ГЛАВА 3

Работа в сети

В этой главе...

- Подключение к серверу
- Реализация серверов
- Прерываемые сокеты
- Получение данных из Интернета
- Отправка электронной почты

Эта глава начинается с описания основных понятий для работы в сети, а затем в ней рассматриваются примеры написания программ на Java, позволяющих устанавливать соединения с серверами. Из нее вы узнаете, как осуществляется реализация сетевых клиентов и серверов. А завершается глава рассмотрением вопросов передачи почтовых сообщений из программы на Java и сбора данных с веб-сервера.

Подключение к серверу

Прежде чем приступать к написанию первой сетевой программы, надо познакомиться с отличным инструментальным средством для отладки сетевых программ: утилитой telnet. Следует, однако, иметь в виду, что она не относится к числу предварительно устанавливаемых компонентов операционной системы, поэтому обязательно проверьте, установлена ли она на вашем компьютере. Если утилиту telnet нельзя запустить из командной строки, еще раз запустите программу установки избранной операционной системы и выберите эту утилиту из списка предлагаемых для установки компонентов.

НА ЗАМЕТКУ! Вместе с ОС Windows Vista устанавливается и утилита `telnet`, но по умолчанию она не активизирована. Чтобы активизировать ее, откройте панель управления, перейдите в раздел Программы, щелкните на ссылке Добавление или удаление компонентов Windows и установите флажок Клиент Telnet. Следует также иметь в виду, что брандмауэр Windows блокирует некоторые сетевые порты, которые будут использоваться в примерах программ из этой главы. А для того чтобы разблокировать эти порты, вы должны обладать полномочиями администратора.

Утилитой `telnet` можно пользоваться не только для соединения с удаленным компьютером. С ее помощью можно также взаимодействовать с различными сетевыми службами. Ниже приводится один из примеров необычного использования этой утилиты. Для этого введите в командной строке следующую команду:

```
telnet time-A.timefreq.bldrdoc.gov 13
```

На рис. 3.1 приведен пример ответной реакции сервера, которая в режиме командной строки будет иметь следующий вид:

```
54276 07-06-25 21:37:31 50 0 0 659.0 UTC(NIST) *
```

```
Terminal
File Edit View Terminal Tabs Help
~$ telnet time-A.timefreq.bldrdoc.gov 13
Trying 132.163.4.103...
Connected to time-A.timefreq.bldrdoc.gov.
Escape character is '^]'.

54276 07-06-25 21:37:31 50 0 0 659.0 UTC(NIST) *
Connection closed by foreign host.
~$
```

Рис. 3.1. Результат, получаемый из службы учета времени дня

Что же в действительности произошло? Утилита `telnet` подключилась к серверу службы учета времени дня, который работает на большинстве компьютеров под управлением операционной системы UNIX. Указанный в этом примере сервер находится в Национальном институте стандартов и технологий (National Institute of Standards and Technology) в г. Боулдер, штат Колорадо. Его системное время

синхронизировано с цезиевыми атомными часами. (Конечно, полученное значение текущего времени будет не совсем точным из-за задержек, связанных с передачей данных по сети.) По принятым правилам сервер службы времени всегда связан с портом **13**.

НА ЗАМЕТКУ! В сетевой терминологии *порт* — это не какое-то конкретное физическое устройство, а абстрактное понятие, упрощающее представление о соединении сервера с клиентом (рис. 3.2).

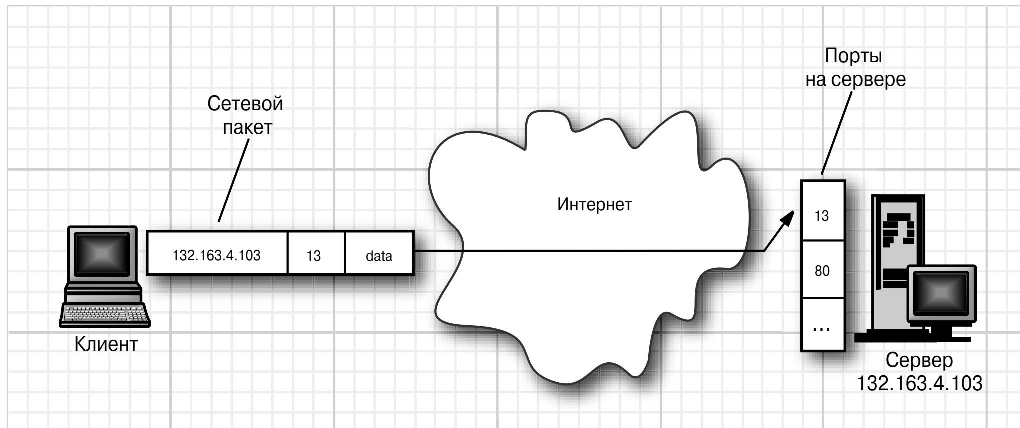

Рис. 3.2. Схема соединения клиента с сервером через конкретный порт

Программное обеспечение сервера постоянно работает на удаленном компьютере и ожидает поступления сетевого трафика через порт **13**. При получении операционной системой на удаленном компьютере сетевого пакета с запросом на подключение к порту **13** на сервере активизируется соответствующий процесс и устанавливается соединение. Такое соединение может быть прервано одним из его участников.

После запуска утилиты `telnet` с параметром `time-A.timefreq.bldrdoc.gov` с целью начать сеанс связи через порт **13** независимое сетевое программное обеспечение преобразует строку `"time-A.timefreq.bldrdoc.gov"` в IP-адрес **132.163.4.104**. Затем посылается запрос на соединение с заданным компьютером через порт **13**. После установления соединения программа на удаленном компьютере передает обратно строку с данными, а затем разрывает соединение. Разумеется, клиенты и серверы могут вести и более сложные диалоги до разрыва соединения.

Проведем еще один, более интересный эксперимент. С этой целью выполните следующие действия.

1. Введите в режиме командной строки команду
2. **telnet horstmann.com 80**
3. Затем аккуратно и точно введите следующее, *дважды* нажав клавишу <Enter> в конце:

```
GET / HTTP/1.1
Host: horstmann.com
пустая строка
```

На рис. 3.3 показана ответная реакция сервера в окне утилиты `telnet`. Она имеет уже знакомый вид страницы текста в формате HTML, а именно начальной страницы веб-сайта Кея Хорстманна. Именно так обычный веб-браузер получает искомые веб-страницы. Для запроса веб-страниц на сервере он применяет сетевой протокол HTTP. Разумеется, браузер отображает данные в намного более удобном для чтения виде, чем формат HTML.

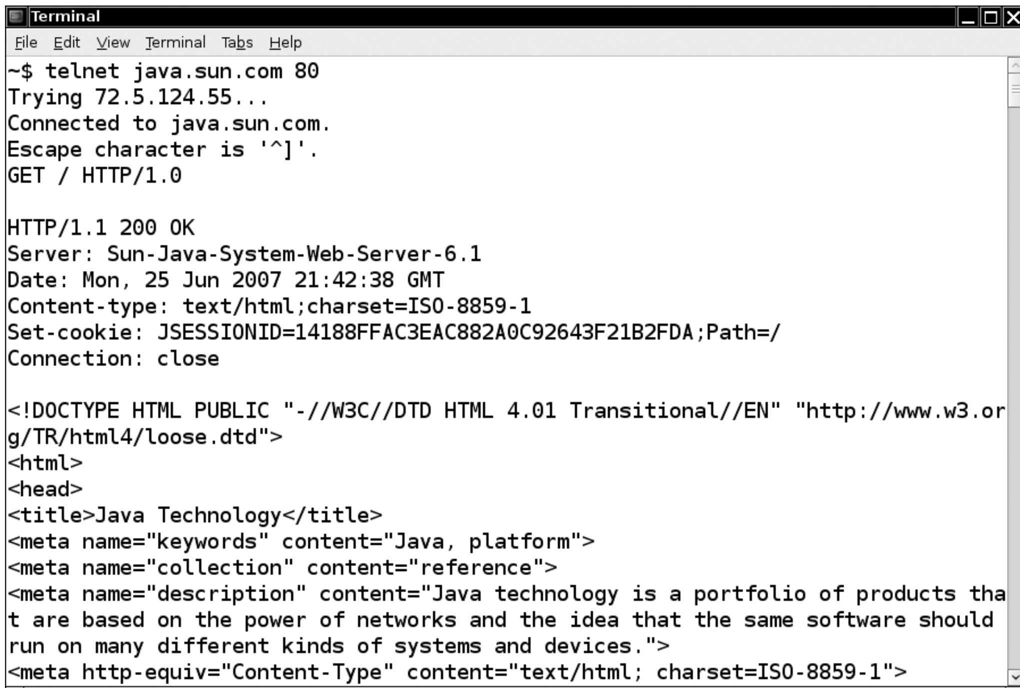

Рис. 3.3. Доступ к HTTP-порту с помощью утилиты `telnet`

НА ЗАМЕТКУ! Пару "ключ-значение" `Host: horstmann.com` требуется указывать для подключения к веб-серверу, на котором под одним и тем же IP-адресом размещаются разные домены. Ее можно не указывать, если на веб-сервере размещается единственный домен.

В первом примере сетевой программы, исходный код которой приведен в листинге 3.1, выполняются те же самые действия, что и при использовании утилиты `telnet`. Она устанавливает соединение с сервером через порт и выводит получаемые в ответ данные.

Листинг 3.1. Исходный код из файла `socket/SocketTest.java`

```
1  package socket;
2
3  import java.io.*;
4  import java.net.*;
5  import java.util.*;
```

Core Java.indb 194 21.11.2013 9:33:40

```
6
7  /**
8   * В этой программе устанавливается сокетное соединение с атомными часами
9   * в г. Боулдере, шт. Колорадо и выводится время, передаваемое из сервера
10  *
11  * @version 1.20 2004-08-03
12  * @author Cay Horstmann
13 */
14 public class SocketTest
15 {
16   public static void main(String[] args) throws IOException
17   {
18     try (Socket s = new Socket("time-A.timefreq.bldrdoc.gov", 13))
19     {
20       InputStream inStream = s.getInputStream();
21       Scanner in = new Scanner(inStream);
22
23       while (in.hasNextLine())
24       {
25         String line = in.nextLine();
26         System.out.println(line);
27       }
28     }
29   }
30 }
```

В данной программе наибольший интерес представляют следующие две строки кода:

```
Socket s = new Socket("time-A.timefreq.bldrdoc.gov", 13);
InputStream inStream = s.getInputStream();
```

В первой строке кода открывается сокет. *Сокет* — это абстрактное понятие, обозначающее возможность для программ устанавливать соединения для обмена данными по сети. Конструктору сокета передается адрес удаленного сервера и номер порта. Если установить соединение не удается, генерируется исключение типа `UnknownHostException`, а при возникновении каких-нибудь других затруднений — исключение типа `IOException`. Класс `UnknownHostException` является подклассом, производным от класса `IOException`, поэтому в данном простом примере обрабатывается только исключение из суперкласса.

После открытия сокета метод `getInputStream()` из класса `java.net.Socket` возвращает объект типа `InputStream`, который можно использовать как любой другой поток ввода. Получив поток ввода, рассматриваемая здесь программа приступает к выводу каждой введенной символьной строки в стандартный поток вывода. Этот процесс продолжается до тех пор, пока не завершится поток ввода или не разорвется соединение с сервером.

