

# Почтовая программа — своими руками!

Дмитрий Рамодин

Написание программ на Java представляет собой весьма интересное занятие, поскольку можно абстрагироваться от многих деталей, с которыми неизбежно сталкиваются пользователи Си++, Паскаля и других языков программирования третьего поколения. С самого начала специалисты корпорации Sun задумали Java как язык, стирающий разницу между разрозненными компьютерами и сетями, и, видимо, поэтому значительная часть библиотеки классов Java абстрагирует сетевые операции и транспортные протоколы.

Простейшая Java-программа, принимающая почту с сервера, — это по силам каждому.

## 1. Передача данных

Прежде чем написать почтовый клиент, неплохо было бы изучить язык, на котором общаются почтовый клиент и сервер, — протокол POP3 (Post Office Protocol). Именно с помощью этого протокола в большинстве случаев принимаются почтовые сообщения такими популярными почтовыми программами, как Microsoft Outlook и Express Netscape Messenger. Для начала примите к сведению, что все почтовые операции на нижнем уровне производятся с применением транспортного протокола TCP. Сокет сервера передает данные в формате POP3 TCP-пакетами в порт 110 (более ранняя версия протокола POP использовала порт 109).

Кроме того, необходимо знать, что команды от клиента к серверу пересылаются в виде текстовых строк, завершающихся парой символов "перевод каретки" (CR) и "перевод строки" (LF). Получив такое сообщение, почтовый сервер приступает к тяжелой работе: либо отвечает на ваши команды, либо пересылает сообщения, имеющиеся в вашем почтовом ящике. Вот здесь начинается настоящее мучение, ибо те, кто

разрабатывал протокол POP3, променяли удобство использования на простоту реализации, не подумав о программистах, которым с ним работать.

## **2. Формат обмена данными**

Команды, которые вы отправляете серверу, делятся на две категории: без параметров и содержащие параметры. Если простые команды без параметров состоят всего из одного слова, то в команды с параметрами добавляются текстовые параметры (или цифровые параметры, преобразованные в текстовую форму и разделенные пробелами). Каждый параметр может иметь длину до 40 символов. В качестве примера простой команды можно привести команду "STAT", которая запрашивает состояние вашего почтового ящика. Примером команды с параметром может служить команда "USER Mitrich", сообщающая серверу, что к нему подключается пользователь с именем Mitrich. Прежде чем отправиться дальше, загляните в таблицу и ознакомьтесь со всеми командами, которые описывает протокол POP3.

Возвращаемые сообщения могут быть двух видов: одно- и многострочные. В первом случае сервер возвращает одну строку; она начинается с признака статуса +OK или -ERR, говорящих об успехе или неудаче выполнения последней команды. Следом за признаком статуса сервер передает полезную информацию различного назначения. Завершается строка символами '\r' и '\n'. Если же возвращаемое сообщение многострочное, то оно передается строка за строкой, каждая из которых завершается '\r' и '\n', а в качестве признака окончания передачи используется строка из единственного символа '.' (точка), также оканчивающаяся символами возврата каретки и перевода строки.

## **3. Класс POPClient**

Ядро почтового клиента — это класс POPClient, реализованный на языке Java (см. листинг 1). Он предоставляет пользователю несколько полезных функций для управления посылкой команд серверу и получения ответа от последнего. Класс POPClient вмещает в себя два внутренних класса — POPCommand и POPResponse. Такого разбиения единого, казалось бы, класса на подклассы требует сама суть объектно-ориентированного программирования: каждая сущность должна быть реализована как отдельный класс. А поскольку и POPCommand и POPResponse

являются вспомогательными по отношению к главному классу POPClient, то они выполнены не как независимые классы, а как классы внутренние. Средства пакета JDK 1.1 предусматривают решение подобных задач.

