
Глава

3

РАБОТА В СЕТИ

В ЭТОЙ ГЛАВЕ...

- ▶ Подключение к серверу
- ▶ Реализация серверов
- ▶ Прерываемые сокеты
- ▶ Отправка электронной почты
- ▶ Создание соединений с URL

Эта глава начинается с описания основ организации сетевых соединений, а затем продолжается рассмотрением Java-программ, позволяющих устанавливать соединения с серверами. Вы узнаете, как производится реализация сетевых клиентов и серверов. Завершается глава рассмотрением вопроса передачи почтовых сообщений из программы на Java и сбора информации с Web-сервера.

Подключение к серверу

Перед созданием первой сетевой программы рекомендуется познакомиться с прекрасным инструментом отладки сетевых программ, а именно – с утилитой `telnet`. Учтите, что она не является обязательным компонентом операционной системы, поэтому проверьте, установлена ли она на вашем компьютере. Если утилиту `telnet` невозможно запустить из командной строки, еще раз запустите программу инсталляции и выберите ее в списке предлагаемых для установки компонентов.



Вместе с ОС Windows Vista устанавливается и утилита `telnet`, однако по умолчанию она не активизирована. Чтобы активизировать ее, откройте панель управления, перейдите в раздел Программы, щелкните на ссылке Добавление или удаление компонентов Windows и отметьте флажок Клиент Telnet. К тому же, брандмауэр Windows блокирует некоторые сетевые порты, которыми мы будем пользоваться в этой главе; чтобы разблокировать порты, вы должны обладать полномочиями администратора.

Утилита `telnet` может использоваться не только для соединения с удаленным компьютером. Посредством ее вы также можете взаимодействовать с различными сетевыми службами. Ниже приводится один из примеров необычного использования этой утилиты. Введите в командной строке команду

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

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

```
53221 04-08-04 02:19:40 50 0 0 513.0 UTC (NIST) *
```

Что же произошло в этом случае? Утилита `telnet` подключилась к серверу службы времени, который работает на большинстве компьютеров под управлением операционной системы UNIX. Указанный в этом примере сервер находится в Национальном институте стандартов и технологий (National Institute of Standards) в Боулдере, штат Колорадо, США. Его системное время синхронизировано с цезиевыми атомными часами. (Конечно, полученное значение текущего времени будет не совсем точным из-за задержек, связанных с передачей данных по сети.) По договоренности сервер службы времени всегда связан с портом 13.



В терминологии, используемой для описания сетевых технологий, порт — это не какое-то физическое устройство, а абстрактное понятие, которое используется для описания организации соединения между сервером и клиентом (рис. 3.2).

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

```

Terminal
File Edit View Terminal Go Help
~$ telnet time-A.timefreq.bldrdoc.gov 13
Trying 132.163.4.101...
Connected to time-A.timefreq.bldrdoc.gov.
Escape character is '^['.

53221 04-08-04 02:19:40 50 0 0 513.0 UTC(NIST) *
Connection closed by foreign host.
~$

```

Рис. 3.1. Результат работы службы времени

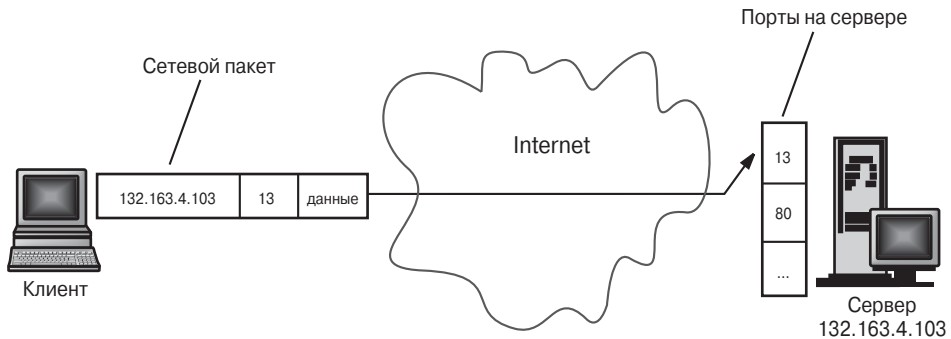


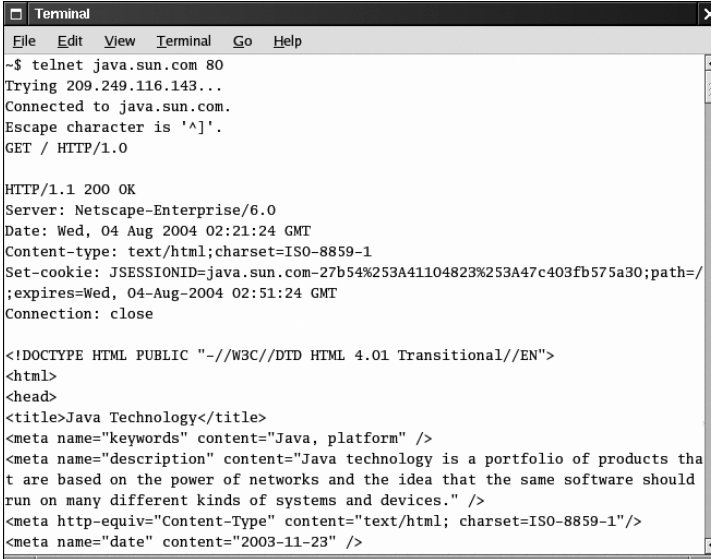
Рис. 3.2. Схема взаимодействия клиента с сервером по определенному порту

После запуска утилиты `telnet` с параметром `time-A.timefreq.bldrdoc.gov` и портом 13 независимое сетевое программное обеспечение преобразует строку `time-A.timefreq.bldrdoc.gov` в IP-адрес `132.163.4.104`. Затем посылается запрос на соединение с заданным компьютером по порту 13. После установления соединения программа на удаленном компьютере передает обратно строку с данными, а затем закрывает соединение. В общем случае клиенты и серверы до закрытия соединения могут участвовать в более сложных диалогах.

Вот еще один более интересный эксперимент. Выполните перечисленные ниже действия.

1. Подключитесь к серверу `java.sun.com` по порту 80.
2. Введите строку `GET / HTTP/1.0` точно в показанном здесь виде без использования клавиши `<Backspace>`.
3. Теперь два раза нажмите клавишу `<Enter>`.

