(String args[]) {
   try {Frame f = new Frame("Compte Client");
        AppletClient appletClient = new AppletClient(args);
        appletClient.init();
        appletClient.start();
        f.add("Center", appletClient);
```

```
        f.setSize(300, 300);
        f.show();}
    catch (Exception e) {
      System.out.println("Exception "+ e.getMessage());
    }
}}
```

Applet 57 : Gestion de compte en banque avec CORBA

On remarquera que les invocations de méthodes sont effectuées comme si l'objet était local au programme, la délocalisation du compte est donc entièrement transparente pour le développeur.

Le lancement du programme doit, comme on s'en doute, se faire avec le même port que pour le référentiel d'implantation.

```
java AppletClient -ORBInitialPort 1050
```

30.6 Persistance des données

Pour clore ce chapitre sur CORBA, nous allons créer une autre implantation de l'objet Compte. Dans tous les exemples vus jusqu'à présent, le montant du compte était volatile, c'est-à-dire perdu lorsqu'on arrête le serveur. En mémorisant le montant du compte dans une base de données, il est possible de rendre persistant l'état du compte.

En modifiant uniquement l'implantation de l'objet compte, il est possible de donner une nouvelle implantation au serveur. Le serveur sera donc à recompiler avec cette nouvelle implantation sans qu'on ait à en retoucher le code. La partie client n'est pas à recompiler, car elle n'est liée à l'objet compte qu'à travers la définition IDL, non modifiée dans notre cas. On voit donc ici le gain dû à l'indépendance du client et du serveur communiquant par des interfaces IDL.

```
import java.sql.*;
import java.net.URL;

public class CompteImpl extends LikeCompte._CompteImplBase {
  // on supprime la variable Float v !!!
  // le compte est entierement gere dans la BD
  String nomCompte;
  static String DML="DML";
  static String SEL="SEL";

  public CompteImpl(java.lang.String name) {
   nomCompte=name;
    System.out.println("Objet Compte disponible : "+name+
                       " = "+SQLService.getMontant(nomCompte));}

public float litSolde() {
   return SQLService.getMontant(nomCompte);}
```

```
public float retrait(float x) {
   doIt(DML, "update BANQUE set MONTANT=MONTANT-"+x+
                   " WHERE NOM_COMPTE='"+nomCompte+"'" );
   return SQLService.getMontant(nomCompte);}

public float depot(float x) {
   doIt(DML, "update BANQUE set MONTANT=MONTANT+"+x+
                   " WHERE NOM_COMPTE='"+nomCompte+"'" );
   return SQLService.getMontant(nomCompte);}

private static void doIt(String typereq, String s){
     if (typereq.equals("DML"))
System.out.println(SQLService.ExecDML(s));
     if (typereq.equals("SEL"))
System.out.println(SQLService.ExecSelect(s)); }

// initialisation de la classe
  static{
      try { // Charger le driver jdbc-odbc bridge
         Class.forName ("sun.jdbc.odbc.JdbcOdbcDriver");
         SQLService.con = DriverManager.getConnection
               ("jdbc:odbc:sgbdOracle", "javatest", "javapass");

         // ici, on est connecté, sinon on a généré une exception
         doIt(SEL, "Select * from BANQUE");

      }
     catch (SQLException e) {SQLService.PrintSQLError(e);}
     catch (Exception e   ) {e.printStackTrace ();}
  }
}
```

La classe CompteImpl avec persistance

Notre code réutilise la classe *SQLService* du chapitre concernant JDBC. Pour faciliter et rendre plus élégant le code de notre classe *Compte*, nous y avons ajouté une méthode *getMontant()* qui nous retourne le montant enregistré dans la base de données pour un compte spécifique.

```
class SQLService {
// . . .
// nous avons ajouté une méthode à cette classe par rapport
// à l'exemple du chapitre sur JDBC

public static float getMontant(String nc) {
  try {stmt = con.createStatement();
       results = stmt.executeQuery(
          "SELECT MONTANT FROM BANQUE "+
              "WHERE NOM_COMPTE='"+nc+"'" );
       if (results.next ())
          {return results.getFloat(1);
          }
      else System.out.print("** pas de donnée pour le compte: "+nc);
```

```
            return 0;
          }
      catch (Exception ex)
         {System.out.print("** Erreur dans la requete pour: "+nc);
           return 0;}
   }
   // . . .
}
```

La classe SQLService étendue

La connexion à la base de données se fait lors de la première instanciation d'un objet de cette classe; nous avons accroché le code dans la partie d'initialisation statique de la classe *Compte*. L'aspect le plus surprenant de cette implantation est la disparition complète de la variable chargée de mémoriser la valeur du compte. Le solde du compte est toujours manipulé dans la base de données.

Pour finir en beauté ce chapitre et ouvrir une polémique, nous poserons une question concernant l'extension possible de cette implantation. Pour implanter un mécanisme transactionnel dans notre applet de manipulation de compte bancaire, à quel niveau doit-on le faire :

- dans l'implantation de l'objet compte en utilisant le mécanisme de synchronisation de Java?
- en utilisant les services transactionnels de CORBA?
- en utilisant les mécanismes transactionnels de la base de données qui mémorise notre objet?

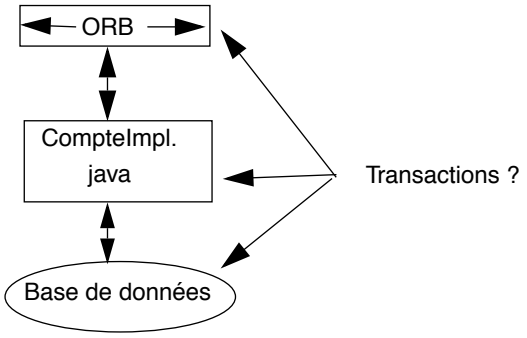


Figure 30.5 Où gérer le mécanisme de transaction ?

Chapitre 31

Les autres API de Java

Les API (*Application Programming Interfaces*) sont le mécanisme d'extension de services pour le langage Java. JDBC, la première API proposée par Sun, a étendu les services du langage à la connexion des bases de données.

Nous donnons la liste actuelle des API.

API	Fonction
Security API	Ensembles de classes pour la gestion de la cryptographie.
Servlet API	Ensembles de classes pour développer des serveurs sur le Web.
Enterprise JavaBeans Components	Plate-forme de création des applications réutilisables.
JavaBeans Components	Plate-forme neutre de composants réutilisables.
Java Telephony API (JTAPI)	Intégration des besoins en informatique et en téléphonie.
Java 2D API	Outils pour la gestion du graphisme en 2 dimensions.
Java 3D API	Outils pour la gestion du graphisme en 3 dimensions.
Advanced Imaging API	Outils pour le traitement d'images.
Java Sound API	Gestion du son en très haute qualité.
Java Speech API	Interfaces avec reconnaissance de la voix.

Tableau 31.1 Fonctions des API

API	Fonction
Java Naming and Directory Interface (JNDI)	Interfaces de gestion d'annuaires.
Java IDL	Interfaces pour le développement avec CORBA.
Java Remote Method Invocation (RMI) API	Interfaces pour la programmation distribuée.
Java Message Service API	Gestion d'un système de messages.
Java Transaction API (JTA)	Mécanisme pour la gestion transactionnelle.
Java Transaction Service (JTS)	Services associés à la gestion transactionnelle.
Java Management API (JMAPI)	Interfaces pour la gestion de la gestion...
JDBC	Gestion de la connexion avec les bases de données.
Java Mail AP	Interface pour la gestion d'une messagerie.
JavaHelp API	Gestion de l'aide en ligne.
Java Communications API	Gestion des communications (du bureau).

Tableau 31.1 Fonctions des API

La lecture de cette liste montre bien à quel point Java est en plein développement et les directions vers lesquelles il s'oriente : réseau, bases de données, multimédia, sécurité, etc.

Partie V

Annexes

Annexe A

Du C++ à Java

Quelle est la différence entre Java et C++? Java est orienté objet.

Ce chapitre présente une revue des différences existant entre C++ et Java. Ces différences sont nombreuses et il convient d'en illustrer toutes les subtilités. Nous débutons par les différences liées au développement de projets puis abordons les différences entre les langages eux-mêmes.

A.1 Développement d'une application en Java

Une application Java est organisée en modules. Un module est spécifié par le mot clé *package* et contient un ou plusieurs fichiers de définition de classes.

En Java, un fichier contient la définition complète d'une classe. Le corps des méthodes (fonctions membres en C++) d'une classe est défini juste après leur déclaration. Contrairement au C++, il n'y a donc pas de fichier d'en-tête (*header file*) décrivant l'interface d'une classe, ni de fichier d'implantation des méthodes de la classe.

En Java, un fichier contient une ou plusieurs définitions de classe dont une seule est publique. Le nom du fichier est nécessairement le nom de la classe publique, nom auquel s'ajoute le suffixe '*.java*'. Une classe ne peut être définie à l'intérieur d'une autre classe.

Commençons par un programme simple pour illustrer les différences dans l'organisation du développement d'application. Nous présentons une version C++ puis la version Java équivalente.

```
// C++
// fichier Temps.h
// note pour les programmeurs avertis : on parle des #ifdef plus tard
class Temps {
private :
   unsigned heure, minute, seconde;
public :
   Temps()
   {heure=0;
    minute=0;
    seconde=0;}
   Temps(unsigned h,unsigned m, unsigned s)
   {heure=h;
    minute=m;
    seconde=s;}
   int tempsValide();
   void affiche();
};
```

Exemple C++ de fichier d'en-tête

```
//C++
// fichier Temps.cpp
#include "Temps.h"
#include <iostream.h>

int Temps::tempsValide()
{ // retourne 1 ssi le temps est une heure correcte, 0 sinon
  if ((heure>=24) || (minute>=60) || (seconde>=60)) return 0;
  return 1;
};

void Temps::affiche()
{cout <<heure <<":" <<minute <<":" <<seconde <<'\n';
}
```

Exemple C++ de fichier d'implantation des méthodes

Le listing suivant est la définition correspondante en Java de la classe *Temps*. On notera que les deux versions de la classe *Temps* n'utilisent pas les mêmes types de données : en effet, le type *unsigned* (défini en C++) n'existe pas en Java, le type *boolean* quant à lui existe et remplace avantageusement le type *int* retourné par la méthode *tempsValide*.

```
// Java
// fichier Temps.java
import java.io.*;
```

```
class Temps {
   private int heure, minute, seconde;
   public Temps()
   {heure=0;
    minute=0;
    seconde=0;}
   public Temps(int h,int m, int s)
   {heure=h;
    minute=m;
    seconde=s;}
   public boolean tempsValide()
   { // retourne true ssi le temps est une heure correcte,
     // false sinon
     if ((heure>=24) || (minute>=60)  || (seconde>=60))
       return false;
     return true;}
   public void affiche()
   {System.out.println(heure +":" +minute +":"+seconde);}
};
```

Définition équivalente en Java de la classe Temps

A.2 Le programme principal : la fonction *main*

En Java, tout est objet (hormis les types simples), une application est une classe : le *main* est une méthode de classe.

```
//C++
// programme utilisant la classe Temps
#include "Temps.h"

void main()
{ Temps t;
  t.affiche();
}
```

Exemple C++ d'utilisation de la classe Temps

```
import Temps;
class Maintemps
{ static public void main(String argv[])
   { Temps t=new Temps();
     t.affiche();}
}
```

Exemple Java d'utilisation de la classe Temps

A.3 Elaboration du code exécutable

Pour les langages compilés, l'élaboration d'un programme exécutable comporte généralement deux phases : la compilation puis l'édition de liens. La compilation en Java produit un programme objet[1] nommé avec le suffixe '*.class*' (voir

point 3.2, p. 32). Il n'y a pas d'édition statique de liens dans Java : l'interpréteur charge, si nécessaire, les différentes classes dont l'application a besoin au cours de son exécution. L'édition de liens est donc dynamique.

Le C++ requiert une phase d'édition de liens statique. Celle-ci a pour but d'assembler les différents programmes objet issus de la compilation pour fabriquer un programme exécutable.

En C++, les fichiers d'en-tête jouent un rôle important : par la directive du préprocesseur *#include*, ils spécifient les modalités de réutilisation d'une classe. Ces modalités sont des références externes qui ne doivent être déclarées qu'une seule fois pour une application (constituée ainsi d'un ensemble de fichiers). Ces déclarations multiples ne sont généralement identifiées qu'à l'édition de liens. Une bonne pratique permet d'éliminer ce problème en utilisant des primitives du préprocesseur. Cette pratique (illustrée dans l'exemple suivant) est une nécessité lorsqu'une application est développée par une équipe de programmeurs, sans que l'on sache si Pierre ou Paul réutilisera telle ou telle classe.

```
#ifndef TEMPS_H
#define TEMPS_H
// en-tete de la classe Temps
class Temps {
private :
   unsigned heure, minute, seconde;
public :
   Temps()
   {heure=0;
    minute=0;
    seconde=0;}
   Temps(unsigned h,unsigned m, unsigned s)
   {heure=h;
    minute=m;
    seconde=s;}
   int tempsValide();
   void affiche();
};
#endif
```

Exemple d'en-tête C++ relevant d'une bonne pratique

L'usage des fichiers d'en-tête et de la séquence *#ifndef...#endif* illustrée sur l'exemple précédent n'est plus nécessaire en Java (il n'y a d'ailleurs pas de préprocesseur). Cependant, ces fichiers étaient fréquemment utilisés par les développeurs C++ pour commenter la classe, préciser les modalités d'utilisation et/ou le rôle des membres d'une classe afin d'en faciliter la réutilisation.

---

1. Le mot objet ici n'est pas pris au sens de « orienté objet ». Un programme objet est le résultat de la compilation d'un programme source.

En Java, il n'est pas toujours facile de se faire une idée précise de la façon de réutiliser telle ou telle classe. En effet, un fichier unique contient la définition complète de la classe, y compris le corps de toutes les méthodes. On ne dispose pas d'une vue synthétique permettant d'appréhender rapidement l'intérêt d'une classe pour sa réutilisation. Les programmeurs Java doivent donc faire un effort particulier de documentation de leurs programmes, éventuellement dans un fichier séparé ou, mieux encore, utiliser les commentaires spéciaux pour l'utilitaire *JavaDoc*. JavaDoc génère ensuite la documentation de vos sources sous forme de documents HTML. Cette pratique est nécessaire pour favoriser la réutilisation, que l'on distribue ou non le code source sur Internet ou à l'intérieur de son équipe de développement.

A.4 Mise à jour d'une application Java

En C++, la modification d'une classe entraîne généralement la recompilation des classes ou de l'application la réutilisant. Votre environnement de développement peut vous proposer de reconstruire une application car l'un des fichiers qui la composent a été modifié. Si vous choisissez de la reconstruire, votre environnement de développement ne recompilera que les parties nécessaires, puis fera une édition de liens.

En Java, ce genre de gestion n'existe pas, et des erreurs peuvent se produire à l'exécution en raison de l'édition dynamique de liens (voir p. 371). Le scénario suivant en est une illustration :

1. On dispose de deux versions exécutables (une C++ et une Java) de l'exemple plus haut.
2. On supprime le constructeur *Temps()*.

On obtient les fichiers suivants :

```
// C++
// fichier Temps.h

class Temps {
private :
   unsigned heure, minute, seconde;
public :
   Temps(unsigned h,unsigned m, unsigned s)
   {heure=h;
    minute=m;
    seconde=s;}
   int tempsValide();
   void affiche();
};
```

En-tête C++ de la classe Temps sans constructeur Temps()

Le fichier *Temps.cpp* d'implantation des méthodes reste identique.

