

Thinking in Java, 2nd edition, Revision 11

©2000 by Bruce Eckel

15 : Informatique Distribuée

De tous temps, la programmation des machines interconnectées s'est avérée difficile, complexe, et sujette à erreurs.

Le programmeur devait connaître le réseau de façon détaillée, et parfois même son hardware. Il était généralement nécessaire d'avoir une bonne compréhension des différentes couches du protocole réseau, et il existait dans chacune des bibliothèques de réseau un tas de fonctions, bien entendu différentes, concernant la connexion, l'empaquetage et le dépaquetage des blocs d'information, l'émission et la réception de ces blocs, la correction des erreurs et le dialogue. C'était décourageant.

Toutefois, l'idée de base de l'informatique distribuée n'est pas vraiment complexe, et elle est encapsulée de manière très agréable dans les bibliothèques Java. Il faut pouvoir :

- obtenir quelques informations de cette machine-là et les amener sur cette machine-ci, ou vice versa. Ceci est réalisé au moyen de la programmation réseau de base ;
- se connecter à une base de données, qui peut résider quelque part sur le réseau. Pour cela, on utilise la Connectivité Bases de Données Java : *Java DataBase Connectivity* (JDBC), qui est une encapsulation des détails confus, et spécifiques à la plate-forme, du SQL (*Structured Query Language* - langage structuré d'interrogation de bases de données - utilisé pour de nombreux échanges avec des bases de données) ;
- fournir des services via un serveur Web. C'est le rôle des *servlets* Java et des Pages Serveur Java : *Java Server Pages* (JSP) ;
- exécuter de manière transparente des méthodes appartenant à des objets Java résidant sur des machines distantes, exactement comme si ces objets résidaient sur la machine locale. Pour cela, on utilise l'Invocation de Méthode Distante de Java : *Remote Method Invocation* (RMI) ;
- utiliser du code écrit dans d'autres langages, tournant sous d'autres architectures. C'est l'objet de *Common Object Request Broker Architecture* (CORBA), qui est directement mis en oeuvre par Java ;
- séparer les questions relatives à la connectivité de la logique concernant le résultat cherché, et en particulier les connexions aux bases de données incluant la gestion des transactions et la sécurité. C'est le domaine des *Enterprise JavaBeans* (EJB). Les EJB ne représentent pas réellement une architecture distribuée, mais les applications qui en découlent sont couramment utilisées dans un système client-serveur en réseau ;
- ajouter et enlever, facilement et dynamiquement, des fonctionnalités provenant d'un réseau considéré comme un système local. C'est ce que propose la fonctionnalité Jini de Java.

Ce chapitre a pour but d'introduire succinctement toutes ces fonctionnalités. Il faut noter que chacune représente un vaste sujet, et pourrait faire l'objet d'un livre à elle toute seule, aussi ce chapitre n'a d'autre but que de vous rendre ces concepts familiers, et en aucun cas de faire de vous un expert (toutefois, vous aurez largement de quoi faire avec l'information que vous trouverez ici à propos de la programmation réseau, des *servlets* et des JSP).

La programmation réseau

Une des grandes forces de Java est de pouvoir travailler en réseau sans douleur. Les concepteurs de la bibliothèque réseau de java en ont fait quelque chose d'équivalent à la lecture et l'écriture de fichiers, avec la différence que les « fichiers » résident sur une machine distante et que c'est elle qui décide de ce qu'elle doit faire au sujet des informations que vous demandez ou de celles que vous envoyez. Autant que possible, les menus détails du travail en réseau ont été cachés et sont pris en charge par la JVM et l'installation locale de Java. Le modèle de programmation utilisé est celui d'un fichier ; en réalité, la connexion réseau (Une *socket* - littéralement douille, ou prise, NdT) est encapsulée dans des objets stream, ce qui permet d'utiliser les mêmes appels de méthode que pour les autres objets stream. De plus, les fonctionnalités de multithreading de Java viennent à point lorsqu'on doit traiter plusieurs connexions simultanées.

Cette section introduit le support réseau de Java en utilisant des exemples triviaux.

Identifier une machine

Il semble évident que, pour pouvoir appeler une machine depuis une autre, en ayant la certitude d'être connecté à une machine en particulier, il doit exister quelque chose comme un identifiant unique sur le réseau. Les anciens réseaux se contentaient de fournir des noms de machines uniques sur le réseau local. Mais Java travaille sur l'Internet, et cela nécessite un moyen d'identifier chaque machine de manière unique par rapport à toutes les autres *dans le monde entier*. C'est la raison d'être de l'adresse IP (Internet Protocol - protocole internet, NdT) qui existe sous deux formes :

1. La forme familière

la forme DNS (*Domain Name System*, Système de Nommage des Domaines). Mon nom de domaine est **bruceeckel.com**, et si j'avais dans mon domaine un ordinateur nommé **Opus**, son nom de domaine serait **Opus.bruceeckel.com**. C'est exactement le type de nom que vous utilisez lorsque vous envoyez du courrier à quelqu'un, et il est souvent associé à une adresse World Wide Web.

2. Sinon, on peut utiliser

la forme du quadruplet pointé, c'est à dire quatre nombre séparés par des points, par exemple **123.255.28.120**.

Dans les deux cas, l'adresse IP est représentée en interne comme un nombre sur 32 bits [\[72\]](#) (et donc chaque nombre du quadruplet ne peut excéder 255), et il existe un objet spécial Java pour représenter ce nombre dans l'une des formes décrites ci-dessus en utilisant la méthode de la bibliothèque **java.net : static InetAddress.getByName()**. Le résultat est un objet du type **InetAddress** qu'on peut utiliser pour construire une socket, comme on le verra plus loin.

Pour montrer un exemple simple d'utilisation de **InetAddress.getByName()**, considérons ce qui se passe lorsque vous êtes en communication avec un Fournisseur d'Accès Internet (FAI) - Internet Service Provider (ISP). Chaque fois que vous vous connectez, il vous assigne une adresse IP temporaire. Tant que vous êtes connecté, votre adresse IP est aussi valide que n'importe quelle autre adresse IP sur Internet. Si quelqu'un se connecte à votre machine au moyen de votre adresse IP, alors il peut se connecter à un serveur Web ou FTP qui tournerait sur votre machine. Bien entendu, il faudrait qu'il connaisse votre adresse IP, mais puisqu'une nouvelle vous est assignée à chaque connexion, comment pourrait-il faire ?

Le programme qui suit utilise **InetAddress.getByName()** pour récupérer votre adresse IP. Pour qu'il fonctionne, vous devez connaître le nom de votre ordinateur. Sous Windows 95/98, aller à Paramètres, Panneau de Contrôle, Réseau, et sélectionnez l'onglet Identification. Le Nom d'ordinateur est le nom à utiliser sur la ligne de commande.

```
//: c15:WhoAmI.java
// Affiche votre adresse de réseau lorsque
// vous êtes connectés à Internet.
import java.net.*;

public class WhoAmI {
    public static void main(String[] args)
        throws Exception {
        if(args.length != 1) {
            System.err.println(
                "Usage: WhoAmI MachineName");
            System.exit(1);
        }
        InetAddress a =
            InetAddress.getByName(args[0]);
        System.out.println(a);
    }
} ////:~
```

Supposons que ma machine ait pour nom `peppy`. Une fois connecté au FAI, je lance le programme :

```
java WhoAmI peppy
```

En retour, j'obtiens un message tel que celui-ci (bien entendu, l'adresse est différente à chaque connexion) :

```
peppy/199.190.87.75
```

Si je donne cette adresse à un ami et qu'il existe un serveur Web tournant sur mon ordinateur, il peut s'y connecter en allant à l'URL `http://199.190.87.75` (du moins tant que je reste connecté). Ceci est parfois une manière commode de distribuer de l'information à d'autres personnes, ou encore de tester la configuration d'un site avant de l'installer sur un « vrai » serveur.

Serveurs et clients

La finalité d'un réseau est de permettre à deux machines de se connecter et ainsi de se « parler ». Une fois que les deux machines se sont trouvées l'une l'autre, elles peuvent entamer une agréable conversation bi-directionnelle. Mais qu'est-ce que chacune peut donc rechercher chez l'autre ? Et d'abord, comment trouvent-elles l'autre ? Tout se passe à peu près comme lorsqu'on est perdu dans un parc d'attractions : une des machines doit attendre l'appel de l'autre sans bouger : « Hé, où êtes-vous ? »

La machine qui attend est appelée *serveur*, celle qui cherche *client*. Cette distinction n'est importante que tant que le client cherche à se connecter au serveur. Une fois les machines connectées, la communication se traduit par un processus bi-directionnel et on n'a plus à se préoccuper de savoir qui est le serveur et qui est le client.

Le rôle du serveur est donc d'être en attente d'une demande de connexion, ceci est réalisé par l'objet serveur qu'on crée dans ce but. Le rôle du client est de tenter d'établir une connexion avec un serveur, ceci est réalisé avec un objet client qu'on crée pour cela. Une fois la connexion établie, aussi bien du côté serveur que du côté client, elle se transforme magiquement en un objet flux d'E/S, et dès lors on peut traiter cette connexion comme une lecture ou une écriture dans un fichier. Ainsi, une fois la connexion établie, il ne reste qu'à utiliser les commandes d'E/S familières vues au Chapitre 11. C'est un des aspects agréables des fonctions de Java en réseau.

Tester les programmes hors réseau

Pour toutes sortes de raisons, il se peut qu'on ne dispose pas de machines client et serveur, pas plus que d'un réseau pour tester nos programmes. Par exemple lorsqu'on réalise des exercices dans une situation d'apprentissage, ou bien qu'on écrive des programmes qui ne sont pas encore suffisamment stables pour être mis sur le réseau. Les créateurs du protocole internet (IP) ont bien appréhendé cette question, et ont créé une adresse spéciale appelée **localhost** qui représente une adresse IP en « boucle locale » permettant d'effectuer des tests en se passant de la présence d'un réseau. Voyez ci-dessous la manière générique de réaliser cette adresse en Java :

```
InetAddress addr = InetAddress.getByName(null) ;
```

En utilisant **getByName()** avec un argument **null**, cette fonction utilise par défaut **localhost**. **InetAddress** est utilisée pour désigner une machine particulière, et on doit l'initialiser avant d'aller plus loin. On ne peut pas manipuler le contenu d'une **InetAddress** (mais il est possible de l'imprimer, comme on va le voir dans l'exemple suivant). L'unique manière de créer une **InetAddress** passe par l'une des méthodes membre **static** surchargées de cette classe : **getByName()** (habituellement utilisée), **getAllByName()** ou **getLocalHost()**.

Une autre manière de réaliser l'adresse de la boucle locale est d'utiliser la chaîne **localhost**:

```
InetAddress.getByName( "localhost" ) ;
```

(en supposant que **localhost** est décrit dans la table des hôtes de la machine), ou encore en se servant de la forme « quadruplet pointé » pour désigner l'adresse IP réservée à la boucle locale :

```
InetAddress.getByName( "127.0.0.1" ) ;
```

Les trois formes aboutissent au même résultat.

Les Ports : un emplacement unique dans la machine

Une adresse IP n'est pas suffisante pour identifier un serveur, en effet plusieurs serveurs peuvent coexister sur une même machine. Chaque machine IP contient aussi des *ports*, et lorsqu'on installe un client ou un serveur il est nécessaire de choisir un port convenant aussi bien à la connexion du client qu'à celle du serveur ; si vous donnez rendez-vous à quelqu'un, l'adresse IP représentera le quartier et le port sera le nom du bar.

Le port n'est pas un emplacement physique dans la machine, mais une abstraction de programmation (surtout pour des raisons de comptabilité). Le programme client sait comment se connecter à la machine via son adresse IP, mais comment se connectera-t-il au service désiré (potentiellement, l'un des nombreux services de cette machine) ? C'est pourquoi les numéros de port représentent un deuxième niveau d'adressage. L'idée sous-jacente est que si on s'adresse à un port particulier, en fait on effectue une demande pour le service associé à ce numéro de port. Un exemple simple de service est l'heure du jour. Typiquement, chaque service est associé à un numéro de port unique sur une machine serveur donnée. Il est de la responsabilité du client de connaître à l'avance quel est le numéro de port associé à un service donné.

Les services système réservent les numéros de ports 1 à 1024, il ne faut donc pas les utiliser, pas davantage que d'autres numéros de ports dont on saurait qu'ils sont utilisés. Le premier nombre choisi pour les exemples de ce livre est le port 8080 (en souvenir de la vénérable vieille puce 8 bits Intel 8080 de mon premier ordinateur, une machine CP/M).

Les sockets

Une *socket* est une abstraction de programmation représentant les extrémités d'une connexion entre deux machines. Pour chaque connexion donnée, il existe une socket sur chaque machine, on peut imaginer un câble virtuel reliant les deux machines, chaque extrémité enfichée dans une socket. Bien entendu, le hardware sous-jacent ainsi que la manière dont les machines sont connectées ne nous intéressent pas. L'essentiel de l'abstraction est qu'on n'a pas à connaître plus que ce qu'il est nécessaire.

En Java, on crée un objet **Socket** pour établir une connexion vers une autre machine, puis on crée un **InputStream** et un **OutputStream** (ou, avec les convertisseurs appropriés, un **Reader** et un **Writer**) à partir de ce **Socket**, afin de traiter la connexion en tant qu'objet flux d'E/S. Il existe deux classes **Socket** basées sur les flux : **ServerSocket** utilisé par un serveur pour écouter les connexions entrantes et **Socket** utilisé par un client afin d'initialiser une connexion. Lorsqu'un client réalise une connexion socket, **ServerSocket** retourne (via la méthode **accept()**) un **Socket** correspondant permettant les communications du côté serveur. À ce moment-là, on a réellement établi une connexion **Socket** à **Socket** et on peut traiter les deux extrémités de la même manière, car elles *sont* alors identiques. On utilise alors les méthodes **getInputStream()** et **getOutputStream()** pour réaliser les objets **InputStream** et **OutputStream** correspondants à partir de chaque **Socket**. À leur tour ils peuvent être encapsulés dans des classes buffers ou de formatage tout comme n'importe quel objet flux décrit au Chapitre 11.

La construction du mot **ServerSocket** est un exemple supplémentaire de la confusion du plan de nommage des bibliothèques Java. **ServerSocket** aurait dû s'appeler « **ServerConnector** » ou bien n'importe quoi qui n'utilise pas le mot **Socket**. Il faut aussi penser que **ServerSocket** et **Socket** héritent tous deux de la même classe de base. Naturellement, les deux classes ont plusieurs méthodes communes, mais pas suffisamment pour qu'elles aient une même classe de base. En fait, le rôle de **ServerSocket** est d'attendre qu'une autre machine se connecte, puis à ce moment-là de renvoyer un **Socket** réel. C'est pourquoi **ServerSocket** semble mal-nommé, puisque son rôle n'est pas d'être une socket mais plus exactement de créer un objet **Socket** lorsque quelqu'un se connecte.

Cependant, un **ServerSocket** crée physiquement un serveur ou, si l'on préfère, une « prise » à l'écoute sur la machine hôte. Cette « prise » est à l'écoute des connexions entrantes et renvoie une « prise » établie (les points terminaux et distants sont définis) via la méthode **accept()**. La confusion vient du fait que ces deux « prises » (celle qui écoute et celle qui représente la communication établie) sont associées à la même « prise » serveur. La « prise » qui écoute accepte uniquement les demandes de nouvelles connexions, jamais les paquets de données. Ainsi, même si **ServerSocket** n'a pas beaucoup de sens en programmation, il en a physiquement.

Lorsqu'on crée un **ServerSocket**, on ne lui assigne qu'un numéro de port. Il n'est pas nécessaire de lui assigner une adresse IP parce qu'il réside toujours sur la machine qu'il représente. En revanche, lorsqu'on crée un **Socket**, il faut lui fournir l'adresse IP et le numéro de port sur lequel on essaie de se connecter (toutefois, le **Socket** résultant de la méthode **ServerSocket.accept()** contient toujours cette information).

Un serveur et un client vraiment simples

Cet exemple montre l'utilisation minimale d'un serveur et d'un client utilisant des sockets. Le serveur se contente d'attendre une demande de connexion, puis se sert du **Socket** résultant de cette connexion pour créer un **InputStream** et un **OutputStream**. Ces derniers sont convertis en **Reader** et **Writer**, puis encapsulés dans un **BufferedReader** et un **PrintWriter**. À partir de là, tout ce que lit le **BufferedReader** est renvoyé en écho au **PrintWriter** jusqu'à ce qu'il reconnaisse la ligne **END**, et dans ce cas il clôt la connexion.

Le client établit une connexion avec le serveur, puis crée un **OutputStream** et réalise le même type d'encapsulation que le serveur. Les lignes de texte sont envoyées vers le **PrintWriter** résultant. Le client crée

également un **InputStream** (ici aussi, avec la conversion et l'encapsulation appropriées) afin d'écouter le serveur (c'est à dire, dans ce cas, simplement les mots renvoyés en écho).