### **3.1. Внутренний класс POPCommand**

Класс POPCommand, как вы, наверное, уже догадались, является инкапсуляцией команды протокола POP3. Для его создания нет конструктора, поэтому этим занимается компилятор Java. С помощью объектов этого класса формируются команды для отправки серверу и производится собственно отправка команды. Кроме того, обязанностью этого класса является чтение ответа сервера на посланную ранее команду. Самую черную работу на нижнем уровне выполняют следующие методы:

- `public boolean user(String name);`
- `public boolean pass(String password);`
- `public boolean quit();`
- `public boolean dele(int number);`
- `public boolean rset();`
- `public boolean list(int number);`
- `public boolean retr(int number);`

Нетрудно догадаться, что это всего лишь оболочки для отправки команд с аналогичными именами. Внутри этих методов происходит стандартная обработка: имеющийся аргумент проверяется, передается вспомогательному методу `transactCommand()` и затем возвращается в виде `true` (команда прошла и обработана сервером) или `false` (произошла ошибка). Когда же сервер прислал вместо вразумительного ответа "тарабарщину", происходит возбуждение исключения под названием `POPException` (о нем ниже в разделе "Класс `POPException`").

Метод `transactCommand` принимает текстовую строку, которая соответствует посылаемой команде, передает ее методу `sendCommand()`, затем читает ответ сервера с помощью метода `readCommandResponse()`. В заключение вызывается метод `isSucceed()` внутреннего класса `POPResponse`. Этот метод возвращает результат последней транзакции.

Если работа `sendCommand()` достаточно понятна, то работа метода `readCommandResponse()` требует некоторого пояснения. В начале своей работы этот

метод создает объект класса `StringBuffer` для временного хранения данных. Потом из потока ввода, подключенного к сокету, командой `ir.readLine()` читается строка ответа, которая добавляется к буферу строки методом `append()`. В момент окончания работы метода значение буфера преобразуется в строку методом `toString()` и записывается во внутреннее хранилище класса `POPResponse` методом `setBuff()`.

Но не все так просто, как может показаться с самого начала. Дело в том, что читать данные из потока сокета можно, а вот установить момент, когда они считаны до конца, нельзя. И документация в этом деле, увы, ничем не поможет. Там рекомендовано читать строки из сокета до тех пор, пока возвращаемое значение не будет равно `null`. Но уверяю вас, этого момента вы не дождетесь. Точно так же вы не сможете воспользоваться методом `read()` для чтения из-за того, что и он блокируется при достижении конца передачи данных. Со стороны это выглядит как зависание, но на самом деле поток, в котором вызывается метод чтения данных, просто переходит в состояние ожидания очередной порции информации. Поэтому мы идем на компромисс и в дополнение к методу `readCommandResponse()` реализуем метод `readMessage()`, который предназначен лишь для считывания многострочного почтового сообщения. Конец передачи данных мы определяем по строке, состоящей из одного символа точки ".". Метод `readMessage()` вызывается из командного метода `getr()` и, вместо того чтобы, как другие команды, сделать вызов метода `transactCommand()`, выполняет следующие три строки на Java:

```
sendCommand("RETR " + Integer.toString(number));  
readMessage();  
return response.isSuccess();
```

Обратите внимание, что многие методы класса `POPCommand` объявлены как `private`, чтобы предотвратить прямой доступ к ним.

### **3.2. Внутренний класс `POPResponse`**

Следующий внутренний класс `POPResponse` отвечает за хранение полученных от сервера данных и предоставление удобного интерфейса для их обработки. Все данные, принятые от сервера, сохраняются методом `setBuff()` в переменной `buff`, которая есть не что иное, как ссылка на строку класса `String`. Для вспомогательных нужд есть метод `getBuff`, с помощью которого можно получить содержимое хранилища в том виде, в

## *Почтовая программа — своими руками!*

каком оно было получено от почтового сервера.

Для предварительной обработки данных вы можете вызывать еще два полезных метода: `cutOffStatus()` и `getServerComment()`. Первый из них берет содержимое временного хранилища, отрезает от него признак статуса (+OK или -ERR) и возвращает оставшуюся строку. Второй метод возвращает текстовое сообщение, которое было передано POP-сервером сразу за признаком статуса. Следует, однако, отметить, что применять этот метод можно лишь после того, как серверу была передана команда, следом за которой сервер непременно пришлет сообщение (см. таблицу), иначе возвращенная методом `getServerComment()` строка будет бессмысленна.

