Содержание

Предисловие 13
К читателю 13
Краткий обзор книги 13
Условные обозначения 16
Примеры исходного кода 16
Благодарности 16
От издательства 18

Глава 1. Потоки данных 19
1.1. От итерации к потоковым операциям 20
1.2. Создание потока данных 22
1.3. Методы `filter()`, `map()` и `flatMap()` 28
1.4. Извлечение подпотоков и объединение потоков данных 30
1.5. Другие операции преобразования потоков данных 31
1.6. Простые методы сведения 32
1.7. Тип `Optional` 34
1.7.1. Получение необязательных значений 34
1.7.2. Употребление необязательных значений 35
1.7.3. Конвейеризация необязательных значений 36
1.7.4. Как не следует обрабатывать необязательные значения 37
1.7.5. Формирование необязательных значений 38
1.7.6. Сочетание функций необязательных значений с методом `flatMap()` 38
1.7.7. Преобразование типа `Optional` в поток данных 39
1.8. Накопление результатов 42
1.9. Накопление результатов в отображениях 47
1.10. Группирование и разделение 51
1.11. Нисходящие коллекторы 52
1.12. Операции сведения 57
1.13. Потоки данных примитивных типов 60
1.14. Параллельные потоки данных 65

Глава 2. Ввод и вывод 71
2.1. Потоки ввода-вывода 71
2.1.1. Чтение и запись байтов 72
2.1.2. Полный комплект потоков ввода-вывода 75
2.1.3. Сочетание фильтров потоков ввода-вывода 79
2.1.4. Ввод-вывод текста 82
2.1.5. Вывод текста 83
2.1.6. Ввод текста 85

2.1.7. Сохранение объектов в текстовом формате 86
2.1.8. Кодировки символов 90
2.2. Чтение и запись двоичных данных 92
2.2.1. Интерфейсы `DataInput` и `DataOutput` 92
2.2.2. Файлы с произвольным доступом 95
2.2.3. ZIP-архивы 99
2.3. Потоки ввода-вывода и сериализация объектов 102
2.3.1. Сохранение и загрузка сериализируемых объектов 102
2.3.2. Представление о формате файлов для сериализации объектов 107
2.3.3. Видоизменение исходного механизма сериализации 113
2.3.4. Сериализация одноэлементных множеств и типизированных перечислений 115
2.3.5. Контроль версий 116
2.3.6. Применение сериализации для клонирования 119
2.4. Манипулирование файлами 121
2.4.1. Пути к файлам 121
2.4.2. Чтение и запись данных в файлы 124
2.4.3. Создание файлов и каталогов 125
2.4.4. Копирование, перемещение и удаление файлов 126
2.4.5. Получение сведений о файлах 128
2.4.6. Обход элементов каталога 130
2.4.7. Применение потоков каталогов 132
2.4.8. Системы ZIP-файлов 135
2.5. Файлы, отображаемые в памяти 136
2.5.1. Эффективность файлов, отображаемых в памяти 136
2.5.2. Структура буфера данных 144
2.6.3. Блокирование файлов 146
2.6. Регулярные выражения 148
2.7.2. Совпадение со строкой 153
2.7.3. Обнаружение многих совпадений 157
2.7.4. Разбиение строк по разделителям 159
2.7.5. Замена совпадений 159

Глава 3. XML 163

3.1. Введение в XML 164
3.2. Структура XML-документа 166
3.3. Синтаксический анализ XML-документов 169
3.4. Проверка достоверности XML-документов 178
3.4.1. Определения типов документов 180
3.4.2. Схема XML-документов 187
3.4.3. Практический пример применения XML-документов 190
3.5. Поиск информации средствами XPath 196
3.6. Использование пространств имен 201
3.7. Потоковые синтаксические анализаторы 204
3.7.1. Применение SAX-анализатора 204
3.7.2. Применение StAX-анализатора 209
3.8. Формирование XML-документов 213
3.8.1. XML-документы без пространств имен 214
3.8.2. XML-документы с пространствами имен 214

3.8.3. Запись XML-документов 215
3.8.4. Запись XML-документов средствами StAX 217
3.8.5. Пример формирования файла в формате SVG 222
3.9. Преобразование XML-документов языковыми средствами XSLT 224

Глава 4. Работа в сети 235

4.1. Подключение к серверу 235
4.1.1. Применение утилиты `telnet` 235
4.1.2. Подключение к серверу из программы на Java 238
4.1.3. Время ожидания для сокетов 240
4.1.4. Межсетевые адреса 241
4.2. Реализация серверов 243
4.2.1. Сокеты сервера 243
4.2.2. Обслуживание многих клиентов 247
4.2.3. Полузакрытие 250
4.2.4. Прерываемые сокеты 251
4.3. Получение данных из Интернета 258
4.3.1. URL и URI 258
4.3.2. Извлечение данных средствами класса `URLConnection` 260
4.3.3. Отправка данных формы 267
4.4. HTTP-клиент 276
4.5. Отправка электронной почты 283