Le serveur ainsi que le client utilisent le même numéro de port, et le client se sert de l'adresse de boucle locale pour se connecter au serveur sur la même machine, ce qui évite de tester en grandeur réelle sur un réseau (pour certaines configurations, on peut être amené à se *connecter physiquement* à un réseau afin que le programme fonctionne, même si on ne *communiqué* pas sur ce réseau.)

Voici le serveur :

```
//: cl5:JabberServer.java
// Serveur simplifié dont le rôle se limite à
// renvoyer en écho tout ce que le client envoie.
import java.io.*;
import java.net.*;

public class JabberServer {
    // Choisir un port hors de la plage 1-1024:
    public static final int PORT = 8080;
    public static void main(String[] args)
        throws IOException {
        ServerSocket s = new ServerSocket(PORT);
        System.out.println("Started: " + s);
        try {
            // Le programme stoppe ici et attend
            // une demande de connexion:
            Socket socket = s.accept();
            try {
                System.out.println(
                    "Connection accepted: " + socket);
                BufferedReader in =
                    new BufferedReader(
                        new InputStreamReader(
                            socket.getInputStream()));
                // Le tampon de sortie est vidé
                // automatiquement par PrintWriter:
                PrintWriter out =
                    new PrintWriter(
                        new BufferedWriter(
                            new OutputStreamWriter(
                                socket.getOutputStream()), true));
                while (true) {
                    String str = in.readLine();
                    if (str.equals("END")) break;
                    System.out.println("Echoing: " + str);
                    out.println(str);
                }
                // Toujours fermer les deux sockets...
            } finally {
                System.out.println("closing...");
                socket.close();
            }
        } finally {
            s.close();
        }
    }
}
```

```

    }
} ///:~

```

On remarque que **ServerSocket** ne nécessite qu'un numéro de port, et non une adresse IP (puisque'il tourne sur *cette* machine !). Lorsqu'on appelle **accept()** la méthode *bloque* jusqu'à ce qu'un client tente de se connecter. Cela signifie qu'elle est en attente d'une demande de connexion, mais elle ne bloque pas les autres processus (voir le Chapitre 14). Une fois la connexion établie, **accept()** renvoie un objet **Socket** qui représente cette connexion.

Nettoyer les sockets est une responsabilité, elle est soigneusement traitée ici. Si le constructeur de **ServerSocket** échoue, le programme se termine simplement (il faut remarquer que nous supposons que le constructeur de **ServerSocket** ne laisse traîner aucune socket de réseau s'il échoue). Dans ce cas, la méthode **main()** renverrait une **IOException** et par conséquent un bloc **try** n'est pas nécessaire. Si le constructeur du **ServerSocket** se termine avec succès, alors tous les autres appels de méthode doivent être inclus dans un bloc **try-finally** afin d'assurer que le **ServerSocket** sera fermé correctement quelle que soit la manière dont le bloc se termine.

La même logique est utilisée pour le **Socket** renvoyé par **accept()**. Si **accept()** échoue, nous devons supposer que le **Socket** n'existe pas et qu'il ne monopolise aucune ressource, et donc qu'il n'est pas besoin de le nettoyer. Au contraire, si **accept()** se termine avec succès, les instructions qui suivent doivent se trouver dans un bloc **try-finally** pour que **Socket** soit toujours nettoyé si l'une d'entre elles échoue. Il faut porter beaucoup d'attention à cela car les sockets sont des ressources importantes qui ne résident pas en mémoire, et on ne doit pas oublier de les fermer (car il n'existe pas en Java de destructeur qui le ferait pour nous).

Le **ServerSocket** et le **Socket** fournis par **accept()** sont imprimés sur **System.out**. Leur méthode **toString()** est donc appelée automatiquement. Voici le résultat :

```

ServerSocket[addr=0.0.0.0,PORT=0,localport=8080]
Socket[addr=127.0.0.1,PORT=1077,localport=8080]

```

En raccourci, on peut voir comment cela « colle » avec ce que fait le client.

Le reste du programme consiste seulement à ouvrir des fichiers pour lire et écrire, sauf que **InputStream** et **OutputStream** sont créés à partir de l'objet **Socket**. Les deux objets **InputStream** et **OutputStream** sont convertis en objets **Reader** et **Writer** au moyen des classes de conversion **InputStreamReader** et **OutputStreamWriter**, respectivement. On aurait pu travailler directement avec les classes Java 1.0 **InputStream** et **OutputStream**, mais pour la sortie il y a un avantage certain à utiliser l'approche **Writer**. C'est évident avec **PrintWriter**, qui possède un constructeur surchargé prenant en compte un deuxième argument, un flag **boolean** indiquant que le tampon de sortie doit être automatiquement vidé après chaque **println()** (mais *non* après les instructions **print()**). Chaque fois qu'on écrit sur **out**, son buffer doit être vidé afin que l'information parte sur le réseau. Le vidage du tampon est important dans cet exemple particulier car aussi bien le serveur que le client attendent de l'autre une ligne complète avant de la traiter. Si le tampon n'est pas vidé à chaque ligne, l'information ne circule pas sur le réseau tant que le buffer n'est pas plein, ce qui occasionnerait de nombreux problèmes dans cet exemple.

Lorsqu'on écrit des programmes réseau, il faut être très attentif à l'utilisation du vidage automatique de buffer. Chaque fois que l'on vide un buffer, un paquet est créé et envoyé. Dans notre exemple, c'est exactement ce que nous recherchons, puisque si le paquet contenant la ligne n'est pas envoyé, l'échange entre serveur et client sera stoppé. Dit d'une autre manière, la fin de ligne est la fin du message. Mais dans de nombreux cas, les messages ne sont pas découpés en lignes et il sera plus efficace de ne pas utiliser le vidage automatique de buffer et à la place de laisser le gestionnaire du buffer décider du moment pour construire et envoyer un paquet. De cette manière, les paquets seront plus importants et le processus plus rapide.

Remarquons que, comme tous les flux qu'on peut ouvrir, ceux-ci sont tamponnés. À la fin de ce chapitre, vous trouverez un exercice montrant ce qui se passe lorsqu'on ne tamponne pas les flux (tout est ralenti).

La boucle **while** infinie lit les lignes depuis le **BufferedReader in** et écrit sur **System.out** et **PrintWriter out**. Remarquons que **in** et **out** peuvent être n'importe quel flux, ils sont simplement connectés au réseau.

Lorsque le client envoie une ligne contenant juste le mot END, la boucle est interrompue et le programme ferme le **Socket**.

Et voici le client :

```
///  
// c15:JabberClient.java  
// Client simplifié se contentant d'envoyer  
// des lignes au serveur et de lire les lignes  
// que le serveur lui envoie.  
import java.net.*;  
import java.io.*;  
  
public class JabberClient {  
    public static void main(String[] args)  
        throws IOException {  
        // Appeler getByName() avec un argument null revient  
        // à utiliser une adresse IP spéciale "Boucle Locale"  
        // pour faire des tests réseau sur une seule machine.  
  
        InetAddress addr =  
            InetAddress.getByName(null);  
        // Il est également possible d'utiliser  
        // l'adresse ou le nom:  
        // InetAddress addr =  
        //     InetAddress.getByName("127.0.0.1");  
        // InetAddress addr =  
        //     InetAddress.getByName("localhost");  
        System.out.println("addr = " + addr);  
        Socket socket =  
            new Socket(addr, JabberServer.PORT);  
        // Le code doit être inclus dans un bloc  
        // try-finally afin de garantir  
        // que socket sera fermé:  
        try {  
            System.out.println("socket = " + socket);  
            BufferedReader in =  
                new BufferedReader(  
                    new InputStreamReader(  
                        socket.getInputStream()));  
            // Le tampon de sortie est automatiquement  
            // vidé par PrintWriter:  
            PrintWriter out =  
                new PrintWriter(  
                    new BufferedWriter(  
                        new OutputStreamWriter(  
                            socket.getOutputStream()), true);  
            for(int i = 0; i < 10; i++) {  
                out.println("howdy " + i);  
                String str = in.readLine();
```



```

        System.out.println(str);
    }
    out.println("END");
} finally {
    System.out.println("closing...");
    socket.close();
}
}
} ///:~

```

La méthode **main()** montre qu'il existe trois manières de produire l'**InetAddress** de l'adresse IP de la boucle locale : avec **null**, **localhost**, ou bien l'adresse réservée et explicite **127.0.0.1**. Bien entendu, si l'on désire se connecter à une machine du réseau, il suffit d'y substituer son adresse. Lorsque **InetAddress addr** est imprimée (via l'appel automatique de sa méthode **toString()**), voici le résultat :

```
localhost/127.0.0.1
```

Lorsque **getByName()** a été appelée avec un argument **null**, elle a cherché par défaut **localhost**, ce qui a fourni l'adresse spéciale **127.0.0.1**.

Remarquons que le **Socket** nommé **socket** est créé avec **InetAddress** ainsi que le numéro de port. Pour comprendre ce qui se passe lorsqu'on imprime un de ces objets **Socket**, il faut se souvenir qu'une connexion Internet est déterminée de manière unique à partir de quatre données : **clientHost**, **clientPortNumber**, **serverHost**, et **serverPortNumber**. Lorsque le serveur démarre, il prend en compte le port qui lui est assigné (8080) sur la machine locale (127.0.0.1). Lorsque le client démarre, le premier port suivant disponible sur sa machine lui est alloué, 1077 dans ce cas, qui se trouve sur la même machine (127.0.0.1) que le serveur. Maintenant, pour que les données circulent entre le client et le serveur, chacun doit savoir où les envoyer. En conséquence, pendant la connexion au serveur connu, le client envoie une adresse de retour afin que le serveur sache où envoyer ses données. Ce qu'on peut voir dans la sortie de l'exemple côté serveur :

```
Socket[addr=127.0.0.1,port=1077,localport=8080]
```

Cela signifie que le serveur vient d'accepter une demande de connexion provenant de 127.0.0.1 sur le port 1077 alors qu'il est à l'écoute sur le port local (8080). Du côté client :

```
Socket[addr=localhost/127.0.0.1,PORT=8080,localport=1077]
```

ce qui signifie que le client vient d'établir une connexion à 127.0.0.1 sur le port 8080 en utilisant le port local 1077.

Il faut remarquer que chaque fois qu'on relance le client, le numéro du port local est incrémenté. Il commence à 1025 (un après le bloc de ports réservé) et ne cesse d'augmenter jusqu'à ce qu'on reboote la machine, auquel cas il recommence à 1025 (sur les machines UNIX, lorsque la limite supérieure de la plage accordée aux sockets est atteinte, on recommence avec la plus petite valeur possible).

L'objet **Socket** créé, le processus consistant à en faire un **BufferedReader** puis un **PrintWriter** est le même que pour le serveur (encore une fois, dans les deux cas on commence par un **Socket**). Ici, le client entame la conversation en envoyant la chaîne « howdy » suivie d'un nombre. Remarquons que le buffer doit être vidé à nouveau (ce qui est automatique via le deuxième argument du constructeur de **PrintWriter**). Si le buffer n'est pas vidé, la conversation est complètement suspendue parce que le « howdy » initial ne sera jamais envoyé (le

buffer n'est pas assez rempli pour que l'envoi se fasse automatiquement). Chaque ligne renvoyée par le serveur est écrite sur **System.out** pour vérifier que tout fonctionne correctement. Pour arrêter l'échange, le client envoie le mot connu **END**. Si le client ne se manifeste plus, le serveur lance une exception.

Remarquons qu'ici aussi le même soin est apporté pour assurer que la ressource réseau que représente le **Socket** est relâchée correctement, au moyen d'un bloc **try-finally**.

Les sockets fournissent une connexion dédiée qui persiste jusqu'à ce qu'elle soit explicitement déconnectée (la connexion dédiée peut encore être rompue de manière non explicite si l'un des deux côtés ou un lien intermédiaire de la connexion se plante). La conséquence est que les deux parties sont verrouillées en communication et que la connexion est constamment ouverte. Cela peut paraître une approche logique du travail en réseau, en fait on surcharge le réseau. Plus loin dans ce chapitre on verra une approche différente du travail en réseau, dans laquelle les connexions seront temporaires.

Servir des clients multiples

Le programme **JabberServer** fonctionne, mais ne peut traiter qu'un client à la fois. Dans un serveur typique, on désire traiter plusieurs clients en même temps. La réponse est le multithreading, et dans les langages ne supportant pas cette fonctionnalité cela entraîne toutes sortes de complications. Dans le Chapitre 14 nous avons vu que le multithreading de Java est aussi simple que possible, si l'on considère le multithreading comme un sujet quelque peu complexe. Parce que gérer des threads est relativement simple en Java, écrire un serveur prenant en compte de multiples clients est relativement facile.

L'idée de base est de construire dans le serveur un seul **ServerSocket** et d'appeler **accept()** pour attendre une nouvelle connexion. Au retour d'**accept()**, on crée un nouveau thread utilisant le **Socket** résultant, thread dont le travail est de servir ce client particulier. Puis on appelle à nouveau **accept()** pour attendre un nouveau client.

Dans le code serveur suivant, on remarquera qu'il ressemble à l'exemple **JabberServer.java** sauf que toutes les opérations destinées à servir un client particulier ont été déplacées dans une classe thread séparée :

```

//: c15:MultiJabberServer.java
// Un serveur utilisant le multithreading
// pour traiter un nombre quelconque de clients.
import java.io.*;
import java.net.*;

class ServeOneJabber extends Thread {
    private Socket socket;
    private BufferedReader in;
    private PrintWriter out;
    public ServeOneJabber(Socket s)
        throws IOException {
        socket = s;
        in =
            new BufferedReader(
                new InputStreamReader(
                    socket.getInputStream()));
        // Autoriser l'auto-vidage:
        out =
            new PrintWriter(
                new BufferedWriter(
                    new OutputStreamWriter(
                        socket.getOutputStream())), true);
    }
}

```

```

        // Si l'un des appels ci-dessus résulte en une
        // exception, l'appelant a la responsabilité
        // de fermer socket. Sinon le thread
        // s'en chargera.
        start(); // Appelle run()
    }
    public void run() {
        try {
            while (true) {
                String str = in.readLine();
                if (str.equals("END")) break;
                System.out.println("Echoing: " + str);
                out.println(str);
            }
            System.out.println("closing...");
        } catch (IOException e) {
            System.err.println("IO Exception");
        } finally {
            try {
                socket.close();
            } catch (IOException e) {
                System.err.println("Socket not closed");
            }
        }
    }
}

public class MultiJabberServer {
    static final int PORT = 8080;
    public static void main(String[] args)
        throws IOException {
        ServerSocket s = new ServerSocket(PORT);
        System.out.println("Server Started");
        try {
            while(true) {
                // On attend ici jusqu'à avoir
                // une demande de connexion:
                Socket socket = s.accept();
                try {
                    new ServeOneJabber(socket);
                } catch (IOException e) {
                    // En cas d'échec, fermer l'objet socket,
                    // sinon le thread le fermera:
                    socket.close();
                }
            }
        } finally {
            s.close();
        }
    }
} //::~~

```

Chaque fois qu'un nouveau client se connecte, le thread **ServeOneJabber** prend l'objet **Socket** produit par **accept()** dans **main()**. Puis, comme auparavant, il crée un **BufferedReader** et un objet **PrintWriter** (avec auto-vidage du buffer) à partir du **Socket**. Finalement, il appelle la méthode spéciale **start()** de la classe

Thread qui initialise le thread puis appelle **run()**. On réalise ainsi le même genre de traitement que dans l'exemple précédent : lire quelque chose sur la socket et le renvoyer en écho jusqu'à l'arrivée du signal spécial **END**.

À nouveau, il nous faut penser à notre responsabilité en ce qui concerne la ressource socket. Dans ce cas, la socket est créée hors de **ServeOneJabber** et donc la responsabilité doit être partagée. Si le constructeur de **ServeOneJabber** échoue, il suffit qu'il lance une exception vers l'appelant, qui nettoiera alors le thread. Mais dans le cas contraire, l'objet **ServeOneJabber** a la responsabilité de nettoyer le thread, dans sa méthode **run()**.

Remarquons la simplicité de **MultiJabberServer**. Comme auparavant, on crée un **ServerSocket**, et on appelle **accept()** pour autoriser une nouvelle connexion. Mais cette fois, la valeur de retour de **accept()** (un **Socket**) est passée au constructeur de **ServeOneJabber**, qui crée un nouveau thread afin de prendre en compte cette connexion. La connexion terminée, le thread se termine tout simplement.

Si la création du **ServerSocket** échoue, à nouveau une exception est lancée par **main()**. En cas de succès, le bloc **try-finally** extérieur garantit le nettoyage. Le bloc **try-catch** intérieur protège d'une défaillance du constructeur **ServeOneJabber** ; en cas de succès, le thread **ServeOneJabber** fermera la socket associée.