На рис. 3.3 показана ответная реакция сервера в окне утилиты `telnet`. Она имеет уже знакомый нам вид страницы текста в формате HTML, а именно – исходной Web-страницы компании Sun, посвященной технологиям Java.



```

Terminal
File Edit View Terminal Go Help
~$ telnet java.sun.com 80
Trying 209.249.116.143...
Connected to java.sun.com.
Escape character is '^]'.
GET / HTTP/1.0

HTTP/1.1 200 OK
Server: Netscape-Enterprise/6.0
Date: Wed, 04 Aug 2004 02:21:24 GMT
Content-type: text/html;charset=ISO-8859-1
Set-cookie: JSESSIONID=java.sun.com-27b54%253A41104823%253A47c403fb575a30;path=/;expires=Wed, 04-Aug-2004 02:51:24 GMT
Connection: close

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
<head>
<title>Java Technology</title>
<meta name="keywords" content="Java, platform" />
<meta name="description" content="Java technology is a portfolio of products that are based on the power of networks and the idea that the same software should run on many different kinds of systems and devices." />
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1"/>
<meta name="date" content="2003-11-23" />

```

Рис. 3.3. Использование утилиты `telnet` для доступа к HTTP-порту

Именно так обычный Web-браузер получает все Web-страницы. Для запроса Web-страниц на сервере он применяет HTTP. Естественно, браузер отображает информацию в виде, гораздо более удобном для чтения.



Если вы попытаетесь проделать это с Web-сервером, на котором размещено множество доменов с одинаковым IP-адресом, вы не получите требуемой страницы. (Так бывает с небольшими Web-сайтами, находящимися на одном сервере, таком как `horstmann.com`.) При подключении к такому серверу нужно определить требуемое имя хоста, как показано ниже:

```
GET / HTTP/1.1
```

```
Host: horstmann.com
```

Затем нажмите клавишу `<Enter>` два раза. (Обратите внимание, что HTTP имеет версию 1.1.)

Программа, приведенная в листинге 3.1, выполняет точно такие же действия, т.е. подключается к порту и выводит полученные данные.

Листинг 3.1. Содержимое файла `SocketTest.java`

```

1. import java.io.*;
2. import java.net.*;
3. import java.util.*;
4.
5. /**
6.  * Данная программа создает сокет для соединения
7.  * с атомными часами в Боулдере, штат Колорадо.
8.  * Она выводит на экран текущее время, полученное с сервера.

```

```
9.  * @version 1.20 2004-08-03
10. * @author Cay Horstmann
11. */
12. public class SocketTest
13. {
14.     public static void main(String[] args)
15.     {
16.         try
17.         {
18.             Socket s = new Socket("time-A.timefreq.bldrdoc.gov", 13);
19.             try
20.             {
21.                 InputStream inStream = s.getInputStream();
22.                 Scanner in = new Scanner(inStream);
23.                 while (in.hasNextLine())
24.                 {
25.                     String line = in.nextLine();
26.                     System.out.println(line);
27.                 }
28.             }
29.             finally
30.             {
31.                 s.close();
32.             }
33.         }
34.         catch (IOException e)
35.         {
36.             e.printStackTrace();
37.         }
38.     }
39. }
```

В данной программе нас больше всего интересуют следующие две строки:

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

Первая строка позволяет открыть *сокет*. Сокет — это абстрактное понятие, которое поддерживается программным обеспечением, позволяющим организовать операции обмена данными по сети. Конструктору сокета передается адрес удаленного сервера и номер порта. При неудачном соединении генерируется исключение `UnknownHostException`, а при наличии каких-то других проблем — исключение `IOException`. Класс `UnknownHostException` является подклассом `IOException`, поэтому в этом простом примере обрабатывается только исключение суперкласса.

После открытия сокета метод `getInputStream()` класса `java.net.Socket` возвращает объект потока `InputStream`, который можно использовать как любой другой поток. После получения потока программа приступает к выводу каждой строки в стандартный выходной поток. Данный процесс продолжается до тех пор, пока поток не будет закрыт или пока не будет разорвано соединение с сервером.

Эта программа может работать только с очень простыми серверами, например со службой времени. В более сложных случаях клиент посылает запрос на получение данных серверу, а сервер может в течение некоторого времени поддерживать это соединение. Примеры подобного поведения представлены далее в главе.

Класс `Socket` очень удобен в работе, поскольку скрывает все сложные подробности установления сетевого соединения и передачи данных по сети. А пакет `java.net` предоставляет тот же программный интерфейс, который используется для работы с файлами.



В этой книге мы рассмотрим лишь протокол TCP (Transmission Control Protocol — протокол управления передачей). Платформа Java поддерживает также протокол UDP (User Datagram Protocol — протокол дейтаграмм пользователя), который может использоваться для отправки пакетов (они называются также *дейтаграммами* (datagram)) с гораздо меньшими затратами, чем в случае протокола TCP. Недостатком является то, что пакеты не нужно доставлять принимающему приложению в последовательном порядке, и они вообще могут быть потеряны. Получатель сам должен позаботиться о том, чтобы пакеты были организованы в определенном порядке, и сам должен повторно запрашивать передачу отсутствующих пакетов. Протокол UDP хорошо подходит для тех приложений, которые могут обходиться без отсутствующих пакетов — например, в аудио- и видеопотоках, или для продолжительных измерений.

Тайм-ауты сокетов

Чтение из блоков сокетов доступно до тех пор, пока данные являются доступными. Если хост является недостижимым, ваше приложение будет ожидать в течение длительного периода времени, и все будет зависеть от того, когда операционная система, под управлением которой работает компьютер, определит момент завершения ожидания.

Вы можете решить, какое значение тайм-аута будет разумным для вашего определенного приложения. Затем вызовите метод `setSoTimeout`, чтобы задать значение тайм-аута (в миллисекундах).

```
Socket s = new Socket(. . .);
s.setSoTimeout(10000); // тайм-аут наступит через 10 секунд
```

Если значение тайм-аута было задано для сокета, то все последующие операции чтения и записи будут генерировать исключение `SocketTimeoutException` в момент наступления тайм-аута, прежде чем операция завершит свою работу. Вы можете перехватить исключение и отреагировать на тайм-аут.

```
try
{
    InputStream in = s.getInputStream(); // чтение из in
    . . .
}
catch (InterruptedException exception)
{
    реакция на тайм-аут
}
```

Вам придется решить еще одну дополнительную проблему, связанную с тайм-аутами. Конструктор

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

может удерживать блокировку в течение неопределенного периода времени до тех пор, пока не будет установлено исходное соединение с хостом.

Эту проблему можно преодолеть, если сначала создать несоединенный сокет, а затем соединиться с ним во время тайм-аута:

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

Если вы хотите позволить пользователям прерывать соединение с сокетом в любой момент времени, прочтите раздел “Прерываемые сокеты” далее в этой главе.

