**Лабораторная работа 5**

Рассмотренный в предыдущем примере способ взаимодействия имеет очень существенный недостаток, который происходит от выбранного нами протокола взаимодействия между клиентом и сервером (что еще раз подчеркивает важность правильного выбора протокола). Этот недостаток состоит в том, что наш протокол полностью не защищен от ошибок приложения. Предположим, например, что наш клиент при передаче операции заведения новой карты вместо того, чтобы передать три параметра (код операции, имя клиента и номер карты), случайно передал четыре - код операции, имя клиента, фамилия клиента и номер карты (такое возможно в случае ошибки программиста, реализующего приложение-клиент, или в случае появления новой, расширенной версии приложения). Поскольку сервер ничего об этом не знает, он считает из потока три параметра, а четвертый (номер карты) будет интерпретирован как код следующей операции. Таким образом, использование явной передачи параметров методов может приводить к трудноуловимым ошибкам.

Попробуем модифицировать наше приложение, чтобы избавиться от этого недостатка. Идея состоит в том, чтобы группировать передаваемые данные некоторым образом так, чтобы даже в случае изменения их структуры не происходило их перемешивания.

Мы могли бы модифицировать протокол из предыдущего примера, введя в него разделители команд или предусмотрев первоначальный обмен клиента и сервера метаданными, описывающими передаваемые объекты - существует масса способов достичь желаемого результата. Однако, поскольку приложение написано на языке Java, разумно воспользоваться средствами, уже встроенными в эту платформу. Речь идет о сериализации объектов.

В самом деле, рассмотрим еще раз нашу задачу. Все, что необходимо, - реализовать механизм обмена структурами данных (или объектами) между клиентом и сервером. И встроенный механизм сериализации в этом случае нам идеально подходит. Мы будем передавать по сети не отдельные "поля", а целые объекты.

Первое, что необходимо сделать, - выделить эти "транспортные" объекты. В нашем случае таких объектов будет два - объект "карта" и объект "операция над картой". Поскольку в нашем примере рассматривается система, выполняющая очень небольшое число действий, все операции системы можно свести к двум типам - операции, в которых аргументом является карта целиком, и операции, в которых в качестве аргумента выступает начисление/списание средств с карты. К первому типу относятся операции заведения новой карты и операция получения баланса карты (баланс мы будем считать ее атрибутом), ко второму - операции пополнения баланса и расчета.

**Класс CardOperation**

Первый класс, который мы реализуем, - класс "операция над картой" ([пример 4.4](https://www.intuit.ru/studies/courses/633/489/lecture/11077?page=3#example.4.4)).

1 package com.asw.net.ex2;

2 import java.util.\*;

3 import java.io.\*;

4

5

6 public class CardOperation implements Serializable {

7 public CardOperation(String card,double amount,Date operationDate){

8 this.card = card;

9 this.amount = amount;

10 this.operationDate = operationDate;

11 }

12 public String card;

13 public double amount;

14 public Date operationDate;

15 }

Листинг 4.4. Транспортный класс CardOperation

Первое, что стоит отметить: класс **CardOperation** объявлен реализующим интерфейс **Serializable**.Этот интерфейс является "тэгирующим" - он не содержит полей или методов и служит только для того, чтобы сообщить о том, что данный класс сериализуем.

Класс содержит три поля, которые и составляют его полезное наполнение - номер карты, сумма начисляемых/списываемых средств и дата операции.

**Класс Card**

Второй транспортный класс представляет собой представление карты и выглядит следующим образом ([пример 4.5](https://www.intuit.ru/studies/courses/633/489/lecture/11077?page=3#example.4.5)).