Afin de tester que le serveur prend réellement en compte plusieurs clients, le programme suivant crée beaucoup de clients (sous la forme de threads) et se connecte au même serveur. Le nombre maximum de threads autorisés est fixé par la constante **final int MAX_THREADS**.

```

//: c15:MultiJabberClient.java
// Client destiné à tester MultiJabberServer
// en lançant des clients multiple.
import java.net.*;
import java.io.*;

class JabberClientThread extends Thread {
    private Socket socket;
    private BufferedReader in;
    private PrintWriter out;
    private static int counter = 0;
    private int id = counter++;
    private static int threadcount = 0;
    public static int threadCount() {
        return threadcount;
    }
    public JabberClientThread(InetAddress addr) {
        System.out.println("Making client " + id);
        threadcount++;
        try {
            socket =
                new Socket(addr, MultiJabberServer.PORT);
        } catch (IOException e) {
            System.err.println("Socket failed");
            // Si la création de socket échoue,
            // il n'y a rien à nettoyer.
        }
        try {
            in =
                new BufferedReader(
                    new InputStreamReader(
                        socket.getInputStream()));

```

```

        // Autoriser l'auto-vidage du tampon:
        out =
            new PrintWriter(
                new BufferedWriter(
                    new OutputStreamWriter(
                        socket.getOutputStream()), true);
        start();
    } catch(IOException e) {
        // socket doit être fermé sur n'importe quelle
        // erreur autre que celle de son constructeur:
        try {
            socket.close();
        } catch(IOException e2) {
            System.err.println("Socket not closed");
        }
    }
    // Sinon socket doit être fermé par
    // la méthode run() du thread.
}
public void run() {
    try {
        for(int i = 0; i < 25; i++) {
            out.println("Client " + id + ": " + i);
            String str = in.readLine();
            System.out.println(str);
        }
        out.println("END");
    } catch(IOException e) {
        System.err.println("IO Exception");
    } finally {
        // Toujours fermer:
        try {
            socket.close();
        } catch(IOException e) {
            System.err.println("Socket not closed");
        }
        threadcount--; // Fin de ce thread
    }
}
}

public class MultiJabberClient {
    static final int MAX_THREADS = 40;
    public static void main(String[] args)
        throws IOException, InterruptedException {
        InetAddress addr =
            InetAddress.getByName(null);
        while(true) {
            if(JabberClientThread.threadCount()
                < MAX_THREADS)
                new JabberClientThread(addr);
            Thread.currentThread().sleep(100);
        }
    }
} ///::~~

```

Le constructeur de **JabberClientThread** prend une **InetAddress** afin d'ouvrir un **Socket**. Vous devinez sans doute la suite : **Socket** est utilisé pour créer un objet du type **Reader** et/ou **Writer** (ou bien **InputStream** et/ou **OutputStream**), ce qui est la seule manière d'utiliser un **Socket** (bien entendu on peut écrire une classe ou deux afin d'automatiser ce travail au lieu de tout retaper à chaque fois, si cela devient trop pénible). À nouveau, **start()** initialise le thread puis appelle **run()**. A ce point, des messages sont envoyés au serveur et l'information en retour du serveur est affichée en écho sur l'écran. Toutefois, la durée de vie du thread est limitée et finalement il se termine. Remarquons que la socket est nettoyée si le constructeur échoue après que la socket soit créée mais avant que l'exécution du constructeur soit terminée. Dans les autres cas la responsabilité d'appeler la méthode **close()** de socket est déléguée à la méthode **run()**.

La variable **threadCount** garde la trace du nombre de **JabberClientThread** existant actuellement. Elle est incrémentée dans le constructeur et décrémentée lorsque **run()** termine (ce qui signifie que le thread est en train de se terminer). Dans **MultiJabberClient.main()** le nombre de threads est testé, on arrête d'en créer s'il y en a trop. Dans ce cas la méthode s'endort. De cette manière, certains threads peuvent se terminer et de nouveaux pourront être créés. Vous pouvez faire l'expérience, en changeant la valeur de **MAX_THREADS**, afin de savoir à partir de quel nombre de connexions votre système commence à avoir des problèmes.

Les Datagrammes

Les exemples que nous venons de voir utilisent le Protocole de Contrôle de Transmission *Transmission Control Protocol* (TCP, connu également sous le nom de *stream-based sockets* - sockets basés sur les flux, NdT), qui est conçu pour une sécurité maximale et qui garantit que les données ne seront pas perdues. Il permet la retransmission des données perdues, il fournit plusieurs chemins au travers des différents routeurs au cas où l'un d'eux tombe en panne, enfin les octets sont délivrés dans l'ordre où ils ont été émis. Ces contrôles et cette sécurité ont un prix : TCP a un overhead élevé.

Il existe un deuxième protocole, appelé *User Datagram Protocol* (UDP), qui ne garantit pas que les paquets seront acheminés ni qu'ils arriveront dans l'ordre d'émission. On l'appelle protocole peu sûr (TCP est un protocole sûr), ce qui n'est pas très vendeur, mais il peut être très utile car il est beaucoup plus rapide. Il existe des applications, comme la transmission d'un signal audio, pour lesquelles il n'est pas critique de perdre quelques paquets çà et là mais où en revanche la vitesse est vitale. Autre exemple, considérons un serveur de date et heure, pour lequel on n'a pas vraiment à se préoccuper de savoir si un message a été perdu. De plus, certaines applications sont en mesure d'envoyer à un serveur un message UDP et ensuite d'estimer que le message a été perdu si elles n'ont pas de réponse dans un délai raisonnable.

En règle générale, la majorité de la programmation réseau directe est réalisée avec TCP, et seulement occasionnellement avec UDP. Vous trouverez un traitement plus complet sur UDP, avec un exemple, dans la première édition de ce livre (disponible sur le CD ROM fourni avec ce livre, ou en téléchargement libre depuis www.BruceEckel.com).

Utiliser des URLs depuis un applet

Un applet a la possibilité d'afficher n'importe quelle URL au moyen du navigateur sur lequel il tourne. Ceci est réalisé avec la ligne suivante :

```
getAppletContext().showDocument(u);
```

dans laquelle **u** est l'objet **URL**. Voici un exemple simple qui nous redirige vers une autre page Web. Bien que nous soyons seulement redirigés vers une page HTML, nous pourrions l'être de la même manière vers la sortie d'un programme CGI.

```

//: c15:ShowHTML.java
// <applet code=ShowHTML width=100 height=50>
// </applet>
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.net.*;
import java.io.*;
import com.bruceeckel.swing.*;

public class ShowHTML extends JApplet {
    JButton send = new JButton("Go");
    JLabel l = new JLabel();
    public void init() {
        Container cp = getContentPane();
        cp.setLayout(new FlowLayout());
        send.addActionListener(new Al());
        cp.add(send);
        cp.add(l);
    }
    class Al implements ActionListener {
        public void actionPerformed(ActionEvent ae) {
            try {
                // Ceci pourrait être un programme CGI
                // au lieu d'une page HTML.
                URL u = new URL(getDocumentBase(),
                    "FetcherFrame.html");
                // Afficher la sortie de l'URL en utilisant
                // le navigateur Web, comme une page ordinaire:
                getAppletContext().showDocument(u);
            } catch (Exception e) {
                l.setText(e.toString());
            }
        }
    }
    public static void main(String[] args) {
        Console.run(new ShowHTML(), 100, 50);
    }
} //::~

```

Il est beau de voir combien la classe **URL** nous évite la complexité. Il est possible de se connecter à un serveur Web sans avoir à connaître ce qui se passe à bas niveau.

Lire un fichier depuis un serveur

Une variante du programme ci-dessus consiste à lire un fichier situé sur le serveur. Dans ce cas, le fichier est décrit par le client :

```

//: c15:Fetcher.java
// <applet code=Fetcher width=500 height=300>
// </applet>
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

```



```

import java.net.*;
import java.io.*;
import com.bruceeckel.swing.*;

public class Fetcher extends JApplet {
    JButton fetchIt= new JButton("Fetch the Data");
    JTextField f =
        new JTextField("Fetcher.java", 20);
    JTextArea t = new JTextArea(10,40);
    public void init() {
        Container cp = getContentPane();
        cp.setLayout(new FlowLayout());
        fetchIt.addActionListener(new FetchL());
        cp.add(new JScrollPane(t));
        cp.add(f); cp.add(fetchIt);
    }
    public class FetchL implements ActionListener {
        public void actionPerformed(ActionEvent e) {
            try {
                URL url = new URL(getDocumentBase(),
                    f.getText());
                t.setText(url + "\n");
                InputStream is = url.openStream();
                BufferedReader in = new BufferedReader(
                    new InputStreamReader(is));
                String line;
                while ((line = in.readLine()) != null)
                    t.append(line + "\n");
            } catch (Exception ex) {
                t.append(ex.toString());
            }
        }
    }
    public static void main(String[] args) {
        Console.run(new Fetcher(), 500, 300);
    }
} ///:~

```

La création de l'objet **URL** est semblable à celle de l'exemple précédent. Comme d'habitude, **getDocumentBase()** est le point de départ, mais cette fois le nom de fichier est lu depuis le **JTextField**. Un fois l'objet **URL** créé, sa version **String** est affichée dans le **JTextArea** afin que nous puissions la visualiser. Puis un **InputStream** est créé à partir de l'**URL**, qui dans ce cas va simplement créer un flux de caractères vers le fichier. Après avoir été convertie vers un **Reader** et bufferisée, chaque ligne est lue et ajoutée au **JTextArea**. Notons que le **JTextArea** est placé dans un **JScrollPane** afin que le scrolling soit pris en compte automatiquement.

En savoir plus sur le travail en réseau

En réalité, bien d'autres choses à propos du travail en réseau pourraient être traitées dans cette introduction. Le travail en réseau Java fournit également un support correct et étendu pour les URLs, incluant des handlers de protocole pour les différents types de contenu que l'on peut trouver sur un site Internet. Vous découvrirez d'autres fonctionnalités réseau de Java, complètement et minutieusement décrites dans *Java Network Programming* de Elliotte Rusty Harold (O'Reilly, 1997).

Se connecter aux bases de données : Java Database Connectivity (JDBC)

On a pu estimer que la moitié du développement de programmes implique des opérations client/serveur. Une des grandes promesses tenues par Java fut sa capacité à construire des applications de base de données client/serveur indépendantes de la plate-forme. C'est ce qui est réalisé avec Java DataBase Connectivity (JDBC).

Un des problèmes majeurs rencontrés avec les bases de données fut la guerre des fonctionnalités entre les compagnies fournissant ces bases de données. Il existe un langage standard de base de données, Structured Query Language (SQL-92), mais il est généralement préférable de connaître le vendeur de la base de données avec laquelle on travaille, malgré ce standard. JDBC est conçu pour être indépendant de la plate-forme, ainsi lorsqu'on programme on n'a pas à se soucier de la base de données qu'on utilise. Il est toutefois possible d'effectuer à partir de JDBC des appels spécifiques vers une base particulière afin de ne pas être limité dans ce que l'on pourrait faire.

Il existe un endroit pour lequel les programmeurs auraient besoin d'utiliser les noms de type SQL, c'est dans l'instruction SQL `TABLE CREATE` lorsqu'on crée une nouvelle table de base de données et qu'on définit le type SQL de chaque colonne. Malheureusement il existe des variantes significatives entre les types SQL supportés par différents produits base de données. Des bases de données différentes supportant des types SQL de même sémantique et de même structure peuvent appeler ces types de manières différentes. La plupart des bases de données importantes supportent un type de données SQL pour les grandes valeurs binaires : dans Oracle ce type s'appelle `LONG RAW`, Sybase le nomme `IMAGE`, Informix `BYTE`, et DB2 `LONG VARCHAR FOR BIT DATA`. Par conséquent, si on a pour but la portabilité des bases de données il faut essayer de n'utiliser que les identifiants de type SQL génériques.

La portabilité est une question d'actualité lorsqu'on écrit pour un livre dont les lecteurs vont tester les exemples avec toute sorte de stockage de données inconnus. J'ai essayé de rendre ces exemples aussi portables qu'il était possible. Remarquez également que le code spécifique à la base de données a été isolé afin de centraliser toutes les modifications que vous seriez obligés d'effectuer pour que ces exemples deviennent opérationnels dans votre environnement.

JDBC, comme bien des APIs Java, est conçu pour être simple. Les appels de méthode que l'on utilise correspondent aux opérations logiques auxquelles on pense pour obtenir des données depuis une base de données, créer une instruction, exécuter la demande, et voir le résultat.

Pour permettre cette indépendance de plate-forme, JDBC fournit un *gestionnaire de driver* qui maintient dynamiquement les objets driver nécessités par les interrogations de la base. Ainsi si on doit se connecter à trois différentes sortes de base, on a besoin de trois objets driver différents. Les objets driver s'enregistrent eux-même auprès du driver manager lors de leur chargement, et on peut forcer le chargement avec la méthode **`Class.forName()`**.

Pour ouvrir une base de données, il faut créer une « URL de base de données » qui spécifie :

1. Qu'on utilise JDBC avec « jdbc ».
2. Le « sous-protocole » : le nom du driver ou le nom du mécanisme de connectivité à la base de données. Parce que JDBC a été inspiré par ODBC, le premier sous-protocole disponible est la « passerelle jdbc-odbc », spécifiée par « odbc ».
3. L'identifiant de la base de données. Il varie avec le driver de base de donnée utilisé, mais il existe généralement un nom logique qui est associé par le software d'administration de la base à un répertoire physique où se trouvent les tables de la base de données. Pour qu'un identifiant de base de données ait une

signification, il faut l'enregistrer en utilisant le software d'administration de la base (cette opération varie d'une plate-forme à l'autre).

Toute cette information est condensée dans une chaîne de caractères, l'URL de la base de données. Par exemple, pour se connecter au moyen du protocole ODBC à une base appelée `people`, l'URL de base de données doit être :

```
String dbUrl = "jdbc:odbc:people";
```

Si on se connecte à travers un réseau, l'URL de base de données doit contenir l'information de connexion identifiant la machine distante, et peut alors devenir intimidante. Voici en exemple la base de données CloudScape que l'on appelle depuis un client éloigné en utilisant RMI :

```
jdbc:rmi://192.168.170.27:1099/jdbc:cloudscape:db
```

En réalité cette URL de base de données comporte deux appels `jdbc` en un. La première partie `jdbc:rmi://192.168.170.27:1099/` utilise RMI pour effectuer une connexion sur le moteur distant de base de données à l'écoute sur le port 1099 de l'adresse IP 192.168.170.27. La deuxième partie de l'URL, `jdbc:cloudscape:db` représente une forme plus connue utilisant le sous-protocole et le nom de la base, mais elle n'entrera en jeu que lorsque la première section aura établi la connexion à la machine distante via RMI.

Lorsqu'on est prêt à se connecter à une base, on appelle la méthode statique **DriverManager.getConnection()** en lui fournissant l'URL de base de données, le nom d'utilisateur et le mot de passe pour accéder à la base. On reçoit en retour un objet **Connection** que l'on peut ensuite utiliser pour effectuer des demandes et manipuler la base de données.

L'exemple suivant ouvre une base d'« information de contact » et cherche le nom de famille d'une personne, donné sur la ligne de commande. Il sélectionne d'abord les noms des personnes qui ont une adresse email, puis imprime celles qui correspondent au nom donné :

```
//: cl15:jdbc:Lookup.java
// Cherche les adresses email dans une
// base de données locale en utilisant JDBC.
import java.sql.*;

public class Lookup {
    public static void main(String[] args)
        throws SQLException, ClassNotFoundException {
        String dbUrl = "jdbc:odbc:people";
        String user = "";
        String password = "";
        // Charger le driver (qui s'enregistrera lui-même)
        Class.forName(
            "sun.jdbc.odbc.JdbcOdbcDriver");
        Connection c = DriverManager.getConnection(
            dbUrl, user, password);
        Statement s = c.createStatement();
        // code SQL:
        ResultSet r =
            s.executeQuery(
                "SELECT FIRST, LAST, EMAIL " +
```

```

        "FROM people.csv people " +
        "WHERE " +
        "(LAST='" + args[0] + "') " +
        " AND (EMAIL Is Not Null) " +
        "ORDER BY FIRST");
while(r.next()) {
    // minuscules et majuscules n'ont
    // aucune importance:
    System.out.println(
        r.getString("Last") + ", "
        + r.getString("FIRST")
        + ": " + r.getString("EMAIL") );
}
s.close(); // fermer également ResultSet
}
} ///:~

```

Une URL de base de données est créée comme précédemment expliqué. Dans cet exemple, il n'y a pas de protection par mot de passe, c'est pourquoi le nom d'utilisateur et le mot de passe sont des chaînes vides.

Une fois la connexion établie avec **DriverManager.getConnection()** l'objet résultant **Connection** sert à créer un objet **Statement** au moyen de la méthode **createStatement()**. Avec cet objet **Statement**, on peut appeler **executeQuery()** en lui passant une chaîne contenant une instruction standard SQL-92 (il n'est pas difficile de voir comment générer cette instruction automatiquement, on n'a donc pas à connaître grand'chose de SQL).

