Chapitre 8

Programmation Internet en Java

Vous avez déjà utilisé *Internet*, le plus connu des inter-réseaux mondiaux d'ordinateurs et quelques-uns de ses *services*, en particulier le *web* et le courrier électronique. On peut évidemment utiliser des progiciels pour accéder à ces services : un *navigateur* tel que *Firefox* ou *Internet Explorer* pour accéder au web, un lecteur de courrier électronique tel que *Mozilla Thunderbird* ou *Outlook*. On peut aussi vouloir savoir comment ça fonctionne, concevoir de tels logiciels ou des utilitaires pour quelques problèmes spécifiques que le paramétrage des progiciels ne permet pas. On a alors à programmer l'accès à Internet. C'est ce que nous allons voir dans ce chapitre à travers Java.

8.1 Rappels sur Internet

Nous allons « rappeler » ce dont nous aurons besoin à propos d'Internet. Nous concevons cette section de telle manière qu'elle puisse servir d'initiation.

8.1.1 Le protocole IP

Réseaux d'ordinateurs.- Des ordinateurs hétérogènes peuvent être reliés entre eux de façon à pouvoir échanger des données. Qu'importe la façon dont ils sont reliés physiquement, ce que l'on veut c'est que l'on puisse trasmettre et recevoir une suite de zéro et de un.

Protocole de réseau.- Dans un réseau d'ordinateurs, il est rare qu'il n'y ait que deux ordinateurs et ce n'est, de toute façon, pas le cas pour Internet. Un ordinateur donné est, plus ou moins, relié à tous les autres ordinateurs du réseau, directement ou indirectement. Lorsqu'on envoie des données, il faut donc indiquer à quel ordinateur elles sont destinées et indiquer l'expéditeur envoie si on veut une réponse.

Une donnée sera donc composée de deux parties :

- les **données de protocole de réseau** permettant d'identifier l'expéditeur, le destinataire ainsi que d'autres paramètres ;
- les données proprement dites, appelées **données d'application**.

Ces données doivent correspondre à un format déterminé par un **protocole de réseau**. Il existe de nombreux protocoles de réseau. TCP/IP qui est la suite de protocoles utilisés par Internet. Nous ne nous intéresserons qu'à TCP/IP dans ce chapitre, la seule suite de protocoles prise en charge par Java.

Notion d'adresse IP.- Chaque ordinateur relié à Internet reçoit un identifiant, évidemment unique, distribuée par un organisme international, appelé **adresse IP** (pour *Internet Protocol*).

En fait cet organisme attribue certaines plages d'adresses à chaque pays, celui-ci ayant un organisme national chargé d'attribuer les adresses. Cet organisme attribue lui-même des plages d'adresses aux grands consommateurs (entreprises, universités...), chacun ayant un responsable chargé d'attribuer les adresses à tel ou tel ordinateur.

Remarquons que lorsqu'on passe par un **FAI** (*Fournisseur d'Accès Internet*, ou **ISP** pour *Internet Service Provider* en anglais), notre ordinateur ne dispose pas d'une adresse IP fixe en général mais d'une **adresse dynamique**, c'est-à-dire que le fournisseur lui attribue une adresse lors d'un accès à son service, cette adresse pouvant quelquefois être différente d'un accès à l'autre.

Format d'une adresse IP.- À l'heure actuelle, on utilise encore souvent une adresse IPv4 tenant sur quatre octets, bien qu'il existe des adresses IPv6 tenant sur six octets au vu de l'engouement pour Internet et donc au nombre d'adresses nécessaires.

La suite de 32 bits est utilisée par le matériel mais il est traditionnel de désigner cette adresse pour l'utilisateur sous la forme de quatre entiers en décimal (compris entre 0 et 255) séparés par des points, par exemple :

```
130.65.86.66
```

Nom de domaine.- Une telle adresse n'étant pas très facile à retenir par l'utilisateur, la plupart des adresses IP sont associées à un **nom de domaine** tel que :

```
java.sun.com
```

Chacun peut essayer de choisir le nom de domaine qu'il veut mais, pour éviter que le même nom corresponde à deux adresses IP différentes, ce nom est autorisé (ou non) par un organisme international.