Глава 5. Работа с базами данных 287

5.1. Структура JDBC 288
5.1.1. Типы драйверов JDBC 288
5.1.2. Типичные примеры применения JDBC 290
5.2. Язык SQL 291
5.3. Конфигурирование JDBC 296
5.3.1. URL баз данных 296
5.3.2. Архивные JAR-файлы драйверов 297
5.3.3. Запуск базы данных 297
5.3.4. Регистрация класса драйвера 298
5.3.5. Подключение к базе данных 299
5.4. Работа с операторами JDBC 302
5.4.1. Выполнение операторов SQL 302
5.4.2. Управление подключениями, операторами и результирующими наборами 305
5.4.3. Анализ исключений SQL 306
5.4.4. Заполнение базы данных 309
5.5. Выполнение запросов 312
5.5.1. Подготовленные операторы для запросов 313
5.5.2. Чтение и запись больших объектов 320
5.5.3. Синтаксис переходов в SQL 321
5.5.4. Множественные результаты 323
5.5.5. Извлечение автоматически генерируемых ключей 324
5.6. Прокручиваемые и обновляемые результирующие наборы 324
5.6.1. Прокручиваемые результирующие наборы 325
5.6.2. Обновляемые результирующие наборы 327

5.7. Наборы строк 331
5.7.1. Построение наборов строк 331
5.7.2. Кешируемые наборы строк 332
5.8. Метаданные 335
5.9. Транзакции 345
5.9.1. Программирование транзакций средствами JDBC 345
5.9.2. Точки сохранения 346
5.9.3. Групповые обновления 346
5.9.4. Расширенные типы данных SQL 349
5.10. Управление подключением к базам данных в веб-приложениях и корпоративных приложениях 350

Глава 6. Прикладной интерфейс API даты и времени 353

6.1. Временн*а*я шкала 354
6.2. Местные даты 358
6.3. Корректоры дат 363
6.4. Местное время 364
6.5. Поясное время 366
6.6. Форматирование и синтаксический анализ даты и времени 370
6.7. Взаимодействие с унаследованным кодом 375

Глава 7. Интернационализация 377

7.1. Региональные настройки 378
7.1.1. Назначение региональных настроек 378
7.1.2. Указание региональных настроек 379
7.1.3. Региональные настройки по умолчанию 381
7.1.4. Отображаемые имена 382
7.2. Форматирование чисел 384
7.2.1. Форматирование числовых значений 384
7.2.2. Форматирование денежных сумм в разных валютах 390
7.3. Форматирование даты и времени 392
7.4. Сортировка и нормализация 400
7.5. Форматирование сообщений 407
7.5.1. Форматирование чисел и дат 407
7.5.2. Форматы выбора 409
7.6. Ввод-вывод текста 411
7.6.1. Текстовые файлы 411
7.6.2. Окончания строк 411
7.6.3. Консольный ввод-вывод 412
7.6.4. Протокольные файлы 413
7.6.5. Отметка порядка следования байтов в кодировке UTF-8 413
7.6.6. Кодирование символов в исходных файлах 414
7.7. Комплекты ресурсов 414
7.7.1. Обнаружение комплектов ресурсов 415
7.7.2. Файлы свойств 416
7.7.3. Классы комплектов ресурсов 417
7.8. Пример интернационализации прикладной программы 419

Глава 8. Написание сценариев, компиляция и обработка аннотаций 435

8.1. Написание сценариев для платформы Java 436
8.1.1. Получение интерпретатора сценариев 436
8.1.2. Выполнение сценариев и привязки 437
8.1.3. Переадресация ввода-вывода 439
8.1.4. Вызов функций и методов из сценариев 440
8.1.5. Компиляция сценариев 442
8.1.6. Пример создания сценария для обработки событий в пользовательском интерфейсе 443
8.2. Прикладной интерфейс API для компилятора 448
8.2.1. Вызов компилятора 448
8.2.2. Запуск заданий на компиляцию 449
8.2.3. Фиксация диагностики 450
8.2.4. Чтение исходных файлов из оперативной памяти 450
8.2.5. Запись байт-кодов в оперативную память 451
8.2.6. Пример динамического генерирования кода Java 453
8.3. Применение аннотаций 459
8.3.1. Введение в аннотации 460
8.3.2. Пример аннотирования обработчиков событий 461
8.4. Синтаксис аннотаций 466
8.4.1. Интерфейсы аннотаций 466
8.4.2. Объявление аннотаций 468
8.4.3. Аннотирование объявлений 470
8.4.4. Аннотирование в местах употребления типов данных 471
8.4.5. Аннотирование по ссылке `this` 472
8.5. Стандартные аннотации 473
8.5.1. Аннотации для компиляции 474
8.5.2. Аннотации для управления ресурсами 475
8.5.3. Мета-аннотации 476
8.6. Обработка аннотаций на уровне исходного кода 478
8.6.1. Процессоры аннотаций 479
8.6.2. Прикладной интерфейс API модели языка 479
8.6.3. Генерирование исходного кода с помощью аннотаций 480
8.7. Конструирование байт-кодов 483
8.7.1. Модификация файлов классов 483
8.7.2. Модификация байт-кодов во время загрузки 489