Настал черед главного класса `POPClient`, с которым нам нужно разобраться. Сразу после объявления его самого описывается странная на первый взгляд переменная `debug`. Значение этой переменной управляет включением и выключением вывода отладочных сообщений в поток `System.out`. Если установить значение `true`, то специально написанный метод `logText()` будет посылать любую текстовую строку, которую вы ему передадите в окно консоли. После окончания отладки класса, вы можете установить `debug` в состояние `false`, и ваши строки, передаваемые `logText()`, уже не будут выводиться. Мало того, компилятор автоматически уберет посылки текста в окно консоли, оптимизируя код. Данная возможность походит на директивы условной компиляции препроцессора языков Си и Си++. О другой функции метода `logText()` мы поговорим чуть позже.

В начале класса `POPClient` объявляется несколько полей, имеющих следующее назначение:

- `POPCommand command` — ссылка на объект внутреннего класса `POPCommand`;
- `POPResponse response` — ссылка на объект внутреннего класса `POPResponse`;
- `Socket socket` — ссылка на сокет, связанный с почтовым сервером;
- `BufferedReader ir` — объект, читающий данные из потока ввода сокета;
- `PrintWriter ow` — объект, записывающий данные в поток вывода сокета;
- `PrintWriter log` — объект, записывающий данные в файл протокола;
- `Vector messages` — хранилище полученных почтовых сообщений;
- `boolean logEnabled` — флаг, разрешающий или запрещающий вывод текста в файл протокола.

В конструкторе по умолчанию производится создание новых экземпляров объектов внутренних классов POPCommand и POPResponse для дальнейшей работы. Если вам требуется протоколировать работу вашего почтового клиента, то воспользуйтесь другим конструктором, принимающим имя файла, который будет служить файлом протокола. Сначала этот конструктор вызовет конструктор по умолчанию, инициализируя таким образом внутренние классы, а затем создаст поток вывода данных в файл. Если таковой поток не удастся создать, то возникает исключение ввода-вывода, которое мы перехватываем, и устанавливаем флаг разрешения протокола в положение "отключено" (false).

Для забывчивых программистов в классе предусмотрен финализатор — метод finalize(), вызываемый сборщиком мусора в момент завершения работы класса. Финализатор вызывает метод disconnect(), отключающий клиента от почтового сервера, если программист по каким-либо причинам не вызывал метод disconnect().

Для соединения с POP-сервером создаваемый нами класс оснащен методом connect(), который достаточно прост. На первом этапе connect() создает сокет для присоединения к серверу с заданным именем и портом. Если сервер с именем hostName не существует, то после некоторого ожидания будет возбуждено исключение UnknownHostException. То же произойдет при возникновении проблем с вводом-выводом данных — будет сгенерировано исключение IOException. Мы перехватываем эти исключения и генерируем свое — POPException. В качестве параметра конструктора задается константа, определяющая вид исключения, которое мы хотим возбудить.

После создания сокета, мы вызываем его методы getInputStream() и getOutputStream(), чтобы получить ссылки на потоки ввода и вывода сокета, через которые мы будем посылать и получать данные. Для удобства работы полученные ссылки преобразуются в экземпляры объектов классов BufferedReader и PrintWriter, пришедших на смену морально устаревшим классам потоков BufferedInputStream и PrintStream. Остается считать ответ сервера:

```
command.readCommandResponse();
```

и определить, произошло ли соединение:

```
return response.isSuccess();
```

## *Почтовая программа — своими руками!*

Для отключения от сервера существует метод `disconnect()`. Он работает довольно прямолинейно: посылает серверу команду `QUIT` и поочередно закрывает потоки протокола сокетов, а затем и сам сокет.

Метод `logText()`, который косвенно упоминался выше, служит для выполнения двух функций: запись строки, полученной через параметр `text`, в файл протокола, если протоколирование разрешено. Эти же данные пересылаются в окно консоли в том случае, если флаг `debug` задан как `true`.

Для получения доступа к почте необходимо вызвать метод `login()`, посылающий имя пользователя и пароль почтовому серверу. Если какой-либо из параметров неверен, то происходит генерация исключения `POPException`.