Serveur de nom.- Si l'on choisit d'envoyer le nom de domaine et non l'adresse IP (ce que l'on fait le plus souvent), il faut bien que quelqu'un fasse la traduction. C'est le but du **serveur de nom** (ou **DNS** pour *Domain Naming Service*) de l'administrateur du réseau d'entreprise ou du fournisseur d'accès à Internet.

Notion d'application et de port.- Un ordinateur relié à Internet peut utiliser plusieurs types de programmes servant à des buts différents : l'ordinateur peut avoir, par exemple, un navigateur Web et un programme pour lire le courrier électronique. On peut, de plus, avoir des sessions différentes pour un même programme : utiliser, par exemple, deux instances du navigateur. On parle d'**application** et d'instance d'application Internet.

Le protocole Internet attribue un **numéro de port**, compris entre 0 et 65 535, à chaque instance d'application.

Modèle client/serveur.- Pour initier une session entre deux ordinateurs, l'un est considéré comme le **serveur** et l'autre comme un **client** de ce serveur : le serveur est en attente perpétuelle de clients ; lorsqu'un client essaie de se connecter à un serveur, celui-ci l'accepte (ou non) et une communication peut commencer.

Les numéros de port des clients sont attribués (de façon plus ou moins aléatoire) par le sous-service réseau du système d'exploitation de l'ordinateur alors que les services bien connus ont un numéro de port attribué par un organisme de régulation international (IANA) : par exemple le serveur Web utilise le port 80, le serveur de lecture du courrier électronique le port 110.

8.1.2 Un exemple de protocole d'application : HTTP

Notion de protocole d'application.- Nous venons de voir comment TCP/IP permet d'établir une connection Internet entre deux ports de façon à ce que deux ordinateurs puissent échanger des données. Chaque application Internet possède son propre **protocole d'application**, qui décrit comment les données de cette application particulière sont transmises.

Le protocole HTTP.- Une des applications les plus utilisées sur Internet, avec le courrier électronique, est le web. Celui-ci utilise deux protocoles : **HTTP** (pour *Hypertext Transfer Protocol*) qui est utilisé pour la transmission des données (qui est du texte) et **HTML** (pour *HyperText Markup Language*) pour la description du formatage.

Intéressons-nous ici à HTPP pour recevoir des données brutes.

Rôle du navigateur.- Supposons que nous tapions l'adresse web (ou **URL** pour *Uniform Resource Locator*, prononcé « Earl ») :

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

sur notre navigateur de façon à charger la page correspondante. Que se passe-t-il alors ? Le navigateur effectue les étapes suivantes :

- il examine la partie de l'URL comprise entre la double oblique et la première oblique simple (ici « `java.sun.com` ») pour identifier l'ordinateur auquel se connecter. Puisque cette partie de l'URL contient des lettres, il doit s'agir d'un nom de domaine, aussi le navigateur effectue une requête auprès d'un serveur DNS pour en obtenir l'adresse IP.
- le navigateur déduit du préfixe « http » de l'URL (c'est-à-dire ce qui précède « `://` ») que le protocole que l'on veut utiliser est HTTP, aussi choisit-il le port 80 par défaut.
- il établit alors une connection Internet au port 80 de cette adresse IP.
- le navigateur déduit du suffixe « `/index.html` » que l'on veut voir afficher le fichier de nom physique « `/index.html` ». Il envoie donc une requête, formatée comme commande HTTP, à travers la connection qu'il vient d'établir. Cette requête est de la forme :

  `GET /index.html HTTP/1.0`
  ligne blanche
- le serveur web de l'ordinateur distant reçoit la requête, la décode, cherche le fichier de nom physique « `/index.html` » dont le chemin est relatif à un certain répertoire et en envoie le contenu au navigateur de votre ordinateur.
- le navigateur reçoit ce fichier, qui est censé être un fichier texte de format HTML, traduit le code HTML pour faire apparaître la page web. Si le fichier contient, par exemple, des images le navigateur effectue une requête GET par image.