La méthode **executeQuery()** renvoie un objet **ResultSet**, qui est un itérateur : la méthode **next()** fait pointer l'objet itérateur sur l'enregistrement suivant dans l'instruction, ou bien renvoie la valeur **false** si la fin de l'ensemble résultat est atteinte. On obtient toujours un objet **ResultSet** en réponse à la méthode **executeQuery()** même lorsqu'une demande débouche sur un ensemble vide (autrement dit aucune exception n'est générée). Notons qu'il faut appeler la méthode **next()** au moins une fois avant de tenter de lire un enregistrement de données. Si l'ensemble résultant est vide, ce premier appel de **next()** renverra la valeur **false**. Pour chaque enregistrement dans l'ensemble résultant, on peut sélectionner les champs en utilisant (entre autres solutions) une chaîne représentant leur nom. Remarquons aussi que la casse du nom de champ est ignorée dans les transactions avec une base de donnée. Le type de données que l'on veut récupérer est déterminé par le nom de la méthode appelée : **getInt()**, **getString()**, **getFloat()**, etc. À ce moment, on dispose des données de la base en format Java natif et on peut les traiter comme bon nous semble au moyen du code Java habituel.

Faire fonctionner l'exemple

Avec JDBC, il est relativement simple de comprendre le code. Le côté difficile est de faire en sorte qu'il fonctionne sur votre système particulier. La raison de cette difficulté est que vous devez savoir comment charger proprement le driver JDBC, et comment initialiser une base de données en utilisant le software d'administration de la base.

Bien entendu, ce travail peut varier de façon radicale d'une machine à l'autre, mais la manière dont j'ai procédé pour le faire fonctionner sur un système Windows 32 bits vous donnera sans doute des indications qui vous aideront à attaquer votre propre situation.

Étape 1 : Trouver le Driver JDBC

Le programme ci-dessus contient l'instruction :

```
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
```

Ceci suggère une structure de répertoire, ce qui est trompeur. Avec cette installation particulière du JDK 1.1, il n'existe pas de fichier nommé **JdbcOdbcDriver.class**, et si vous avez cherché ce fichier après avoir regardé l'exemple vous avez dû être déçus. D'autres exemples publiés utilisent un pseudo nom, comme `myDriver.ClassName`, ce qui nous aide encore moins. En fait, l'instruction de chargement du driver `jdbc-odbc` (le seul existant actuellement dans le JDK) apparaît en peu d'endroits dans la documentation en ligne (en particulier, une page appelée JDBC-ODBC Bridge Driver). Si l'instruction de chargement ci-dessus ne fonctionne pas, il est possible que le nom ait été changé à l'occasion d'une évolution de version Java, et vous devez repartir en chasse dans la documentation.

Si l'instruction de chargement est incorrecte, on récupérera une exception à ce moment-là. Pour tester si l'instruction de chargement du driver fonctionne correctement, mettez en commentaire le code après l'instruction jusqu'à la clause **catch** ; si le programme ne lance pas d'exceptions cela signifie que le driver est correctement chargé.

Étape 2 : Configurer la base de données

Ici aussi, ceci est spécifique à Windows 32-bits ; vous devrez sans doute rechercher quelque peu pour savoir comment faire sur votre plate-forme.

Pour commencer, ouvrez le panneau de configuration. Vous devriez trouver deux icônes traitant de ODBC. Il faut utiliser celle qui parle de ODBC 32-bits, l'autre n'étant là que pour la compatibilité ascendante avec le logiciel ODBC 16-bits, et ne serait d'aucune utilité pour JDBC. En ouvrant l'icône ODBC 32-bits, vous allez voir une boîte de dialogue à onglets, parmi lesquels User DSN, System DSN, File DSN, etc., DSN signifiant Data Source Name. Il apparaît que pour la passerelle JDBC-ODBC, le seul endroit important pour créer votre base de données est System DSN, mais vous devez également tester votre configuration et créer des demandes, et pour cela vous devez également créer votre base dans File DSN. Ceci permet à l'outil Microsoft Query (qui fait partie de Microsoft Office) de trouver la base. Remarquez que d'autres outils de demande sont disponibles chez d'autres vendeurs.

La base de données la plus intéressante est l'une de celles que vous utilisez déjà. Le standard ODBC supporte différents formats de fichier y compris les vénérables chevaux de labour tels que DBase. Cependant, il inclut aussi le format « ASCII, champs séparés par des virgules », que n'importe quel outil de données est capable de créer. En ce qui me concerne, j'ai simplement pris ma base de données « people » que j'ai maintenue depuis des années au moyen de divers outils de gestion d'adresse et que j'ai exportée en tant que fichier « ASCII à champs séparés par des virgules » (ils ont généralement une extension **.csv**). Dans la section System DSN j'ai choisi Add, puis le driver texte pour mon fichier ASCII csv, puis dé-coché « use current directory » pour me permettre de spécifier le répertoire où j'avais exporté mon fichier de données.

Remarquez bien qu'en faisant cela vous ne spécifiez pas réellement un fichier, mais seulement un répertoire. Ceci parce qu'une base de données se trouve souvent sous la forme d'un ensemble de fichiers situés dans un même répertoire (bien qu'elle puisse aussi bien se trouver sous d'autres formes). Chaque fichier contient généralement une seule table, et une instruction SQL peut produire des résultats issus de plusieurs tables de la base (on appelle ceci une *relation*). Une base contenant une seule table (comme ma base « people ») est généralement appelée *flat-file database*. La plupart des problèmes qui vont au-delà du simple stockage et déstockage de données nécessitent des tables multiples mises en relation par des *relations* afin de fournir les résultats voulus, on les appelle bases de données *relationnelles*.

Étape 3 : Tester la configuration

Pour tester la configuration il faut trouver un moyen de savoir si la base est visible depuis un programme qui

l'interrogerait. Bien entendu, on peut tout simplement lancer le programme exemple JDBC ci-dessus, en incluant l'instruction :

```
Connection c = DriverManager.getConnection(
    dbUrl, user, password);
```

Si une exception est lancée, c'est que la configuration était incorrecte.

Cependant, à ce point, il est très utile d'utiliser un outil de génération de requêtes. J'ai utilisé Microsoft Query qui est livré avec Microsoft Office, mais vous pourriez préférer un autre outil. L'outil de requête doit connaître l'emplacement de la base, et Microsoft Query exigeait que j'ouvre l'onglet administrateur ODBC File DSN et que j'ajoute une nouvelle entrée, en spécifiant à nouveau le driver texte et le répertoire contenant ma base de données. On peut donner n'importe quel nom à cette entrée, mais il est préférable d'utiliser le nom déjà fourni dans l'onglet System DSN.

Ceci fait, vous saurez si votre base est disponible en créant une nouvelle requête au moyen de votre générateur de requêtes.

Étape 4 : Générer votre requête SQL

La requête créée avec Microsoft Query m'a montré que ma base de données était là et prête à fonctionner, mais a aussi généré automatiquement le code SQL dont j'avais besoin pour l'insérer dans mon programme Java. Je voulais une requête qui recherche les enregistrements contenant le même nom que celui qui était fourni sur la ligne de commande d'appel du programme. Ainsi pour commencer, j'ai cherché un nom particulier, Eckel.

Je voulais également n'afficher que ceux qui avaient une adresse email associée. Voici les étapes que j'ai suivies pour créer cette requête :

1. Ouvrir une nouvelle requête au moyen du Query Wizard. Sélectionner la base « people » (ceci est équivalent à ouvrir la connexion avec la base au moyen de l'URL de base de données appropriée).
2. Dans la base, sélectionner la table « people ». Dans la table, choisir les colonnes FIRST, LAST, et EMAIL.
3. Sous « Filter Data », choisir LAST et sélectionner « equals » avec comme argument « Eckel ». Cliquer sur le bouton radio « And ».
4. Choisir EMAIL et sélectionner « Is not Null ».
5. Sous « Sort By », choisir FIRST.

Le résultat de cette requête vous montrera si vous obtenez ce que vous vouliez.

Maintenant cliquez sur le bouton SQL et sans aucun effort de votre part vous verrez apparaître le code SQL correct, prêt pour un copier-coller. Le voici pour cette requête :

```
SELECT people.FIRST, people.LAST, people.EMAIL
FROM people.csv people
WHERE (people.LAST='Eckel') AND
(people.EMAIL Is Not Null)
ORDER BY people.FIRST
```

Il serait très facile de se tromper dans le cas de requêtes plus compliquées, mais en utilisant un générateur de requêtes vous pouvez tester votre requête interactivement et générer automatiquement le code correct. Il serait difficile d'affirmer qu'il est préférable de réaliser cela manuellement.

Étape 5 : Modifier et insérer votre requête

Remarquons que le code ci-dessus ne ressemble pas à celui qui est utilisé dans le programme. Ceci est dû au fait que le générateur de requêtes utilise des noms complètement qualifiés, même lorsqu'une seule table est en jeu (lorsqu'on utilise plus d'une table, la qualification empêche les collisions entre colonnes de même nom appartenant à des tables différentes). Puisque cette requête ne concerne qu'une seule table, on peut - optionnellement - supprimer le qualificateur « *people* » dans la plupart des noms, de cette manière :

```
SELECT FIRST, LAST, EMAIL
FROM people.csv people
WHERE (LAST='Eckel') AND
(EMAIL Is Not Null)
ORDER BY FIRST
```

En outre, on ne va pas coder en dur le nom à rechercher. Au lieu de cela, il sera créé à partir du nom fourni en argument sur la ligne de commande d'appel du programme. Ces changements effectués l'instruction SQL générée dynamiquement dans un objet **String** devient :

```
"SELECT FIRST, LAST, EMAIL " +
"FROM people.csv people " +
"WHERE " +
"(LAST='" + args[0] + "') " +
" AND (EMAIL Is Not Null) " +
"ORDER BY FIRST");
```

SQL possède un autre mécanisme d'insertion de noms dans une requête, appelé procédures stockées, *stored procedures*, utilisées pour leur vitesse. Mais pour la plupart de vos expérimentations sur les bases de données et pour votre apprentissage, il est bon de construire vos chaînes de requêtes en Java.

On peut voir à partir de cet exemple que l'utilisation d'outils généralement disponibles et en particulier le générateur de requêtes simplifie la programmation avec SQL et JDBC..

Une version GUI du programme de recherche

Il serait plus pratique de laisser tourner le programme en permanence et de basculer vers lui pour taper un nom et entamer une recherche lorsqu'on en a besoin. Le programme suivant crée le programme de recherche en tant qu'application/applet, et ajoute également une fonctionnalité de complétion des noms afin qu'on puisse voir les données sans être obligé de taper le nom complet :

```
//: c15:jdbc:VLookup.java
// version GUI de Lookup.java.
// <applet code=VLookup
// width=500 height=200></applet>
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import javax.swing.event.*;
import java.sql.*;
import com.bruceeckel.swing.*;

public class VLookup extends JApplet {
    String dbUrl = "jdbc:odbc:people";
    String user = "";
    String password = "";
    Statement s;
```



```

JTextField searchFor = new JTextField(20);
JLabel completion =
    new JLabel(" ");
JTextArea results = new JTextArea(40, 20);
public void init() {
    searchFor.getDocument().addDocumentListener(
        new SearchL());
    JPanel p = new JPanel();
    p.add(new Label("Last name to search for:"));
    p.add(searchFor);
    p.add(completion);
    Container cp = getContentPane();
    cp.add(p, BorderLayout.NORTH);
    cp.add(results, BorderLayout.CENTER);
    try {
        // Charger le driver (qui s'enregistrera lui-même)
        Class.forName(
            "sun.jdbc.odbc.JdbcOdbcDriver");
        Connection c = DriverManager.getConnection(
            dbUrl, user, password);
        s = c.createStatement();
    } catch (Exception e) {
        results.setText(e.toString());
    }
}
class SearchL implements DocumentListener {
    public void changedUpdate(DocumentEvent e) {}
    public void insertUpdate(DocumentEvent e) {
        textValueChanged();
    }
    public void removeUpdate(DocumentEvent e) {
        textValueChanged();
    }
}
public void textValueChanged() {
    ResultSet r;
    if (searchFor.getText().length() == 0) {
        completion.setText("");
        results.setText("");
        return;
    }
    try {
        // compléter automatiquement le nom:
        r = s.executeQuery(
            "SELECT LAST FROM people.csv people " +
            "WHERE (LAST Like " +
            searchFor.getText() +
            "%') ORDER BY LAST");
        if (r.next())
            completion.setText(
                r.getString("last"));
        r = s.executeQuery(
            "SELECT FIRST, LAST, EMAIL " +
            "FROM people.csv people " +
            "WHERE (LAST=" +

```

```

        completion.getText() +
        "'" AND (EMAIL Is Not Null) " +
        "ORDER BY FIRST");
    } catch(Exception e) {
        results.setText(
            searchFor.getText() + "\n");
        results.append(e.toString());
        return;
    }
    results.setText("");
    try {
        while(r.next()) {
            results.append(
                r.getString("Last") + ", "
                + r.getString("fIRST") +
                ": " + r.getString("EMAIL") + "\n");
        }
    } catch(Exception e) {
        results.setText(e.toString());
    }
}

public static void main(String[] args) {
    Console.run(new VLookup(), 500, 200);
}
} ///:~

```

La logique « base de données » est en gros la même, mais on peut remarquer qu'un **DocumentListener** est ajouté pour écouter l'entrée **JTextField** (pour plus de détails, voir l'entrée **javax.swing.JTextField** dans la documentation HTML Java sur java.sun.com), afin qu'à chaque nouveau caractère frappé le programme tente de compléter le nom en le cherchant dans la base et en utilisant le premier qu'il trouve (en le plaçant dans le **JLabel completion**, et en l'utilisant comme un texte de recherche). De cette manière, on peut arrêter de frapper des caractères dès qu'on en a tapé suffisamment pour que le programme puisse identifier de manière unique le nom qu'on cherche.

Pourquoi l'API JDBC paraît si complexe

Naviguer dans la documentation en ligne de JDBC semble décourageant. En particulier, dans l'interface **DatabaseMetaData** qui est vraiment énorme, à l'inverse de la plupart des interfaces rencontrées jusque là en Java, on trouve des méthodes telles que **dataDefinitionCausesTransactionCommit()**, **getMaxColumnNameLength()**, **getMaxStatementLength()**, **storesMixedCaseQuotedIdentifiers()**, **supportsANSI92IntermediateSQL()**, **supportsLimitedOuterJoins()**, et ainsi de suite. Qu'en est-il de tout cela ?

Ainsi qu'on l'a dit précédemment, les bases de données semblent être depuis leur création dans un constant état de bouleversement, principalement à cause de la très grande demande en applications de base de données, et donc en outils de base de données. Ce n'est que récemment qu'on a vu une certaine convergence vers le langage commun SQL (et il existe beaucoup d'autres langages d'utilisation courante de base de données). Mais même le « standard » SQL possède tellement de variantes que JDBC doit fournir la grande interface **DatabaseMetaData** afin que le code puisse utiliser les possibilités de la base SQL « standard » particulière avec laquelle on est connecté. Bref, il est possible d'écrire du code SQL simple et portable, mais si on veut optimiser la vitesse le code va se multiplier terriblement au fur et à mesure qu'on découvre les possibilités d'une base de données d'un vendeur particulier.

Bien entendu, ce n'est pas la faute de Java. Ce dernier tente seulement de nous aider à compenser les disparités entre les divers produits de base de données . Mais gardons à l'esprit que la vie est plus facile si l'on peut aussi bien écrire des requêtes génériques sans trop se soucier des performances, ou bien, si l'on veut améliorer les performances, connaître la plate-forme pour laquelle on écrit afin de ne pas avoir à traîner trop de code générique.

Un exemple plus sophistiqué

Un exemple plus intéressant [73] est celui d'une base de données multi-tables résidant sur un serveur. Ici, la base sert de dépôt pour les activités d'une communauté et doit permettre aux gens de s'inscrire pour réaliser ces actions, c'est pourquoi on l'appelle une base de données de communauté d'intérêts, *Community Interests Database* (CID). Cet exemple fournira seulement une vue générale de la base et de son implémentation, il n'est en aucun cas un tutoriel complet à propos du développement des bases de données. De nombreux livres, séminaires, et packages de programmes existent pour vous aider dans la conception et le développement des bases de données.

De plus, cet exemple implique l'installation préalable d'une base SQL sur un serveur (bien qu'elle puisse aussi bien tourner sur la machine locale), ainsi que la recherche d'un driver JDBC approprié pour cette base. Il existe plusieurs bases SQL libres, et certaines sont installées automatiquement avec diverses distributions de Linux. Il est de votre responsabilité de faire le choix de la base de données et de localiser son driver JDBC ; cet exemple-ci est basé sur une base SQL nommée « Cloudscape ».