Данная программа может взаимодействовать только с очень простыми серверами, например со службой учета текущего времени. В более сложных случаях клиент посылает серверу запрос на получение данных, а сервер может поддерживать установленное соединение в течение некоторого времени после отправки ответа на запрос. Примеры реализации подобного поведения представлены далее в этой главе.

Класс Socket очень удобен для работы в сети, поскольку он скрывает все сложности и подробности установления сетевого соединения и передачи данных по сети, реализуемые средствами библиотеки Java. А пакет java.net, по существу, предоставляет тот же самый программный интерфейс, который используется для работы с файлами.

НА ЗАМЕТКУ! В этой книге рассматривается только сетевой протокол TCP (Transmission Control Protocol — протокол управления передачей). На платформе Java поддерживается также протокол UDP (User Datagram Protocol — протокол пользовательских дейтаграмм), который может служить для отправки пакетов (называемых иначе *дейтаграммами*) с гораздо меньшими издержками, чем по протоколу TCP. Недостаток такого способа обмена данными по сети заключается в том, что пакеты необязательно доставлять получателю в последовательном порядке, и они вообще могут быть потеряны. Получатель сам должен позаботиться о том, чтобы пакеты были организованы в определенном порядке, а кроме того, он должен сам запрашивать повторно передачу отсутствующих пакетов. Протокол UDP хорошо подходит для тех приложений, которые могут обходиться без отсутствующих пакетов, например, для организации аудио- и видеопотоков или продолжительных измерений.

java.net.Socket 1.0

- Socket(String *host*, int *port*)

  Создает сокет для соединения с указанным хостом или портом.

- InputStream getInputStream()
- OutputStream getOutputStream()

  Получают поток ввода для чтения данных из сокета или записи данных в сокет.

Время ожидания для сокетов

Чтение данных из сокета продолжается до тех пор, пока данные доступны. Если хост (т.е. сетевой узел) недоступен, прикладная программа будет ожидать очень долго, и все будет зависеть от того, когда операционная система, под управлением которой работает компьютер, определит момент завершения периода ожидания.

Для конкретной прикладной программы можно самостоятельно определить наиболее подходящую величину времени ожидания для сокета, а затем вызвать метод setSoTimeout(), чтобы установить эту величину в миллисекундах. В приведенном ниже фрагменте когда показано, как это делается.

```
Socket s = new Socket(. . .);
s.setSoTimeout(10000); // истечение времени ожидания через 10 секунд
```

Если величина времени ожидания была задана для сокета, то при выполнении всех последующих операций чтения и записи данных будет генерироваться исключение типа SocketTimeoutException по истечении времени ожидания до фактического завершения текущей операции. Но это исключение можно перехватить, чтобы отреагировать на данное событие надлежащим образом, как показано ниже.

```
try
{
  InputStream in = s.getInputStream(); // читать данные из потока ввода in
```

```
  . . .
}
catch (InterruptedIOException exception)
{
  отреагировать на истечение времени ожидания
}
```

Что касается времени ожидания для сокетов, то остается еще одно затруднение, которое придется каким-то образом разрешить. Так, приведенный ниже конструктор может установить блокировку в течение неопределенного периода времени до тех пор, пока не будет установлено первоначальное соединение с хостом.

```
Socket(String host, int port)
```

Это затруднение можно преодолеть, если сначала создать несоединяемый сокет, а затем соединиться с заданным временем ожидания:

```
Socket s = new Socket();
s.connect(new InetSocketAddress(host, port), timeout);
```

Если же требуется предоставить пользователям возможность прерывать соединение с сокетом в любой момент, то ниже, в разделе "Прерываемые сокеты" поясняется, как этого добиться.

java.net.Socket 1.0

- `Socket()` **1.1**

  Создает сокет, который еще не соединен в данный момент времени.

- `void connect(SocketAddress address)` **1.4**

  Соединяет данный сокет по указанному адресу.

- `void connect(SocketAddress address, int timeoutInMilliseconds)` **1.4**

  Соединяет данный сокет по указанному адресу или осуществляет возврат, если заданный промежуток времени истек.

- `void setSoTimeout(int timeoutInMilliseconds)` **1.1**

  Задает время блокировки для чтения *запросов* в данном сокете. По истечении времени блокировки возникает исключение типа `InterruptedIOException`.

- `boolean isConnected()` **1.4**

  Возвращает логическое значение `true`, если установлено соединение с сокетом.

- `boolean isClosed()` **1.4**

  Возвращает логическое значение `true`, если разорвано соединение с сокетом.

Межсетевые адреса

Как правило, нет особой нужды беспокоиться о межсетевых адресах в Интернете — числовых адресах хостов, состоящих из четырех байт (или из шестнадцати — по протоколу IPv6), как, например, **132.163.4.102**. Тем не менее, если требуется выполнить взаимное преобразование имен хостов и межсетевых адресов, то для этой цели можно воспользоваться классом `InetAddress`.

В пакете `java.net` поддерживаются межсетевые адреса по протоколу IPv6, при условии, что их поддержка обеспечивается и со стороны операционной системы хоста. В частности, статический метод `getByName()` возвращает объект типа `InetAddress` для хоста, как показано в приведенной ниже строке кода. Например, в следующей строке кода возвращается объект типа `InetAddress`, инкапсулирующий последовательность из четырех байт **`132.163.4.104`**:

```
InetAddress address = InetAddress.getByName("time-A.timefreq.bldrdoc.gov");
```

Получить доступ к этим байтам можно с помощью метода `getAddress()`:

```
byte[] addressBytes = address.getAddress();
```

Имена некоторых хостов с большим объемом трафика соответствуют нескольким межсетевым адресам, что объясняется попыткой сбалансировать нагрузку. Так, на момент написания данной книги имя хоста `google.com` соответствовало двенадцати различным сетевым адресам. Один из них выбирается случайным образом во время доступа к хосту. Получить межсетевые адреса всех хостов можно, вызвав метод `getAllByName()`:

```
InetAddress[] addresses = InetAddress.getAllByName(host);
```

И, наконец, иногда требуется адрес локального хоста. Если вы просто запросите адрес локального хоста, указав `localhost`, то неизменно получите в ответ локальный петлевой адрес **`127.0.0.1`**, которым другие не смогут воспользоваться для подключения к вашему компьютеру. Вместо этого вызовите метод `getLocalHost()`, чтобы получить адрес вашего локального хоста, как показано ниже.

```
InetAddress address = InetAddress.getLocalHost();
```

В листинге 3.2 приведен пример простой программы, выводящей межсетевой адрес локального хоста, если не указать дополнительные параметры в командной строке, или же все межсетевые адреса другого хоста, если указать имя хоста в командной строке, как в следующем примере:

```
java inetAddress/InetAddressTest www.horstmann.com
```

Листинг 3.2. Исходный код из файла `inetAddress/InetAddressTest.java`

```
1  package inetAddress;
2
3  import java.io.*;
4  import java.net.*;
5  /**
6   * В этой программе демонстрируется применение класса InetAddress.
7   * В качестве аргумента в командной строке следует указать имя хоста
8   * или же запустить программу без аргументов, чтобы получить в ответ
9   * адрес локального хоста
10  * @version 1.02 2012-06-05
11  * @author Cay Horstmann
12 */
13 public class InetAddressTest
14 {
15   public static void main(String[] args) throws IOException
16   {
17     if (args.length > 0)
18     {
```

```
19        String host = args[0];
20        InetAddress[] addresses = InetAddress.getAllByName(host);
21        for (InetAddress a : addresses)
22          System.out.println(a);
23      }
24      else
25      {
26        InetAddress localHostAddress = InetAddress.getLocalHost();
27        System.out.println(localHostAddress);
28      }
29    }
30 }
```

java.net.InetAddress 1.0

- `static InetAddress getByName(String` *host*`)`
- `static InetAddress[] getAllByName(String` *host*`)`

  Конструируют объект типа `InetAddress` или массив всех межсетевых адресов для заданного имени хоста.

- `static InetAddress getLocalHost()`

  Конструирует объект типа `InetAddress` для локального хоста.

- `byte[] getAddress()`

  Возвращает массив байтов, содержащий числовой адрес.

- `String getHostAddress()`

  Возвращает адрес хоста в виде символьной строки с десятичными числами, разделенными точками, например `"132.163.4.102"`.

- `String getHostName()`

  Возвращает имя хоста.

Реализация серверов

Итак, рассмотрев особенности реализации элементарного сетевого клиента, способного получать данные из сети, перейдем к обсуждению реализации простого сервера, способного посылать данные. После запуска серверная программа переходит в режим ожидания от клиентов подключения к портам сервера. Для рассматриваемого здесь примера выбран номер порта **8189**, который не используется ни одним из стандартных устройств. В следующей строке кода создается сервер с контролируемым портом **8189**:

```
ServerSocket s = new ServerSocket(8189);
```

А в приведенной ниже строке кода серверной программе предписывается ожидать подключения клиентов к заданному порту.

```
Socket incoming = s.accept();
```

Как только какой-нибудь клиент подключится к данному порту, отправив по сети запрос на сервер, данный метод возвратит объект типа `Socket`,

представляющий установленное соединение. Этот объект можно использовать для чтения и записи данных в потоки ввода-вывода, как показано в приведенном ниже фрагменте кода.

```
InputStream inStream = incoming.getInputStream();
OutputStream outStream = incoming.getOutputStream();
```

Все данные, направляемые в поток вывода серверной программы, поступают в поток ввода клиентской программы. А все данные, направляемые в поток вывода из клиентской программы, поступают в поток ввода серверной программы. Во всех примерах, приведенных в этой главе, обмен текстовыми данными осуществляется через сокеты. Поэтому соответствующие потоки ввода-вывода через сокет преобразуются в сканирующие и записывающие потоки типа `Scanner` и `Writer` следующим образом:

```
Scanner in = new Scanner(inStream);
PrintWriter out = new PrintWriter(outStream, true); // автоматическая очистка
```

Допустим, клиентская программа посылает следующее приветствие:

```
out.println("Hello! Enter BYE to exit.");
```

Если для подключения к серверной программе через порт **8189** используется утилита `telnet`, это приветствие отображается на экране терминала.

В рассматриваемой здесь простой серверной программе вводимые данные, отправленные клиентской программой, считываются построчно и посылаются обратно клиентской программе в режиме эхопередачи, как показано в приведенном ниже фрагменте кода. Этим наглядно демонстрируется получение данных от клиентской программы. А настоящая серверная программа должна обработать полученные данные и выдать соответствующий ответ.

```
String line = in.nextLine();
out.println("Echo: " + line);
if (line.trim().equals("BYE")) done = true;
```

По завершении сеанса связи открытый сокет закрывается следующим образом:

```
incoming.close();
```

Вот, собственно, и все, что делает данная программа. Любая серверная программа, например, веб-сервер, работающий по протоколу HTTP, выполняет аналогичный цикл следующих действий.

1. Получение из потока ввода входящих данных запроса на конкретную информацию от клиентской программы.
2. Расшифровка клиентского запроса.
3. Сбор информации, запрашиваемой клиентом.
4. Передача обнаруженной информации клиентской программе через поток вывода исходящих данных.

В листинге 3.3 приведен весь исходный код описанного выше примера серверной программы.

