ФЕДЕРАЛЬНОЕ АГЕНТСТВО ПО ОБРАЗОВАНИЮ
КЕМЕРОВСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
Кафедра ЮНЕСКО по новым информационным технологиям

С.Н. Карабцев

ВВЕДЕНИЕ В ПРОГРАММИРОВАНИЕ НА JAVA

Лабораторный практикум

Математический факультет

Специальность 010503 – математическое обеспечение и администрирование информационных систем

Кемерово, 2009

ЛАБОРАТОРНАЯ РАБОТА №5.
ПАКЕТ java.io. СЕРИАЛИЗАЦИЯ ОБЪЕКТОВ

1. Цель работы

Целью работы является ознакомление со средствами стандартного пакета языка Java java.io для успешного выполнения операций ввода-вывода из создаваемых программ, а также приобретение практических навыков работы при обмене данными через файл и консоль.

2. Методические указания

Лабораторная работа направлена на приобретение знаний о системе ввода-вывода, реализованной в языке Java с помощью пакета java.io, а также приобретение основных понятии о сериализации и десериализации объектов для подготовки их к дальнейшей отправке по сети.

Требования к результатам выполнения лабораторного практикума:

- при выполнении задания необходимо сопровождать все реализованные процедуры и функции набором тестовых входных и выходных данных и описаниями к ним;
- реализацию программы можно осуществлять как из интегрированной среды разработки Eclipse, так и в блокноте с применением вызовов функций командной строки;
- по завершении выполнения задания составить отчет о проделанной работе.

При составлении и оформлении отчета следует придерживаться рекомендаций, представленных на следующей странице: http://unesco.kemsu.ru/student/rule/rule.html.

3. Теоретический материал

Подавляющее большинство программ обменивается данными с внешним миром. Это, безусловно, делают любые сетевые приложения - они передают и получают информацию от других компьютеров и специальных устройств, подключенных к сети. Оказывается удобным точно таким же образом представлять обмен данными между устройствами внутри одной машины. Так, например, программа может считывать данные с клавиатуры и записывать их в файл, или же наоборот - считывать данные из файла и выводить их на экран. Таким образом, устройства, откуда может производиться считывание информации, могут быть самыми разнообразными - файл, клавиатура, (входящее) сетевое соединение и т.д. То же самое касается и устройств вывода - это может быть файл, экран монитора, принтер, (исходящее) сетевое соединение и т.п. В конечном счете, все данные в компьютерной системе в процессе обработки передаются от устройств ввода к устройствам вывода. Обычно часть вычислительной платформы, которая отвечает за обмен данным, так и называется - система ввода/вывода. В Java она представлена пакетом java.io (input/output). Реализация системы ввода/вывода осложняется не только широким спектром источников и получателей данных, но еще и различными форматами передачи информации. Ею можно обмениваться в двоичном представлении, символьном или текстовом с применением некоторой кодировки.

Поток данных

В Java для описания работы по вводу/выводу используется специальное понятие ***поток данных*** (stream). Поток данных связан с некоторым ***источником*** или ***приемником*** данных, способных получать или предоставлять информацию. Соответственно, потоки делятся на ***входные*** - читающие данные, и на ***выходные*** - передающие (записывающие) данные. Введение концепции stream позволяет отделить программу, обменивающуюся информацией одинаковым образом с любыми

устройствами, от низкоуровневых операций с такими устройствами ввода/вывода.

В Java потоки естественным образом представляются объектами. Описывающие их классы как раз и составляют основную часть пакета java.io. Они довольно разнообразны и отвечают за различную функциональность. Все классы разделены на две части - одни осуществляют ввод данных, другие вывод.

Минимальной "порцией" информации является бит, принимающий значение 0 или 1. Традиционно используется более крупная единица измерения байт, объединяющая 8 бит. Таким образом, значение, представленное 1 байтом, находится в диапазоне от 0 до 28-1=255, или, если использовать знак, от -128 до +127. Примитивный тип **byte** в Java в точности соответствует последнему, знаковому диапазону.