API `java.net.Socket` 1.0

- `Socket()` 1.1
Создает сокет, который на данный момент времени не имеет соединения.
- `void connect(SocketAddress address)` 1.4
Производит подключение сокета к данному адресу.
- `void connect(SocketAddress address, int timeoutInMilliseconds)` 1.4
Производит подключение сокета к данному адресу или возвращает значение, если временной интервал истек.
- `void setSoTimeout(int timeoutInMilliseconds)` 1.1
Задаёт время блокирования чтения запросов на данном сокете. Если наступит тайм-аут, возникнет исключение `InterruptedIOException`.
- `boolean isConnected()` 1.4
Возвращает значение `true`, если существует соединение с сокетом.
- `boolean isClosed()` 1.4
Возвращает значение `true`, если соединение с сокетом закрыто.

Internet-адреса

Как правило, вам не нужно излишне беспокоиться об Internet-адресах — числовых адресах хостов, состоящих из четырех байт (или из шестнадцати — в версии протокола IPv6), таких как 132.163.4.102. Тем не менее, вы можете использовать класс `InetAddress`, если необходимо выполнить преобразование между именами хостов и Internet-адресами. Пакет `java.net` поддерживает Internet-адреса в формате протокола IPv6, при условии, что и операционная система хоста тоже их поддерживает.

Статический метод `getByName` возвращает объект `InetAddress` хоста. Например,

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

возвращает объект `InetAddress`, инкапсулирующий последовательность из четырех байт 132.163.4.104. Вы можете получить доступ к этим байтам с помощью метода `getAddress`:

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

Имена некоторых хостов, обслуживающих большой объем трафика, соответствуют нескольким Internet-адресам, что объясняется попыткой сбалансировать нагрузку. Например, на момент написания этой книги имя хоста `java.sun.com` соответствовало трем различным Internet-адресам. Один из них выбирается случайным образом во время доступа к хосту. Получить все хосты можно посредством метода `getAllByName`:

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

И, наконец, в некоторых случаях вам будет необходим адрес локального хоста. Если вы запросите адрес `localhost`, то неизменно получите адрес 127.0.0.1, который другие не могут использовать для подключения к вашему компьютеру. Вместо этого нужно применить метод `getLocalHost`, чтобы получить адрес вашего локального хоста:

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

В листинге 3.2 приведен пример простой программы, выводящей на экран Internet-адрес вашего локального хоста, если не будут определены какие-либо параметры командной строки, или все Internet-адреса другого хоста, если вы определите имя хоста в командной строке, например:

```
java InetAddressTest java.sun.com
```

Листинг 3.2. Содержимое файла InetAddressTest.java

```
1. import java.net.*;
2.
3. /**
4.  * Данная программа демонстрирует пример использования класса InetAddress.
5.  * Вам необходимо задать имя хоста в качестве аргумента командной строки
6.  * или запустить программу без аргументов командной строки, чтобы
7.  * посмотреть адрес локального хоста.
8.  * @version 1.01 2001-06-26
9.  * @author Cay Horstmann
10. */
11. public class InetAddressTest
12. {
13.     public static void main(String[] args)
14.     {
15.         try
16.         {
17.             if (args.length > 0)
18.             {
19.                 String host = args[0];
20.                 InetAddress[] addresses = InetAddress.getAllByName(host);
21.                 for (InetAddress a : addresses)
22.                     System.out.println(a);
23.             }
24.             else
25.             {
26.                 InetAddress localhostAddress = InetAddress.getLocalHost();
27.                 System.out.println(localhostAddress);
28.             }
29.         }
30.         catch (Exception e)
31.         {
32.             e.printStackTrace();
33.         }
34.     }
35. }
```

API java.net.InetAddress 1.0

- static InetAddress getByName(String host)
- static InetAddress[] getAllByName(String host)
Создает InetAddress, или массив всех Internet-адресов, для данного имени хоста.
- static InetAddress getLocalHost()
Создает InetAddress для локального хоста.
- byte[] getAddress()
Возвращает массив байтов, содержащий числовой адрес.
- String getAddress()
Возвращает строку с десятичными числами, например, "132.163.4.102".
- String getHostName()
Возвращает имя хоста.

Реализация серверов

Теперь после создания клиента, который может получать информацию из сети, попробуем создать простой сервер, который может посылать данные. После запуска программа-сервер переходит в режим ожидания обращения клиентов по своему порту. Выберите номер порта 8189, который не используется никакими стандартными устройствами. Приведенная ниже команда позволяет создать сервер с портом 8189:

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

Следующая команда сообщает программе, что она должна ожидать обращения клиентов по заданному порту:

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

Сразу после обращения клиента (посредством переданного по сети запроса) этот метод возвращает объект `Socket`, который представляет установленное соединение. Данный объект можно использовать для чтения входных и записи выходных данных так, как показано ниже.

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

Все данные выходного потока сервера становятся данными входного потока клиента и наоборот: все данные выходного потока клиента поступают во входной поток сервера. Во всех примерах этой главы текстовые данные в потоках будут передаваться через сокет. Потоки, связанные с сокетом, будут преобразовываться в объекты `Scanner` и `Writer`.

```
Scanner in = new Scanner(inStream);
PrintWriter out = new PrintWriter(outStream, true
    /* автоматическая передача оставшихся данных */);
```

Предположим, что программа-клиент отправила приветствие:

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

При использовании утилиты `telnet` для подключения к программе-серверу через порт 8189 данное приветствие будет отображено в окне терминала.

В этой простой программе-сервере построчно считываются входные данные, отправленные программой-клиентом, и отображаются на экране в режиме “эхо”. Подобным образом пример демонстрирует получение входных данных от программы-клиента. Программа-сервер должна обработать полученные данные и дать адекватный ответ.

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

В конце работы следует закрыть используемый сокет:

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

Вот и все. Любая программа-сервер, например, Web-сервер, выполняет такой же цикл основных действий.

1. Получение команды от программы-клиента (“дайте мне нужную информацию”) из входного потока.
2. Расшифровка команды клиента.
3. Сбор информации, запрошенной клиентом.
4. Передача клиенту найденной информации через выходной поток.

В листинге 3.3 показан полный текст описанной выше программы-сервера.