Еще два метода `deleteMessage()` и `undoDeletes()` служат для удаления и отмены удаления сообщений. Если вы хотите удалить сообщение, то вызываете `deleteMessage()`, передавая номер почтового сообщения, которое подлежит удалению. Учтите, что нумерация начинается с 1, как это принято у почтового сервера. Само сообщение при этом лишь помечается на удаление и в дальнейших транзакциях не участвует. Реально оно будет удалено в момент отключения клиента от сервера. Если же вы поняли, что совершили ошибку, то можете просто вызвать `undoDeletes()`, после чего пометка на удаление будет снята, и вы сможете читать это сообщение. Правда, придется заново загрузить все почтовые сообщения на локальный компьютер.

## **4. Класс `POPException`**

Следуя концепции объектно-ориентированного программирования, просто необходимо создать свой собственный класс обработки исключительной ситуации — `POPException`. Он не только расширяет стандартный класс исключения `Exception`, но и добавляет специальное расширение - поле `why`, хранящее в виде числа причину возникновения исключения. Для удобства в классе определяются несколько констант, дающих этому числу осмысленное название. Если вы, создавая объект класса `POPException`, зададите в его конструкторе причину сбоя, то внутри класса вспомогательный метод `assignMessage()` подберет соответствующее текстовое сообщение. Перехватив исключительную ситуацию класса `POPException`, можно получить или разумное текстовое описание методом `getMessage()`, или же код причины

методом `why()`.

Завершая описание, советуем в дополнение к протоколу POP3 ознакомиться со следующими рабочими документами:

RFC821 — Simple Mail Transfer Protocol; RFC1321 — The MD5 Message-Digest Algorithm.

## 5. Команды протокола POP3

Успех и неудача выполнения команды отмечаются признаками статуса +OK или -ERR соответственно

Команда	Назначение	Возможные возвращаемые значения
USER <имя пользователя>	Посылка имени пользователя серверу	+OK <комментарий сервера> — если имя пользователя правильное -ERR <комментарий сервера> — если имя пользователя неверное
PASS <пароль>	Посылка пароля серверу	+OK<комментарий сервера> — если пароль принят сервером -ERR <комментарий сервера> — если пароль неверный или с почтовым ящиком уже кто-то работает
QUIT	Окончание сеанса работы	+OK
STAT	Получить состояние почтового ящика	+OK <кол-во сообщений> <общий размер всех сообщений>
UST [<номер сообщения>]	Получить параметры всех сообщений в ящике пользователя. Если задан номер сообщения, то будут получены только его параметры	+OK <параметры сообщений> -ERR <комментарий сервера> — если запрошенного сообщения в ящике нет.  Возвращаемые параметры сообщений зависят

*Почтовая программа — своими руками!*

		от того, был ли задан номер сообщения. Если — да, то сразу после +OK следует сообщение сервера. Затем строка за строкой передаются параметры всех сообщений в формате <номер сообщения> <размер сообщения>
RETR <номер сообщения>	Получить сообщение с сервера	+OK <тест запрошенного сообщения> — если команда прошла удачно -ERR <комментарий сервера> — если запрошенное сообщение отсутствует на сервере
DELE <номер сообщения>	Пометить сообщение на сервере как удаленное. Реально оно будет удалено после команды QUIT	+OK <комментарий сервера> — если сообщение было помечено на удаление -ERR <комментарий сервера> — если сообщение не существует или уже отмечено как удаленное
NOOP	Пустая операция	+OK
RSET	Отменить удаление удаленных сообщений, помеченных как удаленные	+OK <комментарий сервера>
TOP <номер сообщения> <кол-во строк>	Считать заголовок сообщения и первые строки в количестве, заданном параметром <кол-во строк>	+OK Далее строка за строкой передается заголовок сообщения. За ним следует пустая строка и, если имеется второй параметр, передаются начальные строки сообщения
UIDL [<номер сообщения>]	Получить уникальные идентификаторы всех сообщений в ящике пользователя. Если задан номер сообщения, то будет получен только его идентификатор	+OK <параметры сообщений> -ERR <комментарий сервера> — если запрошенного сообщения в ящике нет.  Возвращаемые параметры сообщений зависят от того, был ли задан номер