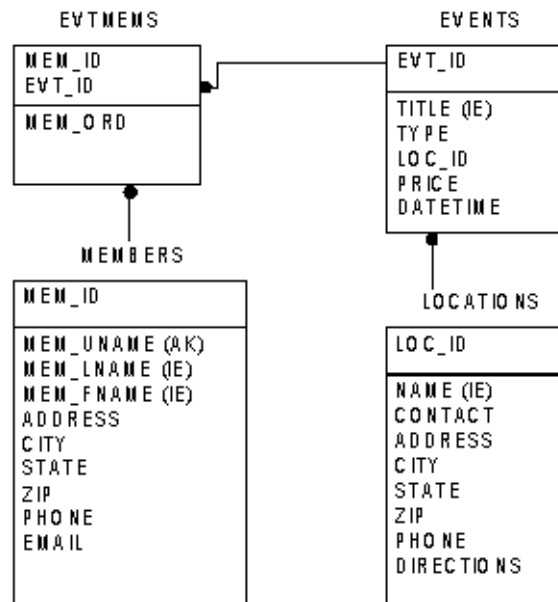
Afin de simplifier les modifications d'information de connexion, le driver de la base, son URL, le nom d'utilisateur et son mot de passe sont placés dans une classe séparée :

```
//: c15:jdbc:CIDConnect.java
// information de connexion à la base de données pour
// la «base de données de communauté d'intérêt» (CID).

public class CIDConnect {
    // Toutes les informations spécifiques à CloudScape:
    public static String dbDriver =
        "COM.cloudscape.core.JDBCDriver";
    public static String dbURL =
        "jdbc:cloudscape:d:/docs/_work/JSapientDB";
    public static String user = "";
    public static String password = "";
} ///:~
```

Dans cet exemple, il n'y a pas de protection de la base par mot de passe, le nom d'utilisateur et le mot de passe sont des chaînes vides.

La base de données comprend un ensemble de tables dont voici la structure :



« Members » contient les informations sur les membres de la communauté, « Events » et « Locations » des informations à propos des activités et où on peut les trouver, et « Evtmems » connecte les événements et les membres qui veulent suivre ces événements. On peut constater qu'à une donnée « membre » d'une table correspond une clef dans une autre table.

La classe suivante contient les chaînes SQL qui vont créer les tables de cette base (référez-vous à un guide SQL pour une explication du code SQL) :

```

//: c15:jdbc:CIDSQl.java
// chaînes SQL créant les tables pour la CID.

public class CIDSQl {
    public static String[] sql = {
        // Créer la table MEMBERS:
        "drop table MEMBERS",
        "create table MEMBERS " +
        "(MEM_ID INTEGER primary key, " +
        "MEM_UNAME VARCHAR(12) not null unique, " +
        "MEM_LNAME VARCHAR(40), " +
        "MEM_FNAME VARCHAR(20), " +
        "ADDRESS VARCHAR(40), " +
        "CITY VARCHAR(20), " +
        "STATE CHAR(4), " +
        "ZIP CHAR(5), " +
        "PHONE CHAR(12), " +
        "EMAIL VARCHAR(30))",
        "create unique index " +
        "LNAME_IDX on MEMBERS(MEM_LNAME)",
        // Créer la table EVENTS:
        "drop table EVENTS",
        "create table EVENTS " +
        "(EVT_ID INTEGER primary key, " +
        "EVT_TITLE VARCHAR(30) not null, " +
        "EVT_TYPE VARCHAR(20), " +
        "LOC_ID INTEGER, " +
        "PRICE DECIMAL, " +
        "DATETIME TIMESTAMP)",
    }
}
  
```

```

"create unique index " +
"TITLE_IDX on EVENTS(EVT_TITLE)",
// Créer la table EVTMEMS:
"drop table EVTMEMS",
"create table EVTMEMS " +
"(MEM_ID INTEGER not null, " +
"EVT_ID INTEGER not null, " +
"MEM_ORD INTEGER)",
"create unique index " +
"EVTMEM_IDX on EVTMEMS(MEM_ID, EVT_ID)",
// Créer la table LOCATIONS:
"drop table LOCATIONS",
"create table LOCATIONS " +
"(LOC_ID INTEGER primary key, " +
"LOC_NAME VARCHAR(30) not null, " +
"CONTACT VARCHAR(50), " +
"ADDRESS VARCHAR(40), " +
"CITY VARCHAR(20), " +
"STATE VARCHAR(4), " +
"ZIP VARCHAR(5), " +
"PHONE CHAR(12), " +
"DIRECTIONS VARCHAR(4096))",
"create unique index " +
"NAME_IDX on LOCATIONS(LOC_NAME)",
};
} ///:~

```

Le programme qui suit utilise l'information de **CIDConnect** et **CIDSQL** pour charger le driver JDBC, se connecter à la base de données, puis créer la structure de la table conforme au diagramme ci-dessus. Pour se connecter à la base, on appelle la méthode **static DriverManager.getConnection()**, en lui passant l'URL de base de données, le nom d'utilisateur et un mot de passe. On obtient en retour un objet **Connection** que l'on peut utiliser pour interroger et manipuler la base. La connexion établie, il suffit d'envoyer le SQL à la base, dans ce cas en balayant le tableau **CIDSQL**. Toutefois, au premier lancement du programme, la commande « drop table » échouera, lançant une exception, qui sera capturée, rapportée, puis ignorée. La commande « drop table » n'a d'autre but que de permettre une expérimentation plus facile : on peut modifier le code SQL définissant les tables puis exécuter à nouveau le programme, les anciennes tables étant remplacées par les nouvelles.

Dans cet exemple, il est intéressant de laisser les exceptions s'afficher sur la console :

```

///: c15:jdbc:CIDCreateTables.java
// Crée les tables d'une base de données pour la
// «community interests database».
import java.sql.*;

public class CIDCreateTables {
    public static void main(String[] args)
        throws SQLException, ClassNotFoundException,
        IllegalAccessException {
        // Charger le driver (qui s'enregistrera lui-même)
        Class.forName(CIDConnect.dbDriver);
        Connection c = DriverManager.getConnection(
            CIDConnect.dbURL, CIDConnect.user,
            CIDConnect.password);
        Statement s = c.createStatement();
        for(int i = 0; i < CIDSQL.sql.length; i++) {

```

```

        System.out.println(CIDSQL.sql[i]);
    try {
        s.executeUpdate(CIDSQL.sql[i]);
    } catch (SQLException sqlEx) {
        System.err.println(
            "Probably a 'drop table' failed");
    }
}
s.close();
c.close();
}
} ///:~

```

Remarquons que les modifications de la base peuvent être contrôlées en changeant **Strings** dans la table **CIDSQL**, sans modifier **CIDCreateTables**.

La méthode **executeUpdate()** renvoie généralement le nombre d'enregistrements affectés par l'instruction SQL. Elle est très souvent utilisée pour exécuter des instructions **INSERT**, **UPDATE**, ou **DELETE** modifiant une ou plusieurs lignes. Pour les instructions telles que **CREATE TABLE**, **DROP TABLE**, et **CREATE INDEX**, **executeUpdate()** renvoie toujours zéro.

Pour tester la base, celle-ci est chargée avec quelques données exemples. Ceci est réalisé au moyen d'une série d'**INSERT** suivie d'un **SELECT** afin de produire le jeu de données. Pour effectuer facilement des additions et des modifications aux données de test, ce dernier est construit comme un tableau d'**Object** à deux dimensions, et la méthode **executeInsert()** peut alors utiliser l'information d'une ligne de la table pour construire la commande SQL appropriée.

```

///: c15:jdbc:LoadDB.java
// Charge et teste la base de données.
import java.sql.*;

class TestSet {
    Object[][] data = {
        { "MEMBERS", new Integer(1),
          "dbartlett", "Bartlett", "David",
          "123 Mockingbird Lane",
          "Gettysburg", "PA", "19312",
          "123.456.7890", "bart@you.net" },
        { "MEMBERS", new Integer(2),
          "beckel", "Eckel", "Bruce",
          "123 Over Rainbow Lane",
          "Crested Butte", "CO", "81224",
          "123.456.7890", "beckel@you.net" },
        { "MEMBERS", new Integer(3),
          "rcastaneda", "Castaneda", "Robert",
          "123 Downunder Lane",
          "Sydney", "NSW", "12345",
          "123.456.7890", "rcastaneda@you.net" },
        { "LOCATIONS", new Integer(1),
          "Center for Arts",
          "Betty Wright", "123 Elk Ave.",
          "Crested Butte", "CO", "81224",
          "123.456.7890",
          "Go this way then that." },
        { "LOCATIONS", new Integer(2),

```

```

        "Witts End Conference Center",
        "John Wittig", "123 Music Drive",
        "Zoneville", "PA", "19123",
        "123.456.7890",
        "Go that way then this." },
    { "EVENTS", new Integer(1),
      "Project Management Myths",
      "Software Development",
      new Integer(1), new Float(2.50),
      "2000-07-17 19:30:00" },
    { "EVENTS", new Integer(2),
      "Life of the Crested Dog",
      "Archeology",
      new Integer(2), new Float(0.00),
      "2000-07-19 19:00:00" },
    // Met en relation personnes et événements
    { "EVTMEMS",
      new Integer(1), // Dave est mis en relation avec
      new Integer(1), // l'événement Software.
      new Integer(0) },
    { "EVTMEMS",
      new Integer(2), // Bruce est mis en relation avec
      new Integer(2), // l'événement Archeology.
      new Integer(0) },
    { "EVTMEMS",
      new Integer(3), // Robert est mis en relation avec
      new Integer(1), // l'événement Software...
      new Integer(1) },
    { "EVTMEMS",
      new Integer(3), // ... et
      new Integer(2), // l'événement Archeology.
      new Integer(1) },
    };
    // Utiliser les données par défaut:
    public TestSet() {}
    // Utiliser un autre ensemble de données:
    public TestSet(Object[][] dat) { data = dat; }
}

public class LoadDB {
    Statement statement;
    Connection connection;
    TestSet tset;
    public LoadDB(TestSet t) throws SQLException {
        tset = t;
        try {
            // Charger le driver (qui s'enregistrera lui-même)
            Class.forName(CIDConnect.dbDriver);
        } catch (java.lang.ClassNotFoundException e) {
            e.printStackTrace(System.err);
        }
        connection = DriverManager.getConnection(
            CIDConnect.dbURL, CIDConnect.user,
            CIDConnect.password);
        statement = connection.createStatement();
    }
}

```



```

public void cleanup() throws SQLException {
    statement.close();
    connection.close();
}
public void executeInsert(Object[] data) {
    String sql = "insert into "
        + data[0] + " values(";
    for(int i = 1; i < data.length; i++) {
        if(data[i] instanceof String)
            sql += "'" + data[i] + "'";
        else
            sql += data[i];
        if(i < data.length - 1)
            sql += ", ";
    }
    sql += ')';
    System.out.println(sql);
    try {
        statement.executeUpdate(sql);
    } catch(SQLException sqlEx) {
        System.err.println("Insert failed.");
        while (sqlEx != null) {
            System.err.println(sqlEx.toString());
            sqlEx = sqlEx.getNextException();
        }
    }
}
public void load() {
    for(int i = 0; i < tset.data.length; i++)
        executeInsert(tset.data[i]);
}
// Lever l'exception en l'envoyant vers la console:
public static void main(String[] args)
throws SQLException {
    LoadDB db = new LoadDB(new TestSet());
    db.load();
    try {
        // Obtenir un ResultSet de la base chargée:
        ResultSet rs = db.statement.executeQuery(
            "select " +
            "e.EVT_TITLE, m.MEM_LNAME, m.MEM_FNAME " +
            "from EVENTS e, MEMBERS m, EVTMEMS em " +
            "where em.EVT_ID = 2 " +
            "and e.EVT_ID = em.EVT_ID " +
            "and m.MEM_ID = em.MEM_ID");
        while (rs.next())
            System.out.println(
                rs.getString(1) + " " +
                rs.getString(2) + ", " +
                rs.getString(3));
    } finally {
        db.cleanup();
    }
}
} ///:~

```

La classe **TestSet** contient un ensemble de données par défaut qui est mis en oeuvre lorsqu'on appelle le constructeur par défaut ; toutefois, il est possible de créer au moyen du deuxième constructeur un objet **TestSet** utilisant un deuxième ensemble de données. L'ensemble de données est contenu dans un tableau à deux dimensions de type **Object** car il peut contenir n'importe quel type, y compris **String** ou des types numériques. La méthode **executeInsert()** utilise RTTI pour différencier les données **String** (qui doivent être entre guillemets) et les données non-**String** en construisant la commande SQL à partir des données. Après avoir affiché cette commande sur la console, **executeUpdate()** l'envoie à la base de données.

Le constructeur de **LoadDB** établit la connexion, et **load()** parcourt les données en appelant **executeInsert()** pour chaque enregistrement. **Cleanup()** termine l'instruction et la connexion ; tout ceci est placé dans une clause **finally** afin d'en garantir l'appel.

Une fois la base chargée, une instruction **executeQuery()** produit un ensemble résultat. La requête concernant plusieurs tables, nous avons bien un exemple de base de données relationnelle.

On trouvera d'autres informations sur JDBC dans les documents électroniques livrés avec la distribution Java de Sun. Pour en savoir plus, consulter le livre *JDBC Database Access with Java* (Hamilton, Cattel, and Fisher, Addison-Wesley, 1997). D'autres livres à propos de JDBC sortent régulièrement.

Les Servlets

Les accès clients sur l'Internet ou les intranets d'entreprise représentent un moyen sûr de permettre à beaucoup d'utilisateurs d'accéder facilement aux données et ressources [74]. Ce type d'accès est basé sur des clients utilisant les standards du World Wide Web Hypertext Markup Language (HTML) et Hypertext Transfer Protocol (HTTP). L'API Servlet fournit une abstraction pour un ensemble de solutions communes en réponse aux requêtes HTTP.

Traditionnellement, la solution permettant à un client Internet de mettre à jour une base de données est de créer une page HTML contenant des champs texte et un bouton « soumission ». L'utilisateur frappe l'information requise dans les champs texte puis clique sur le bouton « soumission ». Les données sont alors soumises au moyen d'une URL qui indique au serveur ce qu'il doit en faire en lui indiquant l'emplacement d'un programme Common Gateway Interface (CGI) lancé par le serveur, qui prend ces données en argument. Le programme CGI est généralement écrit en Perl, Python, C, C++, ou n'importe quel langage capable de lire sur l'entrée standard et d'écrire sur la sortie standard. Le rôle du serveur Web s'arrête là : le programme CGI est appelé, et des flux standard (ou, optionnellement pour l'entrée, une variable d'environnement) sont utilisés pour l'entrée et la sortie. Le programme CGI est responsable de toute la suite. Il commence par examiner les données et voir si leur format est correct. Si ce n'est pas le cas, le programme CGI doit fournir une page HTML décrivant le problème ; cette page est prise en compte par le serveur Web (via la sortie standard du programme CGI), qui la renvoie à l'utilisateur. Habituellement, l'utilisateur revient à la page précédente et fait une nouvelle tentative. Si les données sont correctes, le programme CGI traite les données de la manière appropriée, par exemple en les ajoutant à une base de données. Il élabore ensuite une page HTML appropriée que le serveur Web enverra à l'utilisateur.

Afin d'avoir une solution basée entièrement sur Java, l'idéal serait d'avoir côté client une applet qui validerait et enverrait les données, et côté serveur une servlet qui les recevrait et les traiterait. Malheureusement, bien que les applets forment une technologie éprouvée et bien supportée, leur utilisation sur le Web s'est révélée problématique car on ne peut être certain de la disponibilité d'une version particulière de Java sur le navigateur Web du client ; en fait, on ne peut même pas être certain que le navigateur Web supporte Java ! Dans un intranet, on peut exiger qu'un support donné soit disponible, ce qui apporte une certaine flexibilité à ce qu'on peut faire, mais sur le Web l'approche la plus sûre est d'effectuer tout le traitement du côté serveur puis de

délivrer une page HTML au client. De cette manière, aucun client ne se verra refuser l'utilisation de votre site simplement parce qu'il ne dispose pas dans sa configuration du software approprié.

Parce que les servlets fournissent une excellente solution pour le support de programmation côté serveur, ils représentent l'une des raisons les plus populaires pour passer à Java. Non seulement ils fournissent un cadre pour remplacer la programmation CGI (et éliminer nombre de problèmes CGI épineux), mais tout le code gagne en portabilité inter plate-forme en utilisant Java, et l'on a accès à toutes les API Java (exceptées, bien entendu, celles qui fournissent des GUI, comme Swing).

Le servlet de base

L'architecture de l'API servlet est celle d'un fournisseur de services classique comportant une méthode **service()** appartenant au software conteneur de la servlet, chargée de recevoir toutes les requêtes client, et les méthodes liées au cycle de vie, **init()** et **destroy()**, qui sont appelées seulement lorsque la servlet est chargée ou déchargée (ce qui arrive rarement).

```
public interface Servlet {
    public void init(ServletConfig config)
        throws ServletException;
    public ServletConfig getServletConfig();
    public void service(ServletRequest req,
        ServletResponse res)
        throws ServletException, IOException;
    public String getServletInfo();
    public void destroy();
}
```

La raison d'être de **getServletConfig()** est de renvoyer un objet **ServletConfig** contenant l'initialisation et les paramètres de départ de cette servlet. La méthode **getServletInfo()** renvoie une chaîne contenant des informations à propos de la servlet, telles que le nom de l'auteur, la version, et le copyright.