Листинг 3.3. Исходный код из файла `server/EchoServer.java`

```
1  package server;
2
3  import java.io.*;
4  import java.net.*;
5  import java.util.*;
6  /**
7   * В этой программе реализуется простой сервер, прослушивающий порт
8   * 8189 и посылающий обратно клиенту все полученные от него данные
9   * @version 1.21 2012-05-19
10 * @author Cay Horstmann
11 */
12 public class EchoServer
13 {
14   public static void main(String[] args) throws IOException
15   {
16     // установить сокет на стороне сервера
17     try (ServerSocket s = new ServerSocket(8189))
18     {
19       // ожидать подключения клиента
20       try (Socket incoming = s.accept())
21       {
22         InputStream inStream = incoming.getInputStream();
23         OutputStream outStream = incoming.getOutputStream();
24
25         try (Scanner in = new Scanner(inStream))
26         {
27           PrintWriter out = new PrintWriter(outStream, true);
28                                   // автоматическая очистка
29           out.println("Hello! Enter BYE to exit.");
30
31           // передать обратно данные, полученные от клиента
32           boolean done = false;
33           while (!done && in.hasNextLine())
34           {
35             String line = in.nextLine();
36             out.println("Echo: " + line);
37             if (line.trim().equals("BYE")) done = true;
38           }
39         }
40       }
41     }
42   }
43 }
```

Для проверки работоспособности данной серверной программы ее нужно скомпилировать и запустить. Затем необходимо подключиться с помощью утилиты `telnet` к локальному серверу `localhost` (или по IP-адресу **127.0.0.1**) через порт **8189**. Если ваш компьютер непосредственно подключен к Интернету, любой пользователь может получить доступ к данной серверной программе, если ему известен IP-адрес и номер порта. При подключении через этот порт будет получено следующее сообщение, как показано на рис. 3.4:

```
Hello! Enter BYE to exit.
(Привет! Введите BYE (Пока), чтобы выйти из программы)
```

Рис. 3.4. Сеанс связи с сервером, передающим обратно данные, полученные от клиента

Введите любую фразу и понаблюдайте за тем, как она будет получена обратно в том же самом виде. Для отключения от сервера введите **`BYE`** (все символы в верхнем регистре). В итоге завершится и серверная программа.

`java.net.ServerSocket 1.0`

- `ServerSocket(int` *`port`*`)`
- Создает сокет на стороне сервера, контролирующего указанный порт.
- `Socket accept()`
- Ожидает соединения. Этот метод блокирует (т.е. переводит в режим ожидания) текущий поток до тех пор, пока не будет установлено соединение. Возвращает объект типа `Socket`, через который программа может взаимодействовать с подключаемым клиентом.
- `void close()`
- Закрывает сокет на стороне сервера.

Обслуживание многих клиентов

В предыдущем простом примере серверной программы не предусмотрена возможность одновременного подключения сразу нескольких клиентских программ. Обычно серверная программа работает на компьютере сервера, а клиентские программы могут одновременно подключаться к ней через Интернет из любой точки мира. Если на сервере не предусмотрена обработка одновременных запросов от

многих клиентов, это может привести к тому, что один клиент может монополизировать доступ к серверной программе в течение длительного времени. Во избежание подобных ситуаций следует прибегнуть к помощи потоков исполнения.

Всякий раз, когда серверная программа устанавливает новое сокетное соединение, т.е. в результате вызова метода `accept()` возвращается сокет, запускается новый поток исполнения для подключения *данного* клиента к серверу. После этого происходит возврат в основную программу, которая переходит в режим ожидания следующего соединения. Для того чтобы все это произошло, в серверной программе следует организовать приведенный ниже основной цикл.

```
while (true)
{
  Socket incoming = s.accept();
  Runnable r = new ThreadedEchoHandler(incoming);

  Thread t = new Thread(r);
  t.start();
}
```

Класс `ThreadedEchoHandler` реализует интерфейс `Runnable` и в своем методе `run()` поддерживает взаимодействие с клиентской программой:

```
class ThreadedEchoHandler implements Runnable
{
  . . .
  public void run()
  {
    try
    {
      InputStream inStream = incoming.getInputStream();
      OutputStream outStream = incoming.getOutputStream();
      обработать полученный запрос и отправить ответ
      incoming.close();
    }
    catch(IOException e)
    {
       обработать исключение
    }
  }
}
```

Когда при каждом соединении запускается новый поток исполнения, несколько клиентских программ могут одновременно подключаться к серверу. Это нетрудно проверить, выполнив следующие действия.

1. Скомпилируйте и запустите на выполнение серверную программу, исходный код которой приведен в листинге 3.4.
2. Откройте несколько окон утилиты `telnet` (рис. 3.5).
3. Переходя из одного окна в другое, введите команды. В итоге каждое отдельное окно утилиты `telnet` будет независимо от других взаимодействовать с серверной программой.
4. Для того чтобы разорвать соединение и закрыть окно утилиты `telnet`, нажмите комбинацию клавиш <Ctrl+C>.

```
Terminal
File Edit View Terminal Tabs Help
~/books/cj8/code/v2ch03/ThreadedEchoServer$ java ThreadedEchoServer
Spawning 1
Spawning 2
```

```
Terminal
File Edit View Terminal Tabs Help
~$ telnet localhost 8189
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
Hello! Enter BYE to exit.
Hello Sailor!
Echo: Hello Sailor!
How are you?
Echo: How are you?
BYE
Echo: BYE
Connection closed by foreign host.
~$
```

Рис. 3.5. Сеанс одновременной связи нескольких клиентов с сервером

НА ЗАМЕТКУ! В рассматриваемой здесь программе для каждого соединения порождается отдельный поток исполнения. Такой прием не вполне подходит для высокопроизводительного сервера. Более эффективной работы сервера можно добиться, используя средства из пакета `java.nio`. Дополнительные сведения по данному вопросу можно получить, обратившись по адресу https://www.ibm.com/developerworks/java/library/j-javaio/.

Листинг 3.4. Исходный код из файла `threaded/ThreadedEchoServer.java`

```
1  package threaded;
2
3  import java.io.*;
4  import java.net.*;
5  import java.util.*;
6  /**
7   * В этой программе реализуется многопоточный сервер, прослушивающий
8   * порт 8189 и передающий обратно все данные, полученные от клиентов
9   * @author Cay Horstmann
10  * @version 1.21 2012-06-04
11 */
```

```
12 public class ThreadedEchoServer
13 {
14   public static void main(String[] args )
15   {
16     try
17     {
18       int i = 1;
19       ServerSocket s = new ServerSocket(8189);
20       while (true)
21       {
22         Socket incoming = s.accept();
23         System.out.println("Spawning " + i);
24         Runnable r = new ThreadedEchoHandler(incoming);
25         Thread t = new Thread(r);
26         t.start();
27         i++;
28       }
29     }
30     catch (IOException e)
31     {
32       e.printStackTrace();
33     }
34   }
35 }
36
37 /**
38  * Этот класс обрабатывает данные, получаемые сервером от
39  * клиента через одно сокетное соединение
40 */
42 class ThreadedEchoHandler implements Runnable
43 {
44   private Socket incoming;
45
46   /**
47    * Конструирует обработчик
48    * @param i Входящий сокет
49   */
50   public ThreadedEchoHandler(Socket i)
51   {
52     incoming = i;
53   }
54
55   public void run()
56   {
57     try
58     {
59       try
60       {
61         InputStream inStream = incoming.getInputStream();
62         OutputStream outStream = incoming.getOutputStream();
63
64         Scanner in = new Scanner(inStream);
65         PrintWriter out = new PrintWriter(outStream, true);
66                                  // автоматическая очистка
67         out.println( "Hello! Enter BYE to exit." );
68         // передать обратно данные, полученные от клиента
69         boolean done = false;
70         while (!done && in.hasNextLine())
71         {
```

```
72              String line = in.nextLine();
73              out.println("Echo: " + line);
74              if (line.trim().equals("BYE"))
75                done = true;
76            }
77          }
78          finally
79          {
80            incoming.close();
81          }
82        }
83        catch (IOException e)
84        {
85          e.printStackTrace();
86        }
87      }
88  }
```

Полузакрытие

Полузакрытие обеспечивает возможность прервать передачу данных на одной стороне сокетного соединения, продолжая в то же время прием данных от другой стороны. Рассмотрим типичную ситуацию. Допустим, данные направляются на сервер, но заранее неизвестно, какой именно объем данных требуется передать. Если речь идет о файле, то его закрытие, по существу, означает завершение передачи данных. Если же закрыть сокет, то соединение с сервером будет немедленно разорвано.

Преодолеть подобное затруднение призвано полузакрытие. Если закрыть поток вывода через сокет, то для сервера это будет означать завершение передачи данных запроса. При этом поток ввода остается открытым, позволяя получить ответ от сервера. Код, реализующий механизм полузакрытия на стороне клиента, приведен ниже.

```
Socket socket = new Socket(host, port);
Scanner in = new Scanner(socket.getInputStream());
PrintWriter writer = new PrintWriter(socket.getOutputStream());
// передать данные запроса
writer.print(. . .);
writer.flush();
socket.shutdownOutput();
// теперь сокет полузакрыт
// принять данные ответа
while (in.hasNextLine() != null) { String line = in.nextLine(); . . . }
socket.close();
```

Серверная программа просто читает данные из потока ввода до тех пор, пока не закроется поток вывода на другом конце соединения. Очевидно, что такой подход применим только для служб однократного действия по протоколам, подобным HTTP, где клиент устанавливает соединение с сервером, передает запрос, получает ответ, после чего соединение разрывается.

java.net.Socket 1.0

- `void shutdownOutput()` **1.3**

  Устанавливает поток вывода в состояние завершения.

- `void shutdownInput()` **1.3**

  Устанавливает поток ввода в состояние завершения.
- `boolean isOutputShutdown()` **1.4**

  Возвращает логическое значение `true`, если вывод данных был остановлен.
- `boolean isInputShutdown()` **1.4**

  Возвращает логическое значение `true`, если ввод данных был остановлен.

Прерываемые сокеты

При подключении через сокет текущий поток исполнения блокируется до тех пор, пока соединение не будет установлено, или же до истечения времени ожидания. Аналогично, если пытаться передать или принять данные через сокет, текущий поток приостановит свое исполнение до успешного завершения операции или до истечения времени ожидания.

В прикладных программах, работающих в диалоговом режиме, желательно предоставить пользователям возможность прервать слишком затянувшийся процесс установления соединения через сокет. Но если поток исполнения блокирован для нереагирующего сокета, то разблокировать его не удастся, вызвав метод `interrupt()`.

Для прерывания сокетных операций служит класс `SocketChannel`, предоставляемый в пакете `java.nio`. Объект типа `SocketChannel` создается следующим образом:

```
SocketChannel channel = SocketChannel.open(new InetSocketAddress(host,
port));
```

У канала отсутствуют связанные с ним потоки ввода-вывода. Вместо этого в канале предоставляются методы `read()` и `write()`, использующие объекты типа `Buffer`. (Подробнее о буферах см. в главе 1.) Эти методы объявляются в интерфейсах `ReadableByteChannel` и `WritableByteChannel`. Если же нет желания иметь дело с буферами, для чтения из канала типа `SocketChannel` можно воспользоваться объектом типа `Scanner`. Для этой цели в классе `Scanner` предусмотрен следующий конструктор с параметром типа `ReadableByteChannel`:

```
Scanner in = new Scanner(channel);
```

Для того чтобы превратить канал в поток вывода, применяется статический метод `Channels.newOutputStream()`:

```
OutputStream outStream = Channels.newOutputStream(channel);
```

Вот, собственно, и все, что нужно сделать для прерывания сокетной операции. Если же поток исполнения будет прерван в процессе установления соединения, чтения или записи, соответствующая операция завершится и сгенерирует исключение.