1 package com.asw.net.ex2;

2 import java.io.Serializable;

3 import java.util.\*;

4

5 public class Card implements Serializable{

6 public Card(String person, Date createDate, String cardNumber,double balance){

7 this.person = person;

8 this.createDate = createDate;

9 this.cardNumber = cardNumber;

10 this.balance = balance;

11 }

12 public String person;

13 public transient Date createDate;

14 public String cardNumber;

15 public double balance;

16 public String toString(){

17 return "Card: cardNumber="+cardNumber+"\tBalance="+balance+"\tPerson="+person+"\tCreateDate="+createDate+"";

18 }

19 }

Листинг 4.5. Транспортный класс Card

Поскольку его тоже предстоит передавать по сети, он также объявлен как сериализуемый и содержит следующие поля: ФИО клиента, дату заведения карты, номер карты и текущий баланс. Для удобства вывода на экран перекрыт метод toString(), который распечатывает значения всех полей класса.

Перейдем теперь к серверным классам.

**Класс BillingService**

Первый из серверных классов - **BillingService** ([пример 4.6](https://www.intuit.ru/studies/courses/633/489/lecture/11077?page=3#example.4.6)).

1 package com.asw.net.ex2;

2 import java.net.\*;

3 import java.util.Hashtable;

4 import java.io.\*;

5

6 public class BillingService extends Thread{

7 private int serverPort = 7896;

8 private ServerSocket ss;

9 private Hashtable hash;

10

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

12 BillingService bs = new BillingService();

13 bs.start();

14 }

15

16 public BillingService(){

17 hash = new Hashtable();

18 }

19

20 public void run(){

21 try {

22 ss = new ServerSocket(serverPort);

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

24 while(true){

25 System.out.println("new client waiting...");

26 Socket s = ss.accept();

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

28 BillingClientService bcs = new BillingClientService(this,s);

29 System.out.println("bcs created");

30 bcs.start();

31 }

32 } catch (IOException e) {

33 e.printStackTrace();

34 }

35

36 }

37

38 public void addNewCard(Card card) {

39 hash.put(card.cardNumber, card);

40 }

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

42 Card c = (Card)hash.get(card);

43 if (c==null) {

44 System.out.println("Bad Card number\n");

45 return;

46 };

47 c.balance+=money;

48 hash.put(card,c);

49 }

50 public Card getCard(String card){

51 return (Card)hash.get(card);

52 }

53 }

Листинг 4.6. Серверный класс BillingService

Несложно заметить, что он очень похож на **BillingService** из предыдущего примера. И действительно, он точно так же хранит хэш-таблицу карт, имеет методы для обработки соответствующих событий - добавления новой карты, запроса баланса карты и обработки операций изменения баланса. Основной цикл **run** (строки 21-32) тоже точно такой же - сначала создается серверный сокет, затем ожидаются соединения клиентов. При появлении нового клиента создается экземпляр класса **BillingClientService** (строка 28), который и занимается всем дальнейшим обслуживанием клиента.

Однако есть и отличие - теперь в качестве элемента хранения в хэш-таблице используются экземпляры класса **Card** (ключом по-прежнему выступает ее номер), а метод **getCard**, заменивший метод **getCardBalance**, возвращает не скалярное значение, а экземпляр класса **Card**.

Наибольшие отличия от примера из предыдущего раздела имеет класс BillingClientService ([пример 4.7](https://www.intuit.ru/studies/courses/633/489/lecture/11077?page=4#example.4.7)).

**Класс BillingClientService**

1 package com.asw.net.ex2;

2 import java.io.\*;

3 import java.net.\*;

4

5 public class BillingClientService extends Thread {

6 ObjectInputStream ois;

7 ObjectOutputStream oos;

8 BillingService bs;

9 Socket s;

10

11 public BillingClientService(BillingService bs,Socket s){

12 System.out.println("Constructor BillingClientService\n");

13 this.bs = bs;

14 this.s = s;

15 try {

16 this.oos = new ObjectOutputStream(s.getOutputStream());

17 this.ois = new ObjectInputStream(s.getInputStream());

18 } catch (IOException e) {

19 // TODO Auto-generated catch block

20 e.printStackTrace();

