Содержание

ГЛАВА 4

Работа в сети

В этой главе...

- Подключение к серверу
- Реализация серверов
- Получение данных из Интернета
- HTTP-клиент
- 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.

4.1. Подключение к серверу

В последующих разделах сначала рассматривается подключение к серверу вручную с помощью утилиты `telnet`, а затем автоматическое подключение из программы на Java.

4.1.1. Применение утилиты `telnet`

Утилита `telnet` служит отличным инструментальным средством для отладки сетевых программ. Она должна запускаться из командной строки по команде **`telnet`**.

НА ЗАМЕТКУ! В Windows утилиту **telnet** необходимо активизировать. С этой целью 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. 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.nist.gov 13
```

На рис. 4.1 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) *
```

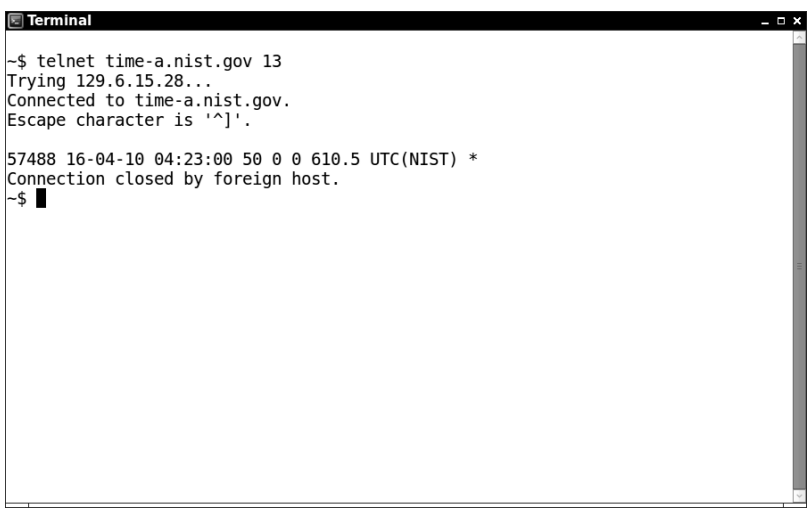

Рис. 4.1. Результат, получаемый из службы учета времени дня

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 США (National Institute of Standards and Technology). Его системное время синхронизировано с цезиевыми атомными часами. (Безусловно, 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. 4.2).

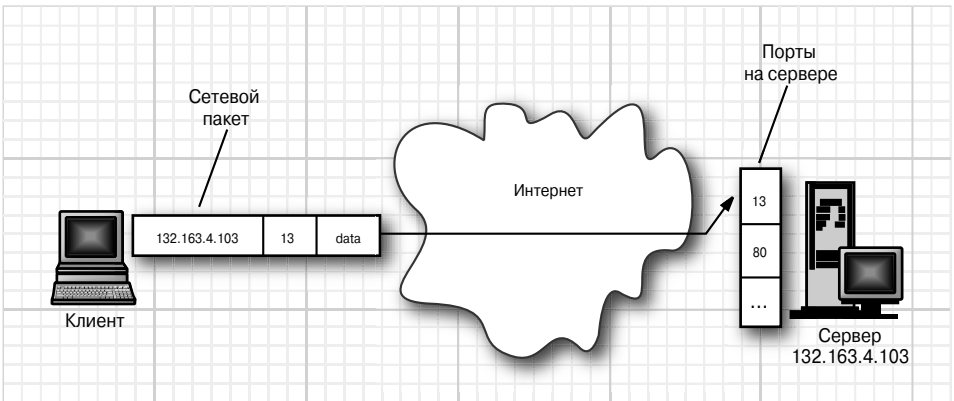

Рис. 4.2. Схема соединения клиента с сервером через конкретный порт

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.

Когда сеанс связи с сервером через порт `13` начинается по команде `telnet` с параметром `time-a.nist.gov`, сетевое программное обеспечение преобразует строку `"time-a.nist.gov"` в IP-адрес `129.6.15.28`. Затем оно посылает по этому адресу запрос на соединение с удаленным 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

   ```
   telnet horstmann.com 80
   ```

2. Затем аккуратно и точно введите следующие строки, *дважды* нажав клавишу <Enter> в конце:

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

На рис. 4.3 показана ответная реакция сервера в окне утилиты `telnet`. Она имеет уже знакомый вам 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. 4.3. 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.

4.1.2. Подключение к серверу из программы на Java

В первом примере сетевой программы, исходный код которой приведен в листинге 4.1, выполняются те же действия, что и при использовании утилиты `telnet`. Она устанавливает соединение с сервером через порт и выводит получаемые в ответ данные.

Листинг 4.1. Исходный код из файла **`socket/SocketTest.java`**

```
1  package socket;
2
3  import java.io.*;
4  import java.net.*;
5  import java.nio.charset.*;
6  import java.util.*;
7
8  /**
9   * В этой программе устанавливается сокетное соединение
10  * с атомными часами в г. Боулдере, шт. Колорадо и
11  * выводится время, передаваемое из сервера
12  * @version 1.22 2018-03-17
13  * @author Cay Horstmann
14  */
```

```
15 public class SocketTest
16 {
17   public static void main(String[] args)
18         throws IOException
19   {
20     try (var s = new Socket("time-a.nist.gov", 13);
21     var in = new Scanner(s.getInputStream(),
22                          StandardCharsets.UTF_8))
23     {
24       while (in.hasNextLine())
25       {
26         String line = in.nextLine();
27         System.out.println(line);
28       }
29     }
30   }
31 }
```

В данной программе наибольший интерес представляют следующие две строки кода:

```
Socket s = new Socket("time-a.nist.gov", 13);
Scanner in = new Scanner(s.getInputStream(), "UTF-8"))
```

В первой строке кода открывается сокет. *Сокет* — это абстрактное понятие, обозначающее возможность для программ устанавливать соединения для обмена данными по сети. Конструктору объекта 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. Пакет `java.net`, по существу, предоставляет тот же самый программный интерфейс, который используется для работы с файлами.

НА ЗАМЕТКУ! Здесь 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 поток вывода для записи данных в сокет.

4.1.3. 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 времени 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 кода показано, как это делается.

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

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 установить соединение с ним, задав время ожидания:

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

Если же пользователям требуется предоставить возможность прерывать соединение с сокетом в любой момент, то далее, в разделе 4.2.4 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 ожидания для чтения запросов в данном сокете. По истечении времени ожидания 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.

4.1.4. Межсетевые адреса

Как правило, нет особой нужды беспокоиться о межсетевых адресах в Интернете — числовых адресах хостов, состоящих из четырех байтов (или из шестнадцати байтов — по протоколу IPv6), как, например, `129.6.15.28`. Но 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. Например, в следующей строке кода возвращается объект типа `InetAddress`, инкапсулирующий последовательность из четырех байтов `129.6.15.28`:

```
InetAddress address = InetAddress.getByName("time-a.nist.gov");
```

Чтобы получить байты межсетевого адреса, достаточно вызвать метод `getAddress()` следующим образом:

```
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);
```

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 4.2 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 4.2. 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
7   * класса InetAddress. В качестве аргумента в командной
8   * строке следует указать имя хоста или запустить
9   * программу без аргументов, чтобы получить в ответ
10  * адрес локального хоста
11  * @version 1.02 2012-06-05
12  * @author Cay Horstmann
13  */
14 public class InetAddressTest
15 {
```

```
16   public static void main(String[] args)
17          throws IOException
18   {
19     if (args.length > 0)
20     {
21       String host = args[0];
22       InetAddress[] addresses =
23                     InetAddress.getAllByName(host);
24       for (InetAddress a : addresses)
25         System.out.println(a);
26     }
27     else
28     {
29       InetAddress localHostAddress =
30                   InetAddress.getLocalHost();
31       System.out.println(localHostAddress);
32     }
33   }
34 }
```

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.

4.2. Реализация серверов

В предыдущем разделе были рассмотрены особенности реализации элементарного сетевого клиента, способного получать данные из сети вообще и Интернета в частности. Теперь перейдем к обсуждению реализации простого сервера, способного посылать данные клиентам.

4.2.1. Сокеты сервера

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:

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

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, метод accept() 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 потоки сканирования (типа Scanner) и записи (типа Writer) следующим образом:

```
var in = new Scanner(inStream, "UTF-8");
var out = new PrintWriter(new OutputStreamWriter(
              outStream, "UTF-8"),
              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. 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 4.3 приведен весь исходный код описанного выше примера серверной программы.