В примере программы, исходный код которой приведен в листинге 3.5, демонстрируется применение прерываемых и блокирующих сокетов. Сервер передает числовые данные, имитируя прерывание их передачи после десятого числа. Если щелкнуть на любой кнопке, запустится поток исполнения, устанавливающий соединение с сервером и выводящий на экран передаваемые данные. В первом потоке исполнения используется прерываемый сокет, а во втором — блокирующий. Если щелкнуть

на кнопке Cancel (Отмена) во время вывода первых десяти чисел, то прервется исполнение любого из двух потоков.

А если щелкнуть на кнопке Cancel после передачи первых десяти чисел, то прервется исполнение только первого потока. Блокировка второго потока исполнения будет продолжаться до тех пор, пока сервер не разорвет окончательно соединение (рис. 3.6).

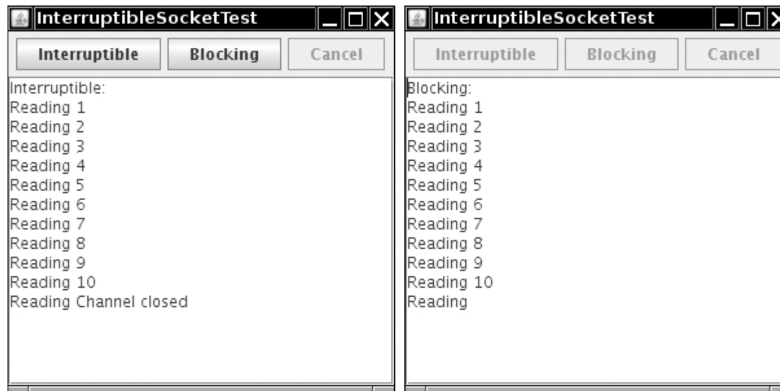

Рис. 3.6. Прерывание сокета

Листинг 3.5. Исходный код из файла `interruptible/InterruptibleSocketTest.java`

```
1   package interruptible;
2
3   import java.awt.*;
4   import java.awt.event.*;
5   import java.util.*;
6   import java.net.*;
7   import java.io.*;
8   import java.nio.channels.*;
9   import javax.swing.*;
10
11  /**
12   * В этой программе демонстрируется прерывание сокета через канал
13   * @author Cay Horstmann
14   * @version 1.03 2012-06-04
15  */
16  public class InterruptibleSocketTest
17  {
18    public static void main(String[] args)
19     {
20    EventQueue.invokeLater(new Runnable()
21       {
22         public void run()
23         {
24           JFrame frame = new InterruptibleSocketFrame();
25           frame.setTitle("InterruptibleSocketTest");
26           frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
27           frame.setVisible(true);
28         }
```

```
29          });
30      }
31  }
32
33  class InterruptibleSocketFrame extends JFrame
34  {
35     public static final int TEXT_ROWS = 20;
36     public static final int TEXT_COLUMNS = 60;
37
38     private Scanner in;
39     private JButton interruptibleButton;
40     private JButton blockingButton;
41     private JButton cancelButton;
42     private JTextArea messages;
43     private TestServer server;
44     private Thread connectThread;
45
46     public InterruptibleSocketFrame()
47     {
48        JPanel northPanel = new JPanel();
49        add(northPanel, BorderLayout.NORTH);
50
51        messages = new JTextArea(TEXT_ROWS, TEXT_COLUMNS);
52        add(new JScrollPane(messages));
53
54        interruptibleButton = new JButton("Interruptible");
55        blockingButton = new JButton("Blocking");
56        northPanel.add(interruptibleButton);
57        northPanel.add(blockingButton);
58
59        interruptibleButton.addActionListener(new ActionListener()
60           {
61              public void actionPerformed(ActionEvent event)
62              {
63                 interruptibleButton.setEnabled(false);
64                 blockingButton.setEnabled(false);
65                 cancelButton.setEnabled(true);
66                 connectThread = new Thread(new Runnable()
67                    {
68                       public void run()
69                       {
70                          try
71                          {
72                             connectInterruptibly();
73                          }
74                          catch (IOException e)
75                          {
76                             messages.append(
77                    "\nInterruptibleSocketTest.connectInterruptibly: " + e);
78                          }
79                       }
80                    });
81                 connectThread.start();
82              }
83           });
84
85        blockingButton.addActionListener(new ActionListener()
86           {
```

```
87          public void actionPerformed(ActionEvent event)
88          {
89            interruptibleButton.setEnabled(false);
90            blockingButton.setEnabled(false);
91            cancelButton.setEnabled(true);
92            connectThread = new Thread(new Runnable()
93              {
94                public void run()
95                {
96                  try
97                  {
98                    connectBlocking();
99                  }
100                 catch (IOException e)
101                 {
102                   messages.append(
103                      "\nInterruptibleSocketTest.connectBlocking: " + e);
104                 }
105               }
106             });
107           connectThread.start();
108         }
109       });
110
111     cancelButton = new JButton("Cancel");
112     cancelButton.setEnabled(false);
113     northPanel.add(cancelButton);
114     cancelButton.addActionListener(new ActionListener()
115       {
116         public void actionPerformed(ActionEvent event)
117         {
118           connectThread.interrupt();
119           cancelButton.setEnabled(false);
120         }
121       });
122     server = new TestServer();
123     new Thread(server).start();
124     pack();
125   }
126
127   /**
128    * Соединяет с проверяемым сервером, используя прерываемый ввод-вывод
129   */
130   public void connectInterruptibly() throws IOException
131   {
132     messages.append("Interruptible:\n");
133     try (SocketChannel channel =
134         SocketChannel.open(new InetSocketAddress("localhost", 8189)))
135     {
136       in = new Scanner(channel);
137       while (!Thread.currentThread().isInterrupted())
138       {
139         messages.append("Reading ");
140         if (in.hasNextLine())
141         {
142           String line = in.nextLine();
143           messages.append(line);
144           messages.append("\n");
```

```
145         }
146       }
147     }
148     finally
149     {
150       EventQueue.invokeLater(new Runnable()
151         {
152           public void run()
153           {
154             messages.append("Channel closed\n");
155             interruptibleButton.setEnabled(true);
156             blockingButton.setEnabled(true);
157           }
158         });
159     }
160   }
161
162   /**
163    * Соединяет с проверяемым сервером, используя блокирующий ввод-вывод
164   */
165   public void connectBlocking() throws IOException
166   {
167     messages.append("Blocking:\n");
168     try (Socket sock = new Socket("localhost", 8189))
169     {
170       in = new Scanner(sock.getInputStream());
171       while (!Thread.currentThread().isInterrupted())
172       {
173         messages.append("Reading ");
174         if (in.hasNextLine())
175         {
176           String line = in.nextLine();
177           messages.append(line);
178           messages.append("\n");
179         }
180       }
181     }
182     finally
183     {
184       EventQueue.invokeLater(new Runnable()
185         {
186           public void run()
187           {
188             messages.append("Socket closed\n");
189             interruptibleButton.setEnabled(true);
190             blockingButton.setEnabled(true);
191           }
192         });
193     }
194   }
195
196   /**
197    * Многопоточный сервер, прослушивающий порт 8189 и посылающий
198    * клиентам числа, имитируя зависание после передачи 10 чисел
199   */
200   class TestServer implements Runnable
201   {
202     public void run()
```

```
203     {
204       try
205       {
206         ServerSocket s = new ServerSocket(8189);
207
208         while (true)
209         {
210           Socket incoming = s.accept();
211           Runnable r = new TestServerHandler(incoming);
212           Thread t = new Thread(r);
213           t.start();
214         }
215       }
216       catch (IOException e)
217       {
218         messages.append("\nTestServer.run: " + e);
219       }
220     }
221   }
222
223   /**
224    * Этот класс обрабатывает данные, получаемые сервером
225    * от клиента через одно сокетное соединение
226   */
227   class TestServerHandler implements Runnable
228   {
229     private Socket incoming;
230     private int counter;
231
232     /**
233      * Конструирует обработчик
234      * @param i Входящий сокет
235     */
236     public TestServerHandler(Socket i)
237     {
238       incoming = i;
239     }
240
241     public void run()
242     {
243       try
244       {
245         try
246         {
247           OutputStream outStream = incoming.getOutputStream();
248           PrintWriter out = new PrintWriter(outStream, true);
249           while (counter < 100)
250           {
251             counter++;
252             if (counter <= 10) out.println(counter);
253             Thread.sleep(100);
254           }
255         }
256         finally
257         {
258           incoming.close();
259            messages.append("Closing server\n");
260         }
```

```
261        }
262        catch (Exception e)
263        {
264          messages.append("\nTestServerHandler.run: " + e);
265        }
266      }
267    }
268 }
```

java.net.InetSocketAddress 1.4

- `InetSocketAddress(String hostname, int port)`
- Создает объект адреса с указанными именем сетевого узла (хоста) и номером порта, преобразуя имя узла в адрес при установлении соединения. Если преобразовать имя сетевого узла в адрес не удается, устанавливается логическое значение `true` свойства `unresolved`.
- `boolean isUnresolved()`
- Возвращает логическое значение `true`, если для данного объекта не удается преобразовать имя сетевого узла в адрес.

java.nio.channels.SocketChannel 1.4

- `static SocketChannel open(SocketAddress address)`
- Открывает канал для сокета и связывает его с удаленным сетевым узлом по указанному адресу.

java.nio.channels.Channels 1.4

- `static InputStream newInputStream(ReadableByteChannel channel)`
- Создает поток ввода для чтения данных из указанного канала.
- `static OutputStream newOutputStream(WritableByteChannel channel)`
- Создает поток вывода для записи данных в указанный канал.

Получение данных из Интернета

Для того чтобы получить доступ к веб-серверам из программы на Java, требуется более высокий уровень сетевого взаимодействия, чем установление соединения через сокет и выдача HTTP-запросов. В последующих разделах будут рассмотрены классы, предоставляемые для этой цели в библиотеке Java.

URL и URI

Классы `URL` и `URLConnection` инкапсулируют бо́льшую часть внутреннего механизма извлечения данных с удаленного веб-сайта. Объект типа `URL` создается следующим образом:

```
URL url = new URL(символьная строка с URL);
```

Если требуется только извлечь содержимое из указанного ресурса, достаточно вызвать метод `openStream()` из класса `URL`. Этот метод возвращает объект типа `InputStream`. Поток ввода данного типа можно использовать обычным образом, например, создать объект типа `Scanner`:

```
InputStream inStream = url.openStream();
Scanner in = new Scanner(inStream);
```

В пакете `java.net` отчетливо различаются унифицированные *указатели* ресурсов (URL) и унифицированные *идентификаторы* ресурсов (URI). В частности, URI — это лишь синтаксическая конструкция, содержащая различные части символьной строки, обозначающей веб-ресурс. А URL — это особая разновидность идентификатора URI с исчерпывающими данными о местоположении ресурса. Имеются и такие URI, как, например, `mailto:cay@hortsmann.com`, которые не являются указателями ресурсов, потому что по ним нельзя обнаружить какие-нибудь данные. Такой URI называется унифицированным *именем* ресурса (URN).