Листинг 3.3. Содержимое файла EchoServer.java

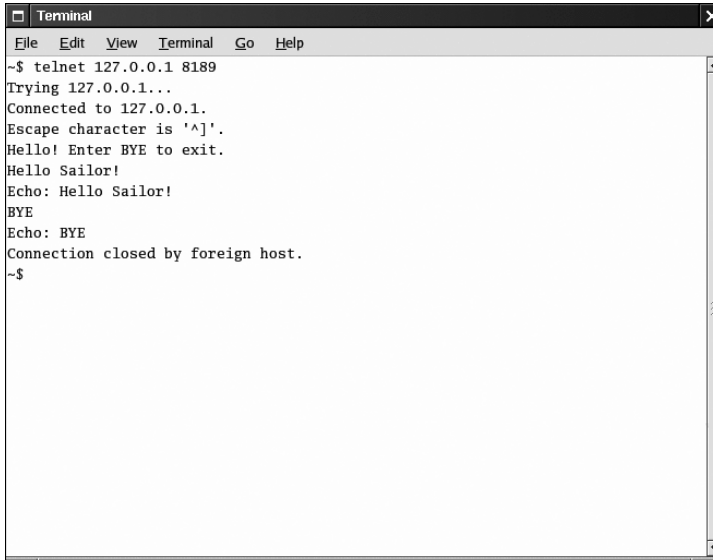
```
1. import java.io.*;
2. import java.net.*;
3. import java.util.*;
4.
5. /**
6.  * Данная программа реализует простой сервер, который ожидает
7.  * обращения по порту 8189 и возвращает клиенту данные,
8.  * полученные от него.
9.  * @version 1.20 2004-08-03
10. * @author Cay Horstmann
11. */
12. public class EchoServer
13. {
14.     public static void main(String[] args )
15.     {
16.         try
17.         {
18.             // Создание сокета на стороне сервера.
19.             ServerSocket s = new ServerSocket(8189);
20.
21.             // Ожидание обращения клиента.
22.             Socket incoming = s.accept( );
23.             try
24.             {
25.                 InputStream inStream = incoming.getInputStream();
26.                 OutputStream outStream = incoming.getOutputStream();
27.
28.                 Scanner in = new Scanner(inStream);
29.                 PrintWriter out = new PrintWriter(outStream, true
30.                     /* автоматическая передача оставшихся данных */);
31.
32.                 out.println( "Hello! Enter BYE to exit." );
33.
34.                 // Передача клиенту полученных от него данных.
35.                 boolean done = false;
36.                 while (!done && in.hasNextLine())
37.                 {
38.                     String line = in.nextLine();
39.                     out.println("Echo: " + line);
40.                     if (line.trim().equals("BYE")) done = true;
41.                 }
42.             }
43.             finally
44.             {
45.                 incoming.close();
46.             }
47.         }
48.         catch (IOException e)
49.         {
50.             e.printStackTrace();
51.         }
52.     }
53. }
```

Для проверки работоспособности программы ее нужно откомпилировать и запустить. Затем необходимо подключиться с помощью утилиты `telnet` к серверу `localhost` (или к IP-адресу `127.0.0.1`) и по порту `8189`.

Если ваш компьютер непосредственно подключен к Internet, любой пользователь может получить доступ к программе-серверу при условии, что ему известен IP-адрес и номер порта.

При обращении по этому порту будет получено сообщение, показанное на рис. 3.4:

```
Hello! Enter BYE to exit.
```



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

Рис. 3.4. Пример работы сервера, который передает клиенту полученные от него данные

Введите любую фразу, и вы получите ее в неизменном виде. Для отключения от программы-сервера введите `BYE` (все символы в верхнем регистре).

API `java.net.ServerSocket 1.0`

- `ServerSocket(int port)`
Создает сокет на стороне сервера, который осуществляет мониторинг порта.
- `Socket accept()`
Ожидает соединения. Этот метод блокирует (т.е. переводит в режим ожидания) текущий поток до тех пор, пока не будет совершено подключение. Метод возвращает объект `Socket`, через который программа может взаимодействовать с подключенным клиентом.
- `void close()`
Закрывает сокет на стороне сервера.

Обслуживание множества серверов

В предыдущем простом примере программы-сервера не предусмотрена возможность одновременного подключения сразу нескольких программ-клиентов. Обычно

программа-сервер работает на компьютере-сервере, а программы-клиенты могут одновременно подключаться к ней по Internet из любой точки мира. Если на сервере не предусмотрена обработка одновременного обращения нескольких клиентов, то это приведет к тому, что один клиент может монополизировать доступ к данной программе в течение длительного времени. Во избежание таких ситуаций следует прибегнуть к помощи потоков.

Для каждого нового соединения, т.е. при успешной обработке запроса на соединение, будет запущен новый поток, который позаботится об организации взаимодействия между программой-сервером и *данной* программой-клиентом. Для этого в программе-сервере следует организовать показанный ниже цикл.

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

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

Класс `ThreadedEchoHandler` реализует интерфейс `Runnable` и в своем методе `run()` поддерживает взаимодействие с программой-клиентом.

```
class ThreadedEchoHandler implements Runnable
{
    . . .
    public void run()
    {
        try
        {
            InputStream inStream = incoming.getInputStream();
            OutputStream outStream = incoming.getOutputStream();
            process input and send response
            incoming.close();
        }
        catch (IOException e)
        {
            обработка исключения
        }
    }
}
```

Теперь несколько программ-клиентов могут одновременно подключаться к серверу, поскольку для каждого соединения создается новый поток. Это можно легко проверить.

1. Скомпилируйте и запустите серверную программу, код которой приведен в листинге 3.4.
2. Откройте несколько окон утилиты `telnet` (рис. 3.5).
3. Переключайтесь между окнами и вводите команды. Теперь каждое отдельное окно утилиты `telnet` может независимо взаимодействовать с программой-сервером.
4. Для того чтобы разорвать соединение и закрыть окно утилиты `telnet`, нажмите комбинацию клавиш `<Ctrl+C>`.



В данной программе мы порождаем отдельный поток для каждого соединения. Этот подход не вполне приемлем для высокопроизводительного сервера. Более эффективной работы сервера можно добиться, используя средства из пакета `java.nio`. Дополнительную информацию по этому вопросу можно получить, обратившись по адресу <http://www-106.ibm.com/developerworks/java/library/j-javaio>.

