**Лабораторная работа 4**

**Первый пример**

Разобранный в предыдущем разделе пример уже достаточно сложен и содержит примеры всех необходимых нам технологий, для того чтобы послужить прототипом для первой версии нашей "ресторанной" системы. Чтобы программы-примеры не получались слишком большими, договоримся, что наша система будет поддерживать очень ограниченный набор операций:

* операцию выдачи клиенту новой карты;
* операцию пополнения счета;
* операцию оплаты картой покупки (снятия средств со счета);
* операцию получения остатка средств на карте.

Кроме того, практически не будут реализованы механизмы защиты от перерасхода средств, механизмы контроля действий пользователя и т.д., поскольку главной задачей рассматриваемых программ является демонстрация тех или иных механизмов взаимодействия, а не демонстрация реализации соответствующей бизнес-логики.

**Класс BillingService**

Первый серверный класс **BillingService** ([пример 4.1](https://www.intuit.ru/studies/courses/633/489/lecture/11077?page=1#example.4.1)) очень похож на класс **TCPServer** из предыдущего примера - так же как и **TCPServer**,он создает серверный сокет, после чего большую часть времени проводит в ожидании соединения клиентов. При подключении клиента создается экземпляр класса **BillingClientService**,в конструктор которого передаются, соответственно, **InputStream** и **OutputStream**,извлеченные из сокета. **BillingClientService** является потомком класса **Thread**,поэтому после запуска (вызов метода **start)** начинает обрабатывать клиентский запрос параллельно, в то время как **BillingService** возвращается к ожиданию соединения со следующим клиентом. Таким образом, предлагаемый класс **BillingService** способен одновременно обрабатывать соединения с несколькими клиентами.

Поскольку основной задачей системы является выполнение операций по обслуживанию клиентских карт, класс **BillingService** предлагает соответствующие методы - всего их четыре:

* **void addNewCard(String personName, String card)** - добавляет в систему новую карту, с идентификатором card и personName - ФИО пользователя;
* **void addMoney(String card, double money)** - увеличивает остаток на карте card на величину money (пополнение счета);
* **void subMoney(String card, double money)** - уменьшает остаток на карте card на величину money (оплата картой);
* **double getCardBalance(String card)** - для карты с идентификатором card возвращает значение баланса.

Для хранения "базы" карт применяется хэш-таблица, в которой ключом служит идентификатор карты, а в качестве значения выступает ее баланс. Соответственно, все операции над картой в качестве обязательного параметра содержат ее идентификатор, используя который в качестве ключа, получают доступ к балансу.

1 package com.asw.net.ex1;

2 import java.net.\*;

3 import java.util.Hashtable;

4 import java.io.\*;

5

6 public class BillingService extends Thread{

7 public static final int ADD\_NEW\_CARD = 1;

8 public static final int ADD\_MONEY = 2;

9 public static final int SUB\_MONEY = 3;

10 public static final int GET\_CARD\_BALANCE = 4;

11 public static final int EXIT\_CLIENT = 5;

12

13 private int serverPort = 7896;

14 private ServerSocket ss;

15 private Hashtable hash;

16

17 public static void main(String[] args) {

18 BillingService bs = new BillingService();

19 bs.start();

20 }

21

22 public BillingService(){

23 hash = new Hashtable();

24 }

25

26 public void run(){

27 try {

28 ss = new ServerSocket(serverPort);

29 System.out.println("Server started");

30 while(true){

31 Socket s = ss.accept();

32 System.out.println("Client accepted");

33 BillingClientService bcs = new BillingClientService(this,

new DataInputStream(s.getInputStream()), new DataOutputStream(s.getOutputStream()));

34 bcs.start();

35 }

36 } catch (IOException e) {

37 e.printStackTrace();

38 }

39 }

40

41 public void addNewCard(String personName, String card) {

42 hash.put(card, new Double(0.0));

43 }

44 public void addMoney(String card, double money) {

45 Double d = (Double)hash.get(card);

46 if (d!=null) hash.put(card,new Double(d.doubleValue()+money));

47 }

48 public void subMoney(String card, double money) {

49 Double d = (Double)hash.get(card);

50 if (d!=null) hash.put(card,new Double(d.doubleValue()-money));

51 }

52 public double getCardBalance(String card) {

53 Double d = (Double)hash.get(card);

54 if (d!=null) return d.doubleValue();

55 return 0;

56 }

57 }

Листинг 4.1. Класс BillingService

Следующий серверный класс - **BillingClientService** - отвечает непосредственно за взаимодействие с клиентом ([пример 4.2](https://www.intuit.ru/studies/courses/633/489/lecture/11077?page=2#example.4.2)). Получив в конструкторе два потока - поток ввода и поток вывода, он способен с их помощью как принимать данные, отправляемые клиентом, так и посылать ему данные.