Глава 9. Модульная система на платформе Java 493

9.1. Понятие модуля 494
9.2. Именование модулей 495
9.3. Пример модульной программы `"Hello, Modular World!"` 496
9.4. Требования модулей 498
9.5. Экспорт пакетов 500
9.6. Модульные архивные JAR-файлы 503
9.7. Модули и рефлексивный доступ 505
9.8. Автоматические модули 508
9.9. Безымянные модули 510

9.10. Параметры командной строки для переноса прикладного кода 511
9.11. Переходные и статические требования 512
9.12. Уточненный экспорт и открытие модулей 514
9.13. Загрузка служб 514
9.14. Инструментальные средства для работы с модулями 517

Глава 10. Безопасность 521

10.1. Загрузчики классов 522
10.1.1. Процесс загрузки классов 522
10.1.2. Иерархия загрузчиков классов 523
10.1.3. Применение загрузчиков классов в качестве пространств имен 526
10.1.4. Создание собственного загрузчика классов 526
10.1.5. Верификация байт-кода 532
10.2. Диспетчеры защиты и полномочия 536
10.2.1. Проверка полномочий 536
10.2.2. Организация защиты на платформе Java 538
10.2.3. Файлы правил защиты 542
10.2.4. Специальные полномочия 548
10.2.5. Реализация класса полномочий 549
10.3. Аутентификация пользователей 555
10.3.1. Каркас JAAS 555
10.3.2. Модули регистрации JAAS 561
10.4. Цифровые подписи 570
10.4.1. Свертки сообщений 571
10.4.2. Подписание сообщений 574
10.4.3. Верификация подписи 577
10.4.4. Проблема аутентификации 580
10.4.5. Подписание сертификатов 582
10.4.6. Запросы сертификатов 584
10.4.7. Подписание кода 585
10.5. Шифрование 587
10.5.1. Симметричные шифры 588
10.5.2. Генерирование ключей шифрования 589
10.5.3. Потоки шифрования 595
10.5.4. Шифрование открытым ключом 596

Глава 11. Расширенные средства Swing и графика 601

11.1. Таблицы 601
11.1.1. Простая таблица 602
11.1.2. Модели таблиц 606
11.1.3. Манипулирование строками и столбцами таблицы 610
11.1.4. Воспроизведение и редактирование ячеек 626
11.2. Деревья 639
11.2.1. Простые деревья 640
11.2.2. Перечисление узлов дерева 657
11.2.3. Воспроизведение узлов дерева 659
11.2.4. Обработка событий в деревьях 662
11.2.5. Специальные модели деревьев 669

11.3. Расширенные средства AWT 678
 11.3.1. Конвейер визуализации 678
 11.3.2. Фигуры 681
11.3.3. Участки 697
 11.3.4. Обводка 699
 11.3.5. Раскраска 707
 11.3.6. Преобразование координат 709
 11.3.7. Отсечение 714
 11.3.8. Прозрачность и композиция 717
11.4. Растровые изображения 726
 11.4.1. Чтение и запись изображений 726
 11.4.2. Манипулирование изображениями 737
11.5. Вывод изображений на печать 753
 11.5.1. Вывод графики на печать 753
 11.5.2. Многостраничная печать 763
 11.5.3. Службы печати 773
 11.5.4. Потоковые службы печати 776
 11.5.5. Атрибуты печати 779

Глава 12. Платформенно-ориентированные методы 787

12.1. Вызов функции на C из программы на Java 788
12.2. Числовые параметры и возвращаемые значения 794
12.3. Строковые параметры 796
12.4. Доступ к полям 803
 12.4.1. Доступ к полям экземпляра 803
 12.4.2. Доступ к статическим полям 807
12.5. Кодирование сигнатур 808
12.6. Вызов методов на Java 810
 12.6.1. Методы экземпляра 810
 12.6.2. Статические методы 811
 12.6.3. Конструкторы 812
 12.6.4. Альтернативные вызовы методов 812
12.7. Доступ к элементам массивов 816
12.8. Обработка ошибок 820
12.9. Применение прикладного интерфейса API для вызовов 825
12.10. Практический пример обращения к реестру Windows 831
 12.10.1. Общее представление о реестре Windows 831
 12.10.2. Интерфейс для доступа к реестру на платформе Java 832
 12.10.3. Реализация функций доступа к реестру в виде платформенно-ориентированных методов 833

Предметный указатель 849

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) *
```

```
Terminal
~$ telnet time-a.nist.gov 13
Trying 129.6.15.28...
Connected to time-a.nist.gov.
Escape character is '^]'.