La classe **GenericServlet** est une implémentation de cette interface et n'est généralement pas utilisée. La classe **HttpServlet** est une extension de **GenericServlet**, elle est explicitement conçue pour traiter le protocole HTTP. C'est cette classe, **HttpServlet**, que vous utiliserez la plupart du temps.

Les attributs les plus commodes de l'API servlet sont les objets auxiliaires fournis par la classe **HttpServlet**. En regardant la méthode **service()** de l'interface **Servlet**, on constate qu'elle a deux paramètres : **ServletRequest** et **ServletResponse**. Dans la classe **HttpServlet**, deux objets sont développés pour HTTP : **HttpServletRequest** and **HttpServletResponse**. Voici un exemple simple montrant l'utilisation de **HttpServletResponse** :

```
//: c15:servlets:ServletsRule.java
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;

public class ServletsRule extends HttpServlet {
    int i = 0; // «persistance» de Servlet
    public void service(HttpServletRequest req,
        HttpServletResponse res) throws IOException {
        res.setContentType("text/html");
        PrintWriter out = res.getWriter();
        out.print("<HEAD><TITLE>");
        out.print("A server-side strategy");
    }
}
```

```

        out.print( "</TITLE></HEAD><BODY>" );
        out.print( "<h1>Servlets Rule! " + i++ );
        out.print( "</h1></BODY>" );
        out.close();
    }
} ///:~

```

La classe **ServletsRule** est la chose la plus simple que peut recevoir une servlet. La servlet est initialisée au démarrage en appelant sa méthode **init()**, en chargeant la servlet après que le conteneur de la servlet soit chargé. Lorsqu'un client envoie une requête à une URL qui semble reliée à une servlet, le conteneur de servlet intercepte cette demande et effectue un appel de la méthode **service()**, après avoir créé les objets **HttpServletRequest** et **HttpServletResponse**.

La principale responsabilité de la méthode **service()** est d'interagir avec la requête HTTP envoyée par le client, et de construire une réponse HTTP basée sur les attributs contenus dans la demande. La méthode **ServletsRule** se contente de manipuler l'objet réponse sans chercher à savoir ce que voulait le client.

Après avoir mis en place le type du contenu de la réponse (ce qui doit toujours être fait avant d'initialiser **Writer** ou **OutputStream**), la méthode **getWriter()** de l'objet réponse renvoie un objet **PrintWriter**, utilisé pour écrire les données en retour sous forme de caractères (de manière similaire, **getOutputStream()** fournit un **OutputStream**, utilisé pour les réponses binaires, uniquement dans des solutions plus spécifiques).

Le reste du programme se contente d'envoyer une page HTML au client (on suppose que le lecteur comprend le langage HTML, qui n'est pas décrit ici) sous la forme d'une séquence de **Strings**. Toutefois, il faut remarquer l'inclusion du « compteur de passages » représenté par la variable **i**. Il est automatiquement converti en **String** dans l'instruction **print()**.

En lançant le programme, on peut remarquer que la valeur de **i** ne change pas entre les requêtes vers la servlet. C'est une propriété essentielle des servlets : tant qu'il n'existe qu'une servlet d'une classe particulière chargée dans le conteneur, et jamais déchargée (sauf en cas de fin du conteneur de servlet, ce qui ne se produit normalement que si l'on reboote l'ordinateur serveur), tous les champs de cette classe servlet sont des objets persistants ! Cela signifie que vous pouvez sans effort supplémentaire garder des valeurs entre les requêtes à la servlet, alors qu'avec CGI vous auriez dû écrire ces valeurs sur disque afin de les préserver, ce qui aurait demandé du temps supplémentaire et fini par déboucher sur une solution qui n'aurait pas été inter-plate-forme.

Bien entendu, le serveur Web ainsi que le conteneur de servlet doivent de temps en temps être rebootés pour des raisons de maintenance ou après une coupure de courant. Pour éviter de perdre toute information persistante, les méthodes de servlet **init()** et **destroy()** sont appelées automatiquement chaque fois que la servlet est chargée ou déchargée, ce qui nous donne l'opportunité de sauver des données lors d'un arrêt, puis de les restaurer après que la machine ait été rebootée. Le conteneur de la servlet appelle la méthode **destroy()** lorsqu'il se termine lui-même, et on a donc toujours une opportunité de sauver des données essentielles pour peu que la machine serveur soit intelligemment configurée.

L'utilisation de **HttpServlet** pose une autre question. Cette classe fournit les méthodes **doGet()** et **doPost()** qui différencient une soumission « GET » CGI de la part du client, et un « POST » CGI. GET et POST se différencient uniquement par les détails de la manière dont ils soumettent les données, ce qui est une chose que personnellement je préfère ignorer. Toutefois, la plupart des informations publiées que j'ai pu voir semblent recommander la création de méthodes **doGet()** et **doPost()** séparées plutôt qu'une seule méthode générique **service()** qui traiterait les deux cas. Ce favoritisme semble faire l'unanimité, mais je ne l'ai jamais entendu expliquer d'une manière qui me laisserait supposer qu'il s'agit d'autre chose que de l'inertie des programmeurs CGI habitués à porter leur attention sur le fait qu'on doit utiliser un GET ou un POST. Aussi, dans l'esprit de faire les choses les plus simples qui fonctionnent [75], j'utiliserai uniquement la méthode **service()** pour ces

exemples, et laisserai au lecteur le soin de choisir entre les GETs et les POSTs. Gardez toutefois à l'esprit qu'il aurait pu m'arriver d'oublier quelque chose et que cette simple dernière remarque pourrait être une excellente raison d'utiliser de préférence les méthodes **doGet()** et **doPost()**.

Quand un formulaire est soumis à un servlet, **HttpServletRequest** est préchargée avec les données du formulaire présentées sous la forme de paires clef/valeur. Si on connaît le nom des champs, il suffit d'y accéder directement avec la méthode **getParameter()** pour connaître leur valeur. Il est également possible d'obtenir un objet **Enumeration** (l'ancienne forme d'un **Iterator**) vers les noms des champs, ainsi que le montre l'exemple qui suit. Cet exemple montre aussi comment un seul servlet peut être utilisé pour produire à la fois la page contenant le formulaire et la réponse à cette page (on verra plus tard une meilleure solution utilisant les JSP). Si **Enumeration** est vide, c'est qu'il n'y a plus de champs ; cela signifie qu'aucun formulaire n'a été soumis. Dans ce cas, le formulaire est élaboré, et le bouton de soumission rappellera la même servlet. Toutefois les champs sont affichés lorsqu'ils existent.

```

//: c15:servlets:EchoForm.java
// Affiche les couples nom-valeur d'un formulaire HTML
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;
import java.util.*;

public class EchoForm extends HttpServlet {
    public void service(HttpServletRequest req,
        HttpServletResponse res) throws IOException {
        res.setContentType("text/html");
        PrintWriter out = res.getWriter();
        Enumeration flds = req.getParameterNames();
        if(!flds.hasMoreElements()) {
            // Pas de formulaire soumis -- on en crée un:
            out.print("<html>");
            out.print("<form method=\"POST\" \" +
                \" action=\"EchoForm\">");
            for(int i = 0; i < 10; i++)
                out.print("<b>Field" + i + "</b> \" +
                    "<input type=\"text\" \" +
                    \" size=\"20\" name=\"Field\" + i +
                    \" value=\"Value\" + i + \"><br>");
            out.print("<INPUT TYPE=submit name=submit\" +
                \" Value=\"Submit\"></form></html>");
        } else {
            out.print("<h1>Your form contained:</h1>");
            while(flds.hasMoreElements()) {
                String field= (String)flds.nextElement();
                String value= req.getParameter(field);
                out.print(field + " = " + value+ "<br>");
            }
        }
        out.close();
    }
} //::~~

```

On peut penser en lisant cela que Java ne semble pas conçu pour traiter des chaînes de caractères car le formatage des pages à renvoyer est pénible à cause des retours à la ligne, des séquences escape, et du signe + inévitable dans la construction des objets **String**. Il n'est pas raisonnable de coder une page HTML quelque peu

conséquence en Java. Une des solutions est de préparer la page en tant que fichier texte séparé, puis de l'ouvrir et de la passer au serveur Web. S'il fallait de plus effectuer des substitutions de chaînes dans le contenu de la page, ce n'est guère mieux car le traitement des chaînes en Java est très pauvre. Si vous rencontrez un de ces cas, il serait préférable d'adopter une solution mieux appropriée (mon choix irait vers Python ; voici une version incluse dans un programme Java appelé JPython) qui génère une page-réponse.

Les Servlets et le multithreading

Le conteneur de servlet dispose d'un ensemble de threads qu'il peut lancer pour traiter les demandes des clients. On peut imaginer cela comme si deux clients arrivant au même moment étaient traités simultanément par la méthode **service()**. En conséquence la méthode **service()** doit être écrite d'une manière sécurisée dans un contexte de thread. Tous les accès aux ressources communes (fichiers, bases de données) demandent à être protégés par le mot clef **synchronized**.

L'exemple très simple qui suit utilise une clause **synchronized** autour de la méthode **sleep()** du thread. En conséquence les autres threads seront bloqués jusqu'à ce que le temps imparti (cinq secondes) soit écoulé. Pour tester cela il faut lancer plusieurs instances d'un navigateur puis lancer ce servlet aussi vite que possible ; remarquez alors que chacun d'eux doit attendre avant de voir le jour.

```

//: c15:servlets:ThreadServlet.java
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;

public class ThreadServlet extends HttpServlet {
    int i;
    public void service(HttpServletRequest req,
        HttpServletResponse res) throws IOException {
        res.setContentType("text/html");
        PrintWriter out = res.getWriter();
        synchronized(this) {
            try {
                Thread.currentThread().sleep(5000);
            } catch (InterruptedException e) {
                System.err.println("Interrupted");
            }
        }
        out.print("<h1>Finished " + i++ + "</h1>");
        out.close();
    }
} //::~~

```

On peut aussi synchroniser complètement la servlet en mettant le mot clef **synchronized** juste avant la méthode **service()**. En réalité, l'unique justification pour utiliser la clause **synchronized** à la place de cela est lorsque la section critique se trouve dans un chemin d'exécution qui ne doit pas être exécuté. Dans un tel cas, il serait préférable d'éviter la contrainte de synchronisation à chaque fois en utilisant une clause **synchronized**. Sinon, chaque thread particulier devrait systématiquement attendre, il vaut donc mieux synchroniser la méthode en entier.

Gérer des sessions avec les servlets

HTTP est un protocole qui ne possède pas la notion de session, on ne peut donc savoir d'un appel serveur à un autre s'il s'agit du même appelant ou s'il s'agit d'une personne complètement différente. Beaucoup d'efforts ont

été faits pour créer des mécanismes permettant aux développeurs Web de suivre les sessions. À titre d'exemple, les compagnies ne pourraient pas faire de e-commerce si elles ne gardaient pas la trace d'un client, ainsi que les renseignements qu'il a saisi sur sa liste de courses.

Il existe plusieurs méthodes pour suivre une session, mais la plus commune utilise les cookies persistants, qui font intégralement partie du standard Internet. Le HTTP Working Group de l'Internet Engineering Task Force a décrit les cookies du standard officiel dans RFC 2109 (*ds.internic.net/rfc/rfc2109.txt* ou voir *www.cookiecentral.com*).

Un cookie n'est pas autre chose qu'une information de petite taille envoyée par un serveur Web à un navigateur. Le navigateur sauvegarde ce cookie sur le disque local, puis lors de chaque appel à l'URL associée au cookie, ce dernier est envoyé de manière transparente en même temps que l'appel, fournissant ainsi au serveur l'information désirée en retour (en lui fournissant généralement d'une certaine manière votre identité). Toutefois les clients peuvent inhiber la capacité de leur navigateur à accepter les cookies. Si votre site doit suivre un client qui a inhibé cette possibilité, alors une autre méthode de suivi de session doit être intégrée à la main (réécriture d'URL ou champs cachés dans un formulaire), car les fonctionnalités de suivi de session intégrées à l'API servlet sont construites autour des cookies.

La classe Cookie

L'API servlet (à partir de la version 2.0) fournit la classe **Cookie**. Cette classe inclut tous les détails de l'en-tête HTTP et permet de définir différents attributs de cookie. L'utilisation d'un cookie consiste simplement à l'ajouter à l'objet réponse. Le constructeur a deux arguments, le premier est un nom du cookie et le deuxième une valeur. Les cookies sont ajoutés à l'objet réponse avant que l'envoi ne soit effectif.

```
Cookie oreo = new Cookie("TIJava", "2000");  
res.addCookie(cookie);
```

Les cookies sont récupérés en appelant la méthode **getCookies()** de l'objet **HttpServletRequest**, qui renvoie un tableau d'objets **Cookie**.

```
Cookie[] cookies = req.getCookies();
```

En appelant **getValue()** pour chaque cookie, on obtient une **String** initialisée avec le contenu du cookie. Dans l'exemple ci-dessus, **getValue("TIJava")** renverrait une String contenant 2000.

La classe Session

Une session consiste en une ou plusieurs requêtes de pages adressées par un client à un site Web durant une période définie. Par exemple, si vous faites vos courses en ligne, la session sera la période démarrant au moment où vous ajoutez un achat dans votre panier jusqu'au moment où vous envoyez effectivement la demande. Chaque achat ajouté au panier déclenchera une nouvelle connexion HTTP, qui n'a aucun rapport ni avec les connexions précédentes ni avec les achats déjà inclus dans votre panier. Pour compenser ce manque d'information, les mécanismes fournis par la spécification des cookies permet au servlet de suivre la session.

Un objet servlet **Session** réside du côté serveur sur le canal de communication ; son rôle est de capturer les données utiles à propos du client pendant qu'il navigue sur votre site Web et qu'il interagit avec lui. Ces données peuvent être pertinentes pour la session actuelle, comme les achats dans le panier, ou bien peuvent être des informations d'authentification fournies lors de l'accès du client au site Web, et qu'il n'y a pas lieu de donner à nouveau durant un ensemble particulier de transactions.

La classe **Session** de l'API servlet utilise la classe **Cookie** pour effectuer ce travail. Toutefois, l'objet **Session** n'a besoin que d'une sorte d'identifiant unique stocké chez le client et passé au serveur. Les sites Web peuvent aussi utiliser les autres systèmes de suivi de session mais ces mécanismes sont plus difficiles à mettre en oeuvre car ils n'existent pas dans l'API servlet (ce qui signifie qu'on doit les écrire à la main pour traiter le cas où le client n'accepte pas les cookies).

Voici un exemple implémentant le suivi de session au moyen de l'API servlet :

```

//: c15:servlets:SessionPeek.java
// Utilise la classe HttpSession.
import java.io.*;
import java.util.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class SessionPeek extends HttpServlet {
    public void service(HttpServletRequest req,
        HttpServletResponse res)
        throws ServletException, IOException {
        // Obtenir l'Objet Session avant tout
        // envoi vers le client.
        HttpSession session = req.getSession();
        res.setContentType("text/html");
        PrintWriter out = res.getWriter();
        out.println("<HEAD><TITLE> SessionPeek ");
        out.println(" </TITLE></HEAD><BODY>");
        out.println("<h1> SessionPeek </h1>");
        // Un simple compteur pour cette session.
        Integer ival = (Integer)
            session.getAttribute("sesspeek.cntr");
        if(ival==null)
            ival = new Integer(1);
        else
            ival = new Integer(ival.intValue() + 1);
        session.setAttribute("sesspeek.cntr", ival);
        out.println("You have hit this page <b>"
            + ival + "</b> times.<p>");
        out.println("<h2>");
        out.println("Saved Session Data </h2>");
        // Boucler au travers de toutes les données de la session:
        Enumeration sesNames =
            session.getAttributeNames();
        while(sesNames.hasMoreElements()) {
            String name =
                sesNames.nextElement().toString();
            Object value = session.getAttribute(name);
            out.println(name + " = " + value + "<br>");
        }
        out.println("<h3> Session Statistics </h3>");
        out.println("Session ID: "
            + session.getId() + "<br>");
        out.println("New Session: " + session.isNew()
            + "<br>");
        out.println("Creation Time: "
            + session.getCreationTime());
    }
}

```

```

        out.println("<I>( " +
            new Date(session.getCreationTime())
            + " )</I><br>");
        out.println("Last Accessed Time: " +
            session.getLastAccessedTime());
        out.println("<I>( " +
            new Date(session.getLastAccessedTime())
            + " )</I><br>");
        out.println("Session Inactive Interval: "
            + session.getMaxInactiveInterval());
        out.println("Session ID in Request: "
            + req.getRequestSessionId() + "<br>");
        out.println("Is session id from Cookie: "
            + req.isRequestedSessionIdFromCookie()
            + "<br>");
        out.println("Is session id from URL: "
            + req.isRequestedSessionIdFromURL()
            + "<br>");
        out.println("Is session id valid: "
            + req.isRequestedSessionIdValid()
            + "<br>");
        out.println("</BODY>");
        out.close();
    }
    public String getServletInfo() {
        return "A session tracking servlet";
    }
} ///:~

```

À l'intérieur de la méthode **service()**, la méthode **getSession()** est appelée pour l'objet requête et renvoie un objet **Session** associé à la requête. L'objet **Session** ne voyage pas sur le réseau, il réside sur le serveur et est associé à un client et à ses requêtes.