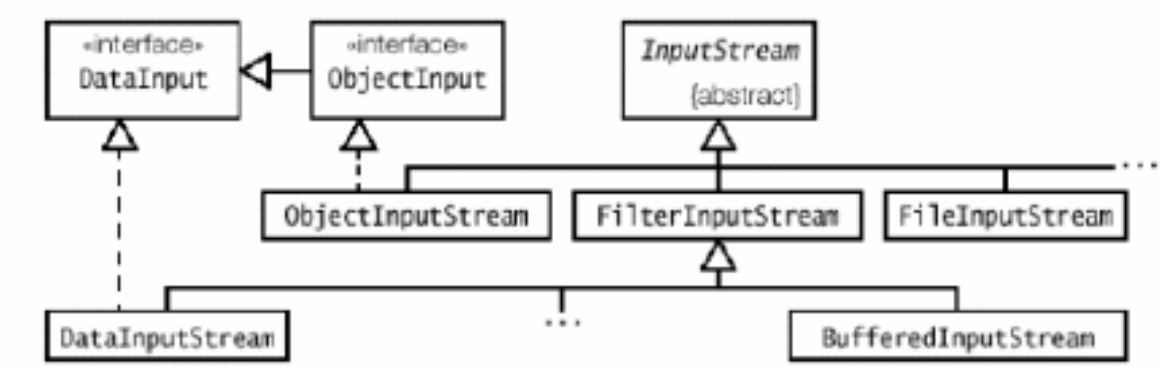
Базовые, наиболее универсальные классы позволяют считывать и записывать информацию именно в виде набора байт. Чтобы их было удобно применять в различных задачах, java.io содержит также классы, преобразующие любые данные в набор байт. Например, если нужно сохранить результаты вычислений - набор значений типа double - в файл, то их можно сначала легко превратить в набор байт, а затем эти байты записать в файл. Аналогичные действия совершаются и в ситуации, когда требуется сохранить объект (т.е. его состояние) - преобразование в набор байт и последующая их запись в файл. Понятно, что при восстановлении данных в обоих рассмотренных случаях проделываются обратные действия - сначала считывается последовательность байт, а затем она преобразовывается в нужный формат.

Байтовые потоки

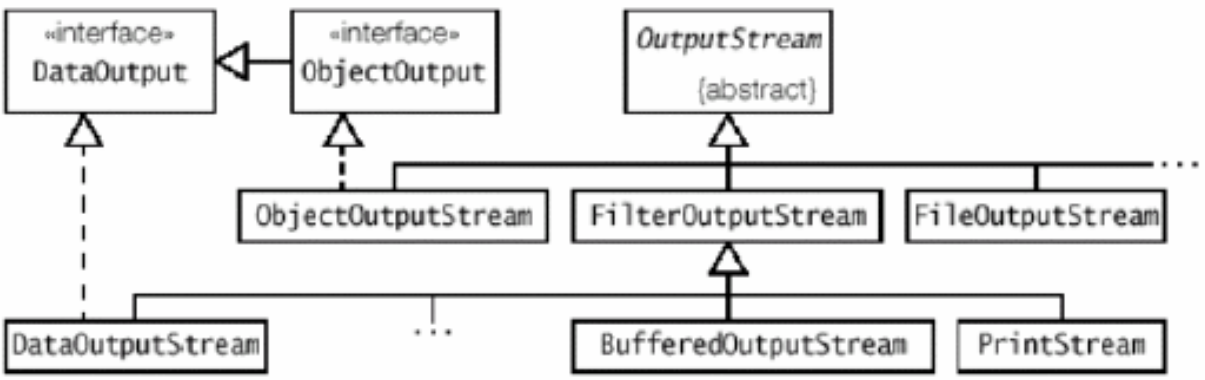
Байтовые потоки определяются в двух иерархиях классов. Наверху этой иерархии – два абстрактных класса: Inputstream и Outputstream (рисунок 1). Каждый из этих абстрактных классов имеет несколько конкретных

подклассов, которые обрабатывают различия между разными устройствами, такими как дисковые файлы, сетевые соединения и даже буферы памяти.

Абстрактные классы Inputstream и Outputstream определяют несколько ключевых методов, которые реализуются другими поточными классами. Два наиболее важных— ***read()*** и ***write()***, которые, соответственно, читают и записывают байты данных. Оба метода объявлены как абстрактные внутри классов inputstream и outputstream и переопределяются производными поточными классами.



а)



б)

Рисунок 1. Иерархия классов. а) – потоки чтения; б) – потоки записи

InputStream - это базовый класс для потоков ввода, т.е. чтения. Соответственно, он описывает базовые методы для работы с байтовыми потоками данных. Эти методы необходимы всем классам, наследующимся от InputStream:

– Метод int ***read()*** – абстрактный. Считывает один байт из потока. Если чтение успешно, то значение между 0..255 представляет собой полученный байт, если конец потока, то значение равно –1.