```
// Java
// fichier Temps.java
import java.io.*;

class Temps {
   private int heure, minute, seconde;
   public Temps(int h,int m, int s)
   {heure=h;
    minute=m;
    seconde=s;}
   public boolean tempsValide()
   { // retourne true si le temps est une heure correcte,
     // false sinon
     if ((heure>=24) || (minute>=60)  || (seconde>=60))
       return false;
     return true;}
   public void affiche()
   {System.out.println(heure +":" +minute +":"+seconde);}
};
```

Classe Temps en Java, sans le constructeur Temps()

3. On recompile la classe *Temps* dans les deux environnements.
4. On teste les deux applications :
   a. la version C++ n'a pas été reconstruite et s'exécute comme avant;
   b. la version Java produit l'erreur suivante à l'exécution :

```
java.lang.NoSuchMethodError: Temps: method <init>()V not found
```

Ce scénario illustre le chargement dynamique par Java des méthodes dont une application a besoin pour son exécution. Cependant, dans les deux cas, l'oubli par le développeur de recompiler l'application entraîne des conséquences dommageables : en C++ l'utilisateur a toujours l'ancienne version, en Java l'utilisateur a une application qui peut produire des bogues.

Tous les problèmes d'identification des classes clientes à recompiler ne sont pas entièrement résolus, mais un pas en avant a été fait relativement au C++ (même si on utilise le « make »).

Nous avons vu dans les paragraphes précédents que la gestion du développement d'une application est différente pour ces deux langages.

Nous allons maintenant aborder les différences existant entre les deux langages proprement dits.

A.5 Les types prédéfinis

En Java, et contrairement au C++, la taille des types est indépendante du système d'exploitation de l'ordinateur.

Les types numériques

Le type *byte* de Java remplace le type *char* du C++.

Java ne permet pas de spécifier des valeurs entières non signées à l'aide du qualificatif *unsigned*.

Type	Taille en bits
`byte`	8
`short`	16
`int`	32
`long`	64
`float`	32
`double`	64

Tableau A.1 Les types numériques en Java

Le type caractère

Le type *char* de Java définit un caractère de 16 bits au format Unicode. Les caractères Unicode sont des valeurs entières de 16 bits non signées (c'est-à-dire entre 0 et 65535).

Le type booléen

Contrairement au C++, Java possède un type booléen nommé *boolean*. Ses valeurs sont *true* et *false* pour vrai et faux respectivement. Le type *boolean* est un type distinct des types numériques : on ne peut pas convertir un type *boolean* en un type numérique quel qu'il soit.

La classe String

Java comporte une classe prédéfinie *String*. Un objet de la classe *String* n'est pas équivalent à un tableau de *char*.

A.6 Définition de nouvelles structures

En C++, on peut construire des structures *(struct)*, des unions (*union*), des énumérations (*enum*), des types (*typedef*) et enfin des classes (*class*).

À l'exception des classes, toutes ces structures sont des rémanences du langage C dont est dérivé le C++. Pour programmer objet proprement, la seule structure à utiliser est celle de classe. Java favorise ainsi une programmation tout objet puisque c'est la seule structure permise.

A.7 Déclaration et initialisation des variables

En Java, une déclaration de variable ne réserve aucun espace mémoire :

- pour une variable d'un type simple, une variable non initialisée ne contient aucune valeur;
- pour une variable définissant une instance d'une classe, une variable non initialisée ne fait référence à aucun objet.

L'utilisation d'une variable non initialisée est détectée à la compilation et provoque une erreur, contrairement au C++ qui initialise par défaut les variables. Java impose donc une plus grande rigueur et permet d'éviter des oublis qui peuvent être dommageables. L'exemple suivant en est une illustration.

Java	C++

`import java.io.*;`
`class demo {`
`  public static void main`
`  (String argv[])`
`  {int c;`
`  System.out.println(c);}`
`}`
	`#include <iostream.h>`
	`main()`
	`  {int c;`
	`  cout << c <<'\n';`
	`  }`

Tableau A.2 Test de l'initialisation de variables en Java et en C++

La version C++ s'exécute normalement et affiche 0, la valeur initiale affectée par le compilateur à la variable *c*. La version Java est refusée à la compilation et ne peut donc être exécutée.

A.8 Déclaration des constantes

En Java, et contrairement au C++, il n'est pas possible de déclarer des instances constantes de classe d'objet. L'instruction suivante n'a pas d'équivalent en Java :

```
//C++
const Temps midi(12,0,0);
```

A.9 Les tableaux

Les différences entre les deux langages portent sur la déclaration et l'initialisation des tableaux. De plus, à l'exécution, Java vérifie les débordements des indices.

Déclaration

Le tableau A.3 récapitule les différences sur les déclarations de tableau. En Java, les instructions légales de cette table ne sont que des déclarations de type, elles ne créent pas le tableau.

En C++, la déclaration d'un tableau en initialise les éléments : la classe doit avoir

Exemple de déclaration d'un tableau	En Java	En C++
`int[] monTableau;`	légal	illégal
`int monTableau[];`	légal	illégal
`Temps [] rendezVous[]; // 2 dimensions`	légal	illégal
`int monTableau[5];`	illégal	légal
`int[5] monTableau;`	illégal	illégal

Tableau A.3 Déclaration de tableaux

un constructeur par défaut, ce constructeur étant appelé automatiquement lors de la déclaration du tableau (si au moins un des éléments n'est pas initialisé explicitement). Dans l'exemple suivant, le constructeur *Temps()* doit être présent.

```
//C++
#include "Temps.h"
void main()
{ Temps intervalle[2];
  for (int i=0;i<2;i++) intervalle[i].affiche();
}
```

Initialisation des tableaux unidimensionnels

L'exemple suivant initialise les valeurs des tableaux aux valeurs *null* pour les objets. Aucun objet n'est créé.

```
//Java
int[] monTableau=new int[5]; //initialise à 0 les 5 éléments du
tableau
Temps intervalle[]=new Temps[2];
//initialise à null les 2 éléments, aucun constructeur n'est appelé
```

En Java, l'allocation effective des objets – éléments du tableau – requiert l'appel explicite d'un constructeur.

```
//Java
import Temps;
class demo {
   public static void main(String argv[])
   {Temps intervalle[]=new Temps[2];
    for (int i=0;i<2;i++)
      {intervalle[i]=new Temps();
       intervalle[i].affiche();}
   }
}
```

Le tableau A.4 illustre la déclaration et l'initialisation avec appel explicite du constructeur désiré.

	Déclaration et initialisation de tableaux d'objets
Java	`Temps[] intervalle= { new Temps(),new Temps(12,15,30)};`
C++	`Temps intervalle[2]= {Temps(),Temps(12,15,30)};`

Tableau A.4 Initialisation de tableaux d'objets

Initialisation des tableaux multidimensionnels

Contrairement au C++, en Java il n'y a pas de tableaux multidimensionnels mais des tableaux de tableaux. L'initialisation de leurs dimensions se fait ainsi :

```
// Java
int t[][]=new int[3][4];
int t[][]={{0,3,-2,5,-1,4},{0,3,-2,3,-1},{0,3,-2,4},{0,3,3}};
```

Il n'existe pas de contrainte sur les tailles : ainsi le tableau précédent est triangulaire. L'attribut *length* permet d'obtenir la longueur d'un tableau. L'exemple ci-dessous montre l'intérêt d'un tel attribut puisqu'il décharge le programmeur de la gestion des longueurs de tableaux. De plus, Java contrôle les valeurs des indices de tableaux, ce qui facilite grandement la mise au point des programmes.

```
//Java
import java.io.*;
class demo {
   public static void main(String argv[])
   { int t[][]={{0,3,-2,5,-1,4},{0,3,-2,3,-1},{0,3,-2,4},{0,3,3}};
      for (int i=0;i<t.length;i++)
       {for (int j=0;j<t[i].length;j++)
          System.out.print(" "+t[i][j]);
        System.out.println();}
   }
}
```

Exemple d'un tableau de tableaux de longueur différente

Son exécution donne le résultat suivant :

```
0 3 -2 5 -1 4
0 3 -2 3 -1
0 3 -2 4
0 3 3
```

A.10 Destruction des objets

En Java, il n'y a pas de destruction explicite d'objets. Lorsqu'un objet n'est plus référencé, l'espace mémoire qu'il occupe pourra être récupéré par le ramasse-miettes (*garbage collector*). Le programmeur est donc libéré de ce travail d'identification des objets à détruire.

En fin de bloc, s'il existe des objets uniquement référencés par des variables locales à ce bloc, l'espace qu'ils occupent sera récupéré ultérieurement par le ramasse-miettes.

Le programmeur peut définir une méthode appelée *finalize()*. Cette méthode, appelée par le ramasse-miettes avant la destruction de l'instance concernée, n'est utile que pour la libération de ressources autres que la mémoire – par exemple un fichier non fermé. Cette méthode joue le rôle du destructeur C++.

A.11 L'instruction *for*

Le tableau A.5 ci-dessous montre les différences de l'instruction *for* entre les deux langages. Ces différences sont dues à ce que Java, contrairement au C++, considère le *for* comme un bloc (voir Pour_faire_ (for), p. 70).

Programme Java refusé à la compilation	Programme C++ valide

`//Java`
`import java.io.*;`
`class demo {`
`  public static void main`
`  (String argv[])`
`{ int t[]=new int[5];`
`  for (int i=0;i<5;i++) t[i]=i;`
`  // i est inconnu`
`  System.out.println(i);`
`}`
`}`
	`//C++`
	`#include <iostream.h>`
	`void main()`
	`{ int t[5];`
	`  for (int i=0;i<5;i++) t[i]=i;`
	`  // i reste defini`
	`  cout << i <<'\n'; }`

Tableau A.5 Portée des variables déclarées dans le for

Le tableau A.6 montre la transformation des boucles *for* pour passer d'un langage à l'autre.

Java	C++

`...`
`for (int i=0;i<5;i++)`
`t[i]=i;`
`...`
	`...`
	`{for (int i=0;i<5;i++)`
	`t[i]=i;}`
	`...`

`...`
`int i; ...`
`for (i=0;i<5;i++)`
`t[i]=i;`
`...`
	`...`
	`for (int i=0;i<5;i++)`
	`t[i]=i;`
	`...`

Tableau A.6 Instructions for en Java et en C++

A.12 L'instruction *continue*

L'instruction *continue* du C est présente en C++. Dans ces langages, elle fait débuter l'itération suivante de la répétitive *(for, while, do)* dans laquelle elle se situe. Ainsi l'exemple ci-dessous permet de n'afficher que les nombres positifs ou nuls du tableau *t*.

Java	C++

`import java.io.*;`
`class demo {`
`public static void`
`main(String argv[])`
`{int t[]={0,3,-2,5,-1};`
`for (int i=0;i<t.length;i++)`
`{if (t[i]<0) continue;`
`System.out.println(t[i]);}`
`}`
`}`
	`#include <iostream.h>`
	`void main()`
	`{ int t[5]={0,3,-2,5,-1};`
	`for (int i=0;i<5;i++)`
	`{if (t[i]<0) continue;`
	`cout << t[i] << endl;}`
	`}`

Tableau A.7 Instruction continue en Java et en C++

En Java, l'instruction *continue* (voir point 5.6, p. 73) peut comporter une étiquette de branchement facultative. Si cette étiquette est absente, l'instruction a la même signification qu'en C++.

L'étiquette de branchement n'équivaut pas à un *goto* qui autorise à aller n'importe où, mais permet dans le cas de répétitives imbriquées de spécifier le niveau d'imbrication à partir duquel on veut continuer les itérations. L'étiquette est nécessairement définie dans un bloc englobant celui dans laquelle l'instruction *continue* se trouve.

Ainsi, le programme suivant recherche toutes les occurrences d'une suite de nombres dans un tableau numérique à deux dimensions. Le programme affiche :

- le tableau dans lequel se fait la recherche;
- la suite à rechercher;
- les coordonnées des occurrences (s'il y en a) de cette suite dans le tableau.

```
//Java
import java.io.*;
class demo {
   public static void main(String argv[])
   {int t[][]={{0,3,-2,5,-1,4},{0,3,-2,3,-1,2},{0,3,-2,4,5,-
1},{0,3,3,-2,3,-1}};
      System.out.println("tableau a parcourir :");
      for (int i=0;i<t.length;i++)
       {for (int j=0;j<t[i].length;j++)
```

```
          System.out.print(" "+t[i][j]);
         System.out.println();
        };

   int search[]={3,-2,3};
      System.out.println("suite a chercher :");
      for (int i=0;i<search.length;i++)
       System.out.print("   "+search[i]);
      System.out.println();
      for (int i=0;i<t.length;i++)
       {debutRecherche:
         for (int k=0;k<t[i].length-search.length;k++)
          {for (int j=0;j<search.length;j++)
             if (t[i][k+j]!=search[j]) continue debutRecherche;
           System.out.println("trouve en "+i+","+k);
          }
       }
   }
}
```

Exemple de programme avec l'instruction continue

Le résultat de l'exécution est le suivant :

```
tableau a parcourir :
   0   3   -2   5   -1   4
   0   3   -2   3   -1   2
   0   3   -2   4   5   -1
   0   3   3   -2   3   -1
suite a chercher :
 3 -2 3
trouve en 1,1
trouve en 3,2
```

Dans cet exemple, l'instruction *continue* permet, en cas d'échec de reconnaissance de la suite dans le tableau (même si on en a trouvé une partie), de continuer la recherche à l'indice *k* suivant. Un programme C++ équivalent contenant des étiquettes de branchement utilise donc le *goto*.

```
//C++
#include <iostream.h>
void main()
{ int t[4][6]={{0,3,-2,5,-1,4},
               {0,3,-2,3,-1,2},
               {0,3,-2,4,5,-1},
               {0,3,3,-2,3,-1}};
     cout << "tableau a parcourir :"<< endl;
     for (int i=0;i<4;i++)
      {for (int j=0;j<6;j++)  cout <<" " << t[i][j] ;
       cout << endl;}

  int search[]={3,-2,3};
     cout << "suite a chercher :" << endl;
     for (i=0;i<3;i++)
      cout <<" " << search[i] ;
```

```
      cout << endl;
      for (i=0;i<4;i++)
        {for (int k=0;k<6-3;k++)
           {for (int j=0;j<3;j++)
              if (t[i][k+j]!=search[j]) goto debutRecherche;
            cout <<"trouve en " << i <<"," << k << endl;
            debutRecherche: ;
           }
        }
}
```

Programme C++ avec goto

A.13 L'instruction *break*

L'instruction *break* du C est encore présente en C++. Dans ces langages, elle fait sortir du bloc *(for, while, switch, do)* dans lequel elle se situe. En Java, l'instruction *break* (voir point 5.5, p. 72) peut comporter une étiquette de branchement qui lui permet de spécifier où doit continuer l'exécution du programme. L'étiquette est nécessairement définie dans un bloc englobant celui dans lequel l'instruction *break* se trouve.

L'exemple suivant est le même que celui illustrant l'instruction *continue*, si ce n'est que l'on s'arrête à la première occurrence trouvée.

```
//Java
import java.io.*;
class demo {
   public static void main(String argv[])
   {int t[][]={{0,3,-2,5,-1,4},{0,3,-2,3,-1,2},
               {0,3,-2,4,5,-1},{0,3,3,-2,3,-1}};
     System.out.println("tableau à parcourir");
     for (int i=0;i<t.length;i++)
      {for (int j=0;j<t[i].length;j++)
           System.out.print(" "+t[i][j]);
       System.out.println();};
    int search[]={3,-2,3};
    System.out.println("suite à chercher");
    for (int i=0;i<search.length;i++)
       System.out.print(" "+search[i]);
    System.out.println();
    sortie:
      for (int i=0;i<t.length;i++)
       {debutRecherche:
         for (int k=0;k<t[i].length-search.length;k++)
          {for (int j=0;j<search.length;j++)
             if (t[i][k+j]!=search[j]) continue debutRecherche;
           System.out.println("trouve en "+i+","+k);
           break sortie;}
       }
    System.out.println("C'est fini!");}
}
```

Exemple de programme avec un break étiqueté

Un programme C++ équivalent contenant des étiquettes de branchement utilise donc le *goto*.

```
//C++
#include <iostream.h>
void main()
{
  int t[4][6]={{0,3,-2,5,-1,4},
               {0,3,-2,3,-1,2},
               {0,3,-2,4,5,-1},
               {0,3,3,-2,3,-1}};
     cout << "tableau à parcourir :"<< endl;
     for (int i=0;i<4;i++)
      {for (int j=0;j<6;j++)  cout <<" " << t[i][j] ;
       cout << endl;}

  int search[]={3,-2,3};
     cout << "suite à chercher :" << endl;
     for (i=0;i<3;i++)
      cout <<" " << search[i] ;
     cout << endl;
     for (i=0;i<4;i++)
       {for (int k=0;k<6-3;k++)
          {for (int j=0;j<3;j++)
             if (t[i][k+j]!=search[j]) goto debutRecherche;
           cout <<"trouve en " << i <<"," << k << endl;
           goto sortie;
           debutRecherche: ;
          }
       }
    sortie:
    cout <<"C'est fini!" << endl;
}
```

Programme équivalent en C++

En définitive, contrairement au C++, Java permet d'éviter une programmation avec des branchements non orthodoxes. Dans les deux langages, les branchements intempestifs peuvent toujours être évités par des variables booléennes qui permettent de sortir de répétitives imbriquées.

A.14 L'instruction *goto*

Le *goto* est un mot clé réservé du langage Java, mais ne correspond actuellement à aucun traitement dans ce langage. Chacun sait que l'usage du *goto* est délicat, le *goto* est cependant très pratique pour sortir de répétitives imbriquées sans avoir à ajouter un booléen à chaque test des répétitives imbriquées. En Java, ce bon usage du *goto* peut se réaliser à l'aide de l'instruction *break* suivie d'une étiquette.

A.15 L'affectation

En Java, toutes les variables désignant les objets sont des références (à l'exception des objets de types numérique, caractère et booléen). C++ utilise les références aux objets notamment pour le passage de paramètres dans les méthodes, fonctions ou procédures. Une référence en C++ est une sorte de pointeur pour lequel l'utilisateur n'a pas à spécifier les indirections permettant l'accès à la variable pointée. Les références C++ ne sont cependant pas des pointeurs : en effet, lorsque l'on affecte deux pointeurs, ils désignent le même objet, ce qui n'est pas le cas des références C++. De plus, contrairement aux pointeurs, aucune opération (++, –) n'est possible sur les références.

Affectation de référence en Java

L'affectation d'une référence à une autre en Java aboutit à ce qu'elles désignent le même objet (voir point 6.9, p. 94). Le même exemple sera repris en C++, mais le résultat de l'affectation sera différent.

L'exemple considère une nouvelle définition de la classe *Temps* à laquelle on a ajouté une méthode *plusUneHeure()* augmentant l'heure d'une unité. Cette nouvelle méthode nous permettra de distinguer les effets de l'affectation.