		сообщения. Если — да, то сразу после +OK идут номер запрошенного сообщения и его идентификатор. Если команда вызвана без параметра, то после статуса +OK следует сообщение сервера. Затем строка за строкой передаются параметры всех сообщений в формате <номер сообщения> <идентификатор>
АРОР <имя пользователя> <дайджест>	Осуществляет подключение к почтовому серверу по закодированной алгоритмом MD5 строке, защищая транзакцию от разглашения пароля пользователя	+OK <комментарий сервера> — если имя пользователя или дайджест соответствуют имеющемуся почтовому ящику пользователя -ERR <комментарий сервера> — если имя пользователя или дайджест неверны

## 6. Листинг 1

```
// Базовый класс для приема сообщений от
// почтового сервера (с использованием
// протокола POP3).

// Формируем пакет
package Mitrich.mail;

// Импортируем нужные классы
import java.io.*;
import java.net.*;
import java.util.Enumeration;
import java.util.Vector;
import java.util.StringTokenizer;
// Вспомогательный класс. См. листинг 2
import Mitrich.mail.POPException;

// Главный класс реализации протокола POP3
```

## Почтовая программа — своими руками!

```
public class POPClient
{
    // После окончания отладки установить false
    private final static boolean debug = true;

    // Номер порта POP3
    public final static short POP3_PORT = 110;

    // Константы класса
    private final String EOL = "\r\n";
    private final String S_OK = "+OK";
    private final String S_ERR = "-ERR";

    // Поля класса
    private POPCommand command = null;
    private POPResponse response = null;
    private Socket socket = null;
    private BufferedReader ir = null;
    private PrintWriter ow = null;
    private PrintWriter log = null;
    private Vector messages = null;
    private boolean logEnabled = false;

    // Конструктор по умолчанию
    public POPClient()
    {
        // Создаем экземпляр класса команды
        command = this.new POPCommand();
        // Создаем экземпляр класса ответа сервера
        response = this.new POPResponse();
    }

    // Этот конструктор создает файл протокола
    // с именем, заданным параметром logName
    public POPClient(String logName)
    {
        this();
        logEnabled = true;
        try
```

```
{
    // Создадим поток вывода в файл протокола
    // с автоматическим сбросом буферов на диск
    log = new PrintWriter(
        new FileOutputStream(logName, true),
        true);
}
catch(IOException e)
{
    // Возникла проблема с созданием файла протокола
    // Протоколирование отключается
    logEnabled = false;
}
}

// Финализатор
protected void finalize() throws Throwable
{
    // Отсоединиться от почтового сервера, если
    // пользователь забыл вызвать метод disconnect()
    disconnect();
}

// Установить соединение с почтовым сервером
public boolean connect(String hostName, int portNumber)
    throws POPException
{
    try
    {
        logText("Creating a socket...");
        // Создаем сокет
        socket = new Socket(hostName, portNumber);

        logText("Creating an input stream...");
        // Получаем ссылку на поток ввода данных от сокета
        ir = new BufferedReader(
            new InputStreamReader(
                socket.getInputStream()));

        logText("Creating an output stream...");
```

## Почтовая программа — своими руками!

```
// Получаем ссылку на поток вывода данных в сокет
ow = new PrintWriter(
    new DataOutputStream(
        socket.getOutputStream()), true);

// Слушаем ответ сервера
command.readCommandResponse();
// Ответ +OK или -ERR ?
return response.isSuccess();
}
catch(UnknownHostException e)
{
    logText("Host unknown");
    // Заданный адрес сервера недействителен
    throw new POPException(POPException.HOST_UNKNOWN);
}
catch(IOException e)
{
    logText("Creating an I/O channel failed");
    // Ошибка сокета. Возможно, сервер отключился
    throw new POPException(POPException.SOCKET_ERROR);
}
}

// Отключаемся от сервера
public void disconnect()
{
    try
    {
        logText("Disconnecting... ");
        // Посылается запрос на отключение
        command.quit();
        if(ir != null)
        {
            ir.close();
            ir = null;
        }
        if(ow != null)
        {
            ow.close();
        }
    }
}
```

```
        ow = null;
    }
    if(socket != null)
    {
        socket.close();
        socket = null;
    }
    if(log != null)
    {
        log.close();
        log = null;
    }
    }
    catch(Exception e)
    { logText("Disconnection failed");}
}

// Записать строку в файл протокола
public void logText(String text)
{
    if(logEnabled) log.println(text);
    if(debug) System.out.println(text);
}

// Входим в почтовый сервер
public void login(String name, String password)
                throws POPException
{
    logText("Sending the user name...");
    // Передаем имя пользователя серверу
    if( !command.user(name) )
        throw new POPException(POPException.BAD_NAME);
    logText("Sending the password...");
    // Если пароль есть, то передаем его серверу
    if(password != null)
        if( !command.pass(password) )
            throw new POPException(POPException.BAD_PASSWORD);
}

// Загрузить всю почту на компьютер пользователя
```