– Метод int ***read(byte[] b)*** возвращает количество считанных байт и сохраняет их в массив b. Метод так же абстрактный.
– Метод ***read(byte[] b, int offset, int length)*** считывает в массив ***b*** начиная с позиции ***offset length*** символов.
– Метод int ***available()*** возвращает количество байт, на данный момент готовых к чтению из потока. Этот метод применяют, что бы операции чтения не приводили к зависанию, когда данные не готовы к считыванию.
– Метод ***close()*** заканчивает работу с потоком.

В классе ***OutputStream***, аналогичным образом, определяются три метода ***write()*** – один принимающий в качестве параметра int, второй byte[], и третий byte[], плюс два int-числа. Все эти методы возвращают void:

– Метод void ***write(int i)*** пишет только старшие 8 бит, остальные игнорирует.
– Метод void ***write(byte[] b)*** записывает массив в поток.
– Метод void ***write(byte[] b, int offset, int length)*** записывает length-элементов из массива байт в поток, начиная с элемента offset.
– Метод ***flush()*** используется для очистки буфера и записи данных

Существует достаточно большое количество классов-наследников от InputStream, часть которых представлена в таблице

Поточный класс	Значение
BufferedInputstream	Буферизованный поток ввода
BufferedOutputStream	Буферизованный поток вывода
ByteArrayInputstream	Поток ввода, который читает из байт-массива
ByteArrayOutputStream	Поток вывода, который записывает в байт-массив
DataInputstream	Поток ввода, который содержит методы для чтения данных стандартных типов
DataOutputStream	Поток вывода, который содержит методы для записи данных стандартных типов
FileInputstream	Поток ввода, который читает из файла
FileOutputStream	Поток вывода, который записывает в файл
FilterInputStream	Реализует Inputstream
FilterOutputStream	Реализует OutputStream

Inputstream	Абстрактный класс, который описывает поточный ввод
OutputStream	Абстрактный класс, который описывает поточный вывод
PipedInputStream	Канал ввода
PipedOutputStream	Канал вывода
Printstream	Поток вывода, который поддерживает print() и println()
PushbackInputStream	Поток (ввода), который поддерживает однобайтовую операцию "unget", возвращающую байт в поток ввода
RandomAccessFile	Поддерживает ввод/вывод файла произвольного
SequenceInputStream	Поток ввода, который является комбинацией двух или нескольких потоков ввода, которые будут читаться последовательно, один за другим

ByteArrayInputStream и ByteArrayOutputStream

Самый естественный и простой источник, откуда можно считывать байты - это, конечно, массив байт. Класс ByteArrayInputStream представляет поток, считывающий данные из массива байт. Этот класс имеет конструктор, которому в качестве параметра передается массив byte[]. Соответственно, при вызове методов read(), возвращаемые данные будут браться именно из этого массива.

```
byte[] bytes ={1,-1,0};
ByteArrayInputStream in =new ByteArrayInputStream(bytes);
int readedInt =in.read();  //readedInt=1
readedInt =in.read();      //readedInt=255
readedInt =in.read();      //readedInt=0
```

Для записи байт в массив, используется класс ByteArrayOutputStream. Этот класс использует внутри себя объект byte[], куда записывает данные, передаваемые при вызове методов write(). Что бы получить записанные в массив данные, вызывается метод toByteArray().

```
ByteArrayOutputStream out =new ByteArrayOutputStream();
out.write(10);
out.write(11);
byte[] bytes =out.toByteArray();
```

FileInputStream и FileOutputStream

Классы FileInputStream используется для чтения данных из файла. Конструктор этого класса в качестве параметра принимает название файла, из которого будет производиться считывание.

Для записи байт в файл используется класс FileOutputStream. При создании объектов этого класса, то есть при вызовах его конструкторов кроме указания файла, так же можно указать, будут ли данные дописываться в конец файла либо файл будет перезаписан. При этом, если указанный файл не существует, то сразу после создания FileOutputStream он будет создан.

```
byte[] bytesToWrite ={1,2,3};          //что записываем
byte[] bytesReaded =new byte[10];  //куда считываем
String fileName ="d:\\test.txt";

try {
FileOutputStream outFile =new FileOutputStream(fileName);
outFile.write(bytesToWrite);            //запись в файл
outFile.close();

FileInputStream inFile =new FileInputStream(fileName);
int bytesAvailable =inFile.available();  //сколько можно считать
int count =inFile.read(bytesReaded,0,bytesAvailable);
inFile.close();}

catch (FileNotFoundException e){
System.out.println("Невозможно произвести запись в файл:"+fileName);}
catch (IOException e){
System.out.println("Ошибка ввода/вывода:"+e.toString());}
```

При работе с FileInputStream метод ***available***() практически наверняка вернет длину файла, то есть число байт, сколько вообще из него можно считать. Но не стоит закладываться на это при написании программ, которые должны устойчиво работать на различных платформах - метод available() возвращает число, сколько байт может быть на данный момент считано без блокирования.