В классе `URI` из библиотеки Java отсутствуют методы доступа к ресурсу по указанному идентификатору, поскольку этот класс предназначен только для синтаксического анализа символьной строки, обозначающей ресурс. В отличие от него, класс `URL` позволяет открыть поток ввода-вывода для данного ресурса. Поэтому в классе `URL` допускается взаимодействие только по тем протоколам и схемам, которые поддерживаются в библиотеке Java, в том числе `http:`, `https:` и `ftp:` — для Интернета, `file:` — для локальной файловой системы, а также `jar:` — для обращения к JAR-файлам.

Синтаксический анализ URI — непростая задача, поскольку идентификаторы ресурсов могут иметь сложную структуру. В качестве примера ниже приведены URI с замысловатой структурой.

```
http://maps.yahoo.com/py/maps.py?csz=Cupertino+CA
ftp://username:password@ftp.yourserver.com/pub/file.txt
```

В обозначении URI задаются правила построения таких идентификаторов. Структура URI выглядит следующим образом:

```
[схема:]специальная_часть_схемы[#фрагмент]
```

где квадратные скобки (**[]**) обозначают необязательную часть, а двоеточие и знак **#** служат в качестве разделителей. Если *схема*: присутствует как составная часть в идентификаторе URI, то он называется *абсолютным*, а иначе — *относительным*. Абсолютный URI называется *непрозрачным*, если *специальная_часть_схемы* не начинается с косой черты (**/**), как, например, показано ниже.

```
mailto:cay@horstmann.com
```

Все абсолютные, прозрачные URI и все относительные URL имеют *иерархическую структуру*. Ниже приведены характерные тому примеры.

```
http://horstmann.com/index.html
../../java/net/Socket.html#Socket()
```

Составляющая *специальная_часть_схемы* иерархического URI имеет следующую структуру:

```
[//полномочия][путь][?запрос]
```

И здесь квадратные скобки (**[]**) обозначают необязательную часть. В URI серверов составляющая *полномочия* имеет приведенную ниже форму, где элемент *порт* должен иметь целочисленное значение.

```
[сведения_о_пользователе@]хост[:порт]
```

В документе RFC 2396, стандартизирующем идентификаторы URI, допускается также механизм указания составляющей *полномочия* в другом формате на основе данных из реестра. Но он не получил широкого распространения.

Одно из назначений класса `URI` состоит в синтаксическом анализе отдельных составляющих идентификатора. Они извлекаются с помощью перечисленных ниже методов.

```
getScheme()
getSchemeSpecificPart()
getAuthority()
getUserInfo()
getHost()
getPort()
getPath()
getQuery()
getFragment()
```

Другое назначение класса `URI` состоит в обработке абсолютных и относительных идентификаторов. Так, если имеются абсолютный и относительный идентификаторы URI:

```
http://docs.mycompany.com/api/java/net/ServerSocket.html
```

и

```
../../java/net/Socket.html#Socket()
```

их можно объединить в абсолютный URI следующим образом:

```
http://docs.mycompany.com/api/java/net/Socket.html#Socket()
```

Такой процесс называется *преобразованием* относительного URI. Обратный процесс называется *преобразованием абсолютных адресов в относительные*. Например, имея *базовый* URI:

```
http://docs.mycompany.com/api
```

можно преобразовать следующий абсолютный URI:

```
http://docs.company.com/api/java/lang/String.html
```

в приведенный ниже относительный URI.

```
java/lang/String.html
```

Для выполнения обоих видов преобразования в классе `URI` предусмотрены два соответствующих метода:

```
relative = base.relativize(combined);
combined = base.resolve(relative);
```

Извлечение данных средствами класса `URLConnection`

Для получения дополнительных сведений о веб-ресурсе следует воспользоваться классом `URLConnection`, предоставляющим намного больше средств управления доступом к веб-ресурсам, чем более простой класс `URL`. Для работы с объектом типа `URLConnection` необходимо тщательно спланировать и выполнить следующие действия.

1. Вызвать метод `openConnection()` из класса `URL` для получения объекта типа `URLConnection` следующим образом:

   ```
   URLConnection connection = url.openConnection();
   ```

2. Задать свойства запроса с помощью перечисленных ниже методов.

```
setDoInput()
setDoOutput()
setIfModifiedSince()
setUseCaches()
setAllowUserInteraction()
setRequestProperty()
setConnectTimeout()
setReadTimeout()
```

3. Эти методы будут подробно рассматриваться далее.
4. Установить соединение с удаленным ресурсом с помощью метода `connect()`:

```
connection.connect();
```

5. Помимо создания сокета, для установления соединения с веб-сервером этот метод запрашивает также у сервера *данные заголовка*.
6. После подключения к веб-серверу становятся доступными поля заголовка. Обращаться к ним можно с помощью универсальных методов `getHeaderFieldKey()` и `getHeaderField()`. Кроме того, для удобства разработки предусмотрены перечисленные ниже методы обработки стандартных полей запроса.

```
getContentType()
getContentLength()
getContentEncoding()
getDate()
getExpiration()
getLastModified()
```

7. И наконец, для доступа к данным указанного ресурса следует вызвать метод `getInputStream()`, предоставляющий поток ввода для чтения данных. (Это тот же самый поток ввода, который возвращается методом `openStream()` из класса `URL`.) Существует также метод `getContent()`, но он не такой удобный. Для обработки содержимого стандартных типов, например, текста (`text/plain`) или изображений (`image/gif`), придется воспользоваться классами из пакета `com.sun`. Кроме того, можно зарегистрировать собственные обработчики содержимого, но они в данной книге не рассматриваются.

НА ЗАМЕТКУ! Некоторые разработчики, пользующиеся классом `URLConnection`, ошибочно считают, что методы `getInputStream()` и `getOutpuStream()` аналогичны одноименным методам из класса `Socket`. Это не совсем так. Класс `URLConnection` способен выполнять много других функций, в том числе обрабатывать заголовки запросов и ответов. Поэтому рекомендуется строго придерживаться указанной выше последовательности действий.

Рассмотрим методы из класса `URLConnection` более подробно. В нем имеется ряд методов, задающих свойства соединения еще до подключения к веб-серверу. Наиболее важными среди них являются методы `setDoInput()` и `setDoOutput()`. По умолчанию при соединении предоставляется поток ввода для чтения данных с веб-сервера, но не поток вывода для записи данных. Для того чтобы получить поток вывода (например, с целью разместить данные на веб-сервере) нужно сделать следующий вызов:

```
connection.setDoOutput(true);
```

Далее можно установить ряд заголовков запроса и послать их веб-серверу в составе единого запроса. Ниже приведен пример заголовков запроса.

```
GET www.server.com/index.html HTTP/1.0
Referer: http://www.somewhere.com/links.html
Proxy-Connection: Keep-Alive
User-Agent: Mozilla/5.0 (X11; U; Linux i686; en-US; rv:1.8.1.4)
Host: www.server.com
Accept: text/html, image/gif, image/jpeg, image/png, */*
Accept-Language: en
Accept-Charset: iso-8859-1,*,utf-8
Cookie: orangemilano=192218887821987
```

Метод `setIfModifiedSince()` служит для уведомления о том, что требуется получить только те данные, которые были изменены после определенной даты.

Методы `setUseCaches()` и `setAllowUserInteraction()` следует вызывать только в аплетах. В частности, метод `setUseCaches()` предписывает браузеру проверить сначала кеш. А метод `setAllowUserInteraction()` позволяет открыть в аплете диалоговое окно с предложением ввести имя и пароль пользователя для защиты доступа к сетевым ресурсам (рис. 3.7).

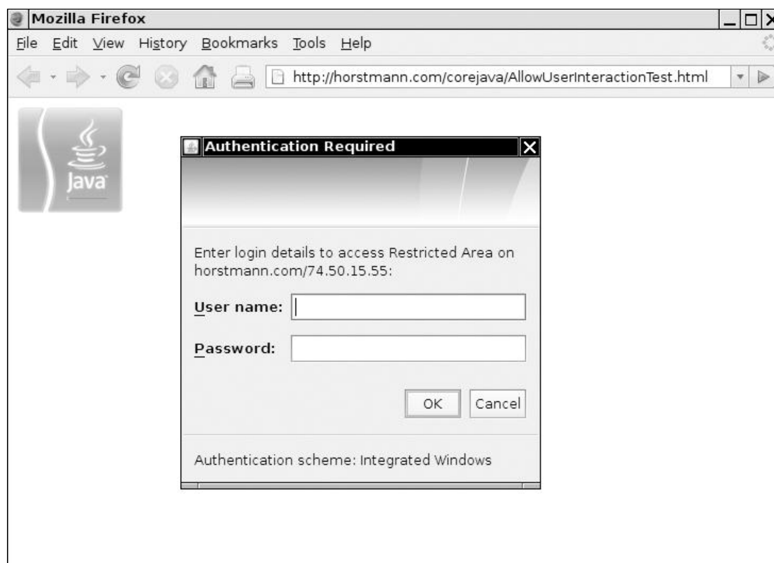

Рис. 3.7. Диалоговое окно для ввода имени и пароля доступа к сетевым ресурсам

И наконец, с помощью метода `setRequestProperty()` можно установить пару "имя–значение", имеющую определенный смысл для конкретного протокола. Формат заголовка запроса по протоколу HTTP описан в документе RFC 2616. Некоторые его параметры не очень хорошо документированы, поэтому за дополнительными разъяснениями зачастую приходится обращаться к другим программистам. Так, для доступа к защищенной паролем веб-странице необходимо выполнить следующие действия.

1. Составьте символьную строку из имени пользователя, двоеточия и пароля:
```
String input = username + ":" + password;
```
2. Перекодируйте полученную в итоге символьную строку по алгоритму кодирования Base64, как показано ниже. (Этот алгоритм преобразует последовательность байтов в последовательность символов в коде ASCII.)
```
String encoding = base64Encode(input);
```
3. Вызовите метод `setRequestProperty()` с именем свойства `"Authorization"` и значением `"Basic " + encoding`, как показано ниже.
```
connection.setRequestProperty("Authorization", "Basic " + encoding);
```

СОВЕТ. Здесь рассматривается способ обращения к защищенной паролем веб-странице. А для доступа к защищенному паролем FTP-файлу применяется совершенно другой подход. В этом случае достаточно сформировать URL следующего вида:

`ftp://`*`имя_пользователя`*`:`*`пароль`*`@ftp.`*`ваш_сервер`*`.com/pub/file.txt`

После вызова метода `connect()` можно запросить данные заголовка из ответа. Рассмотрим сначала способ перечисления всех полей заголовка. Создатели рассматриваемого здесь класса посчитали нужным создать собственный способ перебора полей. Так, при вызове приведенного ниже метода получается *n*-й ключ заголовка, причем нумерация начинается с единицы! В итоге возвращается пустое значение `null`, если *n* равно нулю или больше общего количества полей заголовка.

```
String key = connection.getHeaderFieldKey(n);
```

Но для определения количества полей не предусмотрено никакого другого метода. Чтобы перебрать все поля, приходится просто вызывать метод `getHeaderFieldKey()` до тех пор, пока не будет получено пустое значение `null`. Аналогично при вызове следующего метода возвращается значение из *n*-го поля:

```
String value = connection.getHeaderField(n);
```

Метод `getHeaderFields()` возвращает объект типа `Map` с полями заголовка, как показано ниже.

```
Map<String,List<String>> headerFields = connection.getHeaderFields();
```

В качестве примера ниже приведен ряд полей заголовка из типичного ответа на запрос по протоколу HTTP.

```
Date: Wed, 27 Aug 2008 00:15:48 GMT
Server: Apache/2.2.2 (Unix)
Last-Modified: Sun, 22 Jun 2008 20:53:38 GMT
Accept-Ranges: bytes
Content-Length: 4813
Connection: close
Content-Type: text/html
```

Для удобства разработки предусмотрены шесть методов, получающих значения из наиболее употребительных полей заголовка и приводящие эти значения к соответствующим числовым типам по мере необходимости. Все эти служебные методы перечислены в табл. 3.1. В методах, возвращающих значения типа `long`, отсчет количества возвращаемых секунд начинается с полуночи 1 января 1970 г.