```
// Java
public void plusUneHeure()
      {heure++;}
```

Le programme utilisant cette nouvelle définition est le suivant :

```
// Java
import Temps;
class mainTemps
{static public void
   main(String argv[])
   { Temps t=new Temps(12,0,0);
     Temps t2=new Temps(14,30,0);
     System.out.println("Valeurs initiales :");
     t.affiche();  // t vaut 12:0:0
     t2.affiche(); // t2 vaut 14:30:0

     t=t2;  // affectation des références
     System.out.println("Apres affectation");
     t.affiche(); // t vaut 14:30:0
     t2.affiche();// t2 vaut 14:30:0

     t.plusUneHeure();
     System.out.println("Apres plusUneHeure");
     t.affiche(); // t vaut 15:30:0
     t2.affiche();  // t2 vaut 15:30:0
     }
}
```

L'application Java utilisant la classe Temps

Après l'affectation, les valeurs des attributs sont bien évidemment identiques. L'appel de la méthode *plusUneHeure()* sur la référence *t* augmente l'heure d'une unité pour l'objet référencé par *t*. L'appel de la méthode *affiche()* sur les deux références nous montre qu'elles désignent le même objet.

Affectation de référence en C++

La version C++ du même programme renvoie des résultats différents :

```
// C++ a rajouter dans la partie publique de la classe Temps
void plusUneHeure() {heure++;}
```

Le programme C++ utilisant cette nouvelle définition est le suivant :

```
//C++
// programme utilisant la classe Temps
#include "Temps.h"
#include <iostream.h>

void main()
 {
 Temps &t=Temps(12,0,0);
 Temps &t2=Temps(14,30,0);
 cout << "Valeurs initiales :"<<'\n';
 t.affiche(); // t vaut 12:0:0
 t2.affiche();// t2 vaut 14:30:0
 t=t2;
 cout << "Apres affectation"<<'\n';
 t.affiche(); // t vaut 14:30:0
 t2.affiche();// t2 vaut 14:30:0

 t.plusUneHeure();
 cout << "Apres plusUneHeure" <<'\n';
 t.affiche(); // t vaut 15:30:0
 t2.affiche();// t2 vaut 14:30:0
}
```

L'application C++ utilisant la classe Temps

Le tableau A.8 montre les traces d'exécution des deux programmes.

Java	C++
Valeurs initiales : 12:0:0 14:30:0 Apres affectation 14:30:0 14:30:0 Apres plusUneHeure 15:30:0 15:30:0	Valeurs initiales : 12:0:0 14:30:0 Apres affectation 14:30:0 14:30:0 Apres plusUneHeure 15:30:0 14:30:0

Tableau A.8 Traces d'exécution : gestion des références

L'affectation de référence en C++ est équivalente à l'affectation de deux objets désignés par des variables simples : elle substitue le contenu du terme gauche par celui du terme droit. Les références ne désignent pas le même objet. Lorsque l'on augmente l'heure d'une unité pour l'un, l'objet désigné par l'autre référence reste inchangé.

Récapitulatif

En C++, il y a trois possibilités de désignation des objets :

- un désignateur simple (une variable);
- un pointeur;
- une référence.

Les références n'existaient pas en C, elles ont été introduites en C++ pour le passage des paramètres. En Java, il n'existe qu'un seul mode de désignation des objets : la référence. Cette uniformité ne constitue pas un appauvrissement du langage (par rapport au C++) mais simplifie plutôt la conception des programmes.

Les références de Java ne sont équivalentes ni aux pointeurs ni aux références du C++.

A.16 La définition de classes

Malgré une syntaxe voisine entre les deux langages, de nombreuses différences méritent notre attention.

Définition des attributs

Les attributs d'une classe sont les variables d'instance et les variables de classes. Les seules différences entre Java et C++ concernent leur initialisation. En C++, seules les constantes peuvent être initialisées. En Java (voir point 7.7, p. 102), les variables d'instance et de classes sont initialisables. Les exemples ci-dessous en illustrent les différents aspects :

```
//Java
import Temps;
class UneClasse
   {// variable de classe
    static int n=3;
    // variables d'instances
    int x=n+5,y;
    java.lang.String nom="toto";
    float valeurs[]=new float [10];
    Temps debut=new Temps(12,0,0);
   }
```

Les variables de classe sont initialisées lors du chargement de la classe, dans l'ordre d'apparition dans la définition de la classe : une référence en avant provo-

que une erreur de compilation :

```
class ClasseErronee {
static int i=j+1; // j: reference en avant, erreur de compilation
static int j=3;
}
```

En Java, lors de la création d'une instance, les variables d'instance sont initialisées avant l'appel au constructeur. Ainsi, lors de la création d'une instance de la classe ci-dessous, la variable d'instance *demo* aura la valeur 6.

```
// Java
class UneClasse
   { int demo=3;
    UneClasse() {demo=6;}
   }
```

Classe avec variable d'instance initialisée

```
//Java
import UneClasse;
import java.io.*;
class Demo {
   public static void main(String argv[])
   {UneClasse c=new UneClasse();
    System.out.println(c.demo);
   }
}
```

Programme illustrant la valeur de la variable d'instance demo

En C++, la norme ANSI interdit l'initialisation d'une variable d'instance (non constante) lors de sa définition. Il faut alors initialiser celle-ci à l'aide d'un constructeur.

Définition des méthodes

En Java, les méthodes (voir point 7.8, p. 102) se définissent comme en C++ aux restrictions suivantes près :

- on ne peut pas spécifier qu'un paramètre est constant;
- Java manipule des références sur des objets et non des pointeurs. Les références ne sont pas modifiables par la méthode, les objets eux-mêmes sont modifiables : on peut invoquer des méthodes sur ces objets et modifier leurs attributs accessibles. Le contenu de l'objet peut changer. Avant et après l'exécution d'une méthode, une référence passée en paramètre désigne toujours le même objet;
- un paramètre d'une méthode ne peut avoir de valeur par défaut;
- le corps d'une méthode se trouve obligatoirement dans la définition de sa classe.

Homonymie des méthodes et des attributs

En Java, une méthode et un attribut définis dans une classe peuvent avoir le même nom. En effet, Java ne permettant pas de manipuler des pointeurs sur des méthodes, il n'y a donc pas de confusion possible.

Le C++ ne peut distinguer si l'on désigne l'adresse de la fonction membre ou la donnée membre : ce langage ne permet donc pas de spécifier des attributs et des méthodes de même nom.

Qualification des attributs des méthodes

Spécification des domaines de protection

Comme en C++, les attributs et les méthodes peuvent être qualifiés par les modificateurs *public, private* et *protected* pour spécifier les domaines de protection de la classe (voir tableau A.9).

Java	C++

`class UneClasse`
`{ UneClasse()`
`  {...}`
`  public int A1, A2;`
`  public char A3;`
`  public void m1()`
`   {//corps de la methode m1`
`   }`
`  protected float A4;`
`  protected double A5;`
`  protected void m2()`
`   {//corps de la methode m2`
`   }`
`  private float A6,A7;`
`  private void m3()`
`   {//corps de la methode m3`
`   }`
`  private void m4()`
`   {//corps de la methode m4`
`   }`
`};`
	`class UneClasse`
	`{ UneClasse()`
	`  {...}`
	`public:`
	`  int A1, A2;`
	`  char A3;`
	`  void m1();`
	`protected:`
	`  float A4;`
	`  double A5;`
	`  void m2();`
	`private:`
	`  float A6,A7;`
	`  void m3();`
	`  void m4();`
	`};`

Tableau A.9 Spécification de la visibilité dans Java et C++

Cependant en C++, une spécification d'un domaine de protection reste valable jusqu'à spécification d'un autre domaine de protection. En Java, cette spécification doit être mentionnée pour chaque attribut et méthode.

public et *private* ont la même signification dans les deux langages, *protected* présente une différence : en C++, la méthode ou la variable d'instance est accessible

(à moins qu'il y ait un masquage) par n'importe quelle sous-classe. En Java, l'accès est étendu aux classes du même *package*.

En C++, l'absence de spécification d'un domaine définit implicitement le domaine *private*. En Java, cette absence ne correspond à aucun des trois domaines. En effet, la variable ou la méthode est accessible par toutes les classes du même *package*, mais non par les sous-classes déclarées dans d'autres *packages*.

Spécification des variables et des méthodes de classe

En Java comme en C++, les variables de classe et les méthodes de classe sont spécifiées à l'aide du préfixe *static* devant leur déclaration.

```
//Java
class UneClasse
{ static int nombresDInstances=0;  // variable de classe
  static void uneInstanceDePlus() // methode de classe
   {nombresDInstances++;
   };
  UneClasse()   // constructeur
   {uneInstanceDePlus();
   }
  private String info;  //variable d'instance
  void Information()//methode d'instance
  {System.out.println("l'instance " + info
      + " est une des "+ nombresDInstances
      + " instances de sa classe" );
  } };
```

Exemple de définition de variable et de méthode de classe

En C++ et en Java, les méthodes de classe n'ont accès qu'aux variables et méthodes de classe. Les méthodes d'instance ont accès aux variables et méthodes de classe et d'instance. En Java, on le rappelle : l'initialisation des variables d'instance et de classe peut se faire au moment de leur déclaration, dans la définition de la classe.

En C++, une variable de classe est accessible depuis la classe ou l'une des instances de la classe. L'accès depuis la classe requiert l'opérateur de résolution de portée *(::)*. En Java, l'accès s'effectue à partir de l'instance ou de la classe de manière uniforme à l'aide de l'opérateur de sélection (.) : la classe est un objet avec des données membres et des méthodes. Le tableau A.10 illustre l'accès aux variables et méthodes de classe dans les deux langages.

En Java, une méthode de classe ne peut être redéfinie dans une sous-classe. Une méthode *static* est donc implicitement *final*.

	Java	C++

Définition de classe avec variable et méthode de classe	`class UneClasse`
	`{ public int x;`
	`  public static int n;`
	`  public static int`
	`  valeur()`
	`  {return n;};`
	`  UneClasse(){x=3;}`
	`};`
		`class UneClasse`
		`{ public:`
		`  int x;`
		`  static int n;`
		`  static int valeur()`
		`  {return n;};`
		`  UneClasse(){x=3;}`
		`};`

Modes d'accès	`//accès à partir de la`
	`// classe`
	`int i=UneClasse.n;`
	`int j=UneClasse.valeur();`
	
	`//accès à partir d'une`
	`//instance`
	`UneClasse c=new UneClasse();`
	`i+= c.n;`
	`j+= c.valeur();`
		`//accès à partir de la`
		`// classe`
		`int i=UneClasse::n;`
		`int`
		`j=UneClasse::valeur();`
		
		`//accès à partir d'une`
		`//instance`
		`UneClasse c;`
		`i+= c.n;`
		`j+= c.valeur();`

Tableau A.10 Accès aux variables et méthodes de classe

Autres qualifications des variables

Une variable d'instance ou de classe peut être constante. Un préfixe (*const* en C++, *final* en Java) donne cette spécification. Java permet de qualifier de *volatile* et *transient* des variables.

	Java	C++

Définition de constantes	`class UneClasse`
	`{ final int`
	`  version=3;}`
		`class UneClasse`
		`  { const int`
		`  version=3;}`

Tableau A.11 Spécification d'attributs constants

Autres qualifications des méthodes

En Java, *final* équivaut à *const* en C++ pour les variables d'instance ou de classe. En Java, une méthode qualifiée de *final* n'est pas une méthode retournant une valeur constante mais une méthode ne pouvant être redéfinie dans une classe dérivée.

Une méthode préfixée par *native* n'est pas implantée par la machine virtuelle, mais par du code dépendant de la plate-forme utilisée, par exemple du C ou de l'assembleur. Ces méthodes n'ont pas de corps. Elles peuvent être des méthodes de classe ou d'instance, héritables, masquables ou non.

Une méthode préfixée par *synchronized* est une méthode qui s'exécute via un moniteur qui en contrôle les accès (voir point 25.2, p. 354). Le verrou est sur la classe pour une méthode de classe, sinon sur l'objet (méthode d'instance).

En Java, une méthode qualifiée de *abstract* ressemble à une méthode virtuelle pure en C++, en ce sens qu'elle n'a pas de corps et requiert une définition explicite dans les sous-classes. Cependant, en Java, de telles méthodes sont définies dans une classe nécessairement abstraite elle aussi (voir point A.20, p. 439).

final signifie en Java :

- pour une classe : pas de dérivation de sous-classes;
- pour un attribut : attribut constant;
- pour une méthode : non redéfinissable dans une sous-classe avec les mêmes paramètres.

A.17 Constructeurs

Le principe de définition des constructeurs en Java est le même qu'en C++. Les constructeurs portent le même nom que la classe. Ils peuvent être surchargés afin d'offrir plusieurs manières d'initialiser des instances.

	Construction d'objet par copie
Java	`Temps t=new Temps(12,0,0);` `Temps t2=t.clone()`
C++	`Temps t(12,0,0);` `Temps t2=t;`

Tableau A.12 Clonage en Java

Il existe cependant quelques différences. En Java, il n'existe pas de constructeur par copie par défaut, la méthode *clone()* de la classe *Object* le remplace (voir tableau A.12).

L'ordre d'appel des constructeurs des super-classes est le même pour Java et C++ : le constructeur de la classe de base est toujours exécuté avant celui de la classe dérivée. Une différence notable a trait aux constructeurs des objets composants. Contrairement au C++, aucun constructeur par défaut d'objet composant n'est appelé en Java.

L'exemple du tableau A.13 en est une illustration. La classe *Cours* est composée de deux variables d'instance de la classe *Temps*. En C++ seulement, le constructeur par défaut de la classe *Temps* sera appelé. S'il n'existe pas, la classe *Temps* devra être redéfinie pour en inclure un.

	Java	C++

Modification du constructeur de la classe *Temps* pour afficher son nom	`Temps()`
	`{heure=0;`
	`minute=0;`
	`seconde=0;`
	`System.out.println`
	`("Temps()");`
	`}`
		`Temps()`
		`{heure=0;`
		`minute=0;`
		`seconde=0;`
		`cout << "Temps()";`
		`cout << endl;}`

Classe *Cours* composée de deux variables d'instance de la classe *Temps*	`import Temps;`
	`class Cours {`
	`public Temps`
	`debut,fin;`
	`public int numero;`
	`public Cours(int num)`
	`{numero=num;}`
	`};`
		`#include "Temps.h"`
		
		`class Cours {`
		`public :`
		`Temps debut,fin;`
		`int numero;`
		`Cours(int num)`
		`{numero=num;}`
		`};`

Déclaration et initialisation d'un cours	`Cours c=new Cours(12);`	`Cours c(12);`
Effet de l'instruction précédente		`Temps()` `Temps()`

Tableau A.13 Appel aux constructeurs des composantes

A.18 Destructeurs

Un destructeur en Java est une méthode spéciale (non nécessaire) appelée *finalize()* qui permet la libération de ressources systèmes non automatiquement libérées par le ramasse-miettes. Cette méthode ne joue pas tout à fait le même rôle que les destructeurs en C++. Elle est appelée par le ramasse-miettes qui, si nécessaire, libère l'espace alloué à des objets qui ne sont plus référencés.

En C++, l'appel au destructeur de la super-classe s'effectue automatiquement après le destructeur de la classe de l'instance à détruire. En Java, le programmeur doit explicitement terminer le texte de sa méthode *finalize()* par l'appel de celui de sa super-classe, soit *super.finalize()*. En Java, il n'est pas utile (ni nécessaire) de programmer des destructeurs, sauf si la classe a acquis une ressource autre que la mémoire que le ramasse-miettes ne peut récupérer comme, par exemple, un accès sur un fichier ou sur un socket.

A.19 Héritage

Héritage simple

Seul l'héritage simple est permis. Le mot clé *extends* équivaut aux : du C++.

	Spécification de l'héritage

Java	`import classeDeBase;`
	`class classeDerivee extends classeDeBase`
	`{ ...`
	`// attributs et méthodes de la classe dérivée`
	`};`

C++	`#include "classeDeBase.h"`
	`class classeDerivee : classeDeBase`
	`{ ...`
	`// attributs et méthodes de la classe dérivée`
	`};`

Tableau A.14 Spécification de l'héritage en Java et en C++

Modes d'héritage

En C++, trois modes d'héritage d'une classe sont permis : *public*, *protected*, *private*. En Java, il n'y en a qu'un seul (implicite) : *public*. Ces différents modes ont des conséquences sur les domaines de protection des membres (variables et méthodes) hérités.

Pour les modes d'héritage *private* et *protected*, le C++ permet de conserver dans la classe dérivée les domaines de protection des membres de la classe de base. En considérant également l'héritage multiple, on constate que le C++ permet d'exprimer de manière fine les relations d'héritage entre classes, au détriment cependant de la simplicité de compréhension pour des schémas de classes conséquents.

Mode d'héritage présent dans :	Mode d'héritage dans la classe dérivée	Protection d'un membre de la classe de base		
		private	protected	public
C++	`private`	inaccessible	`private`	`private`
C++	`protected`	inaccessible	`protected`	`protected`
C++ et Java	`public`	inaccessible	`protected`	`public`

Tableau A.15 Modes d'héritage en Java et en C++

A.20 Classe abstraite

En Java, une classe est abstraite si l'une de ses méthodes l'est. Le mot clé *abstract* n'est pas un remplacement du mot *virtual* du C++. En C++, le préfixe *virtual* sur une méthode sert à spécifier la liaison dynamique. De plus, si une méthode est ainsi préfixée, on peut la définir virtuelle pure en indiquant que son corps est vide. En Java, il y a toujours une liaison dynamique. Le préfixe *abstract* signifie virtuelle pure : les méthodes abstraites n'ont pas de corps.

	Java	C++

Méthode avec liaison dynamique	`void m(...)`
	`{ // corps de m`
	`}`
		`virtual void m(...)`
		`{ // corps de m`
		`}`

Méthode abstraite pure	`abstract void m(...);`	`virtual void m(...)=0;`

Tableau A.16 Méthode virtuelle avec corps

Le mot clé *abstract* doit également préfixer le nom de la classe si cette classe a au moins une méthode abstraite.

	Spécification de méthode abstraite

Java	`abstract class UneClasse`
	`{ abstract void uneMethode(...); // methode abstraite`
	`  ...`
	`};`

C++	`class UneClasse`
	`{  virtual void uneMethode(...)=0;`
	`  // methode virtuelle pure`
	`  ....`
	`};`

Tableau A.17 Spécification de classe abstraite (virtuelle)

A.21 Classe terminale

Java permet d'interdir la dérivation d'autres classes à partir d'une classe considérée comme terminale. Le mot clé *final* préfixant la définition de la classe permet de spécifier cette interdiction.