21 }

22

23 System.out.println("Stream`s done \n socket="+s);

24 }

25 public void run(){

26 System.out.println("ClientService thread started\n");

27 boolean work = true;

28 while (work) {

29 int command;

30 Object o;

31 try {

32 o = ois.readObject();

33 if (o instanceof Card[]) {

34 Card[] cards = (Card[])o;

35 for (int i=0;i<cards.length;i++){

36 bs.addNewCard(cards[i]);

37 }

38 }else if (o instanceof CardOperation[]){

39 CardOperation[] co = (CardOperation[])o;

40 for (int i=0;i<co.length;i++){

41 bs.addMoney(co[i].card,co[i].amount);

42 }

43 }else if (o instanceof String){

44 oos.writeObject(bs.getCard((String)o));

45 }else System.out.println("Bad operation");

46

47 } catch (IOException e) {

48 e.printStackTrace();

49 } catch (ClassNotFoundException e) {

50 e.printStackTrace();

51 }

52 }

53 }

54 }

Листинг 4.7. Класс BillingClientService

Во-первых, в качестве потоков ввода-вывода он использует, соответственно, **ObjectInputStream** и **ObjectOutputStream**.Эти высокоуровневые потомки от **InputStream** и **OutputStream** предоставляют возможность оперировать с целыми объектами, а не со значениями примитивных типов, как **DataInputStream** и **DataOutputStream** из предыдущего примера. Конструируются они, как и следовало ожидать, принимая в качестве параметров базовый класс (строки 16,17).

Поскольку класс **BillingClientService** является потомком **Thread**,вся его функциональность сконцентрирована в методе **run**.Так же как и в предыдущем примере, этот метод представляет собой цикл ожидания сообщений от клиента. Однако метод чтения клиентских сообщений значительно отличается от метода, рассмотренного в предыдущем примере. В данном случае, при использовании метода **readObject()** класса **ObjectInputStream**,считывается экземпляр посланного клиентом класса (строка 32). Поскольку неизвестно заранее, какие именно классы прислал клиент (напомним, транспортных классов в нашей системе два - класс карт и класс операций над ними), осуществляется проверка типов (строки 33, 38, 43). Если присланный объект оказался массивом **Card**,то осуществляется его преобразование к этому типу (строка 34), после чего в цикле производится вызов соответствующего метода создания новой карты, в качестве параметра которому передается соответствующий элемент массива, т.е. значение типа **Card**.

В случае если присланный объект оказывается массивом **CardOperation**,осуществляется его приведение к типу "массив CardOperation\* (строка 39) и в цикле вызывается метод изменения баланса.

Если же присланный объект оказался объектом класса **String**,то мы интерпретируем этот вызов как запрос карты по ее номеру и отправляем карту клиенту (строка 44).

Стоит обратить внимание на простоту этой операции. Все, что нужно сделать, - вызвать метод **writeObject** класса **ObjectOutputStream**.Поскольку класс **Card** объявлен как сериализуемый, экземпляр этого класса, возвращенный методом **getCard** класса **BillingService**,будет автоматически "упакован", т.е. преобразован в некий служебный формат, который может быть сохранен во внешнем файле, отправлен по сети и т.д. В этом формате будут сохранены все текущие значения полей объекта, а значит, после восстановления мы получим точную копию сохраненного объекта.

Итак, для передачи копии карты клиенту сервер вызывает метод **writeObject**,а клиент должен прочитать карту соответствующим вызовом **readObject**.

При использовании сериализации следует отчетливо понимать, что принимающая сторона получает копию объекта, а не сам объект. Так, если сервер передаст клиенту карту с номером "1", а клиент, получив ее, изменит в ней поле баланса, значение баланса у карты "1" на сервере не изменится, поскольку изменения эти произойдут лишь в клиентской копии.