Les commandes HTTP.- Le tableau ci-dessous donne les commandes HTTP :

Commande	Signification
GET	Renvoie l'item demandé
HEAD	Renvoie seulement l'en-tête de cet item
OPTIONS	Renvoie les options de communication d'un item
POST	Fournit des données à une commande du serveur et renvoie le résultat
PUT	Stocke un item sur le serveur
DELETE	Efface un item du serveur
TRACE	Donne le chemin de la communication

Seule la commande `GET` nous intéressera ici.

Exemple avec Telnet.- Utilisons la commande `GET` avec l'application `telnet`. Cette application, que vous connaissez certainement déjà, permet à un utilisateur d'envoyer des caractères à un ordinateur distant et de visualiser les caractères que cet ordinateur renvoie.
Lançons `telnet` à partir d'une ligne de commande :

```
telnet java.sun.com 80
```

Soyez maintenant très soigneux car vous ne pouvez pas corriger votre envoi et il n'y a pas nécessairement d'écho à l'écran. Tapez :

```
GET /index.html HTTP/1.0
host: java.sun.com
ligne suivante
ligne suivante
```

Remarquez qu'il faut deux passages à la ligne pour terminer la requête.

Le serveur envoie comme réponse le fichier (texte) demandé, qui est affiché à l'écran sans pause. On ne voit donc que la fin du fichier, dans lequel on reconnaît quelques balises HTML.

Remarque.- Cet exemple avec Telnet nous montre l'extrême sensibilité de HTTP (on ne peut pas revenir en arrière lorsqu'on fait une erreur, syntaxe très rigide) et on comprend pourquoi on a intérêt à utiliser un programme, ce que nous allons faire maintenant.

8.2 Programmation des clients

Nous allons voir comment écrire une application Java permettant d'établir une connection TCP/IP de notre ordinateur (qui sera un client) à un serveur, d'envoyer une requête à ce serveur et d'afficher la réponse.

8.2.1 Les sockets

Notion de socket.- Depuis Unix BSD, on utilise la plupart du temps les **sockets** [1] pour la programmation des communications. La communication par sockets permet aux applications d'effectuer les communications comme des entrées-sorties sur des fichiers : un programme peut lire dans un socket ou écrire dans un socket.

Sockets de flux et de datagrammes.- Java fournit des *sockets de flux* et des *sockets de datagrammes* pour manipuler les deux protocoles de transport d'Internet : TCP et UDP respectivement.

Nous allons utiliser les sockets de flux.

8.2.2 Mise en place des sockets en Java

Classe associée.- En Java, les sockets (bidirectionnels) sont des objets de la classe :

```
Socket
```

du paquetage :

```
java.net
```

Instantiation des sockets.- Le constructeur :

Socket(String, int) ;

permet d'instantier un socket bidirectionnel, le paramètre chaîne de caractères spécifiant l'adresse IP (soit sous forme d'un nom de domaine, soit sous forme de quatre entiers décimaux séparés par des points) et le paramètre entier le port.
On a par exemple :

```
Socket s;
s = new Socket("java.sun.com", 80);
```

Le constructeur lève une exception :

```
UnknownHostException
```

s'il ne trouve pas l'hôte.

Accès aux flots d'entrée et de sortie.- Lorsqu'un client a réussi à créer un socket avec un serveur, il faut pouvoir accéder aux flots d'entrée (pour récupérer l'information du serveur) et de sortie (pour envoyer l'information au serveur), ce qui se fait caractère par caractère.

Ces deux flots de caractères sont des instances des deux classes :

```
InputStream
```

et :

```
OutputStream
```

du paquetage `java.io`.

1. Prise (électrique) en anglais.

L'association s'effectue grâce aux méthodes :

```
getInputStream()
```

et :

```
getOutputStream()
```

de la classe `Socket`. Dans notre cas :

```
InputStream in = s.getInputStream();
OutputStream out = s.getOutputStream();
```

Fermeture d'un socket.- Lorsqu'on a terminé la communication avec le serveur, on doit fermer le socket, grâce à la méthode :

```
close()
```

de la classe `Socket`.

8.2.3 Utilisation des flux

Comme d'habitude, nous utiliserons des chaîne de caractères et non des caractères.

Flux d'entrée.- On associera un tampon d'entrée, de la classe :

```
BufferedReader
```

instancié avec le constructeur dont le paramètre est un objet de la classe `InputStreamReader`, lui-même instancié avec le constructeur dont le paramètre est un objet de la classe `InputStream`.