```
final class classeTerminale extends ...
{ ...
// attributs et méthodes de la classe
};
```

Spécification de classe terminale en Java

A.22 Amitié entre classes et entre instances

En Java, l'amitié autorise l'accès de toutes les variables et méthodes sans qualificatif aux autres classes du même package. Les membres non qualifiés ne sont pas accessibles par les classes d'un package différent. Les membres qualifiés de *private* ne sont pas accessibles par les autres classes du même package. Seules des instances de la même classe peuvent accéder à leurs membres privés respec-

tifs. C'est donc l'appartenance ou non d'une classe à un package qui décide de ses relations d'amitié avec d'autres classes.

En C++, l'amitié entre classes nécessite une définition explicite dans la classe autorisant les accès à ses membres privés. Ce qui entraîne de fréquentes recompilations au fur et à mesure que de nouvelles classes sont ajoutées et qu'elles doivent accéder à des membres privés d'une classe particulière. En Java, de telles nouvelles classes sont simplement à ajouter dans le package, ce qui ne requiert aucune compilation supplémentaire.

A.23 Gestion des exceptions

Java étend la gestion des exceptions de C++ par l'ajout de la clause optionnelle *finally*. Si cette clause est présente, elle est exécutée même s'il n'y a pas eu de levée d'exception dans le bloc *try {...} catch* la précédant.

En Java (voir point 8.1, p. 105), toutes les classes d'exception doivent être sous-classes de *Throwable*.

Déclaration des exceptions levées par une méthode

Cette déclaration présente des différences de syntaxe dans les deux langages (attention au *s* de *throws* en Java).

	Déclaration des exceptions levées par une méthode

Java	`void uneMethode(...) throws A,B,C`
	`{ //corps de la méthode`
	`}`

C++	`void uneMethode(...) throw (A,B,C)`
	`{ //corps de la méthode`
	`}`

Tableau A.18 Déclaration des exceptions levées par une méthode

Levée des exceptions

La levée de l'exception s'exprime en Java et en C++ par l'instruction *throw* qui est nécessairement suivie d'une instance d'exception. Si l'instance doit être créée, le procédé est le même que pour la construction de n'importe quelle instance en Java.

Propagation d'une exception en cours de traitement

En C++, cette propagation se fait par l'instruction *throw* seulement. En Java cette propagation requiert le nom de l'instance d'exception concernée.

	Propagation d'une exception en cours de traitement

Java	`....`
	`catch (IOException e)`
	`{ ...`
	`throw e;}`

C++	`....`
	`catch (IOException e)`
	`{ ...`
	`throw ;}`

Tableau A.19 Propagation d'une exception en cours de traitement

A.24 Caractéristiques de C++ absentes de Java

Outre celles que nous avons illustrées dans l'étude détaillée ci-dessus, il est à noter quelques caractéristiques du C++ non reprises dans Java.

Java n'autorise pas la surcharge des opérateurs. L'opérateur +, par exemple, ne peut être redéfini pour des objets de la classe *Matrice*. Il faudra définir une méthode *plus()* permettant l'addition matricielle.

Java n'a pas la notion de pointeur, on ne peut avoir de pointeur sur des méthodes. De nombreux cas d'utilisation des fonctions en paramètres peuvent être résolus avec la notion d'interface en Java.

Java n'incorpore pas la notion de classe générique (*template*).

A.25 Caractéristiques de Java non supportées par C++

Java présente de nombreuses qualités qui ont suscité un engouement assez large sur la planète. La portabilité, la sécurité (voir chapitre 27), le fonctionnement multiprocessus (voir p. 107) des applications sont des points majeurs. En outre, Java contrôle l'accès aux ressources partagées à l'aide de moniteurs.

Gageons que le programmeur familier du C++, motivé par les avantages de Java, apprendra vite ce langage, car il a été bien pensé sur de nombreux aspects dont la pertinence apparaît lorsque l'on développe des applications assez volumineuses.

Annexe B

Grammaire BNF de Java

Nous avons utilisé les conventions suivantes pour les règles de grammaire :

- [nom] : nom est répété 0 ou 1 fois;
- < nom > : nom est répété 0 ou n fois;
- a | b : soit a soit b;
- "chaîne" : chaîne terminale.

B.1 Niveau syntaxique

```
goal = compilationUnit.

literal = integerLiteral | floatingPointLiteral | booleanLiteral
            | characterLiteral | stringLiteral | "null".

type = primitiveType | referenceType.

primitiveType = numericType | "boolean".

numericType = integralType | floatingPointType.

integralType = "byte" | "short" | "int" | "long" | "char".

floatingPointType = "float" | "double".

referenceType = classOrInterfaceType | arrayType.

classOrInterfaceType = name.
```

```
classType = classOrInterfaceType.

interfaceType = classOrInterfaceType.

arrayType  = ( primitiveType | name ) "[" "]" < "[" "]" >.

name  = simpleName | qualifiedName.

simpleName  = identifier.

qualifiedName  = name "." identifier.

compilationUnit  = [   packageDeclaration ] <   importDeclaration >
< typeDeclaration >.

packageDeclaration  = "package"   name ";".

importDeclaration  = "import"   name [ ".*" ] ";".

typeDeclaration  = classDeclaration |   interfaceDeclaration | ";".

modifier  =    "public" | "protected" | "private" | "static" |
               "abstract" | "final" | "native" | "synchronized" |
               "transient" | "volatile".

classDeclaration  = < modifier > "class"   identifier
     [ super ] [ interface ] classBody.

super  = "extends" classType.

interfaces  = "implements"   interfaceType < ","   interfaceType >.

classBody  = "{" < classBodyDeclaration > "}".

classBodyDeclaration  =
     classMemberDeclaration |
     staticInitializer |
     constructorDeclaration.

classMemberDeclaration  =   fieldDeclaration |   methodDeclaration.

fieldDeclaration  = <   modifier > type
     variableDeclarator < ","   variableDeclarator > ";".

variableDeclarator  =   variableDeclaratorId [ "="
variableInitializer ].

variableDeclaratorId  =   identifier < "[" "]" >.

variableInitializer  =  expression | arrayInitializer.

methodDeclaration  =   methodHeader methodBody.
```

```
methodHeader = < modifier > ( type | "void" )
        methodDeclarator [ throws ].

methodDeclarator = identifier "(" [ formalParameterList ] ")"
        < "[" "]" >.

formalParameterList = formalParameter < "," formalParameter >.

formalParameter = type variableDeclaratorId.

throws = "throws" classTypeList.

classTypeList = classType < "," classType >.

methodBody = block | ";".

staticInitializer = "static" block.

constructorDeclaration = < modifier > constructorDeclarator
        [ throws ] constructorBody.

constructorDeclarator = simpleName "(" [ formalParameterList ]
")".

constructorBody = "{" [ explicitConstructorInvocation ]
                    [ blockStatements ] "}" .

explicitConstructorInvocation =
        "this" "(" [ argumentList ] ")" ";"
        | "super" "(" [ argumentList ] ")" ";".

interfaceDeclaration = < modifier > "interface" identifier
[ extendsInterfaces ] interfaceBody.

extendsInterfaces = "extends" interfaceType < "," interfaceType
>.

interfaceBody = "{" < interfaceMemberDeclaration > "}".

interfaceMemberDeclaration =
     constantDeclaration | asbstractMethodDeclaration.

constantDeclaration = fieldDeclaration.

abstractMethodDeclaration = methodHeader ";".

arrayInitializer = "{" [ variableInitializer
        < "," variableInitializer > ] [ "," ] "}".

block = "{" < blockStatement > "}".

blockStatement = localVariableDeclarationStatement | statement.
```

```
localVariableDeclarationStatement  =   localVariableDeclaration ";".

localVariableDeclaration  =  type   variableDeclarator
        < ","   variableDeclarator >.

statement  =
   statementWithoutTrailingSubstatement
   |   labeledStatement
   |   ifThenStatement
   |   ifThenElseStatement
   |   whileStatement
   |   forStatement.

statementNoShortIf  =
   statementWithoutTrailingSubstatement
   |   labeledStatementNoShortIf
   |   ifThenElseStatementNoShortIf
   |   whileStatementNoShortIf
   |   forStatementNoShortIf.

statementWithoutTrailingSubstatement  =
   block
   | emptyStatement
   | expressionStatement
   | switchStatement
   | doStatement
   | breakStatement
   | continueStatement
   | returnStatement
   | synchronizedStatement
   | throwStatement
   | tryStatement.

emptyStatement  = ";".

labeledStatement  =   identifier ":"   statement.

labeledStatementNoShortIf  =   identifier ":"   statementNoShortIf.

expressionStatement  =   statementExpression ";".

statementExpression  =
   assignment
   |   preIncrementExpression
   |   preDecrementExpression
   |   postIncrementExpression
   |   postDecrementExpression
   |   methodInvocation
   | classInstanceCreationExpression.

ifThenStatement  = "if" "("  expression ")"   statement.
```

```
ifThenElseStatement  =
   "if" "("  expression ")"
        statementNoShortIf
   "else"   statement.

ifThenElseStatementNoShortIf  =
   "if" "("  expression ")"
        statementNoShortIf
   "else"   statementNoShortIf.

switchStatement  = "switch" "("  expression ")"   switchBlock.

switchBlock  = "{" [   switchBlockStatementGroups ]
   [   switchLabels ] "}".

switchBlockStatementGroups  =   switchBlockStatementGroup
      <   switchBlockStatementGroup >.

switchBlockStatementGroup  =   switchLabels < blockStatement >.

switchLabels  =   switchLabel <   switchLabel >.

switchLabel  = "case" constantExpression  ":" | "default" ":".

whileStatement  = while "("  expression ")"   statement.

whileStatementNoShortIf  =
   while "("  expression ")"
      statementNoShortIf.

doStatement  = "do"   statement "while" "("  expression ")" ";" .

forStatement  =
   "for" "("
            [   forInit ] ";"
            [  expression ] ";"
            [   forUpdate ] ")"
      statement.

forStatementNoShortIf  =
   "for" "("
            [   forInit ] ";"
            [  expression ] ";"
            [   forUpdate ] ")"
      statementNoShortIf.

forInit  =   statementExpressionList |   localVariableDeclaration.

forUpdate  =   statementExpressionList.

statementExpressionList  =
   statementExpression < ","   statementExpression >.
```

```
breakStatement  = "break" [   identifier ] ";".

continueStatement  = "continue" [   identifier ] ";".

returnStatement  = "return" [  expression ] ";".

throwStatement  = "throw"  expression ";".

synchronizedStatement  = "synchronized" "("  expression ")" block.

tryStatement  = "try" block ( catches | [ catches ]   finally ).

catches  = catchClause < catchClause >.

catchClause  = "catch" "("   formalParameter ")" block.

finally  = "finally" block.

primary  =   primaryNoNewArray | arrayCreationExpression.

primaryNoNewArray  =
      literal
      | this
      | "("  expression ")"
      | classInstanceCreationExpression
      | fieldAccess
      | methodInvocation
      | arrayAccess.

classInstanceCreationExpression  =
      "new" classType "(" [ argumentList ] ")".

argumentList  =  expression < ","  expression >.

arrayCreationExpression  =
      "new" (   primitiveType | classOrInterfaceType )
      "["  expression "]" < "["  expression "]" > < "[" "]" >.

fieldAccess  =
      primary "."   identifier
      | "super" "."   identifier.

methodInvocation  =
      name "(" [ argumentList ] ")"
      | primary "."   identifier "(" [ argumentList ] ")"
      | "super" "."   identifier "(" [ argumentList ] ")".

arrayAccess  =
      name "["  expression "]"
      | primaryNoNewArray "["  expression "]".

postfixExpression  =
```

```
      primary
      |   name
      |   postIncrementExpression
      |   postDecrementExpression.

postIncrementExpression  =   postfixExpression "++".

postDecrementExpression  =   postfixExpression "--".

unaryExpression  =
      preIncrementExpression
      |   preDecrementExpression
      | "+" unaryExpression
      | "-" unaryExpression
      | unaryExpressionNotPlusMinus.

preIncrementExpression  = "++" unaryExpression.

preDecrementExpression  = "--" unaryExpression.

unaryExpressionNotPlusMinus  =
      postfixExpression
      | "~" unaryExpression
      | "!" unaryExpression
      | castExpression.

castExpression  =
      "("   primitiveType < "[" "]" > ")" unaryExpression
      | "("  expression ")" unaryExpressionNotPlusMinus
      | "("   name "[" "]" < "[" "]" > ")"
unaryExpressionNotPlusMinus.

multiplicativeExpression  =
      unaryExpression
      |   multiplicativeExpression "*" unaryExpression
      |   multiplicativeExpression "/" unaryExpression
      |   multiplicativeExpression "%" unaryExpression.

additiveExpression  =
      multiplicativeExpression
      | additiveExpression "+"   multiplicativeExpression
      | additiveExpression "-"   multiplicativeExpression.

shiftExpression  =
      additiveExpression
      |   shiftExpression "<<" additiveExpression
      |   shiftExpression ">>" additiveExpression
      |   shiftExpression ">>>" additiveExpression.

relationalExpression  =
      shiftExpression
      |   relationalExpression "<"   shiftExpression
```

```
      |   relationalExpression ">"   shiftExpression
      |   relationalExpression "<="   shiftExpression
      |   relationalExpression ">="   shiftExpression
      |   relationalExpression "instanceof"   referenceType.

equalityExpression  =
      relationalExpression
      |  equalityExpression "=="   relationalExpression
      |  equalityExpression "!="   relationalExpression.

andExpression  =
      equalityExpression
      | andExpression "&"  equalityExpression.

exclusiveOrExpression  =
      andExpression
      |  exclusiveOrExpression "^" andExpression.

inclusiveOrExpression  =
      exclusiveOrExpression
      |   inclusiveOrExpression "|"  exclusiveOrExpression.

conditionalAndExpression  =
      inclusiveOrExpression
      | conditionalAndExpression "&&"   inclusiveOrExpression.

conditionalOrExpression  =
      conditionalAndExpression
      | conditionalOrExpression "||" conditionalAndExpression.

conditionalExpression  =
      conditionalOrExpression
      | conditionalOrExpression "?"  expression ":"
conditionalExpression.

assignmentExpression  = conditionalExpression | assignment.

assignment  =   leftHandSide assignmentOperator
assignmentExpression.

leftHandSide  =   name |   fieldAccess | arrayAccess.

assignmentOperator  =
      "=" | "*=" | "/=" | "%=" | "+=" | "-="
      | "<<=" | ">>=" | ">>>=" | "&=" | "^=" | "|=".

expression  = assignmentExpression.

constantExpression  =  expression.
```

B.2 Niveau lexical

```
unicodeInputCharacter = unicodeEscape | rawInputCharacter.

unicodeEscape = "\" unicodeMarker hexDigit hexDigit hexDigit
hexDigit.

unicodeMarker = "u" < "u" >.

rawInputCharacter = "any Unicode character".

lineTerminator = "LF" | "CR" | "CRLF".

inputCharacter = unicodeInputCharacter "but not CR or LF".

input = [ inputElements ] [ sub ].

inputElements = inputElement < inputElement >.

inputElement = whiteSpace | comment | token.

token = identifier | keyword | literal | separator | operator.

sub = "the ASCII SUB character, also known as control-Z".

whiteSpace = " " | "TAB" | "FF" | lineTerminator.

comment = traditionalComment | endOfLineComment |
documentationComment.

traditionalComment = "/*" notStar commentTail.

endOfLineComment = "//" < inputCharacter > lineTerminator.

documentationComment = "/**" commentTailStar.

commentTail = "*" commentTailStar | notStar commentTail.

commentTailStar =
      "/"
      | "*" commentTailStar
      | notStarNotSlash commentTail.

notStar = inputCharacter "but not *" | lineTerminator.

notStarNotSlash = inputCharacter "but not * or /" | lineTerminator.

identifier = identifierChars "but not" (keyword | booleanLiteral |
nullLiteral).

identifierChars = javaLetter | identifierChars javaLetterOrDigit.
```

```
javaLetter = "any Unicode character that is a Java letter".

javaLetterOrDigit = "any Unicode character that is a Java letter-or-
digit".