**Класс BillingClient**

Реализация клиента (класс **BillingClient** - [пример 4.8](https://www.intuit.ru/studies/courses/633/489/lecture/11077?page=4#example.4.8)) похожа на реализацию из предыдущего примера, за исключением, конечно, способа передачи данных.

После запуска теста - вызов **startTest** из метода **main** (строка 77) - происходит соединение с сервером - **connectToServer** (строка 18), после чего формируется два массива. Первый массив содержит объекты класса **Card** (это те карты, которые затем будут переданы на сервер для их заведения), второй - массив из 30000 операций над этими картами. Вызов метода **processCard** (строка 23) передает на сервер массив карт, вызов **processOperation** (строка 34) передает на сервер массив операций.

Для передачи методы **processCard** и **processOperation** используют уже знакомый нам метод **writeObject** класса **ObjectOutputStream**.

После того, как соответствующие карты и операции по ним переданы для обработки на сервер, с помощью вызовов метода **getCard** с сервера запрашиваются копии соответствующих объектов. Поскольку у класса **Card** переопределен метод **toString**,полученные копии могут быть выведены в консоль, что позволяет убедиться в том, что балансы карт действительно изменены в соответствии с переданными операциями.

1 package com.asw.net.ex2;

2 import java.net.\*;

3 import java.util.Date;

4 import java.io.\*;

5

6

7 public class BillingClient {

8

9 int serverPort = 7896;

10 String serverName;

11 Socket s;

12 ObjectInputStream ois;

13 ObjectOutputStream oos;

14 public BillingClient(String serverName){

15 this.serverName = serverName;

16 }

17 public void startTest() throws IOException{

18 connectToServer();

19

20 Card[] cards = {new Card("Piter",new Date(),"1",0.0),

21 new Card("Stefan",new Date(),"2",0.0),

22 new Card("Nataly",new Date(),"3",0.0)};

23 processCard(cards);

24

25 int cnt = 30000;

26 CardOperation[] co = new CardOperation[cnt];

27 for (int i = 0; i < cnt; i++) {

28 switch (i%3){

29 case 0: co[i] = new CardOperation("1",1,new Date());break;

30 case 1: co[i] = new CardOperation("2",2,new Date());break;

31 case 2: co[i] = new CardOperation("3",3,new Date());break;

32 }

33 }

34 processOperation(co);

35 try {

36 System.out.println("getCard: "+getCard("1"));

37 System.out.println("getCard: "+getCard("2"));

38 System.out.println("getCard: "+getCard("3"));

39 } catch (IOException e1) {

40 e1.printStackTrace();

41 } catch (ClassNotFoundException e1) {

42 e1.printStackTrace();

43 }

44

45 java.lang.Object sync = new java.lang.Object();

46 synchronized (sync) {

47 try {

48 sync.wait();

49 } catch (InterruptedException e) {

50 e.printStackTrace();

51 }

52 }

53

54 }

55 void connectToServer() throws UnknownHostException, IOException{

56 s = new Socket(serverName, serverPort);

57 System.out.println("connection established\n");

58 ois = new ObjectInputStream(s.getInputStream());

59 oos = new ObjectOutputStream(s.getOutputStream());

60 System.out.println("Stream`s done \n ois="+ois+"\n oos="+oos);

61 }

62 void processOperation(CardOperation[] co) throws IOException{

63 System.out.println(co);

64 oos.writeObject(co);

65 }

66 void processCard(Card[] c) throws IOException{

67 System.out.println("processCard: c="+c);

68 oos.writeObject(c);

69 }

70 Card getCard(String card) throws IOException, ClassNotFoundException{

71 oos.writeObject(card);

72 return (Card)ois.readObject();

73 }

74 public static void main(String[] args) throws Exception{

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

76 try {

77 bc.startTest();

78 } catch (IOException e) {

79 e.printStackTrace();

80 }

81 }

82 }

Листинг 4.8. Класс BillingClient