## Почтовая программа — своими руками!

```
public void downloadMessages()
    throws POPException
{
    StringTokenizer st = null;
    String tmpStr = null;
    // Создаем пустой вектор для хранения сообщений
    messages = new Vector();

    // Перебираем сообщения на сервере
    for(int i = 1; command.list(i); i++)
    {
        // Если сообщение с заданным номером имеется,
        // считать его с сервера
        if(command.retr(i))
        {
            // Найти место в данных, где находится первая
            // отметка конца строки
            int offset = response.buff.indexOf(EOL);
            // Записать сообщение в вектор, отрезав от
            // него строку статуса
            messages.addElement(
                response.buff.substring(
                    offset + EOL.length()));
        }
        else throw new POPException(POPException.RETR_ERROR);
    }
}

// Возвращает пользователю текст сообщения.
// Нумерация начинается с 1
public String getMessage(int number)
{
    return (String)messages.elementAt(number - 1);
}

// Удаляет сообщение. Нумерация начинается с 1
public boolean deleteMessage(int number)
    throws POPException
{
    if(command.dele(number))
```

```
{
    // Удалить сообщение из вектора
    messages.removeElementAt(number-1);
    return true;
}
return false;
}

// Отменить удаление сообщений, которые были
// удалены вызовом метода deleteMessage()
public boolean undoDeletes() throws POPException
{
    if(command.rset())
    {
        messages = null;
        // Перезагрузить почту с сервера
        downloadMessages();
        return true;
    }
    return false;
}

// Этот внутренний класс предварительно сохраняет
// необработанные данные
class POPResponse
{
    // Временное хранилище полученных данных
    private String buff = "";

    // Возвращает true, если последняя команда
    // выполнена успешно, false — если
    // неудачно. Возбуждает исключение, если
    // сервер прислал неопределенный ответ
    public boolean isSuccess() throws POPException
    {
        boolean result = true;

        if(!buff.startsWith(S_OK))
        {
            if(!buff.startsWith(S_ERR))
```

## Почтовая программа — своими руками!

```
        {
            throw new POPException(POPException.BAD_RESPONSE);
        }
        result = false;
    }
    return result;
}

// Записать данные во временный буфер
protected void setBuff(String s)
{
    buff = s;
}

// Считать данные из временного буфера
protected String getBuff()
{
    return buff;
}

// Возвратить данные без статусного признака
protected String cutOffStatus()
{
    int offset = buff.indexOf(' ');
    if(offset != -1)
    {
        String tmpStr = buff.substring(offset);
        return tmpStr.trim();
    }
    return null;
}

// Получить комментарий сервера к последней
// выполненной команде
public String getServerComment()
{
    String tmpStr = null;

    tmpStr = cutOffStatus();
    int offset = tmpStr.indexOf(EOL);
```

```
        if(offset != -1)
            return tmpStr.substring(0, offset);
        return null;
    }
}

// Внутренний класс для представления команды,
// посылаемой серверу
class POPCommand
{
    // Посылает строку почтовому серверу
    private void sendCommand(String command)
        throws POPException
    {
        logText("Sending command... ");
        try{ ow.println(command); }
        catch(IOException e)
        { throw new POPException(POPException.IO_ERROR); }
    }

    // Получить ответ на служебную команду
    private void readCommandResponse() throws POPException
    {
        logText("Reading response...");
        StringBuffer tmpBuff = new StringBuffer();

        try { tmpBuff.append(ir.readLine()); }
        catch(IOException e)
        {
            throw new POPException(POPException.IO_ERROR);
        }
        response.setBuff(tmpBuff.toString());
    }

    // Считать почтовое сообщение
    private void readMessage() throws POPException
    {
        logText("Reading message...");
        String tmpStr = new String("");
    }
}
```