Клиентское приложение должно иметь возможность передать серверу информацию о том, какие действия должен выполнить сервер - создать новую карту, изменить баланс существующей и т.д. Причем для многих из этих операций нужно передать дополнительную информацию - об идентификаторе карты, о сумме, которую с нее предстоит списать, и т.д. Таким образом, поскольку все, что у нас есть - это потоки, нам предстоит разработать некое соглашение о том, как передавать информацию от клиента к серверу, или, говоря другими словами, создать собственный прикладной протокол для нашей системы.

Разработка таких протоколов, вообще говоря, задача не простая, поскольку от ее решения во многом зависит производительность и надежность будущей системы. В нашем же случае мы реализуем простейший протокол, сообщения в котором будут иметь следующий вид:

|  |  |
| --- | --- |
| Код операции | Данные |

При этом количество и типы данных для каждой операции различны, а вот коды операции имеют целый тип, и их значения заранее известны.

Зафиксировав такое соглашение, мы можем построить обработку клиентского соединения следующим образом: в цикле читаем из потока целое число, которое интерпретируем как код операции, и, в зависимости от прочитанного кода операции, вызываем соответствующий метод. Вызываемый метод выполняет следующие действия:

* дочитывает из потока необходимые для работы параметры (поскольку это конкретный метод, например, увеличения баланса карты, список параметров для него известен);
* вызывает соответствующий метод **BillingService**,передавая ему прочитанные параметры, который и выполняет все необходимые действия.

Проиллюстрируем этот механизм на примере обработки операции по заведению новой карты.

Клиент посылает серверу данные следующего формата: код операции (в данном случае операция ADD\_NEW\_CARD имеет код 1), затем строку с именем владельца, затем строку, содержащую идентификатор карты. **BillingClientService** в методе **run** считывает код операции, после чего вызывает свой метод **addNewCard**.Этот метод "знает", что для операции заведения новой карты необходимо два строковых параметра. Он считывает их из потока, после чего вызывает метод **addNewCard**,который заносит новую запись в хэш-таблицу. Остальные операции выполняются аналогично. Единственный метод, о котором стоит сказать особо, - это метод **getCardBalance**,который отличается от других тем, что не только читает данные от клиента (идентификатор карты), но и отправляет ему данные, используя поток вывода **OutputStream**.

1 package com.asw.net.ex1;

2 import java.io.\*;

3

4 public class BillingClientService extends Thread {

5 DataInputStream dis;

6 DataOutputStream dos;

7 BillingService bs;

8

9 public BillingClientService (BillingService bs,DataInputStream dis,DataOutputStream dos){

10 this.bs = bs;

11 this.dis = dis;

12 this.dos = dos;

13 }

14

15 public void run(){

16 System.out.println("ClientService thread started");

17 boolean work = true;

18 while (work) {

19 int command;

20 try {

21 command = dis.readInt();

22 switch (command) {

23 case BillingService.ADD\_NEW\_CARD:

24 addNewCard();

25 break;

26 case BillingService.ADD\_MONEY:

27 addMoney();

28 break;

29 case BillingService.SUB\_MONEY:

30 subMoney();

31 break;

32 case BillingService.GET\_CARD\_BALANCE:

33 getCardBalance();

34 break;

35 case BillingService.EXIT\_CLIENT:

36 work = false;

37 break;

38 default:

39 System.out.println("Bad operation:" + command);

40 }

41 } catch (IOException e) {

42 e.printStackTrace();

43 }

44 }

45 }

46

47 void addNewCard() throws IOException{

48 String personName = dis.readUTF();

49 String card = dis.readUTF();

50 bs.addNewCard(personName,card);