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

integerLiteral =
   ( decimalNumeral | hexNumeral | octalNumeral ) [ "l" | "L" ] .

decimalNumeral  = "0" | ( "1..9" < "0..9" > ).

digits = "0..9" < "0..9" >.

hexNumeral = "0" ( "x" | "X" ) hexDigit < hexDigit >.

hexDigit = "0..9" | "a..f" | "A..F".

octalNumeral = "0" "0..7" < "0..7" >.


floatingPointLiteral =
   ( ( (digits "." [ digits ] |  "." digits) [ exponentPart ]
      | digits exponentPart ) [ "f" | "F" | "d" | "D" ] )
      | digits [ exponentPart ] ("f" | "F" | "d" | "D" )  .


exponentPart = ("e" | "E") [ "+" | "-" ] digits.

booleanLiteral = "true" | "false".

characterLiteral = "'" ( singleCharacter | escapeSequence ) "'".

singleCharacter = inputCharacter "but not ' or \".

stringLiteral = """ [ stringCharacters ] """.

stringCharacters = stringCharacter < stringCharacter >.

stringCharacter = inputCharacter "but not double quote or \" |
escapeSequence.

escapeSequence =  "\" ("b" | "t" | "n" | "r" | """  | "'" | "\" |
octalEscape ).
```

```
octalEscape =  "\" [ "0..3" ] [ "0..7" ] "0..7".

separator = "(" | ")" | "{" | "}" | "[" | "]" | ";" | "," | ".".

operator =
"=" | ">" | "<" | "!" | "~" | "?" | ":" | "==" | "<=" | ">=" | "!=" |
"&&" | "||" | "++" | "--" | "+" | "-" | "*" | "/" | "&" | "|" | "^"
| "%" | "<<" | ">>" | ">>>" | "+=" | "-=" | "*=" | "/=" | "&=" | "|="
| "^=" | "%=" | "<<="
| ">>=" | ">>>=" .
```

Annexe C

Les fichiers d'archives .jar

Une application Java peut être constituée d'un nombre important de classes et, par conséquent, d'autant de fichiers. Le transfert d'une applet vers le butineur peut s'avérer relativement long, car le chargement se fait classe par classe. Pour accélérer le chargement, il est possible de construire un fichier unique compressé comportant plusieurs fichiers. On parle alors d'un *.jar* (pour java archive), équivalent des *.tar* d'Unix ou des *.zip* du monde des PC. Les fichiers *.jar* ont l'avantage d'être indépendants et portables.

C.1 L'option ARCHIVES

En spécifiant dans la balise <applet> d'un fichier HTML l'option ARCHIVES, on modifie le scénario de l'exécution d'une applet. Les classes contenues dans les fichiers *.jar* sont chargées en une seule fois avant le début de l'exécution de l'applet. Cela représente essentiellement une optimisation pour leur exécution.

Examinons un exemple. Le jeu du serpent présenté dans ce livre, contient (pour une certaine version) les classes suivantes (avec leur taille en bytes) :

```
876   Appat.class
3,021 Jeu.class
2,008 LeMonde.class
1,590 Serpent.class
222   appletcode.html
```

Le butineur WWW chargera donc la page *appletcode.html* suivante :

```
<title>jeux1</title>
<hr>
<applet code=Jeu.class width=500 height=300>
<PARAM NAME=vitesse VALUE="20">
<PARAM NAME=hauteur VALUE="300">
<PARAM NAME=largeur VALUE="500">
<PARAM NAME=grain    VALUE="10">
</applet>
```

L'interprétation de la balise <applet> entraîne le chargement de la classe *Jeu* puis son exécution. L'exécution de ce code contient une instruction pour instancier un objet de la classe *LeMonde*. Comme la machine virtuelle de Java fait du chargement dynamique, le chargement de cette classe sera effectué, puis l'exécution reprendra. Et ainsi de suite. Finalement les cinq classes seront chargées et le jeu continuera sans autres temps morts.

En construisant un fichier *.jar* des classes chargées par *Jeu*, on évite ces interruptions dans l'exécution.

On utilise l'outil *jar* pour construire ce fichier. Dans notre cas, on peut exécuter la commande suivante (la signification des paramètres cvf est expliquée en fin de chapitre) :

```
jar cvf Serpent.jar *.class
```

L'outil montre les classes qu'il inclut dans l'archive ainsi que les gains obtenus grâce à la compression.

```
added manifest
adding: Appat.class (in=876) (out=579) (deflated 33%)
adding: Jeu.class (in=3021) (out=1683) (deflated 44%)
adding: LeMonde.class (in=2008) (out=1163) (deflated 42%)
adding: PointMobile.class (in=483) (out=336) (deflated 30%)
adding: Serpent.class (in=1590) (out=900) (deflated 43%)
```

Dans un nouveau répertoire, nous pouvons mettre les fichiers suivants, la page *html* et l'archive *.jar* :

```
  244 appletcodejar.html
3,797 Serpent.jar
```

Le butineur WWW chargera donc la page *appletcodejar.html* suivante :

```
<title>jeux1</title>
<hr>
<applet archives="Serpent.jar" code=Jeu.class width=500 height=300>
<PARAM NAME=vitesse VALUE="20">
<PARAM NAME=hauteur VALUE="300">
<PARAM NAME=largeur VALUE="500">
<PARAM NAME=grain    VALUE="10">
</applet>
```

Et la première chose sera de charger l'archive *Serpent.jar*. Ensuite, le butineur cherchera dans les classes chargées celle demandée dans l'option `code` (dans notre cas *Jeu.class*), si elle s'y trouve. Il commence alors l'exécution de celle-ci. Sinon, il la charge en utilisant éventuellement l'option *codebase*.

Il existe une autre possibilité pour déclarer une archive. Cette archive peut être spécifiée dans les paramètres de l'applet (la balise `<param>`) :

```
<applet code=Jeu.class width=500 height=300>
<PARAM NAME=ARCHIVES VALUE="Serpent.jar">
. . .
</applet>
```

Il est possible de spécifier plusieurs archives lors du chargement de l'applet. Il suffit de séparer les noms des fichiers par un signe +.

```
<applet archives="Serpent.jar+Autres.jar"
        code=Jeu.class width=500 height=300>
. . .
</applet>
```

L'utilisation des archives est intéressante dans les cas suivants :

- vous désirez diminuer le temps de chargement de vos applets;
- vous désirez simplifier la distribution d'une application ou d'un package;
- vous désirez certifier et authentifier votre code.

Remarque: l'utilisation des archives n'est pas toujours une optimisation du temps de chargement. En effet, si votre applet contient beaucoup de classes, généralement peu d'entre elles sont effectivement chargées lors d'une exécution. En utilisant une archive, elles seront toutes chargées alors que beaucoup ne seront pas utilisées. Il faut dont choisir entre une solution optimiste et une solution pessimiste.

C.2 Retour sur l'outil jar

L'outil *jar* permet :

- d'archiver un ensemble de fichiers (en mode création et en mode de mise à jour);
- de lister le contenu d'une archive;
- d'extraire des fichiers d'une archive;
- de manipuler le fichier *manifest* associé à l'archive.

Le fichier *manifest* associé à l'archive contient des méta-informations sur l'archive. Sans mécanisme de sécurité ou d'authentification, il contient également des informations sur les check-sum des archives.

Nous reproduisons, en la traduisant, l'aide de la commande `jar` afin d'en expliciter toutes les options :

```
>jar
Usage: jar {ctxu}[vfmOM] [jar-file] [manifest-file] [-C dir] files
...
Options:
  -c créer une nouvelle archive
  -t lister le contenu d'une archive
  -x extraire les fichiers nommés(ou all) de l'archive
  -u mettre à jour une archive existante archive
  -v generer une sortie longue sur la sortie standard
  -f spéficier le nom de l'archive
  -m inclure des informations manifestes d'un fichier manifest
spécifié
  -O stocker seulement;  pas de compression ZIP
  -M ne pas créer de fichier manifest pour les entrées
  -C changer de directoire et inclure les fichiers suivants
```

L'authentification et la sécurisation du code sont également possibles, mais le lecteur doit être bien informé sur les mécanismes de sécurité (clé publique, signature, etc.). Nous le renvoyons à la documentation de Sun pour cet aspect particulier.

Le dernier point à souligner est que l'outil *jar* peut être utilisé pour la compression de n'importe quel fichier, et pas seulement des fichiers *.class*. Vous pouvez aussi l'utiliser comme n'importe quel autre outil de compression avec l'avantage de la portabilité. Il permet notamment d'associer des images, des sons ou d'autres ressources qui pourraient être nécessaires à l'applet, et qui seront donc distribuées avec elle.

Annexe D

Rappel sur HTML

Ce court rappel sur le langage HTML et ses principaux délimiteurs (balises) est destiné à permettre une meilleure intégration des applets Java sur le Web.

Le programmeur Java doit intégrer son applet dans une page HTML. Cette page constitue donc un emballage de l'applet. Le succès d'un produit dépendant aussi de son emballage... vous voilà averti.

D.1 SGML

HTML (*Hyper Text Markup Language*) est issu d'un système beaucoup plus puissant appelé SGML (*Standard General Markup Language*).

SGML est un standard international pour la représentation de textes sous une forme informatisée indépendante du système et des périphériques d'affichage (imprimantes, écrans, etc.). Il s'agit d'un métalangage qui permet de définir d'autres langages basés sur les délimiteurs (*markup* en anglais).

La description obtenue est une DTD (*Document Type Definition*). HTML est ainsi une DTD de SGML. SGML sera appelé dans le futur à jouer un rôle de plus en plus important pour la gestion des documents électroniques, car sa description indépendante des plates-formes permet de manipuler facilement le texte et de structurer fortement les documents.

D.2 Délimiteurs

Tags, balises, signets ou bornes sont autant de termes utilisés pour désigner les délimiteurs. Ceux-ci ont la forme générale suivante :

```
<Nom_Délimituer>
    subit l'action du délimiteur
</Nom_Délimiteur>
```

Le nom du délimiteur représente une action de structuration ou de mise en forme sur le texte qui le suit, jusqu'à l'indication de la fin de l'action </>. Certains délimiteurs ne nécessitent pas de fin, comme par exemple <*HR*> qui indique une ligne horizontale.

Il existe plusieurs « standards HTML » : les *versions* (1.0, 2.0 et maintenant 3.2[1]) ainsi que les *dialectes*, soutenus par les acteurs du marché du logiciel tels que Netscape ou Microsoft.

Normalement, tous les butineurs actuels devraient supporter le HTML 2.0. Dans la suite de ce chapitre, nous signalons par l'indication *(*)* les extensions propres au dialecte de Netscape, par ailleurs généralement comprises par l'Internet Explorer de Microsoft.

Parmi ces extensions, les suivantes sont prises en compte dans la norme 3.2 :

- les tables;
- le délimiteur *APPLET*;
- l'écoulement du texte autour des images (*ALIGN*);
- les indices et les exposants (*SUB*, *SUP*).

Délimiteurs de structuration

Les délimiteurs suivants (voir tableau D.1) permettent d'indiquer un minimum de structuration pour un document HTML.

Délimiteur	Signification
<HTML>...</HTML>	Type du document
<!...>	Commentaire
<HEAD>...</HEAD>	En-tête du document (non affiché)
<TITLE>...</TITLE>	Titre du document (non affiché)
<BODY>...</BODY>	Corps du document
<Hi>...<Hi>	Titres de niveau : i =1 à 6

Tableau D.1 HTML, délimiteurs de structuration

1. Voir les annonces dans [14].

Délimiteur	Signification
<P>...</P>	Paragraphe
 	Rupture de ligne
<HR>	Ligne horizontale de séparation

Tableau D.1 HTML, délimiteurs de structuration

Exemple de document HTML structuré (*ex1.html*) :

```
<HTML>
<HEAD>
<! le titre apparait dans les outils de navigation>
<! et dans les indexeurs Yahoo, altavista>
<TITLE> Titre du document visible </TITLE>
</HEAD>
<BODY>
<H1>JAVA</H1>
texte du document
<H2>Une introduction</H2>
Il etait une fois une ile ....
qui etait au milieu de l'ocean ...<P>
Cette ile ...
<H2>Une ...</H2>
texte du document
<H2>Une conclusion</H2>
Il cultivait un delicieux cafe ....<BR>
qui se buvait en faisant des programmes ...
</BODY>
</HTML>
```

Figure D.1 Document HTML structuré (ex1.html)

L'image obtenue dans un butineur permet d'observer que les titres, les ruptures de lignes et les fins de paragraphes sont bien respectés.

Délimiteurs de listes

Les délimiteurs suivants (tableau D.2) permettent de structurer des listes ayant différentes présentations.

Délimiteur	Signification

`<UL>`
`  <LI>`
`  <LI>`
`</UL>`
	UL - Unnumbered List
	LI - List Item
	Liste non numérotée

`<OL>`
`  <LI>`
`  <LI>`
`</OL>`
	OL - Ordered List
	LI - List Item
	Liste numérotée

`<DL>`
`  <DT>`
`  <DD>`
`</DL>`
	DL - Definition List
	DT - Definition Title
	DD - Definition Description
	Liste de définition

`<UL TYPE=disc | circle | square> (*)`	Type de mise en évidence (disque, cercle, carré)
`<OL TYPE=a | A | i | I | 1 |> (*)`	Type de numérotation

Tableau D.2 HTML, délimiteurs de listes

Exemple de listes (*ex2.html*) :

```
<HTML>

<HEAD>
<TITLE> liste ... </TITLE>
</HEAD>

<BODY>
<H1>Les listes</H1>
<OL> Liste à ne pas oublier
   <LI> les nombres premiers
   <LI> les décimales de pi
</OL>
mais avec en plus ...
<OL TYPE=i> Liste des numérations gadgets
   <LI> i, ii, iii,
   <LI> a, b, c,
   <LI> 1, 2 , 3
</OL>
```

```
et encore
<UL>Listes OUX
   <LI>bijoux
   <LI>cailloux
   <LI>poux
   <LI>...
</UL>
et enfin

<DL> Ici quelques définitions importantes
   <DT>Classe
   <DD>Une classe est une description ....
   <DT>Interface
   <DD>Une Interface est une description ....
</DL>