## Почтовая программа — своими руками!

```
StringBuffer tmpBuff = new StringBuffer();

try
{
    // Читать строку за строкой, пока не будет
    // найдена строка-терминатор.
    while(!(tmpStr = ir.readLine()).equals("."))
        tmpBuff.append(tmpStr + "\r\n");
}
catch(IOException e)
{
    throw new POPException(POPException.IO_ERROR);
}
tmpStr = tmpBuff.toString();
response.setBuff(tmpStr);
}

// Метод-оболочка для выполнения типичной команды
private boolean transactCommand(String command)
    throws POPException
{
    sendCommand(command);
    readCommandResponse();
    return response.isSuccess();
}

// Команды передачи имени пользователя
public boolean user(String name) throws POPException
{
    return transactCommand("USER " + name);
}

// Команды передачи пароля
public boolean pass(String password) throws POPException
{
    return transactCommand("PASS " + password);
}

// Команда завершения сеанса работы с почтой
public boolean quit() throws POPException
```

```
{
    return transactCommand("QUIT");
}

// Команда удаления сообщения
public boolean dele(int number) throws POPException
{
    if(number != 0)
        return transactCommand("DELE " + Integer.toString(number));
    else return false;
}

// Команда отмены удаления
public boolean rset() throws POPException
{
    return transactCommand("RSET");
}

// Получить информацию о сообщении
public boolean list(int number) throws POPException
{
    if(number != 0)
        return transactCommand("LIST " + Integer.toString(number));
    else return false;
}

// Команда чтения сообщения с сервера
public boolean retr(int number) throws POPException
{
    if(number != 0)
    {
        sendCommand("RETR " + Integer.toString(number));
        readMessage();
        return response.isSuccess();
    }
    else return false;
}
}
```

<hr>

## 7. Листинг 2

```
// Вспомогательный класс представляет исключения,  
// которые могут возникнуть в процессе работы  
// Автор — Дмитрий Рамодин  
// Изд. дом "Открытые Системы"  
//  
  
// Формируем пакет  
package Mitrich.mail;  
  
public class POPException extends Exception  
{  
    // Константы типов исключительных ситуаций  
    public static final int NOT_AVAILABLE = 0;  
    public static final int BAD_RESPONSE = 1;  
    public static final int BAD_NAME = 2;  
    public static final int BAD_PASSWORD = 3;  
    public static final int SOCKET_ERROR = 4;  
    public static final int HOST_UNKNOWN = 5;  
    public static final int IO_ERROR = 6;  
    public static final int RETR_ERROR = 7;  
  
    // Причина возникновения исключения  
    private static int why = NOT_AVAILABLE;  
  
    // Конструктор по умолчанию  
    public POPException()  
    {  
        super();  
    }  
  
    // Конструктор со строкой  
    public POPException(String message)  
    {
```

```
super(message);
}

// Конструктор, в котором задается причина
// исключительной ситуации
public POPException(int reason)
{
    super(POPException.assignMessage(reason));
}

// Задать строку, соответствующую причине
private static String assignMessage(int reason)
{
    why = reason;

    switch(reason)
    {
        case BAD_RESPONSE:
            return new String(
                "Bad response from the mail server\n");
        case SOCKET_ERROR:
            return new String(
                "Socket I/O couldn't be established\n");
        case BAD_NAME:
            return new String(
                "There is not such user name\n");
        case BAD_PASSWORD:
            return new String(
                "Invalid password\n");
        case HOST_UNKNOWN:
            return new String(
                "Wrong hostname\n");
        case IO_ERROR:
            return new String(
                "I/O operation error.\n");
        case RETR_ERROR:
            return new String(
                "Fatal error ocurred during message reading\n");
        default:
            return new String(
```

## Почтовая программа — своими руками!

```
        "Unknown POPClient failure\n");
    }
}

// Возвращает причину исключительной ситуации
public int why()
{
    return why;
}
}
```

[Мир ПК #01/98](#)