Flux de sortie.- On associera un objet de la classe :

```
PrintWriter
```

pour la sortie (instancié avec le constructeur dont le paramètre est un objet de `OutputStream`). On pourra utiliser sa méthode :

```
print(String)
```

déjà rencontrée pour une classe dérivée.

La classe `PrintWriter` utilise un tampon et n'envoie l'information que lorsque celui-ci est plein. Pour envoyer une commande, il faudra donc vider explicitement le tampon grâce à la méthode :

```
flush()
```

On aura donc :

```
PrintWriter pw;
------------------
pw.print(commande);
pw.flush();
```

8.2.4 Un programme

Le programme suivant permet de se connecter à un serveur web et d'afficher (toujours avec un défilement rapide) une page de ce serveur sous forme texte :

```
import java.io.*;
import java.net.*;
public class WebGet
   {
   public static void main(String[] args) throws IOException
      {
      // Recuperation des arguments
      String host = args[0];
      String resource = args[1];

      // Ouverture du socket
      final int HTTP_PORT = 80;
      Socket s = new Socket(host, HTTP_PORT);

      // Obtention des flux d'octets
      InputStream in = s.getInputStream();
      OutputStream out = s.getOutputStream();

      // Obtention des flux de chaines de caracteres
      BufferedReader reader = new BufferedReader(
          new InputStreamReader(in));
      PrintWriter writer = new PrintWriter(out);

      // Envoi de la commande
      String command = "GET /" + resource
          + " HTTP/1.1\r\n" + "host: "
          + host + "\r\n\r\n";
      writer.print(command);
      writer.flush();

      // Lecture de la reponse
      boolean done = false;
      String input;
      while (!done)
         {
         input = reader.readLine();
         if (input == null) done = true;
         else System.out.println(input);
         }

      // Fermeture du socket
      s.close();
      }
   }
```

On l'utilisera sous la forme :

```
java WebGet java.sun.com index.html
```

Remarque.- Le protocole HTTP a choisi la convention Microsoft de passage à la ligne (retour chariot suivi d'un retour à la ligne d'en dessous) et non la convention Unix.

8.3 Programmation d'un serveur à un seul client

Nous allons voir maintenant comment écrire un programme Java mettant en place un serveur (rudimentaire).

8.3.1 Principe

Socket serveur.- Java encapsule tout ce qui est nécessaire pour un serveur dans la classe :

`ServerSocket`

du paquetage `java.net`.

Instanciation.- Le constructeur le plus utilisé pour cette classe est celle dont le seul paramètre est le numéro du port à utiliser. Pour le port utilisateur 8888, par exemple, on aura donc :

```
ServerSocket server = new ServerSocket(8888);
```

Acceptation de client.- Un serveur va en général indéfiniment attendre un client et effectuer un service lorsqu'un client se manifeste. Pour attendre le client suivant et se connecter, on utilise la méthode :

`Socket accept()`

de la classe `ServerSocket`. Le socket ainsi obtenu permet la transmission de données de la façon vue à propos du client.

Fin de connection.- Le serveur doit fermer la connection avec le client soit lorsqu'il reçoit une ligne vide (ceci signifie que le client s'est déconnecté), soit lorsqu'il reçoit une valeur sentinelle convenue (par exemple 'QUIT').

Le socket serveur, quant à lui, n'est jamais fermé en général puisque le rôle d'un serveur est d'attendre un client. On le fermera par un CTRL-C dans la fenêtre de commande au moment opportun.

Numéro IP de notre ordinateur.- Nous allons implémenter le serveur sur notre ordinateur. Nous avons donc besoin de connaître son numéro IP. Il suffit en fait de son nom de domaine qui est toujours "`localhost`".

8.3.2 Une application client-serveur

Nous allons concevoir un couple client-serveur. Le client essaie de se connecter à un serveur (de port 8888 de notre ordinateur), lui envoie un entier ou 'QUIT' et attend un entier en retour, qu'il affiche. Le serveur effectue une boucle infinie qui consiste à attendr un client et à effectuer un certain service lorsqu'il en accepte un. Ce service consistera ici à calculer le carré de l'entier reçu et de l'envoyer.