Листинг 4.3. Исходный код из файла **`server/EchoServer.java`**

```
1  package server;
2
3  import java.io.*;
4  import java.net.*;
5  import java.nio.charset.*;
6  import java.util.*;
7
8  /**
9   * В этой программе реализуется простой сервер,
10  * прослушивающий порт 8189 и посылающий обратно
11  * клиенту все полученные от него данные
12  * client input.
13  * @version 1.22 2018-03-17
14  * @author Cay Horstmann
15  */
16 public class EchoServer
17 {
18   public static void main(String[] args)
19          throws IOException
20   {
21     // установить сокет на стороне сервера
22     try (var s = new ServerSocket(8189))
23     {
24       // ожидать подключения клиента
25       try (Socket incoming = s.accept())
26       {
27         InputStream inStream =
28                     incoming.getInputStream();
29         OutputStream outStream =
30                      incoming.getOutputStream();
31
32         try (var in = new Scanner(inStream,
33                       StandardCharsets.UTF_8))
34         {
35           var out = new PrintWriter(
36               new OutputStreamWriter(
37                   outStream, StandardCharsets.UTF_8),
38                   true /* автоматическая очистка */);
39
40           out.println("Hello! Enter BYE to exit.");
41
```

```
42            // передать обратно данные,
43            // полученные от клиента
44            var done = false;
45            while (!done && in.hasNextLine())
46            {
47              String line = in.nextLine();
48              out.println("Echo: " + line);
49              if (line.trim().equals("BYE")) done = true;
50            }
51          }
52        }
53      }
54    }
55 }
```

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 (рис. 4.4):

```
Hello! Enter BYE to exit.[1]
```

Рис. 4.4. 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.

---

[1] `Привет! Введите BYE (Пока), чтобы выйти из программы.`

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.

4.2.2. 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, один из клиентов 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();
  var r = new ThreadedEchoHandler(incoming);

  var 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
    }
```

```
    catch(IOException e)
    {
      обработать исключение
    }
  }
}
```

Когда новый поток исполнения запускается при каждом соединении, 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 4.4.
2. Откройте несколько окон утилиты `telnet` (рис. 4.5).
3. Переходя из одного окна в другое, введите команды. В итоге каждое отдельное окно утилиты `telnet` будет взаимодействовать с серверной программой независимо от других окон.
4. Чтобы разорвать соединение и закрыть окно утилиты `telnet`, нажмите комбинацию клавиш <Ctrl+C>.

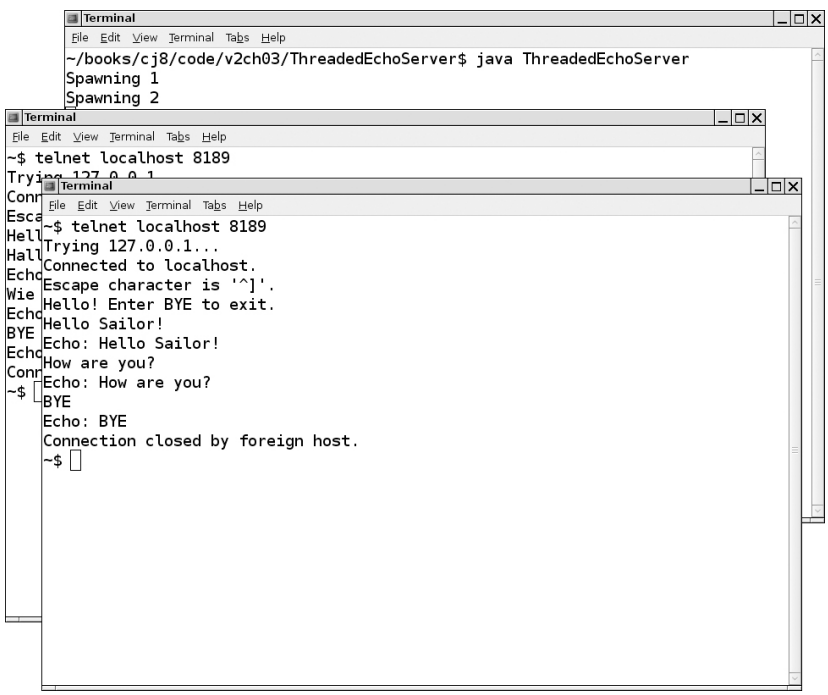

Рис. 4.5. Сеанс одновременной связи нескольких клиентов с сервером

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 4.4. Исходный код из файла **threaded/ThreadedEchoServer.java**

```
1  package threaded;
2
3  import java.io.*;
4  import java.net.*;
5  import java.nio.charset.*;
6  import java.util.*;
7
8  /**
9   * В этой программе реализуется многопоточный сервер,
10  * прослушивающий порт 8189 и передающий обратно все данные,
11  * полученные от всех клиентов
12  * @author Cay Horstmann
13  * @version 1.23 2018-03-17
14  */
15 public class ThreadedEchoServer
16 {
17   public static void main(String[] args )
18   {
19     try (var s = new ServerSocket(8189))
20     {
21       int i = 1;
22
23       while (true)
24       {
25         Socket incoming = s.accept();
26         System.out.println("Spawning " + i);
27         Runnable r = new ThreadedEchoHandler(incoming);
28         var t = new Thread(r);
29         t.start();
30         i++;
31       }
32     }
33     catch (IOException e)
34     {
35       e.printStackTrace();
36     }
37   }
38 }
39
40 /**
41  * Этот класс обрабатывает данные, получаемые сервером
42  * от клиента через одно сокетное соединение
43  */
44 class ThreadedEchoHandler implements Runnable
45 {
46   private Socket incoming;
47
48   /**
49      Конструирует обработчик
50      @param incomingSocket Входящий сокет
51   */
52   public ThreadedEchoHandler(Socket incomingSocket)
53   {
```

```
54      incoming = incomingSocket;
55    }
56
57    public void run()
58    {
59      try (InputStream inStream =
60           incoming.getInputStream();
61           OutputStream outStream =
62                       incoming.getOutputStream();
63           var in = new Scanner(inStream,
64                                StandardCharsets.UTF_8);
65           var out = new PrintWriter(
66                     new OutputStreamWriter(outStream,
67                         StandardCharsets.UTF_8),
68                         true /* autoFlush */))
69      {
70        out.println("Hello! Enter BYE to exit.");
71
72        // передать обратно данные, полученные от клиента
73        var done = false;
74        while (!done && in.hasNextLine())
75        {
76          String line = in.nextLine();
77          out.println("Echo: " + line);
78          if (line.trim().equals("BYE"))
79            done = true;
80        }
81      }
82      catch (IOException e)
83      {
84        e.printStackTrace();
85      }
86    }
87 }
```

4.2.3. 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.

Для преодоления подобного затруднения служит 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.

```
try (var socket = new Socket(host, port))
{
  var in = new Scanner(socket.getInputStream(), "UTF-8");
```

```
  var 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();
    . . .
  }
}
```

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 сетевым 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.

4.2.4. 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 принять данные через сокет, текущий поток приостановит свое исполнение до успешного завершения операции или до истечения времени ожидания. (Для передачи данных время ожидания не устанавливается.)

В прикладных программах, работающих в диалоговом режиме, пользователям желательно предоставить 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 из системы ввода-вывода NIO см. в главе 2.) 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`:

```
var in = new Scanner(channel, StandardCharsets.UTF_8);
```

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 генерированием исключения.

В примере программы, исходный код которой приведен в листинге 4.5, 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.

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. 4.6).

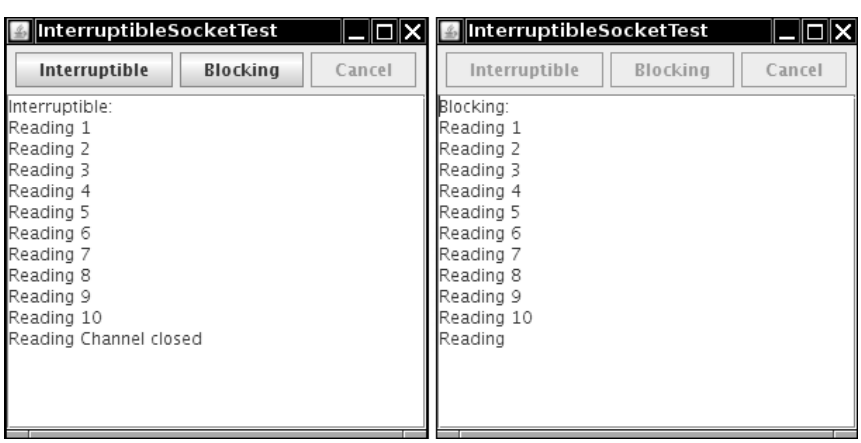

Рис. 4.6. Прерывание сокета

Листинг 4.5. 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.charset.*;
9   import java.nio.channels.*;
10  import javax.swing.*;
11
12  /**
13   * В этой программе демонстрируется прерывание
14   * сокета через канал
15   * @author Cay Horstmann
16   * @version 1.05 2018-03-17
17   */
18  public class InterruptibleSocketTest
19  {
20    public static void main(String[] args)
21    {
22      EventQueue.invokeLater(() ->
23       {
24         var frame = new InterruptibleSocketFrame();
25         frame.setTitle("InterruptibleSocketTest");
26         frame.setDefaultCloseOperation(
27               JFrame.EXIT_ON_CLOSE);
28         frame.setVisible(true);
29       });
30    }
31  }
32
33  class InterruptibleSocketFrame extends JFrame
34  {
35    private Scanner in;
36    private JButton interruptibleButton;
37    private JButton blockingButton;
38    private JButton cancelButton;
39    private JTextArea messages;
40    private TestServer server;
41    private Thread connectThread;
42
43    public InterruptibleSocketFrame()
44    {
45      var northPanel = new JPanel();
46      add(northPanel, BorderLayout.NORTH);
47
48      final int TEXT_ROWS = 20;
49      final int TEXT_COLUMNS = 60;
50      messages = new JTextArea(TEXT_ROWS, TEXT_COLUMNS);
51      add(new JScrollPane(messages));
52
53      interruptibleButton = new JButton("Interruptible");
54      blockingButton = new JButton("Blocking");
```

```
55
56      northPanel.add(interruptibleButton);
57      northPanel.add(blockingButton);
58
59      interruptibleButton.addActionListener(event ->
60        {
61          interruptibleButton.setEnabled(false);
62          blockingButton.setEnabled(false);
63          cancelButton.setEnabled(true);
64          connectThread = new Thread(() ->
65            {
66              try
67              {
68                connectInterruptibly();
69              }
70              catch (IOException e)
71              {
72                messages.append(
73                         "\nInterruptibleSocketTest."
74                       + "connectInterruptibly: " + e);
75              }
76            });
77          connectThread.start();
78        });
79
80      blockingButton.addActionListener(event ->
81       {
82         interruptibleButton.setEnabled(false);
83         blockingButton.setEnabled(false);
84         cancelButton.setEnabled(true);
85         connectThread = new Thread(() ->
86           {
87             try
88             {
89               connectBlocking();
90             }
91             catch (IOException e)
92             {
93               messages.append(
94                        "\nInterruptibleSocketTest."
95                      + "connectBlocking: " + e);
96             }
97           });
98         connectThread.start();
99       });
100
101     cancelButton = new JButton("Cancel");
102     cancelButton.setEnabled(false);
103     northPanel.add(cancelButton);
104     cancelButton.addActionListener(event ->
105       {
106         connectThread.interrupt();
107         cancelButton.setEnabled(false);
108       });
109     server = new TestServer();
110     new Thread(server).start();
```

```
111     pack();
112   }
113
114   /**
115    * Соединяет с проверяемым сервером,
116    * используя прерываемый ввод-вывод
117    */
118   public void connectInterruptibly() throws IOException
119   {
120     messages.append("Interruptible:\n");
121     try (SocketChannel channel = SocketChannel
122        .open(new InetSocketAddress("localhost", 8189)))
123     {
124       in = new Scanner(channel, StandardCharsets.UTF_8);
125       while (!Thread.currentThread().isInterrupted())
126       {
127         messages.append("Reading ");
128         if (in.hasNextLine())
129         {
130           String line = in.nextLine();
131           messages.append(line);
132           messages.append("\n");
133         }
134       }
135     }
136     finally
137     {
138       EventQueue.invokeLater(() ->
139         {
140           messages.append("Channel closed\n");
141           interruptibleButton.setEnabled(true);
142           blockingButton.setEnabled(true);
143         });
144     }
145   }
146
147   /**
148    * Соединяет с проверяемым сервером,
149    * используя блокирующий ввод-вывод
150    */
151   public void connectBlocking() throws IOException
152   {
153     messages.append("Blocking:\n");
154     try (var sock = new Socket("localhost", 8189))
155     {
156       in = new Scanner(sock.getInputStream(),
157                        StandardCharsets.UTF_8);
158       while (!Thread.currentThread().isInterrupted())
159       {
160         messages.append("Reading ");
161         if (in.hasNextLine())
162         {
163           String line = in.nextLine();
164           messages.append(line);
165           messages.append("\n");
166         }
```

```
167       }
168     }
169     finally
170     {
171       EventQueue.invokeLater(() ->
172        {
173          messages.append("Socket closed\n");
174          interruptibleButton.setEnabled(true);
175          blockingButton.setEnabled(true);
176        });
177     }
178   }
179
180   /**
181    * Многопоточный сервер, прослушивающий порт 8189 и
182    * посылающий клиентам числа, имитируя зависание
183    * после передачи 10 чисел
184    *
185    */
186   class TestServer implements Runnable
187   {
188     public void run()
189     {
190       try (var s = new ServerSocket(8189))
191       {
192         while (true)
193         {
194           Socket incoming = s.accept();
195           Runnable r = new TestServerHandler(incoming);
196           new Thread(r).start();
197         }
198       }
199       catch (IOException e)
200       {
201         messages.append("\nTestServer.run: " + e);
202       }
203     }
204   }
205
206   /**
207    * Этот класс обрабатывает данные, получаемые
208    * сервером от клиента через одно сокетное соединение
209    */
210   class TestServerHandler implements Runnable
211   {
212     private Socket incoming;
213     private int counter;
214
215     /**
216      * Конструирует обработчик
217      * @param i Входящий сокет
218      */
219     public TestServerHandler(Socket i)
220     {
221       incoming = i;
222     }
```

```
223
224     public void run()
225     {
226       try
227       {
228         try
229         {
230           OutputStream outStream =
231                 incoming.getOutputStream();
232           var out = new PrintWriter(
233           new OutputStreamWriter(outStream,
234                     StandardCharsets.UTF_8),
235           true /* автоматическая очистка */);
236           while (counter < 100)
237           {
238             counter++;
239             if (counter <= 10) out.println(counter);
240             Thread.sleep(100);
241           }
242         }
243         finally
244         {
245           incoming.close();
246           messages.append("Closing server\n");
247         }
248       }
249       catch (Exception e)
250       {
251         messages.append("\nTestServerHandler.run: " + e);
252       }
253     }
254   }
255 }
```

java.net.InetSocketAddress 1.4

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

java.nio.channels.SocketChannel 1.4

- **static SocketChannel open(SocketAddress *address*)**
  Открывает канал для сокета и связывает его с удаленным хостом 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.

4.3. Получение данных из Интернета

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.

4.3.1. URL и URI

Классы `URL` и `URLConnection` инкапсулируют большую 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();
var in = new Scanner(inStream, StandardCharsets.UTF_8);
```

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. 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 архивным JAR-файлам.

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

```
http:/google.com?q=Beach+Chalet
ftp://username:password@ftp.yourserver.com/pub/file.txt
```

В обозначении идентификаторов URI задаются правила их построения. 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, непрозрачные URI и все относительные URL имеют *иерархическую структуру*. Например:

```
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. Составляющая *полномочия* в URI серверов 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 адресов* 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);
```

4.3.2. 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. Наконец, для доступа к данным указанного ресурса следует вызвать метод `getInputStream()`, предоставляющий поток ввода для чтения данных. (Это тот же 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 приема данных с веб-сервера, но не поток вывода для передачи данных. Чтобы получить поток вывода (например, с целью разместить данные на веб-сервере), необходимо 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.

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

1. Составить символьную строку из имени пользователя, двоеточия и пароля:

```
String input = username + ":" + password;
```

2. Перекодировать полученную в итоге символьную строку по алгоритму кодирования Base64, как показано ниже. (Этот алгоритм преобразует последовательность байтов в последовательность символов в коде ASCII.)

```
Base64.Encoder encoder = Base64.getEncoder();
String encoding = encoder.encodeToString(
                  input.getBytes(StandardCharsets.UTF_8));
```

3. Вызвать 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. 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, в результате вызова 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 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:

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

В качестве примера ниже приведен ряд полей заголовка из типичного ответа на запрос по сетевому 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
```

НА ЗАМЕТКУ! Получить в ответ строку состояния (например, **"HTTP/1.1 200 OK"**) можно, сделав вызов **connection.getHeaderField(0)** или **headerFields.get(null)**.