La méthode **getSession()** possède deux versions : sans paramètres, ainsi qu'elle est utilisée ici, et **getSession(boolean)**. L'appel de **getSession(true)** est équivalent à **getSession()**. Le **boolean** sert à indiquer si on désire créer l'objet session lorsqu'on ne le trouve pas. L'appel le plus probable est **getSession(true)**, d'où la forme **getSession()**.

L'objet **Session**, s'il n'est pas nouveau, nous donne des informations sur le client à partir de visites antérieures. Si l'objet **Session** est nouveau alors le programme commencera à recueillir des informations à propos des activités du client lors de cette visite. Le recueil de cette information est effectué au moyen des méthodes **setAttribute()** et **getAttribute()** de l'objet session.

```

java.lang.Object getAttribute(java.lang.String)
void setAttribute(java.lang.String name,
                  java.lang.Object value)

```

L'objet **Session** utilise une simple paire nom/valeur pour garder l'information. Le nom est du type **String**, et la valeur peut être n'importe quel objet dérivé de **java.lang.Object**. **SessionPeek** garde la trace du nombre de fois où le client est revenu pendant cette session, au moyen d'un objet **Integer** nommé **sesspeek.cntr**. Si le nom n'existe pas on crée un **Integer** avec une valeur de un, sinon on crée un **Integer** en incrémentant la valeur du précédent. Le nouvel **Integer** est rangé dans l'objet **Session**. Si on utilise la même clef dans un appel à **setAttribute()**, le nouvel objet écrase l'ancien. Le compteur incrémenté sert à afficher le nombre de visites du client pendant cette session.

La méthode **getAttributeNames()** est en relation avec **getAttribute()** et **setAttribute()** et renvoie une énumération des noms des objets associés à l'objet **Session**. Une boucle **while** de **SessionPeek** montre cette méthode en action.

Vous vous interrogez sans doute sur la durée de vie d'un objet **Session**. La réponse dépend du conteneur de servlet qu'on utilise ; généralement la durée de vie par défaut est de 30 minutes (1800 secondes), ce que l'on peut voir au travers de l'appel de **getMaxInactiveInterval()** par **ServletPeek**. Les tests semblent montrer des résultats différents suivant le conteneur de servlet utilisé. De temps en temps l'objet **Session** peut faire le tour du cadran, mais je n'ai jamais rencontré de cas où l'objet **Session** disparaît avant que le temps spécifié par « inactive interval » soit écoulé. On peut tester cela en initialisant « inactive interval » à 5 secondes au moyen de **setMaxInactiveInterval()** puis voir si l'objet **Session** est toujours là ou au contraire a été détruit à l'heure déterminée. Il se pourrait que vous ayez à étudier cet attribut lorsque vous choisirez un conteneur de servlet.

Faire fonctionner les exemples de servlet

Si vous ne travaillez pas encore sur un serveur d'applications gérant les servlets Sun ainsi que les technologies JSP, il vous faudra télécharger l'implémentation Tomcat des servlets Java et des JSP, qui est une implémentation libre et « open-source » des servlets, et de plus l'implémentation officielle de référence de Sun. Elle se trouve à jakarta.apache.org.

Suivez les instructions d'installation de l'implémentation Tomcat, puis éditez le fichier **server.xml** pour décrire l'emplacement de votre répertoire qui contiendra vos servlets. Une fois lancé le programme Tomcat vous pouvez tester vos programmes servlet.

Ceci n'était qu'une brève introduction aux servlets ; il existe des livres entiers traitant de ce sujet. Toutefois, cette introduction devrait vous donner suffisamment d'idées pour démarrer. De plus, beaucoup de thèmes développés dans la section suivante ont une compatibilité ascendante avec les servlets.

Les Pages Java Serveur - Java Server Pages

Les Java Server Pages (JSP) sont une extension standard Java définie au-dessus des extensions servlet. Le propos des JSP est de simplifier la création et la gestion des pages Web dynamiques.

L'implémentation de référence Tomcat, déjà mentionnée et disponible librement sur jakarta.apache.org, supporte automatiquement les JSP.

Les JSP permettent de mélanger le code HTML d'une page Web et du code Java dans le même document. Le code Java est entouré de tags spéciaux qui indiquent au conteneur JSP qu'il doit utiliser le code pour générer une servlet complètement ou en partie. L'avantage que procurent les JSP est de maintenir un seul document qui est à la fois la page HTML et le code Java qui la gère. Le revers est que celui qui maintient la page JSP doit être autant qualifié en HTML qu'en Java (toutefois, des environnements GUI de construction de JSP devraient apparaître sur le marché).

La première fois qu'une JSP est chargée par un conteneur JSP (qui est typiquement associé à un serveur Web, ou bien en fait partie), le code nécessaire à la réalisation des tags JSP est automatiquement généré, compilé, et chargé dans le conteneur de la servlet. Les parties invariables de la page HTML sont créées en envoyant des objets **String** à **write()**. Les parties dynamiques sont incluses directement dans la servlet.

À partir de là, et tant que le code source JSP de la page n'est pas modifié, tout se passe comme si on avait une page HTML statique associée à des servlets (en réalité, le code HTML est généré par la servlet). Si on modifie

le code source de la JSP, il est automatiquement recompilé et rechargé dès que cette page sera redemandée. Bien entendu, à cause de ce dynamisme le temps de réponse sera long lors du premier accès à une JSP. Toutefois, étant donné qu'une JSP est généralement plus utilisée qu'elle n'est modifiée, on ne sera pas généralement affecté par ce délai.

La structure d'une page JSP est à mi-chemin d'une servlet et d'une page HTML. Les tags JSP commencent et finissent comme les tags HTML, sauf qu'ils utilisent également le caractère pourcent (%), ainsi tous les tags JSP ont cette structure :

```
<% ici, le code JSP %>
```

Le premier caractère pourcent doit être suivi d'un autre caractères qui

Voici un un exemple extrêmement simple de JSP utilisant un appel standard à une bibliothèque Java pour récupérer l'heure courante en millisecondes, et diviser le résultat par 1000 pour produire l'heure en secondes. Une *expression JSP* (<%=) est utilisée, puis le résultat du calcul est mis dans une **String** et intégré à la page Web générée :

```
//:! c15:jsp:ShowSeconds.jsp
<html><body>
<H1>The time in seconds is:
<%= System.currentTimeMillis()/1000 %></H1>
</body></html>
///:~
```

Dans les exemples JSP de ce livre, la première et la dernière ligne ne font pas partie du fichier code réel qui est extrait et placé dans l'arborescence du code source de ce livre.

Lorsque le client demande une page JSP, le serveur Web doit avoir été configuré pour relayer la demande vers le conteneur JSP, qui à son tour appelle la page. Comme on l'a déjà dit plus haut, lors du premier appel de la page, les composants spécifiés par la page sont générés et compilés par le conteneur JSP en tant qu'une ou plusieurs servlets. Dans les exemples ci-dessus, la servlet doit contenir le code destiné à configurer l'objet **HttpServletResponse**, produire un objet **PrintWriter** (toujours nommé **out**), et enfin transformer le résultat du calcul en un objet **String** qui sera envoyé vers **out**. Ainsi qu'on peut le voir, tout ceci est réalisé au travers d'une instruction très succincte, mais en moyenne les programmeurs HTML/concepteurs de site Web ne seront pas qualifiés pour écrire un tel code.

Les objets implicites

Les servlets comportent des classes fournissant des utilitaires pratiques, comme **HttpServletRequest**, **HttpServletResponse**, **Session**, etc. Les objets de ces classes font partie de la spécification JSP et sont automatiquement disponibles pour vos JSP sans avoir à écrire une ligne de code supplémentaire. Les objets implicites d'une JSP sont décrits dans le tableau ci-dessous.

Variable implicite	Du Type (javax.servlet)	Description
demande (request)	Sous-type de HttpServletRequest dépendant du protocole	La demande qui déclenche l'appel du service.
réponse	Sous-type de HttpServletResponse dépendant du protocole	La réponse à la demande.
pageContext	jsp.PageContext	Le contexte de page encapsule les choses qui dépendent de l'implémentation et fournissent des méthodes communes à un espace de nommage pour ce JSP.
session	Sous-type de http.HttpSession dépendant du protocole	L'objet session créé pour le client demandeur. Voir l'API des servlets.
application	ServletContext	Le contexte de servlet obtenu depuis l'objet configuré : getServletConfig() , getContext() .
out	jsp.JspWriter	L'objet qui écrit dans le flux sortant.
config	ServletConfig	Le ServletConfig pour ce JSP.
page	java.lang.Object	L'instance de cette classe d'implémentation de page courante.

La visibilité de chaque objet est extrêmement variable. Par exemple, l'objet **session** a une visibilité qui dépasse la page, car il englobe plusieurs demandes client et plusieurs pages. L'objet **application** peut fournir des services à un groupe de pages JSP représentant une application Web.

Les directives JSP

Les directives sont des messages adressés au conteneur de JSP et sont reconnaissables au caractère `@` :

```
<%@ directive {attr="value"}* %>
```

Les directives n'envoient rien sur le flux **out**, mais sont importantes lorsqu'on définit les attributs et les dépendances de pages avec le conteneur JSP. Par exemple, la ligne :

```
<%@ page language="java" %>
```

exprime le fait que le langage de scripting utilisé dans la page JSP est Java. En fait, la spécification JSP décrit *seulement* la sémantique des scripts pour un attribut de langage égal à `Java`. La raison d'être de cette directive est d'introduire la flexibilité dans la technologie JSP. Dans le futur, si vous aviez à choisir un autre langage, par exemple Python (un bon choix de langage de scripting), alors ce langage devra supporter le Java Run-time Environment en exposant la technologie du modèle objet Java à l'environnement de scripting, en particulier les variables implicites définies plus haut, les propriétés JavaBeans, et les méthodes publiques.

La directive la plus importante est la directive de page. Elle définit un certain nombre d'attributs dépendant de la page et les communique au conteneur JSP. Parmi ces attributs : **language**, **extends**, **import**, **session**, **buffer**, **autoFlush**, **isThreadSafe**, **info** et **errorPage**. Par exemple :

```
<%@ page session= true import= java.util.* %>
```

Cette ligne indique tout d'abord que la page nécessite une participation à une session HTTP. Puisque nous n'avons pas décrit de directive de langage le conteneur JSP utilisera Java par défaut et la variable de langage de scripting implicite appelée `session` sera du type **`javax.servlet.http.HttpSession`**. Si la directive avait été `false` alors la variable implicite **`session`** n'aurait pas été disponible. Si la variable **`session`** n'est pas spécifiée, sa valeur par défaut est `true`.

L'attribut **`import`** décrit les types disponibles pour l'environnement de scripting. Cet attribut est utilisé tout comme il le serait dans le langage de programmation Java, c'est à dire une liste d'expressions **`import`** ordinaires séparées par des virgules. Cette liste est importée par l'implémentation de la page JSP traduite et reste disponible pour l'environnement de scripting. À nouveau, ceci est actuellement défini uniquement lorsque la valeur de la directive de langage est `java`.

Les éléments de scripting JSP

Une fois l'environnement de scripting mis en place au moyen des directives on peut utiliser les éléments du langage de scripting. JSP 1.1 possède trois éléments de langage de scripting *declaration*, *scriptlet*, et *expression*. Une déclaration déclare des éléments, un scriptlet est un fragment d'instruction, et une expression est une expression complète du langage. En JSP chaque élément de scripting commence par `<%`. Voici la syntaxe de chacun :

```
<%! declaration %>
<% scriptlet %>
<%= expression %>
```

L'espace est facultatif après `<%!`, `<%`, `<%=`, et avant `%>`.

Tous ces tags s'appuient sur la norme XML ; on pourrait dire qu'une page JSP est un document XML. La syntaxe équivalente en XML pour les éléments de scripting ci-dessus serait :

```
<jsp:declaration> declaration </jsp:declaration>
<jsp:scriptlet> scriptlet </jsp:scriptlet>
<jsp:expression> expression </jsp:expression>
```

De plus, il existe deux types de commentaires :

```
<%-- jsp comment --%>
<!-- html comment -->
```

La première forme crée dans les pages sources JSP des commentaires qui n'apparaîtront jamais dans la page HTML envoyée au client. Naturellement, la deuxième forme n'est pas spécifique à JSP, c'est un commentaire HTML ordinaire. Ceci est intéressant car on peut insérer du code JSP dans un commentaire HTML, le commentaire étant inclus dans la page résultante ainsi que le résultat du code JSP.

Les déclarations servent à déclarer des variables et des méthodes dans les langages de scripting utilisés dans une page JSP (uniquement Java pour le moment). La déclaration doit être une instruction Java complète et ne doit pas écrire dans le flux **`out`**. Dans l'exemple ci-dessous **`Hello.jsp`**, les déclarations des variables **`loadTime`**, **`loadDate`** et **`hitCount`** sont toutes des instructions Java complètes qui déclarent et initialisent de nouvelles variables.

```
//:! c15:jsp:Hello.jsp
<%-- This JSP comment will not appear in the
```

```

generated html --%>
<!-- This is a JSP directive: --%>
<%@ page import="java.util.*" %>
<!-- These are declarations: --%>
<%!
    long loadTime= System.currentTimeMillis();
    Date loadDate = new Date();
    int hitCount = 0;
%>
<html><body>
<!-- The next several lines are the result of a
JSP expression inserted in the generated html;
the '=' indicates a JSP expression --%>
<H1>This page was loaded at <%= loadDate %> </H1>
<H1>Hello, world! It's <%= new Date() %></H1>
<H2>Here's an object: <%= new Object() %></H2>
<H2>This page has been up
<%= (System.currentTimeMillis()-loadTime)/1000 %>
seconds</H2>
<H3>Page has been accessed <%= ++hitCount %>
times since <%= loadDate %></H3>
<!-- A "scriptlet" that writes to the server
console and to the client page.
Note that the ';' is required: --%>
<%
    System.out.println("Goodbye");
    out.println("Cheerio");
%>
</body></html>
///:~

```

Lorsque ce programme fonctionne, on constate que les variables **loadTime**, **loadDate** et **hitCount** gardent leurs valeurs entre les appels de la page, il s'agit donc clairement de champs et non de variables locales.

À la fin de l'exemple un scriptlet écrit `Goodbye` sur la console du serveur Web et `Cheerio` sur l'objet **JspWriter** implicite **out**. Les scriptlets peuvent contenir tout fragment de code composé d'instructions Java valides. Les scriptlets sont exécutés au moment du traitement de la demande. Lorsque tous les fragments de scriptlet d'une page JSP donnée sont combinés dans l'ordre où ils apparaissent dans la page JSP, ils doivent contenir une instruction valide telle que définie dans le langage de programmation Java. Le fait qu'ils écrivent ou pas dans le flux **out** dépend du code du scriptlet. Il faut garder à l'esprit que les scriptlets peuvent produire des effets de bord en modifiant les objets se trouvant dans leur champ de visibilité.

Les expressions JSP sont mêlées au code HTML dans la section médiane de **Hello.jsp**. Les expressions doivent être des instructions Java complètes, qui sont évaluées, traduites en **String**, et envoyées à **out**. Si le résultat d'une expression ne peut pas être traduit en **String** alors une exception **ClassCastException** est lancée.

Extraire des champs et des valeurs

L'exemple suivant ressemble à un autre vu précédemment dans la section servlet. La première fois que la page est appelée il détecte s'il n'existe pas de champ et renvoie une page contenant un formulaire, en utilisant le même code que dans l'exemple de la servlet, mais au format JSP. Lorsque le formulaire contenant des champs remplis est envoyé à la même URL JSP, il détecte les champs et les affiche. C'est une technique agréable parce qu'elle permet d'avoir dans un seul fichier, le formulaire destiné au client et le code de réponse pour cette page, ce qui rend les choses plus simples à créer et maintenir.


```

//:!! c15:jsp:DisplayFormData.jsp
<!-- Fetching the data from an HTML form. -->
<!-- This JSP also generates the form. -->
<%@ page import="java.util.*" %>
<html><body>
<H1>DisplayFormData</H1><H3>
<%
    Enumeration flds = request.getParameterNames();
    if(!flds.hasMoreElements()) { // No fields %>
        <form method="POST"
            action="DisplayFormData.jsp">
<%     for(int i = 0; i < 10; i++) { %>
            Field<%=i%>: <input type="text" size="20"
                name="Field<%=i%>" value="Value<%=i%>"><br>
<%     } %>
            <INPUT TYPE=submit name=submit
                value="Submit"></form>
<% } else {
        while(flds.hasMoreElements()) {
            String field = (String)flds.nextElement();
            String value = request.getParameter(field);
%>
            <li><%= field %> = <%= value %></li>
<%     }
    } %>
</H3></body></html>
//:~

```

Ce qui est intéressant dans cet exemple est de montrer comment le code scriptlet et le code HTML peuvent être entremêlés, au point de générer une page HTML à l'intérieur d'une boucle Java **for**. En particulier ceci est très pratique pour construire tout type de formulaire qui sans cela nécessiterait du code HTML répétitif.