8.3.2.1 Le client

Écrivons une application Java client pour le problème que nous avons pris en exemple :

```
import java.io.*;
import java.net.*;


public class SquareClient
   {
   public static void main(String[] args) throws IOException
      {
      final int SQUARE_PORT = 8888;
      Socket s = new Socket("localhost", SQUARE_PORT);
      InputStream in = s.getInputStream();
      OutputStream out = s.getOutputStream();
      BufferedReader reader = new BufferedReader(
         new InputStreamReader(in));
      PrintWriter writer = new PrintWriter(out);

      String commande = "5\n";
      System.out.print("Envoi : " + commande);
      writer.print(commande);
      writer.flush();
      String reponse = reader.readLine();
      System.out.println("Recu : " + reponse);

      commande = "3\n";
      System.out.print("Envoi : " + commande);
      writer.print(commande);
      writer.flush();
      reponse = reader.readLine();
      System.out.println("Recu : " + reponse);

      commande = "1000\n";
      System.out.print("Envoi : " + commande);
      writer.print(commande);
      writer.flush();
      reponse = reader.readLine();
      System.out.println("Recu : " + reponse);

      commande = "500\n";
      System.out.print("Envoi : " + commande);
```

```
      writer.print(commande);
      writer.flush();
      reponse = reader.readLine();
      System.out.println("Recu : " + reponse);

      commande = "QUIT\n";
      System.out.print("Envoi : " + commande);
      writer.print(commande);
      writer.flush();

      s.close();
      }
   }
```

8.3.2.2 Le serveur

Écrivons l'application java serveur. Le rôle du serveur est presque le même pour tous les serveurs, on a donc intérêt à séparer la partie serveur proprement dite de la partie service :

```
import java.io.*;
import java.net.*;

public class SquareServer
   {
   public static void main(String[] args) throws IOException
      {
      final int SQUARE_PORT = 8888;
      ServerSocket server = new ServerSocket(SQUARE_PORT);
      System.out.println("En attente de client...");
      while(true)
         {
         Socket s = server.accept();
         SquareService service = new SquareService(s);
         service.doService();
         s.close();
         }
      }
   }


class SquareService
   {
   private Socket s;

   public SquareService(Socket aSocket)
      {
      s = aSocket;
      }
```

```
   public void doService() throws IOException
      {
      BufferedReader in = new BufferedReader(
         new InputStreamReader(s.getInputStream()));
      PrintWriter out = new PrintWriter(
      s.getOutputStream());
      String ligne;

      ligne = in.readLine();
      while(!(ligne.equals("QUIT") || ligne == null))
         {
         System.out.println("Recu : " + ligne);
         int n = Integer.parseInt(ligne);
         String reponse = "" + n*n;
         System.out.println("Envoye : " + reponse);
         out.println(reponse);
         out.flush();
         ligne = in.readLine();
         }
      }
   }
```

8.3.2.3 Exécution

Exemple.- Commençons par lancer le serveur dans une fenêtre de commande. On voit apparaître :

```
linux:# java SquareServer
En attente de client...
```

Lançons le client dans une autre fenêtre de commande. Nous voyons dans cette fenêtre :

```
linux: # java SquareClient
Envoi : 5
Recu : 25
Envoi : 3
Recu : 9
Envoi : 1000
Recu : 1000000
Envoi : 500
Recu : 250000
Envoi : QUIT
linux: #
```

et dans la fenêtre serveur :

```
En attente de client...
Recu : 5
Envoye : 25
Recu : 3
Envoye : 9
Recu : 1000
```

```
Envoye : 1000000
Recu : 500
Envoye : 250000
```

Remarques.- 1^o) Si nous ne disposons que d'un seul ordinateur, nous avons besoin de deux fenêtres de commande et donc d'un système multi-tâches.

2^o) Le serveur que nous avons conçu n'est pas très réaliste puisqu'il n'accepte qu'un seul client à la fois. Nous allons voir dans la section suivante comment accepter plusieurs clients à la fois.

8.4 Programmation d'un serveur à plusieurs clients simultanés

Le serveur que nous venons de concevoir est très rudimentaire puisqu'il n'accepte qu'un seul client à la fois. Nous allons voir comment concevoir un serveur qui accepte plusieurs clients à la fois.