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 удобные методы перечислены в табл. 4.1. В методах, возвращающих значения типа `long`, отсчет количества возвращаемых секунд начинается с полуночи 1 января 1970 г.

Таблица 4.1. Удобные 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 из листинга 4.6 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.ваш_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 (см. табл. 4.1).
- Первые `10` символьных строк из запрашиваемого ресурса.

Листинг 4.6. Исходный код из файла **urlConnection/URLConnectionTest.java**

```
1  package urlConnection;
2
3  import java.io.*;
4  import java.net.*;
5  import java.nio.charset.*;
6  import java.util.*;
7
8  /**
9   * В этой программе устанавливается соединение по
```

```
10  * заданному URL и отображаются данные заголовка из
11  * получаемого ответа, а также первые 10 строк
12  * запрашиваемых данных. Для этого в командной строке
13  * следует указать конкретный URL, дополнительно имя
14  * пользователя и пароль (для элементарной аутентификации
15  * по сетевому протоколу HTTP)
16  * @version 1.12 2018-03-17
17  * @author Cay Horstmann
18  */
19 public class URLConnectionTest
20 {
21   public static void main(String[] args)
22   {
23     try
24     {
25       String urlName;
26       if (args.length > 0) urlName = args[0];
27       else urlName = "http://horstmann.com";
28
29       var url = new URL(urlName);
30       URLConnection connection = url.openConnection();
31
32       // установить имя пользователя и пароль, если они
33       // указаны в командной строке
34
35       if (args.length > 2)
36       {
37         String username = args[1];
38         String password = args[2];
39         String input = username + ":" + password;
40         Base64.Encoder encoder = Base64.getEncoder();
41         String encoding = encoder.encodeToString(
42                input.getBytes(StandardCharsets.UTF_8));
43         connection.setRequestProperty("Authorization",
44                    "Basic " + encoding);
45       }
46
47       connection.connect();
48
49       // вывести поля заголовка
50
51       Map<String, List<String>> headers =
52                   connection.getHeaderFields();
53       for (Map.Entry<String, List<String>> entry :
54            headers.entrySet())
55       {
56         String key = entry.getKey();
57         for (String value : entry.getValue())
58           System.out.println(key + ": " + value);
59       }
60
61       // вывести значения полей заголовка,
62       // используя удобные методы
63
64       System.out.println("----------");
65       System.out.println("getContentType: "
66                          + connection.getContentType());
```

```
67        System.out.println("getContentLength: "
68                           + connection.getContentLength());
69        System.out.println("getContentEncoding: "
70                         + connection.getContentEncoding());
71        System.out.println("getDate: "
72                           + connection.getDate());
73        System.out.println("getExpiration: "
74                           + connection.getExpiration());
75        System.out.println("getLastModifed: "
76                           + connection.getLastModified());
77        System.out.println("----------");
78
79        String encoding = connection.getContentEncoding();
80        if (encoding == null) encoding = "UTF-8";
81        try (var in = new Scanner(
82             connection.getInputStream(), encoding))
83        {
84          // вывести первые десять строк
85          // запрашиваемого содержимого
86
87          for (int n = 1; in.hasNextLine() && n <= 10; n++)
88            System.out.println(in.nextLine());
89          if (in.hasNextLine()) System.out.println(". . .");
90        }
91      }
92      catch (IOException e)
93      {
94        e.printStackTrace();
95      }
96    }
97 }
```

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 задано логическое значение **true** параметра ***doInput***, пользователь может принимать вводимые данные из текущего объекта типа **URLConnection**.
- **void setDoOutput(boolean *doOutput*)**
- **boolean getDoOutput()**
  Если задано логическое значение **true** параметра ***doOutput***, пользователь может передавать выводимые данные в текущий объект типа **URLConnection**.

java.net.URLConnection 1.0 *(продолжение)*

- **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 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 по одному и тому же ключу 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 истекает срок действия ресурса. Время указывается в секундах, начиная с 1 января 1970 г. по Гринвичу.

java.net.URLConnection 1.0 *(окончание)*

- **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.

4.3.3. Отправка данных формы

В предыдущем разделе описывался способ приема данных с веб-сервера, 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. 4.7.

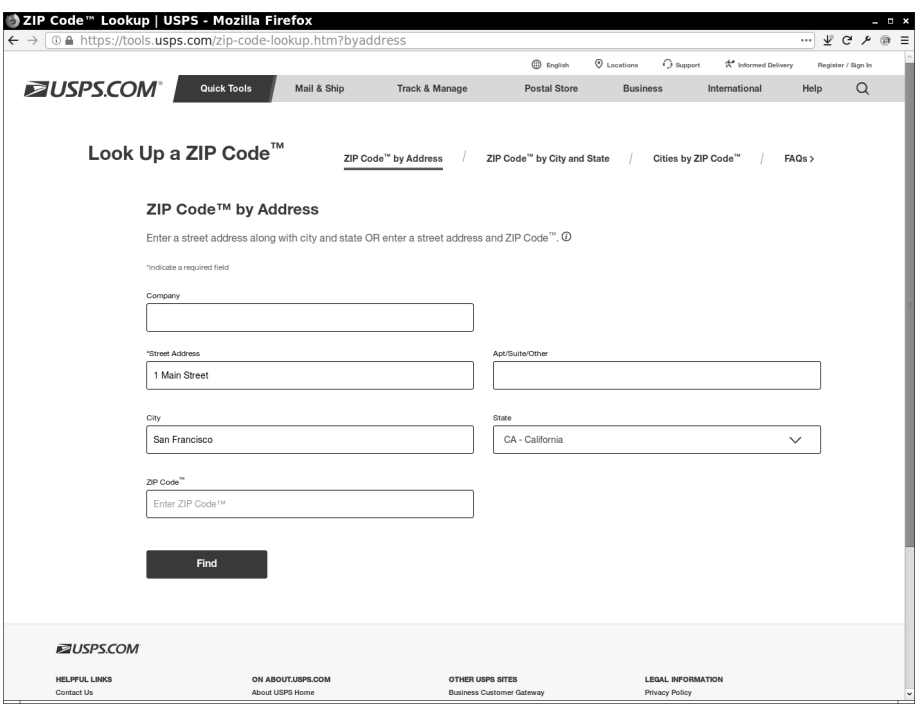

Рис. 4.7. HTML-форма

Когда пользователь щелкает на кнопке Submit (Отправить), 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, платформы JavaServer Faces и Microsoft ASP (Active Server Pages — активные серверные страницы), а также сценарии CGI (Common Gateway Interface — общий шлюзовой интерфейс).

Программа, выполняющаяся 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 схематически показана на рис. 4.8. Ответная страница, сформированная сервером, может содержать новые данные (например, результаты поиска) или только подтверждение о получении введенных данных. Здесь и далее не рассматриваются вопросы реализации серверных программ, а основное внимание уделяется написанию клиентских программ, предназначенных для взаимодействия с готовыми сценариями.

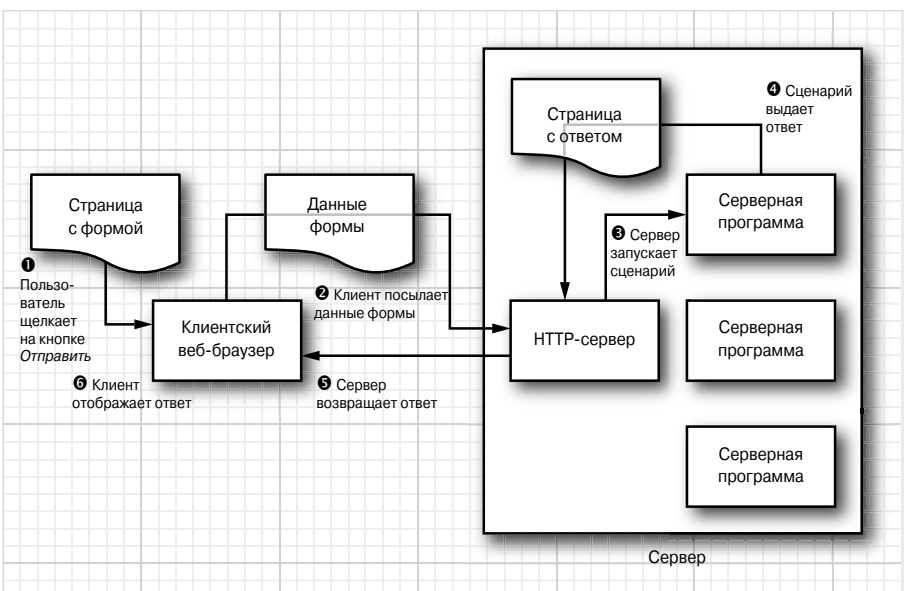

Рис. 4.8. Порядок обработки данных в серверной программе

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 запроса указываются в конце URL в следующем формате:

```
http://хост/путь?запрос
```

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 в вид %UV, где UV — двухзначное шестнадцатеричное число.