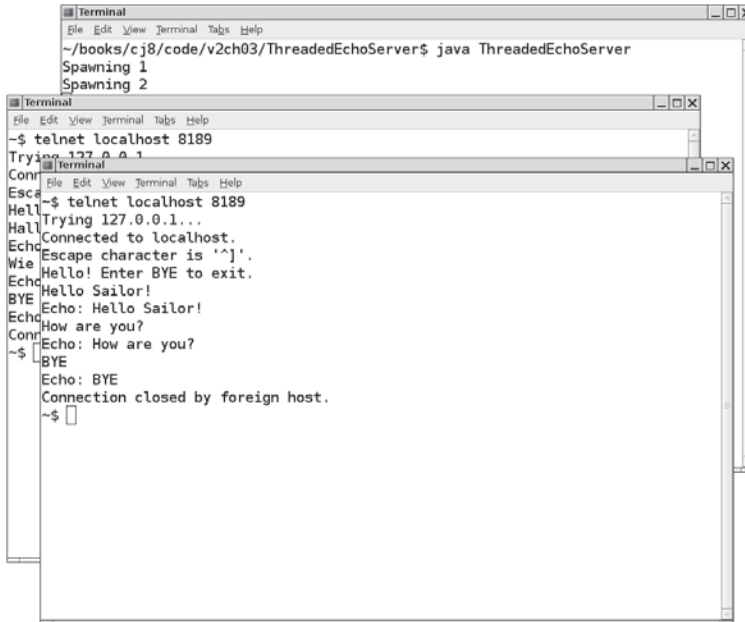


Рис. 3.5. Пример одновременного доступа к многопоточной программе-серверу

Листинг 3.4. Содержимое файла `ThreadedEchoServer.java`

```

1. import java.io.*;
2. import java.net.*;
3. import java.util.*;
4.
5. /**
6.  * Данная программа реализует многопоточный сервер, который
7.  * ожидает обращения по порту 8189 и возвращает клиенту
8.  * переданные им данные.
9.  * @author Cay Horstmann
10.  * @version 1.20 2004-08-03
11.  */
12. public class ThreadedEchoServer
13. {
14.     public static void main(String[] args )
15.     {
16.         try
17.         {
18.             int i = 1;
19.             ServerSocket s = new ServerSocket(8189);
20.

```

```
21.         while (true)
22.         {
23.             Socket incoming = s.accept();
24.             System.out.println("Spawning " + i);
25.             Runnable r = new ThreadedEchoHandler(incoming, i);
26.             Thread t = new Thread(r);
27.             t.start();
28.             i++;
29.         }
30.     }
31.     catch (IOException e)
32.     {
33.         e.printStackTrace();
34.     }
35. }
36. }
37.
38. /**
39.     Этот класс обрабатывает данные введенные клиентом
40.     в рамках одного соединения.
41. */
42. class ThreadedEchoHandler implements Runnable
43. {
44.     /**
45.         Конструктор обработчика.
46.         @param i Сокет для соединения
47.         @param c Счетчик обработчиков
48.     */
49.     public ThreadedEchoHandler(Socket i)
50.     {
51.         incoming = i;
52.     }
53.     public void run()
54.     {
55.         try
56.         {
57.             try
58.             {
59.                 InputStream inStream = incoming.getInputStream();
60.                 OutputStream outStream = incoming.getOutputStream();
61.
62.                 Scanner in = new Scanner(inStream);
63.                 PrintWriter out = new PrintWriter(outStream, true
64.                     /* автоматическая передача оставшихся данных */);
65.
66.                 out.println( "Hello! Enter BYE to exit." );
67.
68.                 // Передача клиенту полученных от него данных
69.                 boolean done = false;
70.                 while (!done && in.hasNextLine())
71.                 {
72.                     String line = in.nextLine();
73.                     out.println("Echo: " + line);
74.                     if (line.trim().equals("BYE"))
75.                         done = true;
76.                 }
77.             }

```

```

78.         finally
79.         {
80.             incoming.close();
81.         }
82.     }
83.     catch (IOException e)
84.     {
85.         e.printStackTrace();
86.     }
87. }
88.
89. private Socket incoming;
90. }

```

Одностороннее закрытие

Одностороннее закрытие (half-close) обеспечивает возможность одной стороне закрыть соединение с сокетом, прекратив отправку своих данных, но при этом получая данные с другой стороны.

Вот типичная ситуация. Предположим, что вы отправляете данные на сервер, но не знаете, какой объем данных необходимо передать. Если вы имеете дело с файлом, то, закрыв его, вы тем самым определяете конец данных. Закрыв же сокет, вы немедленно разорвете соединение с сервером.

Данную проблему решает одностороннее закрытие. Если вы закроете выходной поток, связанный с сокетом, вы тем самым укажете серверу на окончание запроса. При этом входной поток останется открытым, и вы сможете прочесть ответ.

Программа на стороне клиента, реализующая данный подход, выглядит приблизительно так, как показано ниже.

```

Socket socket = new Socket(host, port);
Scanner in = new Scanner(socket.getInputStream());
PrintWriter writer = new PrintWriter(socket.getOutputStream());
// Передача данных запроса.
writer.print(. . .);
writer.flush();
socket.shutdownOutput();
// Теперь один из потоков, связанных с сокетом, закрыт.
// Чтение ответа.
while (in.hasNextLine()) != null)
    { String line = in.nextLine(); . . . }
socket.close();

```

Программа на стороне сервера лишь читает данные из входного потока до тех пор, пока не закроется выходной поток на другом конце соединения.

Очевидно, что данный подход применим только при работе с протоколами, подобными HTTP, где клиент устанавливает соединение с сервером, передает запрос, получает ответ, после чего соединение разрывается.

API **java.net.Socket** 1.0

- `void shutdownOutput()` 1.3
Определяет “конец потока” для выходного потока.
- `void shutdownInput()` 1.3
Определяет “конец потока” для входного потока.

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

Прерываемые сокеты

При подключении с помощью сокета текущий поток блокируется до тех пор, пока соединение не будет установлено, или до истечения времени тайм-аута. Аналогично, если вы пытаетесь передать или прочитать данные посредством сокета, поток приостановит выполнение до успешного завершения операции или до окончания времени тайм-аута.

В реальных приложениях желательно предоставить пользователям возможность прервать слишком затянувшийся процесс установления соединения с помощью сокета. Однако если поток заблокирован, поскольку не получает ответа от сокета, вы не можете разблокировать его, вызвав метод `interrupt()`.

Для прерывания операций с сокетом используется класс `SocketChannel`, предоставляемый пакетом `java.nio`. Объект `SocketChannel` создается следующим образом:

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

С каналом не связываются потоки. Вместо этого он предоставляет методы `read()` и `write()`, использующие объекты `Buffer`. (Дополнительную информацию о буферах NIO можно найти в главе 1.) Данные методы объявлены в интерфейсах `ReadableByteChannel` и `WritableByteChannel`.

Если вы не хотите работать с буферами, можете использовать для чтения из `SocketChannel` объект `Scanner`. В классе `Scanner` предусмотрен конструктор с параметром `ReadableByteChannel`:

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

Для формирования выходного потока на базе канала применяется статический метод `Channels.newOutputStream()`:

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

Вот и все, что необходимо сделать. Если поток будет прерван в процессе открытия соединения, чтения или записи, соответствующая операция завершится и сгенерирует исключение.