51 }

52 void addMoney() throws IOException{

53 String card = dis.readUTF();

54 double money = dis.readDouble();

55 bs.addMoney(card,money);

56 }

57 void subMoney() throws IOException{

58 String card = dis.readUTF();

59 double money = dis.readDouble();

60 bs.subMoney(card,money);

61 }

62 void getCardBalance() throws IOException{

63 String card = dis.readUTF();

64 double money = bs.getCardBalance(card);

65 dos.writeDouble(money);

66 }

67 }

Листинг 4.2. Класс BillingClientService

**Класс BillingClient**

Что касается клиентского приложения, то в данном случае оно устроено очень просто. Сетевое имя сервера оно принимает в качестве аргумента командной строки, порт известен заранее. Первое, что делается, - устанавливается соединение с сервером (метод **connectToServer)**.Затем создаются три новые карты с разными идентификаторами. Создаются они с помощью метода **sendNewCardOperation**,который действует в соответствии с установленными нами протоколом: он записывает в поток сначала код операции, затем имя владельца карты, затем его идентификатор.

После создания карт в цикле выполняется метод **sendAddMoneyOperation**,который посылает серверу команды об увеличении баланса для только что добавленных карт. Цикл выполняется 1000 раз, поэтому средств на каждой из карт в итоге оказывается много.

С помощью вызовов **sendGetCardBalanceOperation** в качестве параметра которого передается идентификатор карты, мы можем в этом убедиться.

Все реализованные в клиентском приложении методы: **send NewCardOperation, sendAddMoneyOperation, sendSubMoneyOperation, sendGetCardBalanceOperation, closeConnection** - "симметричны" соответствующим методам **BillingClientService** и неукоснительно соблюдают разработанный протокол.

1 package com.asw.net.ex1;

2 import java.net.\*;

3 import java.io.\*;

4

5 public class BillingClient {

6 int serverPort = 7896;

7 String serverName;

8 Socket s;

9 DataInputStream dis;

10 DataOutputStream dos;

11

12 public BillingClient(String serverName){

13 this.serverName = serverName;

14 }

15

16 public static void main(String[] args) {

17 BillingClient bc = new BillingClient(args[0]);

18 try {

19 bc.startTest();

20 } catch (IOException e) {

21 e.printStackTrace();

22 }

23 }

24

25 public void startTest() throws IOException{

26 connectToServer();

27 sendNewCardOperation("Piter","1");

28 sendNewCardOperation("Stefan","2");

29 sendNewCardOperation("Nataly","3");

30 for (int i=0; i<1000;i++){

31 sendAddMoneyOperation("1", i%10);

32 sendAddMoneyOperation("2", i%20);

33 sendAddMoneyOperation("3", i%30);

34 }

35 System.out.println("1:"+sendGetCardBalanceOperation("1"));

36 System.out.println("2:"+sendGetCardBalanceOperation("2"));

37 System.out.println("3:"+sendGetCardBalanceOperation("3"));

38 closeConnection();

39 }

40

41 void connectToServer() throws UnknownHostException, IOException{

42 s = new Socket(serverName, serverPort);

43 dis = new DataInputStream(s.getInputStream());

44 dos = new DataOutputStream(s.getOutputStream());

45 }

46

47 void sendNewCardOperation(String personName, String card) throws IOException{

48 dos.writeInt(BillingService.ADD\_NEW\_CARD);

49 dos.writeUTF(personName);

50 dos.writeUTF(card);

51 }

52

53 void sendAddMoneyOperation(String card, double money) throws IOException{

54 dos.writeInt(BillingService.ADD\_MONEY);

55 dos.writeUTF(card);

56 dos.writeDouble(money);

57 }

58 void sendSubMoneyOperation(String card, double money) throws IOException{

59 dos.writeInt(BillingService.SUB\_MONEY);

60 dos.writeUTF(card);

61 dos.writeDouble(money);

62 }

63 double sendGetCardBalanceOperation(String card) throws IOException{

64 dos.writeInt(BillingService.GET\_CARD\_BALANCE);

65 dos.writeUTF(card);

66 return dis.readDouble();

67 }

68 void closeConnection() throws IOException{

69 dos.writeInt(BillingService.EXIT\_CLIENT);

70 }

71 }

Листинг 4.3. Класс BillingClient