Например, название города и штата *San Francisco, CA* передается в закодированном виде как San+Francisco%2c+CA. Здесь шестнадцатеричное число 2c (или десятичное 44) обозначает запятую в кодировке UTF-8. Благодаря такому способу кодирования промежуточные программы не будут путаться в пробелах и смогут правильно интерпретировать другие символы.

На момент написания данной книги веб-сайт Google Maps (www.google.com/maps) принимал параметры запроса с именами q и hl, значения которых определяют местоположение и естественный язык в ответе. Чтобы получить карту местности по адресу Маркет-стрит, 1, г. Сан-Франциско на немецком языке, необходимо указать следующий URL:

```
http://www.google.com/maps?q=1+Market+Street+San+Francisco&hl=de
```

Очень длинные строки запроса могут выглядеть непривлекательно в большинстве браузеров, а в старых браузерах и промежуточных серверах накладывается ограничение на количество символов, включаемых в запрос по команде GET. Именно поэтому запрос по команде POST чаще всего употребляется для форм, содержащих немало данных. Параметры запроса по команде POST не следует включать в состав URL. Вместо этого следует 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 серверной программе сначала создается объект типа URLConnection:

```
var url = new URL("http://хост/путь");
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:

```
var out = new PrintWriter(connection.getOutputStream(),
                          StandardCharsets.UTF_8);
```

Теперь можно передать данные на сервер, как показано ниже.

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

После передачи данных поток вывода закрывается следующим образом:

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

Наконец, вызывается метод getInputStream(), чтобы прочитать ответ с сервера.

Рассмотрим конкретный практический пример. Веб-сайт, доступный по адресу `https://tools.usps.com/zip-code-lookup.htm?byaddress`, содержит страницу с формой для поиска почтового индекса по введенному адресу улицы (см. рис. 4.7). Чтобы воспользоваться этой формой в программе на Java, следует знать URL и параметры запроса по команде `POST`.

Эту информацию можно было бы получить, просмотрев код HTML-разметки формы, но ее, как правило, проще "выудить" из запроса с помощью сетевого монитора, входящего в набор инструментальных средств веб-разработки, предоставляемый в большинстве браузеров. В качестве примера на рис. 4.9 приведен моментальный снимок, сделанный сетевым монитором браузера Firefox при передаче на обработку данных выбранному для данного примера веб-сайту. На этом моментальном снимке можно выявить URL, параметры и значения, указанные при передаче данных на обработку.

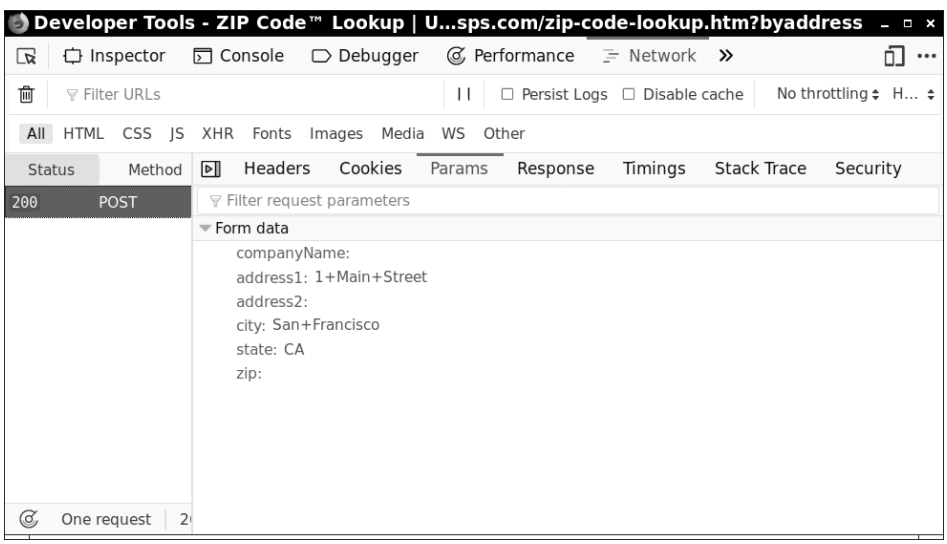

Рис. 4.9. Текущий контроль передачи HTML-формы на обработку

При передаче данных формы на обработку в HTTP-заголовок включается тип содержимого и его длина:

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

Данные можно передать и в других форматах. Так, если данные посылаются в формате JSON (JavaScript Object Notation — представление объектов JavaScript), в HTTP-заголовке запроса по команде `POST` должен быть указан тип содержимого `application/json`, а также его длина, как показано в следующем примере:

```
Content-Length: 124
```

В листинге 4.7 приведен исходный код примера программы, посылающей данные на сервер по команде `POST`. В файл свойств с расширением `.properties` вводятся следующие данные:

```
url=https://tools.usps.com/tools/app/ziplookup/zipByAddress
User-Agent=HTTPie/0.9.2
```

```
address1=1 Market Street
address2=
city=San Francisco
state=CA
companyName=
. . .
```

Программа удаляет элементы `url` и `User-Agent`, а все остальные элементы направляет методу `doPost()`. В методе `doPost()` сначала устанавливается соединение, а затем пользовательский посредник. (Служба определения почтовых индексов не обрабатывает устанавливаемый по умолчанию параметр запроса `User-Agent`, содержащий строку `"Java"`, вероятно, потому, что почтовая служба не намерена обслуживать программные запросы.)

Далее вызывается метод `setDoOutput(true)`, открывается поток вывода и 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, StandardCharsets.UTF_8));
if (дополнительные пары "ключ-значение") out.print('&');
```

Взаимодействие с сервером фактически происходит при переходе от записи к чтению любой части ответа. В заголовке `Content-Length` задается длина выводимых данных, а в заголовке `Content-Type` — тип `application/x-www-form-urlencoded`, если только не был указан другой тип содержимого. Заголовки и данные запроса посылаются серверу. Затем читаются заголовки и данные ответа сервера, которые могут быть запрошены. В данном примере программы такой переход от записи к чтению происходит при вызове метода `connection.getContentEncoding()`.

Следует, однако, иметь в виду, что если при выполнении серверной программы 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 фиксации страницы с этим сообщением об ошибке следует вызвать метод `getErrorStream()`:

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

НА ЗАМЕТКУ! Метод **`getErrorStream()`**, а также ряд других методов, применяемых в рассматриваемом здесь примере программы, относятся к классу **`HttpURLConnection`**, производному от класса **`URLConnection`**. Если сделать запрос по URL, начинающемуся с префикса **`http://`** или **`https://`**, то полученный в итоге объект соединения можно привести к типу **`HttpURLConnection`**.

При передаче данных по команде `POST` на сервер серверная программа может *переадресовать* его по другому URL для получения искомой информации. Сервер может сделать это потому, что искомая информация находится в каком-нибудь другом месте. С другой стороны, он может предоставить отмеченный закладкой URL. Как правило, класс `HttpURLConnection` может осуществить переадресацию.

НА ЗАМЕТКУ! Если при переадресации требуется переслать cookie-файлы из одного сайта на другой, с этой целью можно настроить глобальный обработчик cookie-файлов следующим образом:

```
CookieHandler.setDefault(new CookieManager(null,
CookiePolicy.ACCEPT_ALL));
```

В этом случае cookie-файлы будут надлежащим образом включены в переадресацию.

Несмотря на то что переадресация, как правило, осуществляется автоматически, иногда это приходится делать вручную. Автоматическая переадресация между сетевыми протоколами HTTP и HTTPS не поддерживается из соображений безопасности. Она может не состояться и по менее ясным причинам. Например, в прежней версии службы определения почтовых индексов обычно применялась переадресация. Напомним, что параметр запроса `User-Agent` был установлен ранее таким образом, чтобы почтовая служба не посчитала, что запрос был сделан через прикладной интерфейс Java API. И хотя в первоначальном запросе можно задать другую строку для пользовательского посредника, такая настройка не используется при автоматической переадресации, при которой всегда посылается типичная строка пользовательского посредника, содержащая строку `"Java"`.