Таблица 3.1. Служебные методы, получающие значения полей заголовка из ответа на запрос

Имя поля (ключа)	Имя метода	Возвращаемое значение
Date	getDate	long
Expires	getExpiration	long
Last-Modified	getLastModified	long
Content-Length	getContentLength	int
Content-Type	getContentType	String
Content-Encoding	getContentEncoding	String

В примере программы, исходный код которой приведен в листинге 3.6, предоставляется возможность поэкспериментировать с соединениями по URL. Запустив программу, вы можете указать в командной строке конкретный URL, имя пользователя и пароль:

```
java urlConnection.URLConnectionTest http://www.сервер.com пользователь пароль
```

В итоге программа выведет на экран следующее.

- Все ключи и значения из полей заголовка.
- Значения, возвращаемые шестью служебными методами доступа к наиболее употребительным полям заголовка, как показано в табл. 3.1.
- Первые 10 символьных строк из запрашиваемого ресурса.

Это довольно простая программа, за исключением той части, где выполняется кодирование по алгоритму Base64. Вместо используемого в данном примере класса можно применить недокументированный класс `sun.misc.BASE64Encoder`. Для этого достаточно заменить используемый вызов метода `base64Encode()` вызовом аналогичного метода, приведенного ниже. Но в данном примере мы не стали полагаться на недокументированные классы, использовав в ней свой класс.

```
String encoding = new sun.misc.BASE64Encoder().encode(input.getBytes());
```

НА ЗАМЕТКУ! Класс `javax.mail.internet.MimeUtility` из стандартного расширения JavaMail библиотеки Java также содержит метод для кодирования по алгоритму Base64. В состав JDK входит также класс `java.util.prefs.Base64`, который служит той же самой цели, но он не является открытым, и поэтому его нельзя использовать в прикладных программах.

Листинг 3.6. Исходный код из файла `urlConnection/URLConnectionTest.java`

```
1   package urlConnection;
2
3   import java.io.*;
4   import java.net.*;
5   import java.util.*;
6
7   /**
8    * В этой программе устанавливается соединение по заданному URL и
9    * отображаются данные заголовка из получаемого ответа, а также
10   * первые 10 строк запрашиваемых данных. Для этого в командной строке
11   * следует указать конкретный URL и дополнительно имя пользователя и
12   * пароль (для элементарной аутентификации по протоколу HTTP)
```

```
13   * @version 1.11 2007-06-26
14   * @author Cay Horstmann
15  */
16  public class URLConnectionTest
17  {
18    public static void main(String[] args)
19    {
20      try
21      {
22        String urlName;
23        if (args.length > 0) urlName = args[0];
24        else urlName = "http://horstmann.com";
25
26        URL url = new URL(urlName);
27        URLConnection connection = url.openConnection();
28
29        // установить имя пользователя и пароль, если они
30        // указаны в командной строке
31        if (args.length > 2)
32        {
33          String username = args[1];
34          String password = args[2];
35          String input = username + ":" + password;
36          String encoding = base64Encode(input);
37          connection.setRequestProperty(
38              "Authorization", "Basic " + encoding);
39        }
40
41        connection.connect();
42
43        // вывести поля заголовка
44        Map<String, List<String>> headers = connection.getHeaderFields();
45        for (Map.Entry<String, List<String>> entry : headers.entrySet())
46        {
47          String key = entry.getKey();
48          for (String value : entry.getValue())
49            System.out.println(key + ": " + value);
50        }
51
52        // вывести значения полей заголовка, используя служебные методы
53        System.out.println("----------");
54        System.out.println(
55            "getContentType: " + connection.getContentType());
56        System.out.println(
57            "getContentLength: " + connection.getContentLength());
58        System.out.println(
59            "getContentEncoding: " + connection.getContentEncoding());
60        System.out.println("getDate: " + connection.getDate());
61        System.out.println(
62            "getExpiration: " + connection.getExpiration());
63        System.out.println(
64            "getLastModifed: " + connection.getLastModified());
65        System.out.println("----------");
66
67        Scanner in = new Scanner(connection.getInputStream());
68
69        // вывести первые десять строк запрашиваемого содержимого
70        for (int n = 1; in.hasNextLine() && n <= 10; n++)
71          System.out.println(in.nextLine());
```

```
72        if (in.hasNextLine()) System.out.println(". . .");
73      }
74      catch (IOException e)
75      {
76        e.printStackTrace();
77      }
78    }
79
80    /**
81     * Кодирует символьную строку по алгоритму Base64
82     * @param s Символьная строка
83     * @return Возвращает символьную строку s,
85     *         закодированную по алгоритму Base64
86    */
87    public static String base64Encode(String s)
88    {
89      ByteArrayOutputStream bOut = new ByteArrayOutputStream();
90      Base64OutputStream out = new Base64OutputStream(bOut);
91      try
92      {
93        out.write(s.getBytes());
94        out.flush();
95      }
96      catch (IOException e)
97      {
98      }
99      return bOut.toString();
100   }
101 }
103 /**
104  * Этот потоковый фильтр преобразует поток вывода байтов в их
105  * кодировку по алгоритму Base64, где каждые три байта кодируются четырьмя
106  * символами. Каждые шесть битов в кодировке |11111122|22223333|33444444|
107  * кодируются по таблице toBase64. Если количество вводимых байтов не
108  * кратно трем, то последняя группа из четырех символов заполняется
109  * одним или двумя знаками равенства. А каждая выводимая строка состоит
110  * не больше, чем из 76 символов
111 */
112 class Base64OutputStream extends FilterOutputStream
113 {
114   private static char[] toBase64 =
115     { 'A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J', 'K', 'L',
116       'M', 'N', 'O', 'P', 'Q', 'R', 'S', 'T', 'U', 'V', 'W', 'X',
117       'Y', 'Z', 'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j',
118       'k', 'l', 'm', 'n', 'o', 'p', 'q', 'r', 's', 't', 'u', 'v',
119       'w', 'x', 'y', 'z', '0', '1', '2', '3', '4', '5', '6', '7',
120       '8', '9', '+', '/' };
121
122   private int col = 0;
123   private int i = 0;
124   private int[] inbuf = new int[3];
125
126   /**
127    * Конструирует потоковый фильтр
128    * @param out Потоковый фильтр
129   */
130   public Base64OutputStream(OutputStream out)
131   {
132     super(out);
```

```
133   }
134
135   public void write(int c) throws IOException
136   {
137     inbuf[i] = c;
138     i++;
139     if (i == 3)
140     {
141       if (col >= 76)
142       {
143         super.write('\n');
144         col = 0;
145       }
146       super.write(toBase64[(inbuf[0] & 0xFC) >> 2]);
147       super.write(
148          toBase64[((inbuf[0] & 0x03) << 4) | ((inbuf[1] & 0xF0) >> 4)]);
149       super.write(
150          toBase64[((inbuf[1] & 0x0F) << 2) | ((inbuf[2] & 0xC0) >> 6)]);
151       super.write(toBase64[inbuf[2] & 0x3F]);
152       col += 4;
153       i = 0;
154     }
155   }
156   public void flush() throws IOException
157   {
158     if (i > 0 && col >= 76)
159     {
160       super.write('\n');
161       col = 0;
162     }
163     if (i == 1)
164     {
165       super.write(toBase64[(inbuf[0] & 0xFC) >> 2]);
166       super.write(toBase64[(inbuf[0] & 0x03) << 4]);
167       super.write('=');
168       super.write('=');
169     }
170     else if (i == 2)
171    {
172      super.write(toBase64[(inbuf[0] & 0xFC) >> 2]);
173      super.write(
174         toBase64[((inbuf[0] & 0x03) << 4) | ((inbuf[1] & 0xF0) >> 4)]);
175      super.write(toBase64[(inbuf[1] & 0x0F) << 2]);
176      super.write('=');
177     }
178   }
179 }
```

java.net.URL 1.0

- `InputStream openStream()`

  Открывает поток ввода для чтения данных из ресурса.

- `URLConnection openConnection();`

  Возвращает объект типа `URLConnection`, управляющий соединением с ресурсом.

java.net.URLConnection 1.0

- `void setDoInput(boolean doInput)`
- `boolean getDoInput()`

  Если параметр `doInput` принимает логическое значение `true`, пользователь может принимать вводимые данные из текущего объекта типа `URLConnection`.

- `void setDoOutput(boolean doOutput)`
- `boolean getDoOutput()`

  Если параметр *`doOutput`* принимает логическое значение `true`, пользователь может передавать выводимые данные в текущий объект типа `URLConnection`.

- `void setIfModifiedSince(long time)`
- `long getIfModifiedSince()`

  Свойство `ifModifiedSince` настраивает данный объект типа `URLConnection` на извлечение только тех данных, которые были изменены после указанного момента времени. Время задается в секундах, начиная с полуночи 1 января 1970 г. по Гринвичу.

- `void setUseCaches(boolean useCaches)`
- `boolean getUseCaches()`

  Если параметр `useCaches` принимает логическое значение `true`, данные можно извлечь из локального кеша. Следует, однако, иметь в виду, что кеш не поддерживается самим объектом типа `URLConnection`. Поэтому кеш должен быть предоставлен внешней программой, например, браузером.

- `void setAllowUserInteraction(boolean allowUsreInteraction)`
- `boolean getAllowUserInteraction()`

  Если параметр *`allowUserInteraction`* принимает логическое значение, у пользователя может запрашиваться пароль. Следует, однако, иметь в виду, что у самого объекта типа `URLConnection` отсутствуют средства, требующиеся для выполнения подобных запросов. Поэтому запрос пароля должен быть организован во внешней программе, например, в браузере или подключаемом модуле.

- `void setConnectTimeout(int timeout)` **5.0**
- `int getConnectTimeout()` **5.0**

  Устанавливают или возвращают величину времени ожидания (в миллисекундах) для соединения. Если время ожидания истечет до установления соединения, метод `connect()` из соответствующего потока ввода сгенерирует исключение типа `SocketTimeoutException`.

- `void setReadTimeout(int timeout)` **5.0**
- `int getReadTimeout()` **5.0**

  Устанавливают или возвращают величину времени ожидания (в миллисекундах) для чтения данных. Если время ожидания истечет до успешного завершения операции чтения, метод `read()` сгенерирует исключение типа `SocketTimeoutException`.

- `void setRequestProperty(String key, String value)`

  Устанавливает значение в поле заголовка.

- `Map<String,List<String>> getRequestProperties()` **1.4**

  Возвращает отображение со свойствами запроса. Все свойства одного и того же ключа вносятся в список.

- `void connect()`

  Устанавливает соединение с удаленным ресурсом и получает данные заголовка из ответа.

- `Map<String,List<String>> getHeaderFields()` **1.4**

  Возвращает отображение с полями заголовка из ответа. Все свойства одного и того же ключа вносятся в список.

- `String getHeaderFieldKey(int n)`

  Возвращает ключ *n*-го поля заголовка из ответа или пустое значение `null`, если *n* меньше или равно нулю или превышает количество полей.