Программа, код которой показан в листинге 3.5, демонстрирует прерываемые и блокирующие сокеты. Сервер передает числа, и должен остановить передачу после десятого числа. Щелкните на любой кнопке, чтобы запустить поток, устанавливающий соединение с сервером и выводящий выходные данные. Первый поток использует прерываемый сокет, а второй — блокирующий сокет. Если щелкнуть на кнопке `Cancel` (Отмена) во время вывода первых десяти чисел, вы прервете любой поток.

А если вы щелкнете на кнопке `Cancel` после первых десяти чисел, то вы прервете только первый поток. Второй поток будет продолжать блокирование до тех пор, пока сервер, в конце концов, не завершит соединение (рис. 3.6).

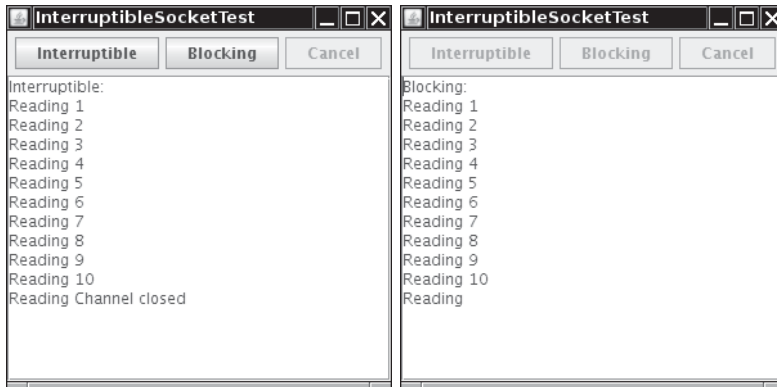


Рис. 3.6. Прерывание операций с сокетом

Листинг 3.5. Содержимое файла `InterruptibleSocketTest.java`

```

1. import java.awt.*;
2. import java.awt.event.*;
3. import java.util.*;
4. import java.net.*;
5. import java.io.*;
6. import java.nio.channels.*;
7. import javax.swing.*;
8.
9. /**
10.  * Данная программа демонстрирует возможность прерывания
11.  * при использовании канала сокета.
12.  * @author Cay Horstmann
13.  * @version 1.01 2007-06-25
14.  */
15. public class InterruptibleSocketTest
16. {
17.     public static void main(String[] args)
18.     {
19.         EventQueue.invokeLater(new Runnable()
20.         {
21.             public void run()
22.             {
23.                 {
24.                     JFrame frame = new InterruptibleSocketFrame();
25.                     frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
26.                     frame.setVisible(true);
27.                 }
28.             }
29.         });
30.     }
31.
32.     class InterruptibleSocketFrame extends JFrame
33.     {
34.         public InterruptibleSocketFrame()
35.         {
36.             setSize(WIDTH, HEIGHT);
37.             setTitle("InterruptibleSocketTest");
38.         }

```

```
39.     JPanel northPanel = new JPanel();
40.     add(northPanel, BorderLayout.NORTH);
41.
42.     messages = new JTextArea();
43.     add(new JScrollPane(messages));
44.
45.     interruptibleButton = new JButton("Interruptible");
46.     blockingButton = new JButton("Blocking");
47.
48.     northPanel.add(interruptibleButton);
49.     northPanel.add(blockingButton);
50.
51.     interruptibleButton.addActionListener(new ActionListener()
52.     {
53.         public void actionPerformed(ActionEvent event)
54.         {
55.             interruptibleButton.setEnabled(false);
56.             blockingButton.setEnabled(false);
57.             cancelButton.setEnabled(true);
58.             connectThread = new Thread(new Runnable()
59.             {
60.                 public void run()
61.                 {
62.                     try
63.                     {
64.                         connectInterruptibly();
65.                     }
66.                     catch (IOException e)
67.                     {
68.                         messages.append(
69.                             "\nInterruptibleSocketTest.connectInterruptibly: " + e);
70.                     }
71.                 }
72.             });
73.             connectThread.start();
74.         }
75.     });
76.
77.     blockingButton.addActionListener(new ActionListener()
78.     {
79.         public void actionPerformed(ActionEvent event)
80.         {
81.             interruptibleButton.setEnabled(false);
82.             blockingButton.setEnabled(false);
83.             cancelButton.setEnabled(true);
84.             connectThread = new Thread(new Runnable()
85.             {
86.                 public void run()
87.                 {
88.                     try
89.                     {
90.                         connectBlocking();
91.                     }
92.                     catch (IOException e)
93.                     {
94.                         messages.append(
95.                             "\nInterruptibleSocketTest.connectBlocking: " + e);
```

```
96.         }
97.     }
98.     });
99.     connectThread.start();
100.    }
101.    });
102.
103.    cancelButton = new JButton("Cancel");
104.    cancelButton.setEnabled(false);
105.    northPanel.add(cancelButton);
106.    cancelButton.addActionListener(new ActionListener()
107.    {
108.        public void actionPerformed(ActionEvent event)
109.        {
110.            connectThread.interrupt();
111.            cancelButton.setEnabled(false);
112.        }
113.    });
114.    server = new TestServer();
115.    new Thread(server).start();
116. }
117. /**
118.  * Соединение с тестовым сервером посредством прерываемого ввода-вывода.
119.  */
120. public void connectInterruptibly() throws IOException
121. {
122.     messages.append("Interruptible:\n");
123.     SocketChannel channel =
124.         SocketChannel.open(new InetSocketAddress("localhost", 8189));
125.     try
126.     {
127.         in = new Scanner(channel);
128.         while (!Thread.currentThread().isInterrupted())
129.         {
130.             messages.append("Reading ");
131.             if (in.hasNextLine())
132.             {
133.                 String line = in.nextLine();
134.                 messages.append(line);
135.                 messages.append("\n");
136.             }
137.         }
138.     }
139.     finally
140.     {
141.         channel.close();
142.         EventQueue.invokeLater(new Runnable()
143.         {
144.             public void run()
145.             {
146.                 messages.append("Channel closed\n");
147.                 interruptibleButton.setEnabled(true);
148.                 blockingButton.setEnabled(true);
149.             }
150.         });
151.     }
152. }
153.
```

```
154.  /**
155.   * Соединение с тестовым сервером посредством блокирующего ввода-вывода.
156.   */
157.  public void connectBlocking() throws IOException
158.  {
159.      messages.append("Blocking:\n");
160.      Socket sock = new Socket("localhost", 8189);
161.      try
162.      {
163.          in = new Scanner(sock.getInputStream());
164.          while (!Thread.currentThread().isInterrupted())
165.          {
166.              messages.append("Reading ");
167.              if (in.hasNextLine())
168.              {
169.                  String line = in.nextLine();
170.                  messages.append(line);
171.                  messages.append("\n");
172.              }
173.          }
174.      }
175.      finally
176.      {
177.          sock.close();
178.          EventQueue.invokeLater(new Runnable()
179.          {
180.              public void run()
181.              {
182.                  messages.append("Socket closed\n");
183.                  interruptibleButton.setEnabled(true);
184.                  blockingButton.setEnabled(true);
185.              }
186.          });
187.      }
188.  }
189.
190.  /**
191.   * Многопоточный сервер, который ожидает обращения
192.   * по порту 8189 и передает клиенту псевдослучайные значения,
193.   * имитируя останов сервера через 10 чисел.
194.   */
195.  class TestServer implements Runnable
196.  {
197.      public void run()
198.      {
199.          try
200.          {
201.              ServerSocket s = new ServerSocket(8189);
202.              while (true)
203.              {
204.                  Socket incoming = s.accept();
205.                  Runnable r = new TestServerHandler(incoming);
206.                  Thread t = new Thread(r);
207.                  t.start();
208.              }
209.          }
```

```
210.         catch (IOException e)
211.         {
212.             messages.append("\nTestServer.run: " + e);
213.         }
214.     }
215. }
216.
217. /**
218.  * Данный класс обрабатывает данные, полученные
219.  * от клиента в рамках одного соединения.
220.  */
221. class TestServerHandler implements Runnable
222. {
223.     /**
224.      * Конструктор обработчика.
225.      * @param i – входящий сокет
226.      */
227.     public TestServerHandler(Socket i)
228.     {
229.         incoming = i;
230.     }
231.
232.     public void run()
233.     {
234.         try
235.         {
236.             OutputStream outputStream = incoming.getOutputStream();
237.             PrintWriter out = new PrintWriter(outputStream, true /* autoFlush */);
238.             while (counter < 100)
239.             {
240.                 counter++;
241.                 if (counter <= 10) out.println(counter);
242.                 Thread.sleep(100);
243.             }
244.             incoming.close();
245.             messages.append("Closing server\n");
246.         }
247.         catch (Exception e)
248.         {
249.             messages.append("\nTestServerHandler.run: " + e);
250.         }
251.     }
252.     private Socket incoming;
253.     private int counter;
254. }
255.
256. private Scanner in;
257. private JButton interruptibleButton;
258. private JButton blockingButton;
259. private JButton cancelButton;
260. private JTextArea messages;
261. private TestServer server;
262. private Thread connectThread;
263.
264. public static final int WIDTH = 300;
265. public static final int HEIGHT = 300;
266. }
```