В подобных случаях переадресацию можно осуществить вручную. Прежде чем подключиться к серверу, необходимо выключить режим автоматической переадресации следующим образом:

```
connection.setInstanceFollowRedirects(false);
```

Сделав запрос, следует получить код ответа:

```
int responseCode = connection.getResponseCode();
```

и проверить, относится ли он к одному из перечисленных ниже кодов.

```
HttpURLConnection.HTTP_MOVED_PERM
HttpURLConnection.HTTP_MOVED_TEMP
HttpURLConnection.HTTP_SEE_OTHER
```

В таком случае следует сначала получить заголовок ответа `Location`, а затем URL для переадресации. Далее необходимо разорвать текущее соединение и установить другое соединение по новому URL, как показано ниже.

```
String location = connection.getHeaderField("Location");
if (location != null)
{
  URL base = connection.getURL();
  connection.disconnect();
  connection = (HttpURLConnection)
               new URL(base, location).openConnection();
  . . .
}
```

Приемы, демонстрируемые в рассматриваемой здесь программе, могут оказаться полезными всякий раз, когда требуется запросить информацию из существующего веб-сайта. Для этого достаточно выяснить сначала параметры, которые требуется послать в запросе, а затем удалить дескрипторы HTML-разметки и прочую ненужную информацию из полученного ответа.

Листинг 4.7. Исходный код из файла **post/PostTest.java**

```
1   package post;
2
3   import java.io.*;
4   import java.net.*;
5   import java.nio.charset.*;
6   import java.nio.file.*;
7   import java.util.*;
8
9   /**
10   * В этой программе демонстрируется применение
11   * класса URLConnection для формирования запроса
12   * по команде POST
13   * @version 1.42 2018-03-17
14   * @author Cay Horstmann
15   */
16  public class PostTest
17  {
18    public static void main(String[] args)
19          throws IOException
20    {
21      String propsFilename = args.length > 0 ? args[0]
22                             : "post/post.properties";
23      var props = new Properties();
24      try (InputStream in = Files.newInputStream(
25                            Paths.get(propsFilename)))
26      {
27        props.load(in);
28      }
29      String urlString = props.remove("url").toString();
30      Object userAgent = props.remove("User-Agent");
31      Object redirects = props.remove("redirects");
32      CookieHandler.setDefault(new CookieManager(null,
33                               CookiePolicy.ACCEPT_ALL));
34      String result = doPost(new URL(urlString), props,
35      userAgent == null ? null : userAgent.toString(),
36      redirects == null ? -1 : Integer.parseInt(
37                               redirects.toString()));
38      System.out.println(result);
39    }
40
41    /**
42     * Сделать HTTP-запрос по команде POST
43     * @param url Конкретный URL для отправки запроса
44     * @param nameValuePairs Параметры запроса
45     * @param userAgent Пользовательский посредник или
46     *                  пустое значение null, если это
47     *                  посредник по умолчанию
48     * @param redirects Количество последующих
49     *                  переадресаций вручную или
50     *                  значение -1, если переадресация
51     *                  производится автоматически
52     * @return Данные, возвращаемые из сервера
53     */
54    public static String doPost(URL url,
```

```
55                  Map<Object, Object> nameValuePairs,
56                  String userAgent, int redirects)
57           throws IOException
58    {
59      var connection = (HttpURLConnection)
60                       url.openConnection();
61      if (userAgent != null)
62        connection.setRequestProperty(
63                  "User-Agent", userAgent);
64      if (redirects >= 0)
65        connection.setInstanceFollowRedirects(false);
66
67      connection.setDoOutput(true);
68
69      try (var out = new PrintWriter(
70                     connection.getOutputStream()))
71      {
72        var first = true;
73        for (Map.Entry<Object, Object> pair :
74             nameValuePairs.entrySet())
75        {
76          if (first) first = false;
77          else out.print('&');
78          String name = pair.getKey().toString();
79          String value = pair.getValue().toString();
80          out.print(name);
81          out.print('=');
82          out.print(URLEncoder.encode(value,
83                    StandardCharsets.UTF_8));
84        }
85      }
86      String encoding = connection.getContentEncoding();
87      if (encoding == null) encoding = "UTF-8";
88
89      if (redirects > 0)
90      {
91        int responseCode = connection.getResponseCode();
92        if (responseCode == HttpURLConnection
93                            .HTTP_MOVED_PERM
94            || responseCode == HttpURLConnection
95                               .HTTP_MOVED_TEMP
96            || responseCode == HttpURLConnection
97                               .HTTP_SEE_OTHER)
98        {
99          String location = connection
100                           .getHeaderField("Location");
101         if (location != null)
102         {
103           URL base = connection.getURL();
104           connection.disconnect();
105           return doPost(new URL(base, location),
106                         nameValuePairs, userAgent,
107                         redirects - 1);
108         }
109       }
110     }
111     else if (redirects == 0)
```

```
112     {
113       throw new IOException("Too many redirects");
114     }
115
116     var response = new StringBuilder();
117     try (var in = new Scanner(
118          connection.getInputStream(), encoding))
119     {
120       while (in.hasNextLine())
121       {
122         response.append(in.nextLine());
123         response.append("\n");
124       }
125     }
126     catch (IOException e)
127     {
128       InputStream err = connection.getErrorStream();
129       if (err == null) throw e;
130       try (var in = new Scanner(err))
131       {
132         response.append(in.nextLine());
133         response.append("\n");
134       }
135     }
136
137     return response.toString();
138   }
139 }
```

java.net.HttpURLConnection 1.0

- **InputStream getErrorStream()**
  Возвращает поток ввода, из которого читаются сообщения сервера об ошибках.

java.net.URLEncoder 1.0

- **static String encode(String *s*, String *encoding*) 1.4**
  Возвращает строку ***s***, закодированную в формате URL с помощью заданной кодировки символов. (Рекомендуется указывать кодировку **"UTF-8"**.) При кодировании в формате URL символы **'A'-'Z'**, **'a'-'z'**, **'0'-'9'**, **'-'**, **'_'**, **'.'** и **'~'** оставляются без изменения. Пробелы заменяются знаками **'+'**, а все остальные символы — последовательностями кодированных байтов в форме **"%*XY*"**, где **0x*XY*** — шестнадцатеричное значение байта.

java.net.URLDecoder 1.2

- **static string decode(String *s*, String *encoding*) 1.4**
  Возвращает форму строки ***s***, закодированной в формате URL и декодированной с помощью заданной кодировки символов.

4.4. HTTP-клиент

Класс URLConnection был разработан еще до того, как сетевой протокол HTTP стал универсальным для Интернета. В этом классе поддерживается целый ряд сетевых протоколов, хотя поддержка протокола HTTP в нем реализована не очень удобно. Когда было принято решение о поддержке сетевого протокола HTTP/2, то стало ясно, что лучше предоставить современный клиентский интерфейс вместо того, чтобы переделывать уже существующий прикладной интерфейс API. Так, в классе HttpClient предоставляется более удобный прикладной интерфейс API для поддержки сетевого протокола HTTP/2. Начиная с версии Java 11 класс HttpClient входит в состав пакета java.net.http.

НА ЗАМЕТКУ! В версиях Java 9 и 10 прикладные программы следует запускать на выполнение из командной строки со следующим параметром:

```
--add-modules jdk.incubator.httpclient
```

В прикладном интерфейсе API для HTTP-клиента предоставляется более простой механизм подключения к веб-серверу, чем в классе URLConnection, где этот процесс дотошно выполняется в течение целого ряда стадий. HTTP-клиент, реализуемый средствами класса HttpClient, может выдавать запросы и получать ответы от веб-сервера. Чтобы получить такой клиент, достаточно сделать следующий вызов:

```
HttpClient client = HttpClient.newHttpClient()
```

Если требуется сконфигурировать клиент, то можно воспользоваться прикладным интерфейсом API его построителя, как показано ниже.

```
HttpClient client = HttpClient.newBuilder()
          .followRedirects(HttpClient.Redirect.ALWAYS)
          .build();
```

Подобным образом получается построитель, вызываются методы для специальной настройки создаваемого клиента, а затем вызывается метод build() с целью завершить весь процесс построения. Это типичный шаблон для построения неизменяемых объектов.

По такому же шаблону построения составляются запросы. В качестве примера ниже демонстрируется составление HTTP-запроса по команде GET.

```
HttpRequest request = HttpRequest.newBuilder()
            .uri(new URI("http://horstmann.com"))
            .GET()
            .build();
```

Универсальный идентификатор ресурса (URI) в сетевом протоколе HTTP равнозначен URL. Но в Java предоставляется класс URL с методами, фактически устанавливающими соединение с веб-ресурсом по заданному URL, тогда как класс URI обеспечивает лишь необходимый синтаксис (схему, хост, порт, путь, запрос, фрагмент и т.д.).