</BODY>

</HTML>
```

Nous obtenons l'affichage suivant (l'image a été volontairement tronquée) :

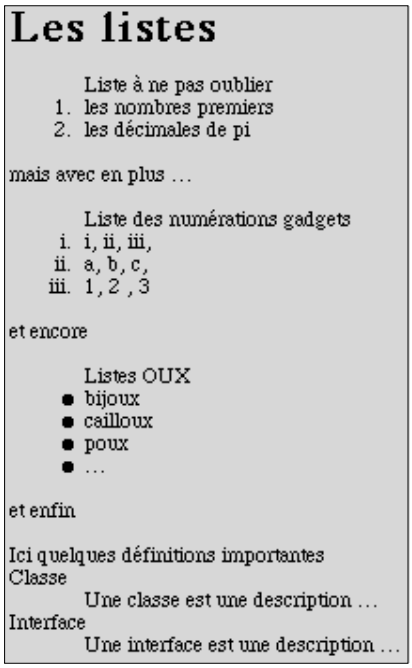


Figure D.2 Document HTML contenant des listes (ex2.html)

Délimiteurs d'attributs logiques textuels

Les délimiteurs suivants permettent de définir le type d'un texte, qui influence généralement sa forme. Il est à noter que ces types sont relativement limités et

qu'ils sont orientés vers des documents de programmation. Exemple d'attributs logiques (*ex3.html*) :

Délimiteur	Signification
<EM>...</EM>	Mise en évidence (EMphase).
<STRONG>...</STRONG>	Mise en évidence forte.
<CITE>...</CITE>	Citation.
<KDB>...</KDB>	Entrée au clavier.
<SAMP>...</SAMP>	Exemple.
<VAR>...</VAR>	Variable.

Tableau D.3 HTML, délimiteurs d'attributs logiques textuels

```
<HTML>
<HEAD> <TITLE> attributs logiques ... </TITLE> </HEAD>
<BODY>
<H1>attribut logique</H1>
Texte "normal" sans attribut logique<p>
<EM>Attention </EM><p>
<STRONG>Important</STRONG><p>
<CITE>"Sur un arbre perché maitre corbeau tenait un ..." La
Fontaine</CITE><p>
<KDB>La touche ESC permet de ...</KDB><p>
<SAMP>Ici on trouve un exemple de multiplication</SAMP><p>
<VAR>variable</VAR><p>
</BODY> </HTML>
```

Nous obtenons l'image de la figure D.3 : le choix d'un type de caractères et de son style repose sur l'interpréteur HTML.

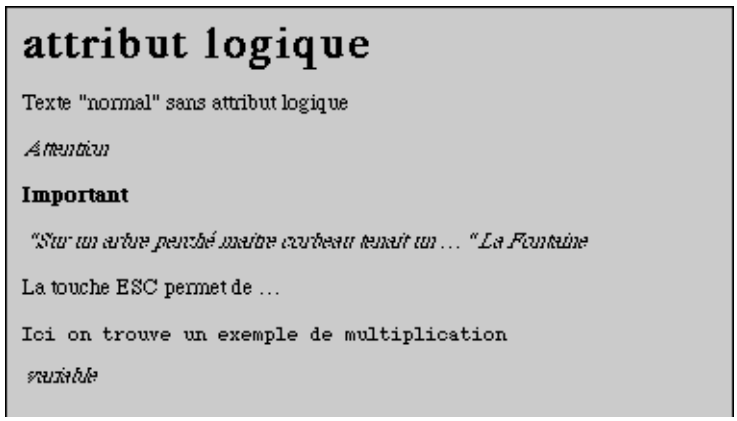


Figure D.3 Attributs logiques du texte en HTML (ex3.html)

Délimiteurs d'attributs typographiques

Les délimiteurs suivants permettent de définir directement les éléments typographiques du texte.

Délimiteur	Signification
<B>...</B>	Gras.
<I>...</I>	Italique.
<TT>...</TT>	(TeleType) fonte non proportionnelle (courrier) respecte les retours de ligne.
^{...}	Variable en exposant, au-dessus de la ligne de base.
_{...}	Variable en indice, en dessous de la ligne de base.
<FONT SIZE=n COLOR=#rrvvbb >	Spécifie la taille de la fonte par rapport à une échelle relative de 1 à 7 et la couleur en hexadécimal.

Tableau D.4 HTML, délimiteurs d'attributs typographiques

Exemple d'attributs typographiques (ex4.html) :

```
<HTML>

<HEAD>
<TITLE> attribut typographiques ... </TITLE>
</HEAD>

<BODY>
<H1>attribut typographique</H1>
<B>en gras</B><P>
<I>en  italique</I><P>
<TT>Comme sur une machine à écrire!</TT><p>
Doit apparaître<SUP>en exposant</SUP><P>
Doit apparaître<SUB>en indice</SUB><P>
<FONT size=1 color=#FF0000>petit rouge</SUB><P>
<FONT size=7 color=#0000FF>grand bleu</SUB><P>
</BODY>

</HTML>
```

Nous obtenons l'image de la figure D.4 : le choix d'un type de caractères et de son style repose sur les indications du concepteur de la page. Néanmoins, ces aspects esthétiques étant fortement dépendants des butineurs, il est conseillé de les utiliser avec parcimonie.

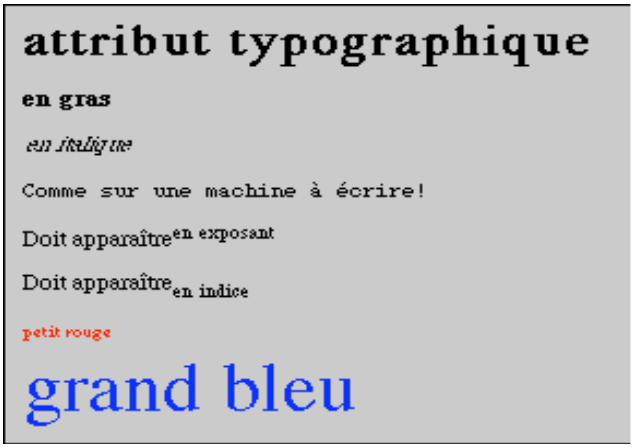


Figure D.4 Attributs typographiques en HTML (*ex4.html*)

Délimiteurs de zones d'écran

Netscape implante, à l'aide du concept de *frame*, la possibilité de partager l'espace de la fenêtre en plusieurs zones séparées. Chaque zone pourra afficher un document indépendamment des autres ou être remplie par un autre document. Le délimiteur <*FRAMESET*> remplace le délimiteur <*BODY*>.

Délimiteur	Signification
`<FRAMESET COLS="hauteur_liste" ROWS="largeur_liste"> ...</FRAMESET>`	Définit la répartition de la zone en colonnes ou en rangées. La répartition est : absolue (n) en points, en pourcentage (n%), relative (*,n*). Les FRAMESET peuvent être imbriqués les uns dans les autres.
`<FRAME SRC="URL" NAME="Nom_Zone" MARGINWIDTH=n MARGINHEIGHT=n SCROLLING="yes|no|auto" NORESIZE> </FRAME>`	Ce délimiteur est contenu dans un FRAMESET, il donne un nom à une zone et indique l'URL pour charger son contenu; les autres paramètres concernent la taille des marges la séparant des autres zones, les ascenseurs de défilement de la zone et la possibilité de la redimensionner (avec la souris, vous pouvez redimensionner une zone en vous positionnant sur l'une de ses marges).
`<NOFRAME> ... </NOFRAME>`	Le texte mis entre ces deux délimiteurs sera exécuté si le navigateur ne connaît pas le concept de FRAME. Cela permet donc de donner aussi accès à ces pages à quelqu'un qui ne possède pas de navigateur adapté.

Tableau D.5 HTML, délimiteurs de zones d'écran

Exemple de délimiteurs de zones (*ex5.html*) :

```
<HTML>
<HEAD><TITLE> délimiteurs de zones... </TITLE></HEAD>
<FRAMESET ROWS=*,*,3*>
   <FRAME SRC="ex5h.html" NAME="haut">
   <FRAME SRC="ex5m.html" NAME="milieu">
   <FRAMESET COLS=40%,60%>
     <FRAME SRC="ex5bd.html" NAME="basdroite">
     <FRAMESET ROWS=*,*>
       <FRAME SRC="ex5bgh.html" NAME="basgauchehaut">
       <FRAME SRC="ex5bgb.html" NAME="basgauchebas">
      </FRAMESET>
   </FRAMESET>
</FRAMESET>
<NOFRAME>
   Cette page est mieux vue avec une version supportant les FRAME,
suite du texte.
</NOFRAME> </HTML>
```

La figure D.5 illustre l'exécution avec un butineur connaissant le concept de *FRAME*, et avec un second qui ne le connaît pas. Conserver une équivalence de navigation en programmant une double interprétation des pages HTML peut être un exercice coûteux.

Figure D.5 Délimiteurs de zones en HTML (*ex5.html*)

Délimiteurs d'ancres et de liens

Les délimiteurs suivants permettent de définir des liens entre les documents. C'est ici que se joue la partie hypertexte de HTML.

Délimiteur	Signification
`<A NAME="...">... </A>`	Définit une ancre à l'intérieur du document. La partie comprise entre les deux délimiteurs est la zone référencée.
`<A HREF="URL"> ... </A>`	Lien vers un autre URL (document ou éventuellement intérieur du même document). La partie comprise entre les deux délimiteurs définit le texte du lien dans le document appelant (généralement souligné).
`<A HREF="URL" TARGET="frame"> (*)`	Lien vers un autre URL (document) à afficher dans la sous-fenêtre désirée.

Tableau D.6 HTML, délimiteurs d'ancres et de liens

Exemple d'ancres et de liens (*ex6.html*) :

```
<HTML>
<HEAD><TITLE> fichier liens hypertextuels ... </TITLE></HEAD>
<BODY>
<H1>liens hypertextuels</H1>
texte ...<p>
attention <a href="ex6a.html"> ici </a> se trouve une liste de site
important.<p>
texte ...<p>
</BODY>
</HTML>
```

Exemple : *ex6a.html*

```
<HTML>
<HEAD><TITLE>liste importante ex6a </TITLE></HEAD>
<BODY>
<A href="ex6.html">sommaire</A><P>
<H1>liens hypertextuels</H1>
<H2>liens hypertextuels</H2>
item 1<P>
item 2<P>
<a href="#java">liens sur java</A><P>
item n<p>
...<p>
<A name="#java"><H2>JAVA</H2></A>
...
<A href="http://java.sun.com/">sun et java</A><P>
...
</BODY>
</HTML>
```

Nous obtenons l'image suivante pour le document *ex6.html*. En cliquant sur ici, on charge le document *ex6a.html*. En cliquant sur sommaire, on retourne à *ex6.html*. En cliquant sur liens sur java, on avance dans le document jusqu'à la zone de référence. En cliquant sur sun et java, on charge la « home page » du serveur Sun.

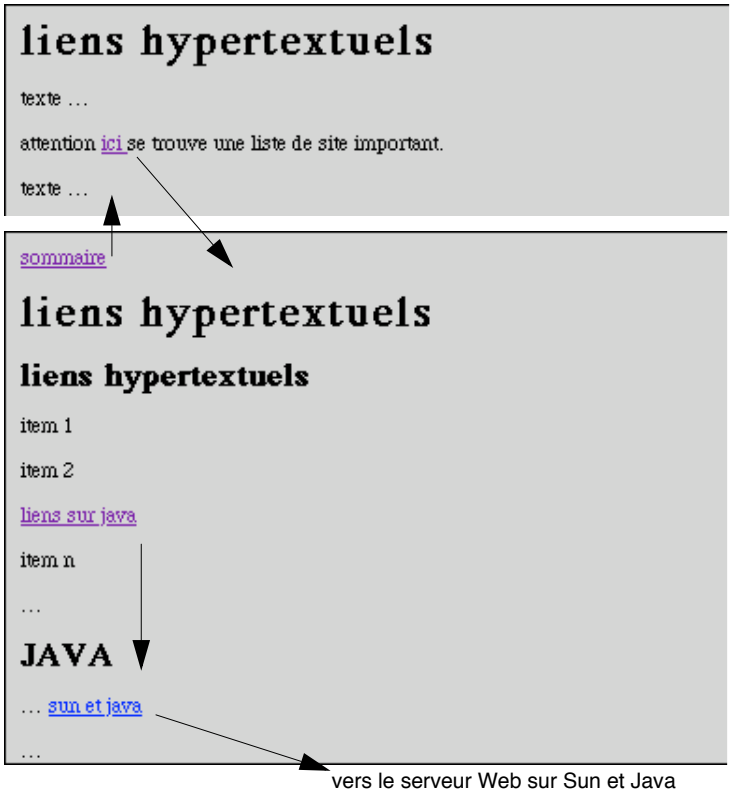


Figure D.6 Liens hypertextuels en HTML (ex6.html)

Délimiteurs de manipulation d'images

Les images sont chargées depuis des fichiers (extension *gif* et *jpg* selon la compression utilisée). Le délimiteur permet de spécifier l'alignement de l'image).

Délimiteur	Signification

`<IMG SRC="URL"`
`ALIGN=top | middle | bottom`
`ALT="..."`
`>`
	Permet d'insérer une image dans le document en indiquant l'URL désignant l'image. ALIGN permet de positionner l'image par rapport à la ligne de texte courante. ALT permet de donner un texte pour les navigateurs n'affichant pas d'image.

Tableau D.7 HTML, délimiteurs de manipulation d'images

Délimiteur	Signification
<IMG SRC="URL" WIDTH=".." HEIGHT=".." VSPACE=".." HSPACE=".." ALIGN="right | left" BORDER="number">	Ces extensions permettent de définir la taille de l'image (HEIGHT, WIDTH), l'espace autour de l'image (VSPACE, HSPACE), l'écoulement du texte autour de l'image (LEFT, RIGHT) et le cadre de l'image (BORDER).
<IMG SRC="URL" ISMAP>	Cette extension permet d'indiquer que l'image est aussi une zone sensible déclenchant des actions.
<BODY BACKGROUND="image.gif">	Permet de choisir un fond d'écran. Il doit être petit en taille et ne pas gêner la lecture.

Tableau D.7 HTML, délimiteurs de manipulation d'images

Exemple d'images (*ex7.html*) :

```
<HTML>
<HEAD>
<TITLE>images ... </TITLE>
</HEAD>

<BODY>

<H1>image</H1>
texte <img src="image7.gif" align=top> haut
<img src="image7.gif" align=middle> milieu
<img src="image7.gif" align=bottom> bas <p>
utilisation d'image pour les ancres<p>
<a href="ex6.html"><img src="flechegauche.gif" align=middle></a>
<a href="ex8.html"><img src="flechedroite.gif" align=middle></a><p>

<img src="bigimg7.jpg" width="100" align=left> taille fixée, image
jpeg, le texte s'écoule autour de l'image à gauche ....
</BODY>

</HTML>
```

L'image suivante (figure D.7) montre le résultat obtenu. Il est aussi possible de définir des ancres avec des images en lieu et place du texte du lien.

Délimiteurs d'images cliquables (cartes)

Les images définissent une zone qui peut être découpée en plusieurs régions auxquelles on associe une action. Le système de coordonnées est identique à celui de Java (voir point 11.1, p. 129). Deux possibilités sont offertes : dans la première, la gestion des régions est confiée au serveur HTTP (on utilise alors l'option ISMAP); dans la deuxième, la carte est gérée par le butineur lui-même.

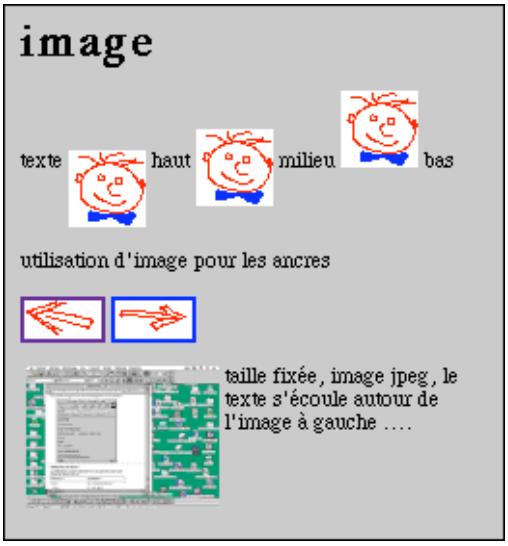


Figure D.7 Insertion d'images en HTML (*ex7.html*)

Délimiteur	Signification
`<IMG SRC="URL" USEMAP=#... >`	Définit la source de la zone sensible et associe le nom de la définition de la carte.
`<MAP NAME="  ">` `...` `</MAP>`	Définit la carte en lui associant plusieurs régions dont les coordonnées sont relatives à l'image associée.
`<AREA SHAPE="rect | poly | circle" COORDS= x1,y1, ..., xn,yn HREF="URL" NOHREF>`	Pour chaque zone sensible, on définit sa forme et les coordonnées de la zone : – rect="x1,y1,x2,y2" – circle="ox,oy,rx,ry" – poly=" x1,y1, ..., xn,yn" – HREF le lien à activer – NOHREF permet de définir des zones insensibles.

Tableau D.8 HTML, délimiteurs d'images cliquables

Exemple d'image cliquable (*ex8.html*) :