8.4.1 Principe

Il n'y a pas de notion syntaxique nouvelle à ce propos. Nous allons utiliser à la fois les possibilités de la programmation Internet et de la programmation parallèle de Java.

Le principe général est le suivant :

— la classe de service doit être un processus léger ; la méthode de départ de cette classe est donc la méthode `run()` ;

— lorsque le socket du serveur accepte une connexion, le serveur doit construire une nouvelle instanciation de la classe de service et démarrer le processus correspondant.

8.4.2 Nouveau client

Programme.- Concevons une nouvelle application client pour se rendre compte du problème de la synchronisation. L'application demande un entier à l'utilisateur, envoie cet entier au serveur, récupère la valeur de retour, l'affiche et recommence jusqu'à ce que l'utilisateur lui donne une valeur sentinelle (disons 'QUIT') :

```
import java.io.*;
import java.net.*;

public class SquareClient2
   {
   public static void main(String[] args) throws IOException
      {
      final int SQUARE_PORT = 8888;

      Socket s;
      InputStream in;
      OutputStream out;
      BufferedReader reader;
      PrintWriter writer;
      String commande;
      String reponse;
      InputStreamReader fich;
      BufferedReader tampon;
      String entree;

      s = new Socket("localhost", SQUARE_PORT);
      in = s.getInputStream();
      out = s.getOutputStream();
      reader = new BufferedReader(
         new InputStreamReader(in));
```

```
      writer = new PrintWriter(out);
      fich = new InputStreamReader(System.in);
      tampon = new BufferedReader(fich);

      System.out.print("entier : ");
      entree = tampon.readLine();
      while (! entree.equals("QUIT"))
         {
         commande = entree + "\n";
         System.out.print("Envoi : " + commande);
         writer.print(commande);
         writer.flush();
         reponse = reader.readLine();
         System.out.println("Recu : " + reponse);
         System.out.print("entier : ");
         entree = tampon.readLine();
         }
      commande = "QUIT\n";
      System.out.print("Envoi : " + commande);
      writer.print(commande);
      writer.flush();

      s.close();
      }
   }
```

Exécution.- Lançons le serveur précédent. Lançons un premier client : la réaction est normale. Lançons un deuxième client : l'action est gelée dès la première saisie d'un entier ; l'entier a bien été envoyé au serveur mais sans aucun retour. Par contre, dès qu'on termine avec le premier client, on reçoit la réponse.

On imagine ce qui arriverait à un serveur Web, par exemple, qui n'accepterait qu'un seul client à la fois. Les utilisateurs seraient vite lassés.

8.4.3 Nouveau serveur

Le programme.- Concevons donc un serveur qui accepte plusieurs clients à la fois :

```
import java.io.*;
import java.net.*;

public class SquareServer2
   {
   public static void main(String[] args) throws IOException
      {
      final int SQUARE_PORT = 8888;
      ServerSocket server;
      Socket s;

      server = new ServerSocket(SQUARE_PORT);
      System.out.println("En attente de client...");
```

```
      while(true)
         {
         s = server.accept();
         SquareService2 service = new SquareService2(s);
         service.start();
         }
      }
   }
class SquareService2 extends Thread
    {
    private Socket s;

    public SquareService2(Socket aSocket)
       {
       s = aSocket;
       }

    public void run()
       {
       try
          {
          BufferedReader in = new BufferedReader(
             new InputStreamReader(s.getInputStream()));
          PrintWriter out = new PrintWriter(
             s.getOutputStream());
          String entree;
          String reponse;
          int n;

          entree = in.readLine();
          while(! (entree.equals("QUIT") || entree == null))
             {
             System.out.println("Recu : " + entree);
             n = Integer.parseInt(entree);
             reponse = "" + n*n;
             System.out.println("Envoye : " + reponse);
             out.println(reponse);
             out.flush();

             yield();

             entree = in.readLine();
             }
          s.close();
          }
       catch(IOException e)
          {
          }
       }
    }
```

Exécution.- Cette fois-ci l'exécution se déroule conformément à nos désirs, même dans le cas de plusieurs clients.