Для составления HTTP-запроса по команде POST требуется "издатель тела запроса", где запрашиваемые данные преобразуются в пересылаемые данные. Имеются издатели тела запроса для символьных строк, массивов байтов и файлов.

Так, если запрос составляется в формате JSON, издателю его тела достаточно предоставить символьную строку в формате JSON, как показано ниже.

```
HttpRequest request = HttpRequest.newBuilder()
           .uri(new URI(url))
           .header("Content-Type", "application/json")
           .POST(HttpRequest.BodyPublishers
           .ofString(jsonString))
           .build();
```

К сожалению, в рассматриваемом здесь прикладном интерфейсе API не поддерживается требующееся форматирование общеупотребительных типов содержимого запросов. В приведенном далее примере программы из листинга 4.8 демонстрируется применение издателей тела запроса для обработки данных формы и выгрузки файлов.

Отправляя запрос на веб-сервер, приходится указывать клиенту порядок обработки получаемого ответа. Если же требуется отравить лишь тело запроса в виде символьной строки, это можно сделать с помощью метода `HttpResponse.BodyHandlers.ofString()` следующим образом:

```
HttpResponse<String> response = client.send(request,
                     HttpResponse.BodyHandlers.ofString());
```

Класс `HttpResponse` является обобщенным, а параметр его типа обозначает тип тела запроса. Получить тело запроса в виде символьной строки можно следующим образом:

```
String bodyString = response.body();
```

Имеются и другие обработчики тела ответа, получающие ответ в виде массива байтов или потока ввода. В частности, метод `BodyHandlers.ofFile(filePath)` возвращает обработчик, сохраняющий ответ в заданном файле, а метод `BodyHandlers.ofFileDownload(directoryPath)` сохраняет ответ в заданном каталоге, используя имя файла из заголовка `Content-Disposition`. Наконец, обработчик, возвращаемый из метода `BodyHandlers.discarding()`, просто отвергает полученный ответ.

Обработка содержимого ответа не обеспечивается как составная часть рассматриваемого здесь прикладного интерфейса API. Так, если ответ получается в формате JSON, для синтаксического анализа его содержимого потребуется отдельная библиотека, поддерживающая обработку данных формата JSON.

В объекте типа `HttpResponse` предоставляется также код состояния и заголовки ответов:

```
int status = response.statusCode();
HttpHeaders responseHeaders = response.headers();
```

Объекты типа `HttpHeaders` можно преобразовать в отображение, как демонстрируется в приведенной ниже строке кода. В качестве значений в таком отображении служат списки, поскольку в сетевом протоколе HTTP для каждого ключа допускается несколько значений.

```
Map<String, List<String>> headerMap = responseHeaders.map();
```

Если же требуется значение для конкретного ключа и заранее известно, что у него не может быть несколько значений, следует вызвать метод `firstValue()`,

как показано ниже. В ответ получается конкретное значение, а если оно не предоставлено — пустое необязательное значение.

```
Optional<String> lastModified =
                 headerMap.firstValue("Last-Modified");
```

Ответы можно обрабатывать и асинхронно. Для этого при построении клиента предоставляется исполнитель:

```
ExecutorService executor = Executors.newCachedThreadPool();
HttpClient client = HttpClient.newBuilder()
                   .executor(executor).build();
```

Сначала составляется запрос, а затем для клиента вызывается метод `sendAsync()`, как показано ниже. В итоге получается завершаемое будущее действие типа `CompletableFuture<HttpResponse<T>>`, где `T` — тип обработчика тела ответа. О том, как применять прикладной интерфейс API для завершаемых будущих действий, см. в главе 12 первого тома настоящего издания.

```
HttpRequest request = HttpRequest.newBuilder().uri(uri)
                                  .GET().build();
client.sendAsync(request,
                 HttpResponse.BodyHandlers.ofString())
    thenAccept(response -> . . .);
```

СОВЕТ. Чтобы активизировать режим протоколирования для HTTP-клиента типа **`HttpClient`**, достаточно ввести следующую строку кода в файл **`net.properties`** свойств комплекта JDK:

`jdk.httpclient.HttpClient.log=all`

Вместо параметра **`all`** можно указать разделяемый запятыми список параметров **`headers`**, **`requests`**, **`content`**, **`errors`**, **`ssl`**, **`trace`** и **`frames`**, а после них дополнительно — параметры **`:control`**, **`:data`**, **`:window`** или **`:all`**, но без пробелов.

После этого можно установить уровень протоколирования **`INFO`** для регистратора **`jdk.httpclient.HttpClient`**, введя, например, следующую строку кода в файл **`logging.properties`** свойств комплекта JDK:

`jdk.httpclient.HttpClient.level=INFO`

Листинг 4.8. Исходный код из файла **`client/HttpClientTest.java`**

```
1   package client;
2
3   import java.io.*;
4   import java.math.*;
5   import java.net.*;
6   import java.nio.charset.*;
7   import java.nio.file.*;
8   import java.util.*;
9
10  import java.net.http.*;
11   import java.net.http.HttpRequest.*;
12
13  class MoreBodyPublishers
14  {
15    public static BodyPublisher ofFormData(
```

```
16                  Map<Object, Object> data)
17    {
18      var first = true;
19      var builder = new StringBuilder();
20      for (Map.Entry<Object, Object> entry :
21           data.entrySet())
22      {
23        if (first) first = false;
24        else builder.append("&");
25        builder.append(URLEncoder.encode(
26                       entry.getKey().toString(),
27                       StandardCharsets.UTF_8));
28        builder.append("=");
29        builder.append(URLEncoder.encode(
30                       entry.getValue().toString(),
31                       StandardCharsets.UTF_8));
32      }
33      return BodyPublishers.ofString(builder.toString());
34    }
35
36    private static byte[] bytes(String s)
37      { return s.getBytes(StandardCharsets.UTF_8); }
38
39    public static BodyPublisher ofMimeMultipartData(
40           Map<Object, Object> data, String boundary)
41           throws IOException
42    {
43      var byteArrays = new ArrayList<byte[]>();
44      byte[] separator = bytes("--" + boundary
45            + "\nContent-Disposition: form-data; name=");
46      for (Map.Entry<Object, Object> entry :
47           data.entrySet())
48      {
49        byteArrays.add(separator);
50
51        if (entry.getValue() instanceof Path)
52        {
53          var path = (Path) entry.getValue();
54          String mimeType = Files.probeContentType(path);
55          byteArrays.add(bytes("\"" + entry.getKey()
56                 + "\"; filename=\"" + path.getFileName()
57                 + "\"\nContent-Type: "
58                 + mimeType + "\n\n"));
59          byteArrays.add(Files.readAllBytes(path));
60        }
61        else
62          byteArrays.add(bytes("\"" + entry.getKey()
63              + "\"\n\n" + entry.getValue() + "\n"));
64      }
65      byteArrays.add(bytes("--" + boundary + "--"));
66      return BodyPublishers.ofByteArrays(byteArrays);
67    }
68
69    public static BodyPublisher ofSimpleJSON(
70                 Map<Object, Object> data)
71    {
72      var builder = new StringBuilder();
```

```
73      builder.append("{");
74      var first = true;
75      for (Map.Entry<Object, Object> entry :
76           data.entrySet())
77      {
78        if (first) first = false;
79        else
80          builder.append(",");
81        builder.append(jsonEscape(entry.getKey()
82                        .toString()))
83               .append(": ")
83               .append(jsonEscape(entry.getValue()
84                        .toString()));
85      }
86      builder.append("}");
87      return BodyPublishers.ofString(builder.toString());
88    }
89
90    private static Map<Character, String> replacements =
91            Map.of('\b', "\\b", '\f', "\\f", \n', '"\\n",
92                   '\r', "\\r", '\t', "\\t", '"', "\\\"",
93                   '\\', "\\\\");
94
95    private static StringBuilder jsonEscape(String str)
96    {
97      var result = new StringBuilder("\"");
98      for (int i = 0; i < str.length(); i++)
99      {
100       char ch = str.charAt(i);
101       String replacement = replacements.get(ch);
102       if (replacement == null) result.append(ch);
103       else result.append(replacement);
104     }
105     result.append("\"");
106     return result;
107   }
108 }
109
110 public class HttpClientTest
111 {
112   public static void main(String[] args)
113         throws IOException, URISyntaxException,
114                InterruptedException
115   {
116     System.setProperty("jdk.httpclient.HttpClient.log",
117                        "headers,errors");
118     String propsFilename = args.length > 0 ? args[0] :
119                            "client/post.properties";
120     Path propsPath = Paths.get(propsFilename);
121     var props = new Properties();
122     try (InputStream in =
123          Files.newInputStream(propsPath))
124     {
126       props.load(in);
127     }
128     String urlString = "" + props.remove("url");
129     String contentType = ""
```

```
130          + props.remove("Content-Type");
131     if (contentType.equals("multipart/form-data"))
132     {
133       var generator = new Random();
134       String boundary = new BigInteger(256, generator)
135                             .toString();
136       contentType += ";boundary=" + boundary;
137       props.replaceAll((k, v) ->
138             v.toString().startsWith("file://")
139                ? propsPath.getParent()
140                  .resolve(Paths.get(v.toString()
141                  .substring(7)))
142                : v);
143     }
144     String result = doPost(urlString,
145                            contentType, props);
146      System.out.println(result);
147   }
148
149   public static String doPost(String url,
150          String contentType, Map<Object, Object> data)
151          throws IOException, URISyntaxException,
152                 InterruptedException
153   {
154     HttpClient client = HttpClient.newBuilder()
155        .followRedirects(HttpClient.Redirect.ALWAYS)
156        .build();
157
158     BodyPublisher publisher = null;
159     if (contentType.startsWith("multipart/form-data"))
160     {
161       String boundary = contentType.substring(
162              contentType.lastIndexOf("=") + 1);
163       publisher = MoreBodyPublishers
164                   .ofMimeMultipartData(data, boundary);
165     }
166     else if (contentType.equals(
167              "application/x-www-form-urlencoded"))
168     publisher = MoreBodyPublishers.ofFormData(data);
169     else
170     {
171       contentType = "application/json";
172       publisher = MoreBodyPublishers.ofSimpleJSON(data);
173     }
174
175     HttpRequest request = HttpRequest.newBuilder()
176                   .uri(new URI(url))
177                   .header("Content-Type", contentType)
178                   .POST(publisher)
179                   .build();
180     HttpResponse<String> response = client.send(
181         request, HttpResponse.BodyHandlers.ofString());
182     return response.body();
183   }
184 }
```