```
<HTML>

<HEAD>
<TITLE>images ... </TITLE>
</HEAD>

<BODY>
<H1>image avec zones sensibles</H1>
<! une image de 100 x 100>
```

```
carte d'accès<img src="image8.gif" usemap=#demo>
<MAP NAME="demo">
   <AREA SHAPE="circle" HREF="ex4.html" COORDS=25,25,35,35 >
   <AREA SHAPE="rect"   HREF="ex5.html" COORDS=0,50,50,100 >
   <AREA SHAPE="rect"   HREF="ex6.html" COORDS=50,0,100,50 >
   <AREA SHAPE="rect"   HREF="ex7.html" COORDS=50,50,100,100 >
</MAP>
</BODY>

</HTML>
```

La carte ci-dessous permet d'accéder à quatre liens définis dans les zones.

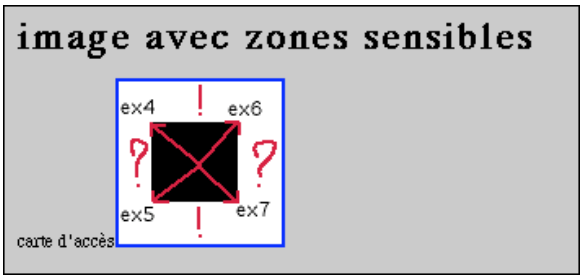


Figure D.8 Images cliquables en HTML (ex8.html)

Délimiteurs de gestion de formulaires

HTML permet de gérer des formulaires comportant des champs, des boutons, des cases à cocher, des menus, des listes déroulantes, etc. Ces formulaires sont obligatoirement associés à une application (un *CGI*) indépendante du serveur HTTP. Dans ce cas, le serveur sert d'intermédiaire entre le client et cette application (par exemple, une base de données). Le contenu des différents champs et l'état des boutons du formulaire sont transmis à l'application.

Délimiteur	Signification
`<FORM METHOD="post | get" ACTION="URL de l'application"> ... </FORM>`	Définit un formulaire, sa méthode d'échange et l'URL de l'application destinataire (généralement une extension .cgi). Entre les deux délimiteurs, on trouve du texte et les objets de formulaire. L'attribut *name* sera le nom du paramètre envoyé à l'application.
`<INPUT TYPE="submit" VALUE="Executer">`	Bouton déclenchant l'exécution du formulaire (envoi des paramètres).
`<INPUT TYPE="reset" VALUE="Effacer">`	Bouton réinitialisant le formulaire (vide les champs).

Tableau D.9 HTML, délimiteurs de gestion de formulaires

Délimiteur	Signification
`<INPUT TYPE="text" SIZE=... NAME="nom_champ" VALUE="val initiale">`	Champ permettant l'entrée de texte sur une ligne de taille définie.
`<INPUT TYPE="password" SIZE=... NAME="nom_champ" VALUE="val initiale">`	Champ permettant l'entrée d'un texte non affiché sur l'écran.
`<INPUT TYPE="radio" NAME="nom_bouton" VALUE="valeur_retournée" CHECKED>`	Affichage d'un bouton radio (actif si CHECKED).
`<INPUT TYPE="checkbox" NAME="nom_du_groupe" VALUE="valeur_retournée" CHECKED>`	Affichage d'une case à cocher dans un groupe. La concaténation des valeurs des cases est retournée.
`<SELECT NAME="nom_liste" SIZE=... MULTIPLE> ... </SELECT>`	Permet d'afficher des menus déroulants si SIZE est absent, ou bien des listes à ascenseurs de la taille SIZE. MULTIPLE indiquant dans ce cas si l'on autorise des choix multiples. Entre les deux délimiteurs, on donne les options à afficher.
`<OPTION> text`	Élément de la liste.
`<TEXTAREA NAME="nom_zone" ROWS=n COLS=n >t exte initial </TEXTAREA>`	Ce délimiteur permet de définir une zone de texte multilignes

Tableau D.9 HTML, délimiteurs de gestion de formulaires

Exemple de formulaire en HTML (*ex9.html*) :

```
<HTML>

<HEAD>
<TITLE>Formulaire ... </TITLE>
</HEAD>
<body background="bg7.gif">
<H1>Formulaires ...</H1>
<hr>
<FORM METHOD="get" ACTION="application.cgi">
Remplissez ce formulaire pour recevoir de la documentation
   <INPUT TYPE="submit" VALUE="Executer">
   <INPUT TYPE="reset" VALUE="Effacer"><p>
nom......<INPUT TYPE="text" SIZE=20 NAME="nom" VALUE="entrez
ici"><br>
e-mail...<INPUT TYPE="text" SIZE=15 NAME="mail" VALUE=""><br>
mot passe<INPUT TYPE="password" SIZE=8 NAME="pwd" VALUE="caché"> si
déjà abonné<p>
```

```
Je m'intéresse à <INPUT TYPE="radio" NAME="int1" VALUE="java"
CHECKED>Java
<INPUT TYPE="radio" NAME="int2" VALUE="java-script" >Java script
<INPUT TYPE="radio" NAME="int3" VALUE="html" >HTML<p>

Je désire l'information par:
<INPUT TYPE="checkbox" NAME="media" VALUE="disk"> disquette
<INPUT TYPE="checkbox" NAME="media" VALUE="email" CHECKED>e-mail
<INPUT TYPE="checkbox" NAME="media" VALUE="cd"> CD-ROM<p>

niveau d'information
<SELECT NAME="niveau">
   <OPTION> débutant
   <OPTION> moyen
   <OPTION> avancé
</SELECT>

système utilisé
<SELECT NAME="syst" SIZE=4 MULTIPLE>
   <OPTION> Mac-PowerPC
   <OPTION> SUN solaris 2.5
   <OPTION> PC Linux
   <OPTION> PC Window 3.xx
   <OPTION> PC Window 95
   <OPTION> PC Window NT
   <OPTION> débutant
</SELECT><p>
Commentaires <TEXTAREA NAME="com" ROWS=3 COLS=40 >placer vos
commentaires ici</TEXTAREA>
<hr>

</FORM>
</BODY>
```

Nous obtenons le formulaire de la figure D.9. On notera l'utilisation d'une image de fond d'écran. Lors de l'activation du bouton *Executer*, la chaîne suivante va être envoyée au serveur HTTP qui activera l'application (cette dernière décodera la liste des paramètres pour la traiter).

```
http://xx.xx.xx.xx/HD500/Desktop%20Folder/JAVA/javadoc/
Exemples%20HTML/
application.cgi?nom=Guyot+Jacques&mail=guyot@acm.org&pwd=cach%8E&i
nt1=java&media=email&media=cd&%D3niveau%D3=moyen&%D3syst%D3=SUN+so
laris+2.5&%D3com%22=placer+vos+commentaires+ici
```

Délimiteurs de gestion de tableaux

HTML (version 3.2) permet de générer des tableaux qui peuvent contenir des textes, des listes, des images, des formulaires et des liens. Les tableaux se définissent rangée par rangée.

Le tableau D.10 présente les délimiteurs de gestion de tableaux.

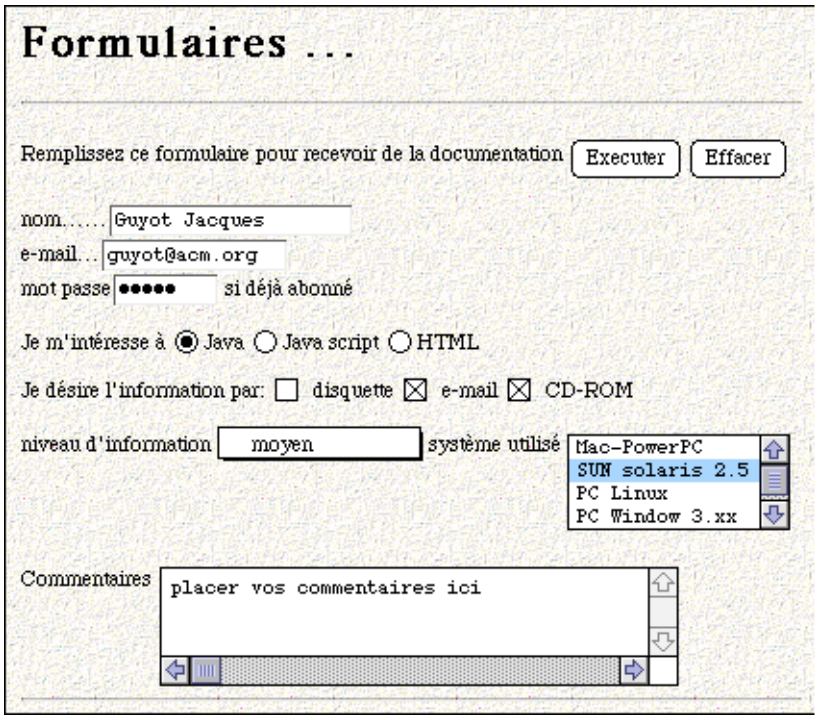


Figure D.9 Un formulaire en HTML (*ex9.html*)

Délimiteur	Signification
`<TABLE BORDER=n CELLPADDING= CELLSPACING= WIDTH=n%> ... </TABLE>`	Définit une table dont la taille (WIDTH) est spécifiée en %. BORDER spécifie la largeur du bord. CELLPADDING spécifie l'espacement entre le contenu de la cellule et son bord. CELLSPACING définit l'espacement entre deux cellules. Les cellules sont définies entre les deux délimiteurs.
`<CAPTION ALIGN=top | bottom> ... titre du tableau </CAPTION>`	Associe un titre au tableau qui se place au-dessus ou en dessous de celui-ci.
`<TR VALIGN= top | middle | bottom ALIGN= left | center | right > ... les cellules </TR>`	Définit une ligne et les attributs par défaut pour cette ligne tels que l'alignement vertical ou horizontal.

Tableau D.10 HTML, délimiteurs de gestion de tableaux

Délimiteur	Signification
<TH VALIGN= top | middle | bottom ALIGN= left | center | bottom COLSPAN=n ROWSPAN=n NOWRAP> ... texte de l'entête	Délimiteur permettant de définir le texte de l'entête du tableau (avec un style prédéfini). L'alignement vertical ou horizontal peut être redéfini. COLSPAN et ROWSPAN permettent d'étendre une cellule sur plusieurs cases. NOWRAP indique que le texte doit être affiché sur une seule ligne.
<TD VALIGN= top | middle | bottom ALIGN= left | center | bottom COLSPAN=n ROWSPAN=n NOWRAP> ... texte d'une cellule	Délimiteur permettant de définir le contenu d'une cellule. L'alignement vertical ou horizontal peut être redéfini. COLSPAN et ROWSPAN permettent d'étendre une cellule sur plusieurs cases. NOWRAP indique que le texte doit être affiché sur une seule ligne.

Tableau D.10 HTML, délimiteurs de gestion de tableaux

Exemple de tableaux (*ex10.html*) :

```
<HTML>
<HEAD><TITLE>Les tableaux ... </TITLE></HEAD>
<BODY background="bg7.gif">
<H1>Tableaux ...</H1>
   <TABLE BORDER=5 CELLPADDING=2 CELLSPACING=3 WIDTH=100%>
    <CAPTION ALIGN=top>
      tableau des carrés
    </CAPTION>
      <TR> <TH> i <TH> i*i </TR>
      <TR> <TD> 1 <TD> 1</TR>
      <TR> <TD> 2 <TD> 4</TR>
      <TR> <TD> 3 <TD> 9</TR>
   </TABLE>
<TABLE BORDER=5 CELLPADDING=2 CELLSPACING=3 WIDTH=100%>
   <CAPTION ALIGN=top>
     tableau de bric et de broc
   </CAPTION>
   <TR VALIGN= middle ALIGN= left  >
     <TH COLSPAN=2 > sur deux case
     <TH> Nom
     <TH> e-mail
   </TR>
   <TR VALIGN= middle ALIGN= left  >
     <TD> A
     <TD COLSPAN=2 ROWSPAN=2 >
       B attention <a href="ex6a.html"> ici </a>
      se trouve une liste de site important
     <TD> C
   </TR>
   <TR VALIGN= middle ALIGN= left  >
     <TD> 1<TD> 2
   </TR>
```

```
    <TR VALIGN= middle ALIGN= left  >
      <TD> a <img src="image7.gif" align=middle><TD> b<TD> c<TD> d
    </TR>
</TABLE>
</BODY>
</HTML>
```

La figure D.10 montre la construction de deux tableaux, l'un très simple, l'autre un peu plus complexe.

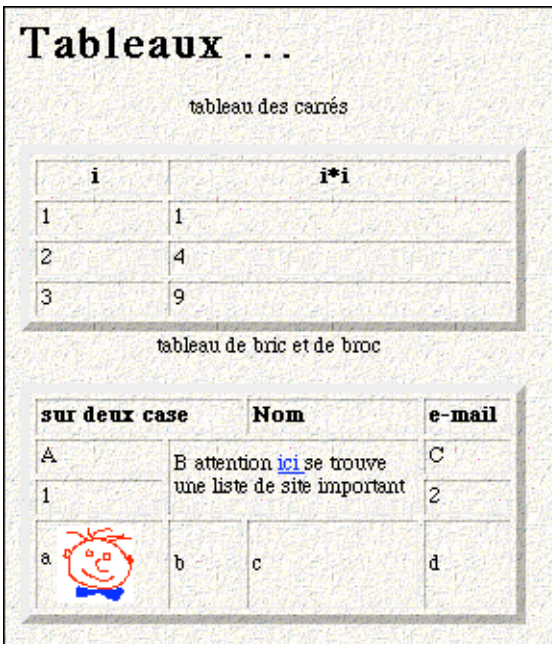


Figure D.10 Tableaux en HTML (*ex10.html*)

Annexe E

JavaScript

JavaScript[1] est un langage développé par les sociétés Sun Microsystems et Netscape. JavaScript ressemble à Java, mais il ne possède ni le typage statique ni la vérification des types. Les expressions et les instructions de contrôle sont pratiquement identiques. JavaScript est entièrement interprété, il n'y a donc pas de phase de compilation comme dans Java.

JavaScript est vu comme un complément de HTML; il permet d'inclure du code directement dans un document HTML, alors que dans Java, on indique uniquement la source du code à exécuter. Le tableau E.1 ci-dessous souligne les principales différences entre les deux langages.

JavaScript	Java
Interprété directement par le client.	Compilé puis chargé depuis le serveur par le client.
Basé sur les objets, utilise et étend des objets préexistants, pas de classes ni d'héritage.	Langage orienté objet, héritage, etc.
Code incorporé dans le document HTML.	Code séparé mais référencé par le document HTML.

Tableau E.1 Principales différences entre JavaScript et Java

1. Un document de référence du langage JavaScript peut être obtenu à l'adresse : http://developer.netscape.com/docs/manuals/index.html.

JavaScript	Java
Pas de déclaration des variables.	Déclaration des variables et typage fort.
Application limitée à la taille du document HTML.	Permet de gérer des projets de taille importante.
Performance limitée due à l'interprétation.	Performance limitée par le processeur du poste client (si l'on utilise la recompilation du byte-code en code natif).
Développement ne nécessitant qu'un butineur interprétant JavaScript.	Développement nécessitant le kit de développement avec le compilateur.

Tableau E.1 Principales différences entre JavaScript et Java

Les objets manipulés par JavaScript sont ceux qui existent dans la page HTLM (les boutons, les champs, etc.). Le programme JavaScript est lié à ces objets afin de traiter les événements qui en sont émis.

Nous pensons qu'un programmeur Java peut sans peine programmer en JavaScript. Il lui faudra s'informer sur les différents gestionnaires d'événements construits autour des composants graphiques de HTML et sur les différentes librairies, *Math* par exemple.

Nous donnons ci-dessous (voir figure E.1) un petit exemple de document HTML incluant du JavaScript pour confirmer notre propos :

- on remarquera l'absence des points-virgules; en effet, une ligne est égale à une instruction;
- le programme est structuré en deux parties : dans la première on décrit la fonction *calculF()* qui vérifie la validité des caractères en entrée, dans la seconde on spécifie le gestionnaire de l'événement *onChange()* attaché au champ *nombreATraiter*.