API java.net.InetSocketAddress 1.4

- `InetSocketAddress(String hostname, int port)`

Создает объект адреса для указанного узла и порта. Преобразование имени узла осуществляется в процессе установления соединения. Если преобразовать имя в адрес не удалось, устанавливается значения `true` свойства `unresolved`.

- `boolean isUnresolved()`

Возвращает `true`, если преобразование имени в адрес выполнить не удалось.

API java.nio.channels.SocketChannel 1.4

- `static SocketChannel open(SocketAddress address)`

Открывает канал сокета и связывает его с удаленным узлом.

API java.nio.channels.Channels 1.4

- `static InputStream newInputStream(ReadableByteChannel channel)`

Создает входной поток, поддерживающий чтение из указанного канала.

- `static OutputStream newOutputStream(WritableByteChannel channel)`

Создает выходной поток, поддерживающий запись в указанный канал.

Отправка электронной почты

В этом разделе приведен пример практического использования программирования сокетов для отправки электронной почты удаленному компьютеру.

Для отправки электронной почты необходимо установить соединение с сокетом по порту 25, который обычно используется для протокола SMTP (Simple Mail Transport Protocol – простой протокол передачи почты). Протокол SMTP описывает формат электронных сообщений. Вы можете подключаться к любому серверу, на котором выполняется служба SMTP. Однако сервер должен быть готов к приему запроса на соединение. Ранее серверы с демонами `sendmail` могли принимать любые электронные сообщения, но в настоящее время из-за большого объема спама (т.е. массовой рассылки электронных сообщений с навязчивой рекламой и другим бесполезным содержанием) в большинстве серверов встроена проверка и допуск почтовых сообщений только от разрешенных пользователей или IP-адресов.

Сразу после соединения с сервером следует послать заголовок сообщения (в формате SMTP, который достаточно просто создать), а затем и текст сообщения так, как показано ниже.

1. Создайте сокет на вашем компьютере:

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

2. В выходном потоке передайте следующую информацию:

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

В спецификации протокола SMTP (документ RFC 821) требуется, чтобы строки заканчивались символами /r и /n.

На некоторых SMTP-серверах достоверность данных не проверяется, т.е. можно передать любую информацию об отправителе. (Имейте в виду, что любой желающий может послать поддельное сообщение, например, с обратным адресом `president@whitehouse.gov` и приглашением посетить прием на лужайке Белого дома.)

В листинге 3.6 приведен очень простой пример почтовой программы. Как показано на рис. 3.7, с ее помощью можно ввести адрес отправителя, адрес получателя, имя SMTP-сервера, текст сообщения, а затем отправить данное сообщение с помощью кнопки Send (Отправить).

Эта программа просто передает SMTP-серверу последовательность команд, которая обсуждалась ранее. На экране эти команды отображаются вместе с ответами.

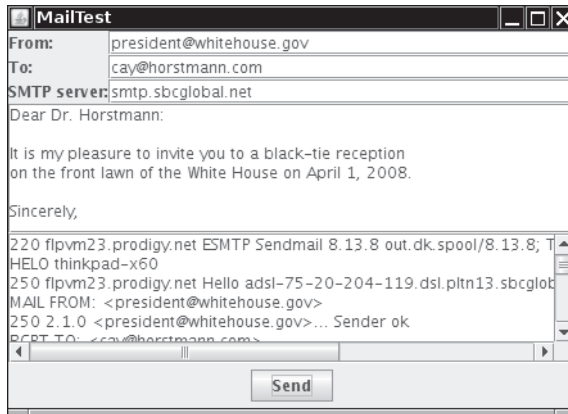


Рис. 3.7. Программа MailTest



Когда эта программа появилась в первом издании настоящей книги в 1996 г., большинство SMTP-серверов принимали соединения, которые поступали откуда угодно, вовсе не проверяя их. В наше время большинство серверов не являются столь доверчивыми, поэтому с запуском этой программы у вас могут возникнуть трудности. Почтовый сервер вашего поставщика услуг Internet может принимать соединения, поступающие из вашего дома или из доверенного IP-адреса. Другие серверы используют правило “POP before SMTP”, требующее, чтобы вы сначала загрузили свою электронную почту (для этого потребуются ввести пароль), прежде чем отправлять какие-либо сообщения. Попробуйте ввести ложный адрес электронной почты, прежде чем отправлять письмо с помощью этой программы. Все большую популярность приобретает расширение SMTP, требующее применения зашифрованного пароля (<http://tools.ietf.org/html/rfc2554>). Наша простая программа не поддерживает упомянутый механизм аутентификации.