57488 16-04-10 04:23:00 50 0 0 610.5 UTC(NIST) *
Connection closed by foreign host.
~$
```

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).

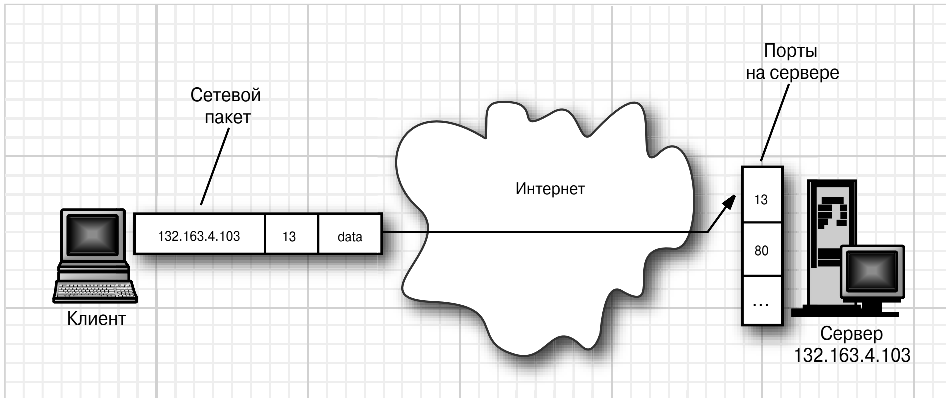

Рис. 4.2. Схема соединения клиента с сервером через конкретный порт

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.

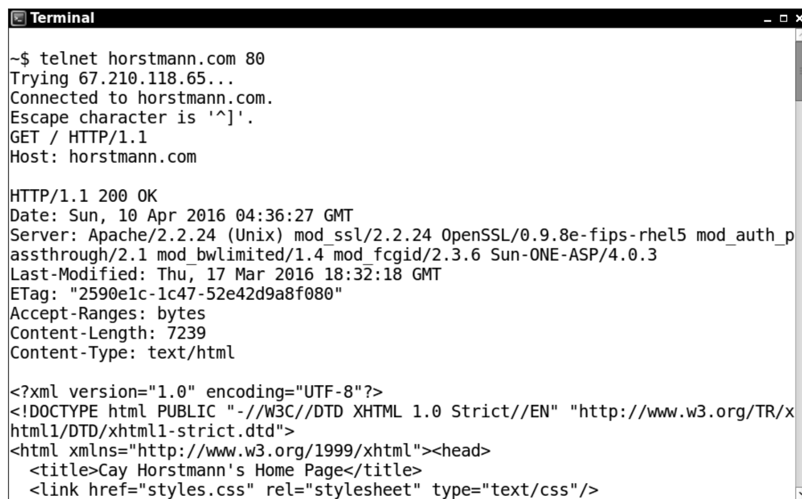

Рис. 4.3. Доступ к HTTP-порту с помощью утилиты **`telnet`**

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]

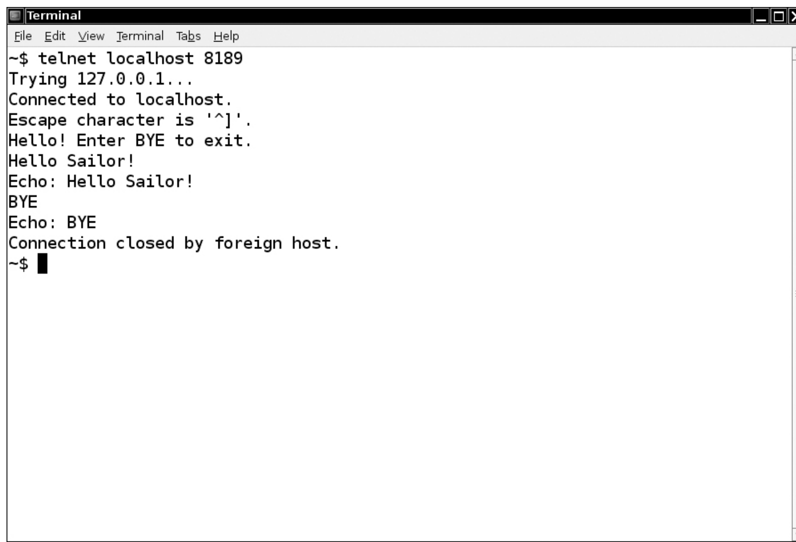

Рис. 4.4. Сеанс связи с сервером, передающим обратно данные, полученные от клиента

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
    }
```