```
<HEAD>
<TITLE>Exemple en JavaScript</TITLE>
<SCRIPT LANGUAGE="JavaScript">
<!---caché pour les butineurs ne connaissant pas JavaScript
function calculF(s) {
    if (s == "") {
        alert("Entrez un nombre entier dans le champ SVP")
        return true
    }
    for (var i = 0; i < s.length; i++) {
        var ch = s.substring(i, i + 1)
        if (ch < "0" || ch > "9") {
            alert("Un nombre entier sans decimale")
            return true
        }
    }
```

```
    var val = parseInt(s, 10)
    racine=Math.sqrt(val)
    alert("racine carre de("+val+")="+racine)
    return false
}
// fin du code -->
</SCRIPT>
</HEAD>

<BODY>
<h1> Calcul de la racine carrée</h1>
<FORM NAME="ExempleJavaScript">
Entrez un nombre entier puis <RETOUR>:
<INPUT NAME="nombreATraiter"
    onChange="if (calculF(this.value))
              {this.focus();this.select();}">
</FORM>
</BODY>
```

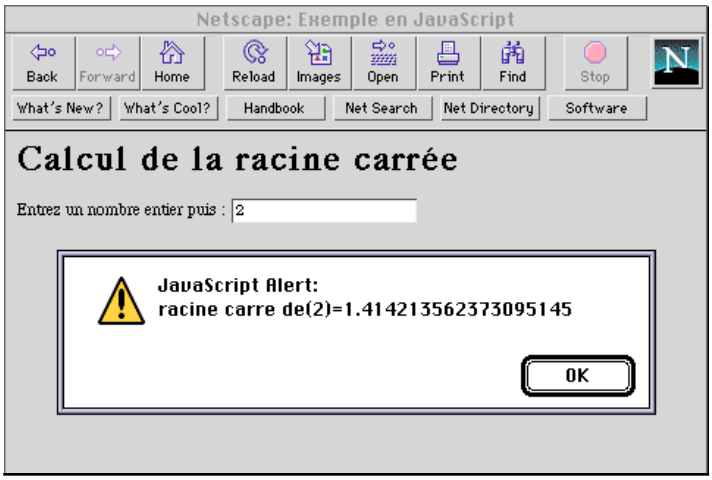


Figure E.1 Exécution d'un programme JavaScript

Glossaire

Adresse électronique

Nom d'utilisateur et de machine permettant à une personne de recevoir du courrier électronique (ex : dupont@moulinsart.be).

Adresse IP

Adresse affectée à toute machine connectée à l'Internet, composée du numéro du réseau et de machine (ex : 193.124.56.99). Les plages d'adresses sont attribuées par une organisation (InterNic Register).

API

(*Application Programming Interface*) Spécification de l'interface de programmation d'un système, ensemble de classes et de méthodes décrivant ses fonctionnalités.

Applet

Une applet est un petit programme écrit en Java devant être invoqué depuis une page HTML.

Archie

Outil permettant la recherche de fichiers sur un ensemble de sites FTP.

Archive

Ensemble de fichiers regroupés et compressés pour la transmission (PC : fichiers *.zip*; Mac, Unix : fichiers *.tar* et *.gz*).

Butineur

Logiciel de navigation sur le Web, permet d'interpréter les documents rédigés selon la syntaxe HTML et le code des programmes Java.

Byte-code

voir Pseudocode.

Champ

(*Field*) Variables et méthodes d'une classe.

Classe

Unité de base d'un programme Java. L'instanciation d'une classe crée un objet ayant le comportement défini dans la classe.

Client

Système (généralement un poste de travail) qui utilise les services d'un autre système (un serveur).

Code natif

Code que le processeur de la machine physique peut directement exécuter, par opposition au pseudocode qui doit être interprété par une machine virtuelle.

Compilateur

Logiciel permettant la compilation.

Compilation

Action transformant le code source d'un programme Java en pseudocode pour la machine virtuelle Java.

Constructeur

Méthode permettant de créer une instance d'une classe.

CORBA

(*Common Object Request Broker Architecture*) Norme qui définit l'interopérabilité et la connection entre des objets répartis.

E-mail

Courrier électronique (adresse électronique, logiciel client-serveur pour accéder aux messages, SMPT).

Exception

Survenance d'un événement qui signale généralement une anomalie à l'exécution. Par extension, code qui traite l'anomalie.

FAQ

(*Frequently Asked Questions*) Document regroupant des questions de base sur un domaine particulier.

FTP

(*File Transfer Protocol*) Protocole de transmission de fichiers entre deux systèmes connectés à Internet.

Garbage collection

voir Ramasse-miettes.

GUI

(*Graphical User Interface*) Ensemble de règles et d'objets qui définissent la construction des interfaces utilisateur.

Héritage

Concept de la programmation objet qui permet de construire une nouvelle classe en étendant le comportement d'une autre, sans en redéfinir toutes les méthodes. La nouvelle (sous-) classe hérite le comportement de sa (super-) classe.

HotJava Le butineur de Sun Microsystems, premier butineur capable d'exécuter du code Java.

HTML (*Hyper Text Markup Language*) Langage permettant de décrire des hyperdocuments (dont le contenu est multimédia et inclut des liens hypertexte).

Instance d'une classe

Objet ayant le comportement de la classe dont il est l'instance.

Interface

Concept du langage Java qui permet de regrouper un ensemble de méthodes définissant un comportement qui sera implanté par une ou plusieurs classes, quelle que soit leur position dans la hiérarchie des classes.

Internet

Historiquement défini comme le protocole de communication TCP/IP, mais devient de plus en plus synonyme de l'ensemble des services et des machines connectés à l'aide de ce protocole.

Interprétation

Action qui, pour chaque pseudocode, exécute les opérations nécessaires (simule la machine virtuelle définie par le pseudocode).

Intranet

Utilisation des techniques liées à l'Internet pour une communauté restreinte (généralement une entreprise).

Java

Langage de programmation orienté objet qui permet d'invoquer des programmes depuis des documents HTML et d'écrire des applications portables.

JavaScript

Langage de programmation orienté objet permettant d'insérer des programmes dans des documents HTML (ici le code est inclus dans le document HTML, à la différence de Java où seule l'invocation est dans le document).

JDBC

(*Java DataBase Connectivity*) Ensemble d'interfaces Java permettant d'interagir avec des systèmes de gestion de bases de données relationnelles.

JDK

(*Java Development Kit*) Kit de développement Java, comprenant un compilateur, un interpréteur, la Core API, un appletviewer, etc.

Machine virtuelle

La machine virtuelle de Java est l'interpréteur du pseudocode. Elle en assure la portabilité et la sécurité.

Méthode

Unité de comportement d'un objet (défini dans sa classe).

Multiprocessus

Programme dont la description fait appel à des activités concurrentes représentant chacune un processus.

Multithread

Voir Multiprocessus.

Navigateur

Voir Butineur.

NCSA

(*National Center for Supercomputer Applications*) L'organisation qui a développé le premier butineur grand public (Mosaic pour Macintosh et PC).

Objet

Possède différentes définitions selon le contexte : 1) instance de classe; 2) par extension, la classe elle-même; 3) par extension,

utilisé comme adjectif : programmation objet, développement objet, base de données objet, etc.

ODMG

(*Object Database Management Group*) Consortium de vendeurs de systèmes de gestion de bases de données orientées objet qui proposent des standards en matière de langage d'interrogation et de format des objets.

Package

En Java, regroupement de classes d'un même domaine fonctionnel.

Pointeur

Donnée dont la valeur est une adresse (généralement un entier pointant sur un emplacement mémoire).

Processus

Il faut distinguer entre : processus lourd, activité au sein d'un système d'exploitation et processus léger, sous-activité d'un processus lourd. Nous faisons toujours référence à des processus légers lorsque nous parlons de processus.

Pseudocode

Méthode pour représenter le jeu d'instructions d'une machine virtuelle ; assure l'indépendance vis-à-vis de la machine physique. Le compilateur Java génère du pseudocode.

Ramasse-miettes

Programme permettant de récupérer la mémoire occupée par des instances d'objets qui ne sont plus référencées et de compacter la mémoire.

RMI

(*Remote Method inovation*) Modèle de distribution d'objets de Java facilitant leur implantation sur des machines différentes.

Serveur

Système qui fournit des services à d'autres systèmes (*voir* Client).

SGBD

(Système de Gestion de Bases de Données) Logiciel capable de gérer une masse d'informations, d'en faciliter l'accès et d'en garantir la cohérence et la persistance.

SGML

Langage de définition pour des langages comme HTML et VRML.

SMTP

(*Simple Mail Transfert Protocol*) Protocole définissant les services de base du courrier électronique.

Sous-classe

Classe dérivée à partir d'une autre classe (super-classe).

Super-classe

Classe étendue par une autre classe (sous-classe).

Surcharge

Concept permettant d'associer plusieurs définitions au même identificateur. En Java, seules les méthodes peuvent être surchargées (mais pas les opérateurs).

Système d'exploitation (SE)

Programme de base gérant les ressources de la machine physique (mémoire, disque, temps CPU). Il est vu comme un ensemble de services pour les applications. MS-DOS, Windows 3.11/95/NT, MacOS, Solaris, Linux, AIX sont autant de systèmes d'exploitation différents. La machine virtuelle de Java rend les applications indépendantes des différents SE.

TCP/IP

(*Transmission Control Protocol/Internet Protocol*) Protocole de transmission entre réseaux. Définition d'Internet.

Télécharger

(*Download*) Transférer un fichier d'un système à un autre (généralement d'un serveur vers un poste utilisateur).

Telnet

Protocole de connexion entre une machine client (en mode émulation de terminal) et une machine serveur.

Thread

Voir Processus.

Unicode

Jeu de caractères défini sur 16 bits (norme ISO 10646).

URL

(*Uniform Resource Locator*) Chaque document du Web a une identité unique sur l'Internet (forme générale : *protocole:/hôte/répertoire/document.html?param*). L'URL permet d'identifier d'autres services Internet, comme FTP par exemple.

Vecteur

(*Array*) Ensemble d'objets désignés par un indice entier.

Web

Voir WWW.

WWW

(*World-Wide Web*) Aussi appelé W3 ou Web. La partie hypertextuelle d'Internet (Butineur, HTML, HTTP, URL).

Références

[1] Bergstra J. A, Heering J., Klint P. (eds.). *Algebraic Specification*, Addison Wesley, 1989.

[2] Booch. G. *Software Components with Ada : Structures, Tools, and Subsystems*, Benjamin/Cummings, 1987.

[3] CERN : Laboratoire européen pour la physique des particules, Genève.
URL : http://www.cern.ch

[4] *JDBC™ : A Java API*, JavaSoft.
URL : http://www.javasoft.com.

[5] Jones C. *Systematic software development using VDM*, Prentice Hall, 1986.

[6] Meyer B. *Conception et programmation par objets : pour du logiciel de qualité*, Interéditions, 1991.

[7] Meyer B. *The many faces of inheritance : A taxonomy of taxonomy*, *IEEE Computer,* Vol. 29, N°. 5, mai 1996.

[8] Network Wizards (statistiques à propos d'Internet).
URL : http://www.nw.com.

[9] ODMG : Object Database Management Group.
URL : http://www.odmg.org.

[10] Spivey J. M.. *The Z notation : A reference manual : based on the work of J. R. Abrial*, Prentice Hall, 1992.

[11] Stroustrup B. *The Design and Evolution of C++*, Addison Wesley, 1994.

[12] Thomas P., Robinson H., Emms J. *Abstract Data Types : Their Specification Representation and Use*, Oxford Applied Mathematics and Computing Science Series, 1988.

[13] The Unicode Consortium.
URL : http://www.unicode.org

[14] The W3 consortium.
URL : http://www.w3.org

Le lecteur intéressé par les aspects sociaux liés à l'introduction de technologies telles que Java trouvera matière à réflexion dans les livres ou documents électroniques suivants :

- *Terminal* : *technologies de l'information, culture et société.* Revue trimestrielle publiée et diffusée par les éditions l'Harmattan.
  http://terminal.ens-cachan.fr/Terminal/serveurText/index.html
- *Le Monde diplomatique* : publie régulièrement des articles sur la mondialisation, les enjeux économiques, politiques et sociaux de l'utilisation des nouvelles techniques.
  http://www.monde-diplomatique.fr/index.html
- *Paul Virilio* : dans *L'art du moteur* et *Cybermonde, la politique du pire*, il met en garde la société contre les logiques sous-jacentes à l'accélération des techniques et l'isolation des individus dans des sociétés très ou trop informatisées.
- *Pierre Lévy* : à travers plusieurs livres dont *L'intelligence collective, pour une anthropologie du cyberspace*, il mène une réflexion sur l'homme, ses outils de cognition et la transformation des sociétés par la technologie.
- *Philippe Queau* : il décrit dans *Le Virtuel* les risques et les bénéfices des techniques basées sur la simulation.
- *Jean Baudrillard* : reconnu comme le théoricien du postmodernisme.
- *Sherry Turkle* : dans *Life on the screen*, elle établit un rapport entre le postmodernisme, l'individu et les techniques liées à Internet.

Liste des applets

Liste des applications

Liste des tableaux

Index

B

C

D

G

H

I

J

Q

R

S

Le CD-ROM du livre

Le CD-ROM qui accompagne de ce livre contient les éléments suivants :

Les sources

Les fichiers sources de toutes les applets et applications du livre (sauf l'applet n° 26, basée sur l'ouvrage *Cent mille milliards de poèmes* de Raymond Queneau, pour des raisons de droits d'auteur).

Les diagrammes syntaxiques

Une applet de dessin des diagrammes syntaxiques pour explorer graphiquement la syntaxe de Java.

La syntaxe commentée

Toute la syntaxe de Java, en format BNF et en diagrammes syntaxiques. Chaque élément syntaxique est commenté. Les règles sont connectées entre elles par des liens hypertextes.

Les concepts

Les concepts du langage Java et de son environnement de programmation expliqués et présentés dans des exemples.

Les notes de cours :

Les copies de transparents utilisés lors de plusieurs cours Java (environ 250 pages, en format PDF).

Ces documents sont également disponibles sur le site Web :
http://cuiwww.unige.ch/java.

L'environnement de développement intégré *CodeWarrior for Java* de Metrowerks (Windows et Macintosh)

Code Warrior est un atelier intégré de développement de logiciel. Il contient les outils de base tels qu'un compilateur, un débogueur, un éditeur de texte, etc. Ces outils sont intégrés, c'est-à-dire qu'ils interagissent entre eux de manière cohérente. Par exemple, lorsque le compilateur détecte une erreur de syntaxe, un simple clic souris sur le texte de l'erreur permet d'afficher, dans l'éditeur, la portion de code concernée. Il en va de même entre le débogueur et l'éditeur, etc. À noter que le compilateur est nettement plus rapide que le « standard » fourni avec le JDK de Sun.

Le cœur de l'environnement de développement est le gestionnaire de projets. Un projet est composé de fichiers sources Java, mais également des fichiers d'images, de son, de code HTML nécessaires à l'application ou à l'applet à développer. La configuration d'un projet peut s'avérer un peu fastidieuse car il faut tout définir : les fichiers àutiliser, le type de fichiers à produire (.class, .jar, etc.), les paramètres de compilation et de déboguage, etc. Cependant, une fois cette phase effectuée, Code Warrior se charge de tout. Lors d'une modification de code source, par exemple, il recompile tout ce qu'il faut, regénère les fichiers de code machine, les archives jar, etc. et relance l'exécution de l'application ou de l'applet.

Il s'agit donc d'un outil professionnel qui peut faire gagner du temps dans le développement et la mise au point de projets. Le développeur Java débutant aura donc tout intérêt à maîtriser rapidement l'utilisation de ce genre d'outils avant de s'attaquer à de plus gros projets.

Un projet porté dans *CodeWarrior for Java*

L'applet du jeu du serpent (voir p. 303) a été portée dans *CodeWarrior for Java*, au sein des environnements Windows NT et Macintosh.

L'éditeur de texte UltraEdit (Windows)

UltraEdit-32 est un éditeur de texte susceptible de remplacer avantageusement Notepad. Il propose de nombreuses fonctionnalités : support de l'accès FTP, vérificateur orthographique multilangue, glisser/déposer de texte, numérotage des lignes, tris, macros, etc. Le clavier et la barre d'outils sont configurables.

Les fichiers peuvent être édités en mode texte ou hexadécimal/binaire, pour des tailles allant jusqu'à 2 Go. Pour les programmeurs, le surlignage de syntaxe est pré-configuré pour les langages HTML, Java, C/C++, VisualBasic et Perl. UltraEdit-32 est disponible en téléchargement (*shareware*, versions anglaise, française et allemande) à http://www.imdcomp.com/

downloads/index.html ou http://www.ultraedit.com/downloads/index.html.

Le compilateur Java peut être lancé à partir d'UltraEdit-32.

Le *Java Development Kit* de Sun