java.net.http.HttpClient 11

- **static HttpClient newHttpClient()**
  Возвращает объект типа **HttpClient** с конфигурацией HTTP-клиента по умолчанию.
- **static HttpClient.Builder newBuilder()**
  Возвращает построитель HTTP-клиентов, представленных объектами типа **HttpClient**.
- **<T> HttpResponse<T> send(HttpRequest *request*, HttpResponse.BodyHandler<T> *responseBodyHandler*)**
- **<T> CompletableFuture<HttpResponse<T>> sendAsync(HttpRequest *request*, HttpResponse .BodyHandler<T> *responseBodyHandler*)**
  Составляют синхронный и асинхронный запрос и обрабатывают тело получаемого ответа с помощью заданного обработчика.

java.net.http.HttpClient.Builder 11

- **HttpClient build()**
  Возвращает объект типа **HttpClient** со свойствами, сконфигурированными данным построителем HTTP-клиентов.
- **HttpClient.Builder followRedirects(HttpClient.Redirect *policy*)**
  Устанавливает правило переадресации, определяемое одной из следующих констант из перечисления **HttpClient.Redirect**: **ALWAYS**, **NEVER** или **NORMAL** (отклонять переадресацию только из сетевого протокола HTTPS в протокол HTTP).
- **HttpClient.Builder executor(Executor *executor*)**
  Устанавливает исполнитель асинхронных запросов.

java.net.http.HttpRequest 11

- **HttpRequest.Builder newBuilder()**
  Возвращает построитель HTTP-запросов, представленных объектами типа **HttpRequest**.

java.net.http.HttpRequest.Builder 11

- **HttpRequest build()**
  Возвращает объект типа **HttpRequest** со свойствами, сконфигурированными данным построителем HTTP-запросов.
- **HttpRequest.Builder uri(URI *uri*)**
  Устанавливает URI для данного запроса.
- **HttpRequest.Builder header(String *name*, String *value*)**
  Устанавливает заголовок для данного запроса.

java.net.http.HttpRequest.Builder 11 *(окончание)*

- **HttpRequest.Builder GET()**
- **HttpRequest.Builder DELETE()**
- **HttpRequest.Builder POST(HttpRequest.BodyPublisher *bodyPublisher*)**
- **HttpRequest.Builder PUT(HttpRequest.BodyPublisher *bodyPublisher*)**

  Устанавливают метод доступа и тело для данного запроса.

java.net.http.HttpResponse<T> 11

- **T body()**

  Возвращает тело данного ответа.
- **int statusCode()**

  Возвращает код состояния для данного ответа.
- **HttpHeaders headers()**

  Возвращает заголовки ответа.

java.net.http.HttpHeaders 11

- **Map<String,List<String>> map()**

  Возвращает отображение типа **Map** данных заголовков.
- **Optional<String> firstValue(String *name*)**

  Возвращает первое значение по имени, указанному в данных заголовках, если таковое имеется.

4.5. Отправка электронной почты

В прошлом для отправки электронной почты достаточно было написать программу, установливавшую соединение с сетевым сокетом через порт 25, который обычно используется для работы сетевого протокола SMTP (Simple Mail Transport Protocol — простой протокол передачи почты), описывающего формат электронных сообщений. После подключения к серверу в данной программе нужно было послать заголовок сообщения, который достаточно просто было создать в формате SMTP, а затем и текст сообщения, выполнив перечисленные ниже действия.

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

```
var s = new Socket("mail.yourserver.com", 25);
    // номер порта 25 соответствует протоколу SMTP
var out = new PrintWriter(s.getOutputStream(),"UTF-8");
```

2. Направить в поток вывода следующие данные:

```
HELO хост отправителя
MAIL FROM: адрес отправителя
```

```
RCPT TO: адрес получателя
DATA
Subject: тема
(пустая строка)
почтовое сообщение
(любое количество строк)

QUIT
```

В спецификации сетевого протокола SMTP (документ RFC 821) требуется, чтобы строки завершались последовательностями символов /r и /n. Первоначально SMTP-серверы исправно направляли электронную почту от любого адресата. Но когда навязчивые сообщения наводнили Интернет, большинство этих серверов было оснащено встроенными проверками и принимали 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 https://javaee.github.io/javamail/ и разархивируйте его на жесткий диск своего компьютера.

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 из листинга 4.9.

```
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 и 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:

```
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)
```

Кроме упомянутого выше прикладного интерфейса JavaMail API, для выполнения данной программы потребуется архивный JAR-файл каркаса JavaBeans Activation Framework, который можно загрузить по адресу `https://www.oracle.com/technetwork/java/javase/downloads/index-135046.html#download` или из центрального хранилища Maven Central по адресу `https://mvnrepository.com/artifact/javax.activation/activation`. Затем выполните следующую команду:

```
java -classpath .:javax.mail.jar:activation-1.1.1.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 программному интерфейсу JavaMail API FAQ), доступную по адресу `https://javaee.github.io/javamail/FAQ`.

Листинг 4.9. 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
13  * прикладного интерфейса JavaMail API для отправки
14  * сообщений по электронной почте
15  * @author Cay Horstmann
16  * @version 1.01 2018-03-17
17  */
18 public class MailTest
19 {
```

```
20   public static void main(String[] args)
21         throws MessagingException, IOException
22   {
23     var props = new Properties();
24     try (InputStream in = Files.newInputStream(
25          Paths.get("mail", "mail.properties")))
26     {
27        props.load(in);
28     }
29     List<String> lines = Files.readAllLines(
30          Paths.get(args[0]), StandardCharsets.UTF_8);
31
32     String from = lines.get(0);
33     String to = lines.get(1);
34     String subject = lines.get(2);
35
36     var builder = new StringBuilder();
37     for (int i = 3; i < lines.size(); i++)
38     {
39       builder.append(lines.get(i));
40       builder.append("\n");
41     }
42
43     Console console = System.console();
44     var password =
45         new String(console.readPassword("Password: "));
46
47     Session mailSession =
48             Session.getDefaultInstance(props);
49     // mailSession.setDebug(true);
50     var message = new MimeMessage(mailSession);
51     message.setFrom(new InternetAddress(from));
52     message.addRecipient(RecipientType.TO,
53                          new InternetAddress(to));
54     message.setSubject(subject);
55     message.setText(builder.toString());
56     Transport tr = mailSession.getTransport();
57     try
58     {
59       tr.connect(null, password);
60       tr.sendMessage(message,
61                      message.getAllRecipients());
62     }
63     finally
64     {
65       tr.close();
66     }
67   }
68 }
```

В этой главе было показано, как на Java пишется исходный код программ для сетевых клиентов и серверов и как организуется сбор данных с веб-серверов. 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.