Attributs et visibilité d'une page JSP

En cherchant dans la documentation HTML des servlets et des JSP, on trouve des fonctionnalités donnant des informations à propos de la servlet ou de la page JSP actuellement en cours. L'exemple suivant montre quelques-unes de ces données.

```

//:!! c15:jsp:PageContext.jsp
<!--Viewing the attributes in the pageContext-->
<!-- Note that you can include any amount of code
inside the scriptlet tags -->
<%@ page import="java.util.*" %>
<html><body>
Servlet Name: <%= config.getServletName() %><br>
Servlet container supports servlet version:
<% out.print(application.getMajorVersion() + "."
+ application.getMinorVersion()); %><br>
<%
    session.setAttribute("My dog", "Ralph");
    for(int scope = 1; scope <= 4; scope++) { %>
        <H3>Scope: <%= scope %> </H3>
<%     Enumeration e =

```



```

        pageContext.getAttributeNamesInScope(scope);
    while(e.hasMoreElements()) {
        out.println("\t<li>" +
            e.nextElement() + "</li>");
    }
}
%>
</body></html>
///  


```

Cet exemple montre également l'utilisation du mélange de code HTML et d'écriture sur **out** pour fabriquer la page HTML résultante.

La première information générée est le nom de la servlet, probablement `JSP` mais cela dépend de votre implémentation. On peut voir également la version courante du conteneur de servlet au moyen de l'objet `application`. Pour finir, après avoir déclaré les attributs de la session, les noms d'attributs sont affichés avec une certaine visibilité. On n'utilise pas beaucoup les visibilités dans la plupart des programmes JSP ; on les a montré ici simplement pour donner de l'intérêt à l'exemple. Il existe quatre attributs de visibilité, qui sont : la *visibilité de page* (visibilité 1), la *visibilité de demande* (visibilité 2), la *visibilité de session* (visibilité 3 : ici, le seul élément disponible dans la visibilité de session est `My dog`, ajouté juste après la boucle **for**), et la *visibilité d'application* (visibilité 4), basée sur l'objet **ServletContext**. Il existe un **ServletContext** pour chaque application Web tournant sur une Machine Virtuelle Java (une application Web est une collection de servlets et de contenus placés dans un sous-ensemble spécifique de l'espace de nommage de l'URL serveur tel que `/catalog`. Ceci est généralement réalisé au moyen d'un fichier de configuration). Au niveau de visibilité de l'application on peut voir des objets représentant les chemins du répertoire de travail et du répertoire temporaire.

Manipuler les sessions en JSP

Les sessions ont été introduites dans les sections précédentes à propos des servlets, et sont également disponibles dans les JSP. L'exemple suivant utilise l'objet **session** et permet de superviser le temps au bout duquel la session deviendra invalide.

```

///  

c15:jsp:SessionObject.jsp
<!--Getting and setting session object values-->
<html><body>
<H1>Session id: <%= session.getId() %></H1>
<H3><li>This session was created at
<%= session.getCreationTime() %></li></H1>
<H3><li>Old MaxInactiveInterval =
    <%= session.getMaxInactiveInterval() %></li>
<% session.setMaxInactiveInterval(5); %>
<li>New MaxInactiveInterval=
    <%= session.getMaxInactiveInterval() %></li>
</H3>
<H2>If the session object "My dog" is
still around, this value will be non-null:<H2>
<H3><li>Session value for "My dog" =
<%= session.getAttribute("My dog") %></li></H3>
<!-- Now add the session object "My dog" -->
<% session.setAttribute("My dog",
    new String("Ralph")); %>
<H1>My dog's name is
<%= session.getAttribute("My dog") %></H1>
<!-- See if "My dog" wanders to another form -->

```

```

<FORM TYPE=POST ACTION=SessionObject2.jsp>
<INPUT TYPE=submit name=submit
Value="Invalidate"></FORM>
<FORM TYPE=POST ACTION=SessionObject3.jsp>
<INPUT TYPE=submit name=submit
Value="Keep Around"></FORM>
</body></html>
///  


```

L'objet **session** est fourni par défaut, il est donc disponible sans code supplémentaire. Les appels de **getID()**, **getCreationTime()** et **getMaxInactiveInterval()** servent à afficher des informations sur l'objet session.

Quand on ouvre la session pour la première fois on a, par exemple, **MaxInactiveInterval** égal à 1800 secondes (30 minutes). Ceci dépend de la configuration du conteneur JSP/servlet. **MaxInactiveInterval** est ramené à 5 secondes afin de rendre les choses intéressantes. Si on rafraîchit la page avant la fin de l'intervalle de 5 secondes, alors on voit :

```
Session value for "My dog" = Ralph
```

Mais si on attend un peu plus longtemps, alors **Ralph** devient **null**.

Pour voir comment les informations de sessions sont répercutées sur les autres pages, ainsi que pour comparer le fait d'invalider l'objet session à celui de le laisser se terminer, deux autres JSP sont créées. La première (qu'on atteint avec le bouton **invalidate** de **SessionObject.jsp**) lit l'information de session et invalide explicitement cette session :

```

///  

c15:jsp:SessionObject2.jsp
<!--The session object carries through-->
<html><body>
<H1>Session id: <%= session.getId() %></H1>
<H1>Session value for "My dog"
<%= session.getValue("My dog") %></H1>
<% session.invalidate(); %>
</body></html>
///  


```

Pour tester cet exemple, rafraîchir **SessionObject.jsp**, puis cliquer immédiatement sur le bouton **invalidate** pour activer **SessionObject2.jsp**. À ce moment on voit toujours **Ralph**, immédiatement (avant que l'intervalle de 5 secondes ait expiré). Rafraîchir **SessionObject2.jsp** pour voir que la session a été invalidée manuellement et que **Ralph** a disparu.

En recommençant avec **SessionObject.jsp**, rafraîchir la page ce qui démarre un nouvel intervalle de 5 secondes, puis cliquer sur le bouton « **Keep Around** », ce qui nous amène à la page suivante, **SessionObject3.jsp**, qui N'invalide PAS la session :

```

///  

c15:jsp:SessionObject3.jsp
<!--The session object carries through-->
<html><body>
<H1>Session id: <%= session.getId() %></H1>
<H1>Session value for "My dog"
<%= session.getValue("My dog") %></H1>
<FORM TYPE=POST ACTION=SessionObject.jsp>
<INPUT TYPE=submit name=submit Value="Return">

```

```

</FORM>
</body></html>
///:~

```

Dû au fait que cette page n'invalide pas la session, `Ralph` est toujours là aussi longtemps qu'on rafraîchit la page avant la fin de l'intervalle de 5 secondes. Ceci n'est pas sans ressembler à un `Tomagotchi`, et `Ralph` restera là tant que vous jouerez avec lui, sinon il disparaîtra.

Créer et modifier des cookies

Les cookies ont été introduits dans la section précédente concernant les servlets. Ici encore, la concision des JSP rend l'utilisation des cookies plus simple que dans les servlets. L'exemple suivant montre [cela](#) en piégeant les cookies liés à une demande en entrée, en lisant et modifiant leur date d'expiration, et en liant un [nouveau](#) cookie à la réponse :

```

//: c15:jsp:Cookies.jsp
<!--This program has different behaviors under
different browsers! -->
<html><body>
<H1>Session id: <%= session.getId() %></H1>
<%
Cookie[] cookies = request.getCookies();
for(int i = 0; i < cookies.length; i++) { %>
    Cookie name: <%= cookies[i].getName() %> <br>
    value: <%= cookies[i].getValue() %><br>
    Old max age in seconds:
    <%= cookies[i].getMaxAge() %><br>
    <% cookies[i].setMaxAge(5); %>
    New max age in seconds:
    <%= cookies[i].getMaxAge() %><br>
<% } %>
<%! int count = 0; int dcount = 0; %>
<% response.addCookie(new Cookie(
    "Bob" + count++, "Dog" + dcount++)); %>
</body></html>
///:~

```

Chaque navigateur ayant sa manière de stocker les cookies, le résultat sera différent suivant le navigateur (ce qui n'est pas rassurant, mais peut-être réparerez-vous un certain nombre de bugs en lisant cela). Par ailleurs, il se peut aussi que l'on ait des résultats différents en arrêtant le navigateur et en le relançant, plutôt que de visiter une autre page puis de revenir à **Cookies.jsp**. Remarquons que l'utilisation des objets session semble plus robuste que l'utilisation directe des cookies.

Après l'affichage de l'identifiant de session, chaque cookie du tableau de cookies arrivant avec l'objet **request** object est affiché, ainsi que sa date d'expiration. La date d'expiration est modifiée et affichée à son tour pour vérifier la nouvelle valeur, puis un nouveau cookie est ajouté à la réponse. Toutefois, il est possible que votre navigateur semble ignorer les dates d'expiration ; il est préférable de jouer avec ce programme en modifiant la date d'expiration pour voir ce qui se passe avec divers navigateurs.

Résumé sur les JSP

Cette section était un bref aperçu des JSP ; cependant avec les sujets abordés ici (ainsi qu'avec le langage Java appris dans le reste du livre, sans oublier votre connaissance personnelle du langage HTML) vous pouvez dès à présent écrire des pages Web sophistiquées via les JSP. La syntaxe JSP n'est pas particulièrement profonde ni compliquée, et si vous avez compris ce qui était présenté dans cette section vous êtes prêts à être productifs en utilisant les JSP. Vous trouverez d'autres informations dans la plupart des livres sur les servlets, ou bien à java.sun.com.

La disponibilité des JSP est très agréable, même lorsque votre but est de produire des servlets. Vous découvrirez que si vous vous posez une question à propos du comportement d'une fonctionnalité servlet, il est plus facile et plus rapide d'y répondre en écrivant un programme de test JSP qu'en écrivant une servlet. Ceci est dû en partie au fait qu'on ait moins de code à écrire et qu'on puisse mélanger le code Java et le code HTML, mais l'avantage devient particulièrement évident lorsqu'on voit que le Conteneur JSP se charge de la recompilation et du chargement du JSP à votre place chaque fois que la source est modifiée.

Toutefois, aussi fantastiques que soient les JSP, il vaut mieux garder à l'esprit que la création de pages JSP requiert un plus haut niveau d'habileté que la simple programmation en Java ou la simple création de pages Web. En outre, debugger une page JSP morcelée n'est pas aussi facile que débbugger un programme Java, car (pour le moment) les messages d'erreur sont assez obscurs. Cela changera avec l'évolution des systèmes de développement, et peut-être verrons nous d'autres technologies construites au-dessus de Java plus adaptées aux qualités des concepteurs de site web.

Suite du chapitre 15 : RMI (Remote Method Invocation)

Exercices

On trouvera les solutions des exercices sélectionnés dans le document électronique *The Thinking in Java Annotated Solution Guide*, disponible pour une participation minimale à www.BruceEckel.com.

1. Compiler et lancer les programmes **JabberServer** et **JabberClient** de ce chapitre. Éditer ensuite les fichiers pour supprimer les « buffering » d'entrée et de sortie, compiler et relancer, observer le résultat.
2. Créer un serveur qui demande un mot de passe avant d'ouvrir un fichier et de l'envoyer sur la connexion réseau. Créer un client qui se connecte à ce serveur, donne le mot de passe requis, puis capture et sauve le fichier. Tester la paire de programmes sur votre machine en utilisant **localhost** (l'adresse IP de boucle locale **127.0.0.1** obtenue en appelant **InetAddress.getByName(null)**).
3. Modifier le serveur de l'Exercice 2 afin qu'il utilise le multithreading pour servir plusieurs clients.
4. Modifier **JabberClient.java** afin qu'il n'y ait pas de vidage du tampon de sortie, observer le résultat.
5. Modifier **MultiJabberServer** afin qu'il utilise la technique de « surveillance de thread » « *thread pooling* ». Au lieu que le thread se termine lorsqu'un client se déconnecte, il intègre de lui-même un « pool » de threads disponibles ». Lorsqu'un nouveau client demande à se connecter, le serveur cherche d'abord dans le pool un thread existant capable de traiter la demande, et s'il n'en trouve pas, en crée un. De cette manière le nombre de threads nécessaires va grossir naturellement jusqu'à la quantité maximale nécessaire. L'intérêt du « thread pooling » est d'éviter l'overhead engendré par la création et la destruction d'un nouveau thread pour chaque client.
6. À partir de **ShowHTML.java**, créer une applet qui fournisse un accès protégé par mot de passe à un sous-ensemble particulier de votre site Web.
7. Modifier **CIDCreateTables.java** afin qu'il lise les chaînes SQL depuis un fichier texte plutôt que depuis **CIDSQL**.
8. Configurer votre système afin d'exécuter avec succès **CIDCreateTables.java** et **LoadDB.java**.
9. Modifier **ServletsRule.java** en surchargeant la méthode **destroy()** afin qu'elle sauvegarde la valeur de **i** dans un fichier, et la méthode **init()** pour qu'elle restaure cette valeur. Montrer que cela fonctionne en rechargeant le conteneur de servlet. Si vous ne possédez pas de conteneur de servlet, il vous faudra télécharger, installer, et exécuter Tomcat depuis jakarta.apache.org afin de travailler avec les servlets.

10. Créer une servlet qui ajoute un cookie à l'objet réponse, lequel sera stocké sur le site client. Ajouter à la servlet le code qui récupère et affiche le cookie. Si vous n'avez pas de conteneur de servlet, il vous faudra télécharger, installer, et exécuter Tomcat depuis *jakarta.apache.org* afin de travailler avec les servlets.
11. Créer une servlet utilisant un objet **Session** stockant l'information de session de votre choix. Dans la même servlet, récupérer et afficher cette information de session. Si vous ne possédez pas de conteneur de servlet, il vous faudra télécharger, installer, et exécuter Tomcat depuis *jakarta.apache.org* afin de travailler avec les servlets.
12. Créer une servlet qui change la valeur de « inactive interval » de l'objet session pour la valeur 5 secondes en appelant **setMaxInactiveInterval()**. Tester pour voir si la session se termine naturellement après 5 secondes. Si vous n'avez pas de conteneur de servlet, il vous faudra télécharger, installer, et exécuter Tomcat depuis *jakarta.apache.org* afin de travailler avec les servlets.
13. Créer une page JSP qui imprime une ligne de texte en utilisant le tag `<H1>`. Générer la couleur de ce texte aléatoirement, au moyen du code Java inclus dans la page JSP. Si vous ne possédez pas de conteneur JSP, il vous faudra télécharger, installer, et exécuter Tomcat depuis *jakarta.apache.org* afin de travailler avec JSP.
14. Modifier la date d'expiration dans **Cookies.jsp** et observer l'effet avec deux navigateurs différents. Constater également les différences entre le fait de visiter à nouveau la même page, et celui de fermer puis réouvrir le navigateur. Si vous ne possédez pas de conteneur JSP, il vous faudra télécharger, installer, et exécuter Tomcat depuis *jakarta.apache.org* afin de travailler avec JSP.
15. Créer une page JSP contenant un champ autorisant l'utilisateur à définir l'heure de fin de session ainsi qu'un second champ contenant les données stockées dans cette session. Le bouton de soumission rafraîchit la page, prend les valeurs courantes de l'heure de fin et les données de la session, et les garde en tant que valeurs par défaut des champs susmentionnés. Si vous ne possédez pas de conteneur JSP, il vous faudra télécharger, installer, et exécuter Tomcat depuis *jakarta.apache.org* afin de travailler avec JSP.
16. (Encore plus difficile). Modifier le programme **VLookup.java** de telle manière qu'un clic sur le nom résultat copie automatiquement ce nom dans les presse-papier (ce qui vous permet de le coller facilement dans votre email). Vous aurez peut-être besoin de revenir en arrière sur le chapitre 13 pour vous remémorer l'utilisation du presse-papier dans les JFC.

[72] Cela signifie un nombre maximum légèrement supérieur à quatre milliards, ce qui s'avère rapidement insuffisant. Le nouveau standard des adresses IP utilisera un nombre représenté sur 128 bits, ce qui devrait fournir suffisamment d'adresses IP uniques pour le futur prévisible.

[73] Créé par Dave Bartlett.

[74] Dave Bartlett participa activement à ce développement, ainsi qu'à la section JSP.

[75] « A primary tenet of Extreme Programming (XP) ». Voir *www.xprogramming.com*.

Last Update:04/24/2000

Dernière mise à jour de la version française : 22/08/2000