- `String getHeaderField(int n)`

  Возвращает значение *n*-го поля заголовка из ответа или пустое значение `null`, если *n* меньше или равно нулю или превышает количество полей.

- `int getContentLength()`

  Возвращает длину доступного содержимого или **-1**, если длина неизвестна.

- `String getContentType()`

  Возвращает тип содержимого, например, `text/plain` или `image/gif`.

- `String getContentEncoding()`

  Возвращает кодировку содержимого, например `gzip`. Применяется редко, потому что используемая по умолчанию кодировка не всегда указывается в поле `identity` заголовка `Content-Encoding`.

- `long getDate()`
- `long getExpiration()`
- `long getLastModifed()`

  Возвращают время создания, последней модификации ресурса или время, когда срок действия ресурса истекает. Время указывается в секундах, начиная с 1 января 1970 г. по Гринвичу.

- `InputStream getInputStream()`
- `OutputStream getOutputStream()`

  Возвращают поток ввода для чтения данных из ресурса или вывода для записи данных в ресурс.

- `Object getContent()`

  Выбирает подходящий обработчик содержимого для чтения данных из ресурса. Этот метод вряд ли полезен для чтения данных стандартного типа, например, `text/plain` или `image/gif`, кроме тех случаев, когда требуется создать собственный обработчик этих типов данных.

Отправка данных формы

В предыдущем разделе описывался способ чтения данных с веб-сервера, а в этом разделе рассматривается способ передачи данных из клиентской программы на веб-сервер, а также другим программам, которые может вызывать веб-сервер. Для передачи данных из браузера на веб-сервер нужно заполнить форму, аналогичную приведенной на рис. 3.8.

Рис. 3.8. HTML-форма

Когда пользователь щелкает на кнопке Submit (Предъявить), данные, введенные в текстовых полях, а также сведения о состоянии флажков и кнопок-переключателей передаются на веб-сервер. Получив данные, введенные пользователем в форме, веб-сервер вызывает программу для их последующей обработки.

Существует целый ряд технологий, позволяющих веб-серверу вызывать программы для обработки данных. Наиболее часто для этой цели используются сервлеты на Java, JavaServer Faces, Microsoft ASP (Active Server Pages) и сценарии CGI. Ради простоты все программы, выполняющиеся на стороне сервера, будут обозначаться единообразно как *сценарий*, независимо от конкретной применяемой технологии.

Сценарий, выполняющийся на стороне сервера, обрабатывает данные, введенные пользователем в форме, и формирует новую HTML-страницу, которую веб-сервер передает обратно браузеру. Последовательность действий по обработке данных из формы условно показана на рис. 3.9. Страница, сформированная сервером, может содержать новые данные (например, результаты поиска) или только подтверждение о получении введенных данных. Здесь и далее не рассматриваются вопросы реализации сценариев, выполняемых на стороне сервера, а основное внимание уделяется написанию клиентских программ, предназначенных для взаимодействия с готовыми сценариями.

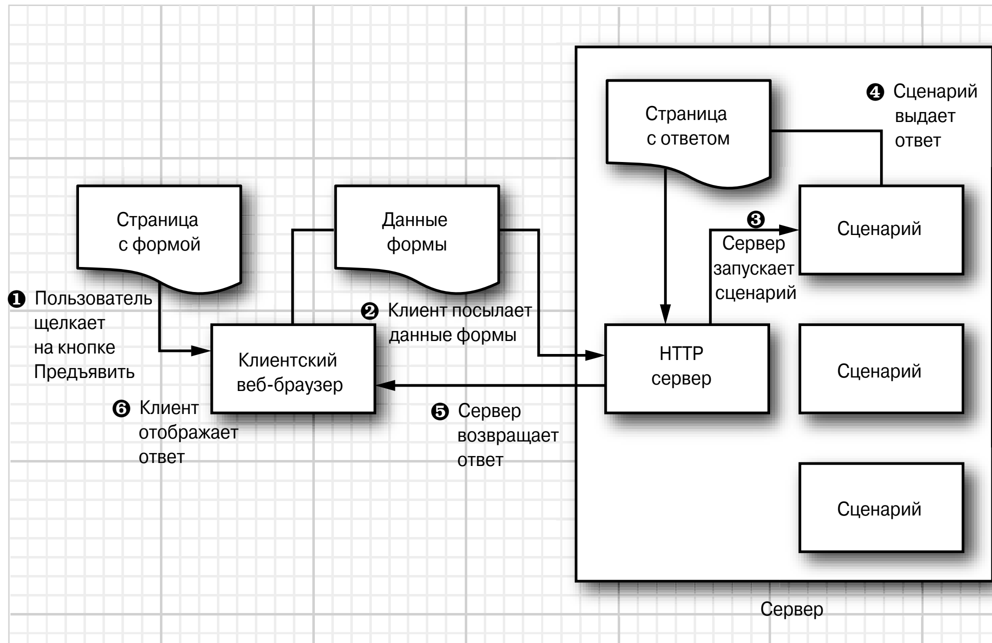

Рис. 3.9. Порядок обработки данных по сценарию на стороне сервера

При передаче данных на веб-сервер не имеет никакого значения, будет ли использован для их интерпретации сценарий CGI, сервлет или программа другого типа. Клиент посылает данные на веб-сервер в стандартном формате, а веб-сервер должен сам найти ту программу, которая выдаст нужный ответ.

Передача данных на веб-сервер может осуществляться по командам `GET` и `POST`. При выдаче команды `GET` параметры просто указываются в конце URL в следующем формате:

```
http://хост/сценарий?параметры
```

Каждый параметр имеет вид "имя–значение". Параметры разделяются знаками **&**. Значения параметров кодируются по схеме кодирования URL, которая подчиняется следующим правилам.

- Символы от **A** до **Z**, от **a** до **z**, от **0** до **9**, а также знаки **.**, **-**, **~** и **_** остаются без изменения.
- Все пробелы заменяются знаками **+**.
- Все остальные символы кодируются в кодировке UTF-8, а каждый байт преобразуется к виду **%UV**, где **UV** — двухзначное шестнадцатеричное число.

Например, название города и штата *New York, NY* передается в закодированном виде как `New+York%2c+NY`. Здесь шестнадцатеричное число **2c** (или десятичное **44**) обозначает запятую в коде ASCII. Благодаря такому способу кодирования промежуточные программы не будут путаться в пробелах и смогут правильно интерпретировать другие символы.

Так, на момент написания этой книги веб-сервер Yahoo! содержал сценарий `py.maps.py` на хосте `maps.yahoo.com`. Для выполнения этого сценария требуются два параметра: `addr` и `csz`. Чтобы получить карту местности по адресу 1 Market Street, San Francisco, CA, нужно сформировать следующий URL:

```
http://maps.yahoo.com/py/maps.py?addr=1+Market+Street&csz=San+Francisco+CA
```

Команда `GET` очень проста в употреблении, но имеет существенный недостаток: в большинстве браузеров количество символов, которые можно включить в запрос по команде `GET`, ограничено.

Параметры команды `POST` не нужно включать в состав URL. Вместо этого нужно получить поток вывода из объекта типа `URLConnection` и записать в него пары "имя–значение". Кроме того, значения, включаемые в URL, необходимо закодировать, разделив их знаком **&**. Рассмотрим этот процесс более подробно. Для передачи данных сценарию сначала создается объект типа `URLConnection` следующим образом:

```
URL url = new URL("http://хост/сценарий");
URLConnection connection = url.openConnection();
```

Затем вызывается метод `setDoOutput()`, чтобы установить соединение для передачи данных:

```
connection.setDoOutput(true);
```

Далее вызывается метод `getOutputStream()`, чтобы получить поток вывода. Для передачи текстовых данных поток вывода удобно инкапсулировать в объект типа `PrintWriter` следующим образом:

```
PrintWriter out = new PrintWriter(connection.getOutputStream());
```

Теперь можно передать данные на сервер, как следует из приведенного ниже фрагмента кода.

```
out.print(name1 + "=" + URLEncoder.encode(value1, "UTF-8") + "&");
out.print(name2 + "=" + URLEncoder.encode(value2, "UTF-8"));
```

После передачи данных выходной поток вывода закрывается следующим образом:

```
out.close();
```

И, наконец, вызывается метод `getInputStream()`, чтобы прочитать ответ с сервера.

Рассмотрим следующий пример. Веб-сайт `http://esa.un.org/unpd/wpp/unpp/panel_population.htm` содержит страницу с формой для ввода запроса на получение статистических сведений о численности и структуре населения (см. рис. 3.8). В HTML-разметке данной формы содержится следующий дескриптор:

```
<form method="post" onSubmit="return checksubmit(. . .)" . . .>
```

Если проанализировать исходный код функции `checksubmit()` сценария на JavaScript, то можно обнаружить, что загрузка значений, разделенных запятыми, вызывает приведенное ниже действие, в котором указывается имя сценария, выполняющего команду `POST`.

```
document.menuForm.action ="p2k0data_script.asp";
```

Далее нужно найти имена полей, которые потребуются указанному сценарию. Внимательно проанализируем элементы разметки формы. У каждого из них имеется свой атрибут `name`, как показано ниже.

```
<select name="Variable">
<option value="12;">Population</option>
  дополнительные параметры . . .
</select>
```

В данном случае поле имеет имя `Variable`. Оно содержит тип заполнения таблиц статистическими данными. Например, таблица типа `"12;"` содержит оценки общей численности населения. Далее в HTML-разметке данной формы можно найти поле `Location` с параметрами, обозначающими сокращенные названия стран, например **900** (весь мир) и **404** (Кения).

Необходимо также установить ряд других полей. Например, чтобы получить оценку численности населения в Кении в период с 1950 по 2050 гг., нужно сформировать следующую символьную строку:

```
Panel=1&Variable=12%3b&Location=404&Varient=2&StartYear=1950&EndYear=2050;
```

Затем эта символьная строка посылается по следующему URL:

`http://esa.un.org/unpd/wpp/unpp/p2k0data.asp`

Обработав полученные данные, сценарий пришлет в итоге следующий ответ:

```
"Country","Variable","Variant","Year","Value"
"Kenya","Population (thousands)","Medium variant","1950",6077
"Kenya","Population (thousands)","Medium variant","1955",6984
"Kenya","Population (thousands)","Medium variant","1960",8115
"Kenya","Population (thousands)","Medium variant","1965",9524
. . .
```

Как видите, сценарий отправляет назад файл с данными, разделенными запятыми. Этот сценарий был выбран для данного примера потому, что в нем нетрудно разобраться, не утруждая себя анализом HTML-дескрипторов в более сложных сценариях.