Remarque.- Il y a obligation, comme d'habitude en Java, de récupérer les exceptions d'entrées-sorties. Nous ne pouvons pas le faire à travers une clause `throws` puisque nous surchargerions la méthode `run()` déjà définie. Nous utilisons donc un couple de bloc d'essai et de récupération.

8.5 Serveur synchronisé

Les processus du serveur précédent ne sont pas synchronisés, ce qui ne doit pas être dramatique. Mais, en général, on a intérêt à bien faire attention aux problèmes de synchronisation.

Imaginons, par exemple, que deux personnes puissent accéder au même compte bancaire (disons le mari et la femme). Supposons qu'elles décident, à peu près au même moment, de retirer de l'argent. Elles commencent par consulter le solde : disons qu'il reste 150 euros. Cette information est envoyé aux deux personnes et l'accès n'est bloqué pour aucune d'elles. Elles décident alors toutes les deux de retirer 100 euros. L'une verra sur son ticket qu'il reste 50 euros, ce qui lui paraît normal et l'autre qu'il reste - 50 euros, de façon incompréhensible.

Nous renvoyons aux exercices pour un exemple explicite.

8.6 Accès direct aux url

Nous avons vu une façon de nous connecter aux serveurs Web et la façon de rappatrier l'information en envoyant des commandes du protocole HTTP. Puisque HTTP est un protocole tellement important, les concepteurs de Java ont conçu toute une bibliothèque à son propos qui évite d'utiliser explicitement les commandes HTTP.

8.6.1 Détermination d'une url

Rappels.- Le protocole HTTP utilise les **url** (pour *Uniform Resource Locator* ou *Universal Resource Locator*) pour localiser les données sur Internet. Les url représentent des fichiers et des répertoires, mais ils peuvent aussi représenter des tâches complexes telles que des recherches dans des bases de données et sur l'Internet dans sa totalité.

La classe URL.- Sun a implémenté une classe :

```
URL
```

dans le paquetage `java.net`. Le constructeur le plus intéressant de cette classe est celui dont l'argument est une chaîne de caractères, qui est l'url au sens de HTTP. On a, par exemple :

```
URL u = new URL("http://java.sun.com/index.html");
```

Application.- Il existe de nombreux constructeurs d'objets dont un des paramètres est un objet URL, en particulier pour les applets. Rappelons, par exemple, que la classe `Applet` possède la méthode :

```
Image getImage(URL, String)
```

permettant de récupérer une image sur internet. Elle a deux arguments : un objet de la classe URL et le nom physique du fichier graphique.

Exemple.- Voici une variation de l'exemple précédent :

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

public class image3 extends Applet
   {
   public void init()
      {
      resize(250, 300);
      }

   public void paint(Graphics g)
      {
      Image img;
      g.drawString("Exemple d'image", 15, 15);
      img = getImage(getDocumentBase(),"PCegielski.gif");
      g.drawImage(img, 40, 50, this);
      }
   }
```

APPENDICE

Rappelons que le programme de la section 8.2.4 s'écrirait de la façon suivante en langage C sous UNIX :

```
/* client6.c */

#include <stdio.h>
#include <sys/socket.h>
#include <netinet/in.h>

void main(void)
   {
   int s;
   struct sockaddr_in serveur;
   int longueur;
   const unsigned char IPno[] = {194, 214, 24, 150}; /* www.univ-paris12.fr */
   char buf[80];
   int n;

   /* Creation d'une socket */
   s = socket(AF_INET, SOCK_STREAM, 0);

   /* Connexion */
   memset(&serveur, 0, sizeof(serveur));

   serveur.sin_family = AF_INET;
   serveur.sin_port = htons(80);
   memcpy(&serveur.sin_addr.s_addr, IPno, 4);

   longueur = sizeof(serveur);

   n = connect(s, (struct sockaddr *) &serveur, longueur);

   /* Emission */
   write(s, "GET /index.html HTTP/1.0\n\n", 26);

   /* Reception */
   printf("Message recu : ");
   n = -1;
   while (n != 0)
      {
      n = read(s, buf, 80);
      buf[n] = '\0';
      printf("%s\n", buf);
      }

   /* Fermeture de la socket */
   close(s);
   }
```