В последнем разделе вы могли видеть, как используются приемы программирования на уровне сокетов для подключения к SMTP-серверу и отправки электронного сообщения. Хорошей новостью является то, что мы это можем сделать; более того, с помощью нашей программы вы сможете понять суть такой важной службы Internet, как электронная почта. Тем не менее, если вы планируете разработать приложение, которое будет включать систему обработки электронной почты, вам придется работать на более высоком уровне и использовать библиотеку, инкапсулирующую подробности протоколов. Например, компания Sun Microsystems разработала JavaMail API в

качестве стандартного расширения платформы Java. Чтобы отправить сообщение в JavaMail API, нужно просто сделать следующий вызов:

```
Transport.send(message);
```

Библиотека позаботится о протоколах сообщений, аутентификации, обработке прикрепленных файлов и т.п.

Листинг 3.6. Содержимое файла MailTest.java

```
1. import java.awt.*;
2. import java.awt.event.*;
3. import java.util.*;
4. import java.net.*;
5. import java.io.*;
6. import javax.swing.*;
7.
8. /**
9.  * Данная программа демонстрирует использование
10. * сокетов для передачи почтовых сообщений.
11. * @author Cay Horstmann
12. * @version 1.11 2007-06-25
13. */
14. public class MailTest
15. {
16.     public static void main(String[] args)
17.     {
18.         EventQueue.invokeLater(new Runnable()
19.         {
20.             public void run()
21.             {
22.                 JFrame frame = new MailTestFrame();
23.                 frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
24.                 frame.setVisible(true);
25.             }
26.         });
27.     }
28. }
29.
30. /**
31. * Фрейм для пользовательского интерфейса.
32. */
33. class MailTestFrame extends JFrame
34. {
35.     public MailTestFrame()
36.     {
37.         setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
38.         setTitle("MailTest");
39.
40.         setLayout(new GridBagLayout());
41.
42.         // Мы используем класс GBC, описанный в главе 9 первого тома.
43.         add(new JLabel("From:"), new GBC(0,
44.             0).setFill(GBC.HORIZONTAL));
45.
46.         from = new JTextField(20);
47.         add(from, new GBC(1,
48.             0).setFill(GBC.HORIZONTAL).setWeight(100, 0));
```



```
49.     add(new JLabel("To:"), new GBC(0,
50.         1).setFill(GBC.HORIZONTAL));
51.
52.     to = new JTextField(20);
53.     add(to, new GBC(1, 1).setFill(GBC.HORIZONTAL).
54.         setWeight(100, 0));
55.
56.     add(new JLabel("SMTP server:"), new GBC(0,
57.         2).setFill(GBC.HORIZONTAL));
58.
59.     smtpServer = new JTextField(20);
60.     add(smtpServer, new GBC(1,
61.         2).setFill(GBC.HORIZONTAL).setWeight(100, 0));
62.
63.     message = new JTextArea();
64.     add(new JScrollPane(message), new GBC(0,
65.         3, 2, 1).setFill(GBC.BOTH).setWeight(100, 100));
66.
67.     comm = new JTextArea();
68.     add(new JScrollPane(comm), new GBC(0,
69.         4, 2, 1).setFill(GBC.BOTH).setWeight(100, 100));
70.
71.     JPanel buttonPanel = new JPanel();
72.     add(buttonPanel, new GBC(0, 5, 2, 1));
73.
74.     JButton sendButton = new JButton("Send");
75.     buttonPanel.add(sendButton);
76.     sendButton.addActionListener(new ActionListener()
77.     {
78.         public void actionPerformed(ActionEvent event)
79.         {
80.             new SwingWorker<Void, Void>()
81.             {
82.                 protected Void doInBackground() throws Exception
83.                 {
84.                     comm.setText("");
85.                     sendMail();
86.                     return null;
87.                 }
88.             }.execute();
89.         }
90.     });
91. }
92. /**
93.  * Передача почтового сообщения, которое было задано
94.  * с применением средств пользовательского интерфейса.
95.  */
96. public void sendMail()
97. {
98.     try
99.     {
100.        Socket s = new Socket(smtpServer.getText(), 25);
101.
102.        InputStream inStream = s.getInputStream();
103.        OutputStream outStream = s.getOutputStream();
104.
105.        in = new Scanner(inStream);
106.        out = new PrintWriter(outStream, true /* autoFlush */);
```

```
107.         String hostName = InetAddress.getLocalHost().getHostName();
108.
109.         receive();
110.         send("HELO " + hostName);
111.         receive();
112.         send("MAIL FROM: <" + from.getText() + ">");
113.         receive();
114.         send("RCPT TO: <" + to.getText() + ">");
115.         receive();
116.         send("DATA");
117.         receive();
118.         send(message.getText());
119.         send(".");
120.         receive();
121.         s.close();
122.     }
123.     catch (IOException e)
124.     {
125.         comm.append("Error: " + e);
126.     }
127. }
128. /**
129.  * Передача строки сокету и воспроизведение ее
130.  * в текстовой области comm.
131.  * @param s Строка для передачи.
132.  */
133. public void send(String s) throws IOException
134. {
135.     comm.append(s);
136.     comm.append("\n");
137.     out.print(s.replaceAll("\n", "\r\n"));
138.     out.print("\r\n");
139.     out.flush();
140. }
141. /**
142.  * Получение строки посредством сокета и отображение
143.  * ее в текстовой области comm.
144.  */
145. public void receive() throws IOException
146. {
147.     String line = in.nextLine();
148.     comm.append(line);
149.     comm.append("\n");
150. }
151.
152. private Scanner in;
153. private PrintWriter out;
154. private JTextField from;
155. private JTextField to;
156. private JTextField smtpServer;
157. private JTextArea message;
158. private JTextArea comm;
159.
160. public static final int DEFAULT_WIDTH = 300;
161. public static final int DEFAULT_HEIGHT = 300;
162. }
```