В листинге 3.7 приведен исходный код примера программы, посылающей данные на сервер по команде `POST`. В файл свойств с расширением `.properties` вводятся следующие данные:

```
url=http://esa.un.org/unpd/wpp/unpp/p2k0data_script.asp
Panel=1
Variable=12;
Location=404
Varient=2
StartYear=1950
EndYear=2050
```

Программа удаляет элемент `url`, а все остальные элементы направляет методу `doPost()`. В методе `doPost()` сначала устанавливается соединение, затем вызывается метод `setDoOutput(true)` и открывается поток вывода. Затем перечисляются все ключи и значения. Для каждой пары "ключ–значение" по очереди передаются ключ, знак **=**, значение и разделительный знак **&**:

```
out.print(key);
out.print('=');
out.print(URLEncoder.encode(value, UTF-8"));
if (дополнительные пары "ключ-значение") out.print('&');
```

И наконец, в данной программе читается ответ с сервера. Следует, однако, иметь в виду, что если при выполнении сценария возникнет ошибка, то вызов метода `connection.getInputStream()` приведет к исключению типа `FileNotFoundException`. Тем не менее сервер продолжит передачу данных,

отправив HTML-страницу с сообщением об ошибке. Обычно это сообщение "Error 404-page not found", уведомляющее о том, что данная страница не найдена. Для перехвата этой страницы нужно привести объект типа URLConnection к типу HttpURLConnection и вызвать метод getErrorStream() следующим образом:

```
InputStream err = ((HttpURLConnection) connection).getErrorStream();
```

Любопытно, какие же данные объект типа URLConnection передает на сервер в дополнение к введенным данным? Объект типа URLConnection сначала посылает на сервер заголовок запроса. При отправке данных формы заголовок содержит следующие сведения:

```
Content-Type: application/x-www-form-urlencoded
```

Заголовок для команды POST должен также включать в себя сведения о длине содержимого, например:

```
Content-Length: 124
```

Конец заголовка обозначается пустой строкой. Затем следует порция передаваемых данных. Веб-сервер отделяет заголовок и направляет полученную порцию данных сценарию, выполняемому на стороне сервера. Следует также иметь в виду, что объект типа URLConnection буферизирует все данные, которые направляются в поток вывода, поскольку он должен сначала определить общую длину содержимого.

Прием, применяемый в данной программе, очень удобен для формирования запросов на получение данных с веб-сервера. Для этого следует проанализировать HTML-разметку веб-страницы, формирующей аналогичный запрос, чтобы выяснить требующиеся параметры и удалить из ответа все лишние HTML-дескрипторы и другие ненужные данные.

Листинг 3.7. Исходный код из файла post/PostTest.java

```
1  package post;
2
3  import java.io.*;
4  import java.net.*;
5  import java.nio.file.*;
6  import java.util.*;
7
8  /**
9   * В этой программе демонстрируется применение класса URLConnection
10  * для формирования запроса по команде POST
11  * @version 1.30 2012-06-04
12  * @author Cay Horstmann
13 */
14 public class PostTest
15 {
16   public static void main(String[] args) throws IOException
17   {
18     Properties props = new Properties();
19     try (InputStream in = Files.newInputStream(Paths.get(args[0])))
20     {
21       props.load(in);
22     }
23     String url = props.remove("url").toString();
24     String result = doPost(url, props);
25     System.out.println(result);
```

```
26   }
27
28   public static String doPost(String urlString,
29       Map<Object,   Object> nameValuePairs) throws IOException
30   {
31     URL url = new URL(urlString);
32     URLConnection connection = url.openConnection();
33     connection.setDoOutput(true);
34
35     try (PrintWriter out =
36          new PrintWriter(connection.getOutputStream()))
37     {
38       boolean first = true;
39       for (Map.Entry<Object, Object> pair : nameValuePairs.entrySet())
40       {
41         if (first) first = false;
42         else out.print('&');
43
44         String name = pair.getKey().toString();
45         String value = pair.getValue().toString();
46         out.print(name);
47         out.print('=');
48         out.print(URLEncoder.encode(value, "UTF-8"));
49       }
50     }
51
52     StringBuilder response = new StringBuilder();
53     try (Scanner in = new Scanner(connection.getInputStream()))
54     {
55       while (in.hasNextLine())
56       {
57         response.append(in.nextLine());
58         response.append("\n");
59       }
60     }
61     catch (IOException e)
62     {
63       if (!(connection instanceof HttpURLConnection)) throw e;
64       InputStream err = ((HttpURLConnection) connection).getErrorStream();
65       if (err == null) throw e;
66       Scanner in = new Scanner(err);
67       response.append(in.nextLine());
68       response.append("\n");
69     }
70
71     return response.toString();
72   }
73 }
```

java.net.HttpURLConnection 1.0

- `InputStream getErrorStream()`

  Возвращает поток для ввода сообщений веб-сервера об ошибках.

java.net.URLEncoder 1.0

- `static String encode(String s, String encoding)` **1.4**

  Возвращает символьную строку s в закодированном для URL виде. Кодирование производится по заданной схеме. При кодировании URL символы **A**–**Z**, **a**–**z**, **0**–**9**, **-**, **_**, **.** и **~** остаются без изменения. Пробелы передаются в виде знаков **+**, а все остальные символы — в виде **%UV**, где **UV** — шестнадцатеричное представление байта.

java.net.URLDecoder 1.2

- `static string decode(String s, String encoding)` **1.4**

  Возвращает в декодированном виде символьную строку s, закодированную для URL. Декодирование производится по заданной схеме.

Отправка электронной почты

Раньше для написания программы, отправляющей электронную почту, достаточно было установить сокетное соединение через порт **25**, который обычно используется для обмена данными по протоколу SMTP. А сам протокол SMTP (Simple Mail Transport Protocol — простой протокол для передачи почты) описывает формат сообщений электронной почты. Как только будет установлено соединение с сервером, нужно послать заголовок почтового сообщения (его нетрудно составить в формате SMTP), а затем и текст самого сообщения, выполнив следующие действия.

1. Откройте сокет на своем компьютере, подключенном к Интернету, как показано ниже.

```
Socket s = new Socket("mail.yourserver.com", 25); // номер порта 25
                                    // соответствует протоколу SMTP
PrintWriter out = new PrintWriter(s.getOutputStream());
```

2. Направьте в поток вывода следующие данные:

```
HELO компьютер отправителя
MAIL FROM: адрес отправителя
RCPT TO: адрес получателя
DATA
почтовое сообщение
(любое количество строк)
.
QUIT
```

В спецификации протокола SMTP (документ RFC 821) требуется, чтобы строки завершались комбинациями символов **/r** и **/n**. Первоначально SMTP-серверы исправно направляли электронную почту от любого адресата. Но когда навязчивые сообщения наводнили Интернет, большинство этих серверов были оснащены встроенными проверками, принимая запросы только по тем IP-адресам, которым они доверяют. Аутентификация обычно происходит через безопасные сокетные соединения.

Реализовать алгоритмы подобной аутентификации вручную — дело непростое. Поэтому в этом разделе будет показано, как пользоваться прикладным интерфейсом JavaMail API для отправки сообщений электронной почты из программы на Java. С этой целью загрузите данный прикладной интерфейс по адресу `www.oracle.com/technetwork/java/javamail` и разархивируйте его на жесткий диск своего компьютера.

Для того чтобы воспользоваться прикладным интерфейсом JavaMail API, необходимо установить некоторые свойства, зависящие от конкретного почтового сервера. В качестве примера ниже приведены свойства, устанавливаемые для почтового сервера GMail. Они считываются из файла свойств в рассматриваемом здесь примере программы, исходный код которой приведен в листинге 3.8.

```
mail.transport.protocol=smtps
mail.smtps.auth=true
mail.smtps.host=smtp.gmail.com
mail.smtps.user=cayhorstmann@gmail.com
```

Из соображений безопасности пароль не вводится в файл свойств, но предлагается для ввода вручную. После чтения из файла свойств сеанс почтовой связи устанавливается следующим образом:

```
Session mailSession = Session.getDefaultInstance(props);
```

Затем составляется почтовое сообщение с указанием требуемого отправителя, получателя, темы и текста самого сообщения, как показано ниже.

```
MimeMessage message = new MimeMessage(mailSession);
message.setFrom(new InternetAddress(from));
message.addRecipient(RecipientType.TO, new InternetAddress(to));
message.setSubject(subject);
message.setText(builder.toString());
```

Далее почтовое сообщение отправляется следующим образом:

```
Transport tr = mailSession.getTransport();
tr.connect(null, password);
tr.sendMessage(message, message.getAllRecipients());
tr.close();
```

Рассматриваемая здесь программа читает почтовое сообщение из текстового файла в приведенном ниже формате.

```
Отправитель
Получатель
Тема
Текст сообщения (любое количество строк)
```

Для запуска данной программы на выполнение введите приведенную ниже команду, где `mail.jar` — архивный JAR-файл, входящий в состав распространяемой версии прикладного интерфейса JavaMail API. (При указании пути к классу пользователи Windows должны ввести после параметра **`-classpath`** точку с запятой вместо двоеточия.)

```
java -classpath .:path/to/mail.jar path/to/message.txt
```

На момент написания данной книги почтовый сервер GMail не проверял достоверность получаемой информации, а следовательно, в почтовом сообщении можно было указать любого отправителя. (Это обстоятельство следует иметь в виду при получении от отправителя по адресу `president@whitehouse.gov` очередного приглашения на официальный прием, организуемый на лужайке перед Белым домом.)

СОВЕТ. Если вам не удастся выяснить причину, по которой соединение с почтовым сервером не действует, сделайте следующий вызов и проверьте почтовые сообщения:

```
mailSession.setDebug(true);
```

Кроме того, обратитесь за полезными советами на веб-страницу `JavaMail API FAQ` (Часто задаваемые вопросы по прикладному интерфейсу JavaMail API FAQ) по адресу `http://www.oracle.com/technetwork/java/javamail/faq-135477.html`.

Листинг 3.8. Исходный код из файла `mail/MailTest.java`

```
1  package mail;
2
3  import java.io.*;
4  import java.nio.charset.*;
5  import java.nio.file.*;
6  import java.util.*;
7  import javax.mail.*;
8  import javax.mail.internet.*;
9  import javax.mail.internet.MimeMessage.RecipientType;
10
11 /**
12  * В этой программе демонстрируется применение прикладного интерфейса
13  * JavaMail API для отправки сообщений по электронной почте
14  * @author Cay Horstmann
15  * @version 1.00 2012-06-04
16 */
17 public class MailTest
18 {
19   public static void main(String[] args)
20          throws MessagingException, IOException
21   {
22     Properties props = new Properties();
23     try (InputStream in =
24          Files.newInputStream(Paths.get("mail", "mail.properties")))
25     {
26       props.load(in);
27     }
28     List<String> lines =
29       Files.readAllLines(Paths.get(args[0]), Charset.forName("UTF-8"));
30
31     String from = lines.get(0);
32     String to = lines.get(1);
33     String subject = lines.get(2);
34
35     StringBuilder builder = new StringBuilder();
36     for (int i = 3; i < lines.size(); i++)
37     {
38       builder.append(lines.get(i));
39       builder.append("\n");
40     }
41
42     Console console = System.console();
43     String password = new String(console.readPassword("Password: "));
44
45     Session mailSession = Session.getDefaultInstance(props);
```

```
46     // mailSession.setDebug(true);
47     MimeMessage message = new MimeMessage(mailSession);
48     message.setFrom(new InternetAddress(from));
49     message.addRecipient(RecipientType.TO, new InternetAddress(to));
50     message.setSubject(subject);
51     message.setText(builder.toString());
52     Transport tr = mailSession.getTransport();
53     try
54     {
55       tr.connect(null, password);
56       tr.sendMessage(message, message.getAllRecipients());
57     }
58     finally
59     {
60       tr.close();
61     }
62   }
63 }
```

В этой главе было показано, каким образом на Java пишется исходный код программ для сетевых клиентов и серверов и как организуется сбор данных с веб-серверов. А в следующей главе речь пойдет о взаимодействии с базами данных. Из нее вы узнаете, как работать с реляционными базами данных в программах на Java, используя прикладной интерфейс JDBC API. В следующей главе дается также краткое введение в управление соединениями с базами данных, а также прикладной интерфейс JNDI API.