Классы-фильтры

Задачи, возникающие при вводе/выводе крайне разнообразны - этот может быть считывание байтов из файлов, объектов из файлов, объектов из массивов, буферизованное считывание строк из массивов. В такой ситуации решение путем простого наследования приводит к возникновению слишком большого числа подклассов. Решение же, когда требуется совмещение нескольких свойств, высокоэффективно в виде надстроек. (В ООП этот паттерн называется адаптер.) ***Надстройки*** - наложение дополнительных объектов для получения новых свойств и функций. Таким образом, необходимо создать несколько дополнительных объектов - адаптеров к классам ввода/вывода. В java.io их еще называют ***фильтрами***. При этом надстройка-фильтр, включает в себя интерфейс объекта, на который надстраивается, и поэтому может быть в свою очередь дополнительно быть надстроена.

В java.io интерфейс для таких надстроек ввода/вывода предоставляют классы ***FilterInputStream*** (для входных потоков) и ***FilterOutputStream*** (для выходных потоков). Эти классы унаследованы от основных базовых классов ввода/вывода - InputStream и OutputStream соответственно. Конструкторы этих классов принимают в качестве параметра объект InputStream и имеют модификатор доступа protected. Сами же эти классы являются базовыми для надстроек. Поэтому только наследники могут вызывать его(при их создании), передавая переданный им поток. Таким образом обеспечивается некоторый общий интерфейс для надстраиваемых объектов.

Классом-фильтром являются классы ***BuffereInputStream*** и ***BufferedOutputStream***. На практике, при считывании с внешних устройств, ввод данных почти всегда необходимо буферизировать. ***BufferedInputStream*** - содержит в себе массив байт, который служит буфером для считываемых данных. То есть, когда байты из потока считываются (вызов метода read()) либо пропускаются (метод skip()), сначала перезаписывается этот буферный массив, при этом считываются сразу много байт за раз. Так же класс

BufferedInputStream добавляет поддержку методов ***mark()*** и ***reset()***. Эти методы определены еще в классе InputStream, но их реализация по умолчанию бросает исключение IOException. Метод mark() запоминает точку во входном потоке и метод reset() приводит к тому, что все байты, считанные после наиболее позднего вызова метода mark(), будут считаны заново, прежде чем новые байты будут считываться из содержащегося входного потока.

BufferedOutputStream - при использовании объекта этого класса, запись производится без необходимости обращения к устройству ввода/вывода при записи каждого байта. Сначала данные записываются во внутренний буфер. Непосредственное обращение к устройству вывода и, соответственно, запись в него произойдет, когда буфер будет полностью заполнен. Освобождение буфера с записью байт на устройство вывода можно обеспечить и непосредственно - вызовом метода flush(). Так же буфер будет освобожден непосредственно перед закрытием потока (вызов метода close()). При вызове этого метода также будет закрыт и поток, над которым буфер надстроен.

```
String fileName ="d:\\file1";
InputStream inStream =null;
OutputStream outStream =null; //Записать в файл некоторое количество байт

long timeStart =System.currentTimeMillis();

outStream =new FileOutputStream(fileName);
outStream =new BufferedOutputStream(outStream);

for(int i=1000000;--i>=0;){
outStream.write(i);}

inStream =new FileInputStream(fileName);
inStream =new BufferedInputStream(inStream);

while(inStream.read()!=-1) {…//чтение данных
}
```

Символьные потоки

Символьные потоки определены в двух иерархиях классов. Наверху этой иерархии два абстрактных класса: Reader и Writer. Они обрабатывают потоки символов Unicode. Абстрактные классы Reader и Writer определяют несколько ключевых методов, которые реализуются другими поточными классами. Два самых важных метода — ***read()*** и ***write()***, которые читают и записывают символы данных, соответственно. Они переопределяются производными поточными классами.

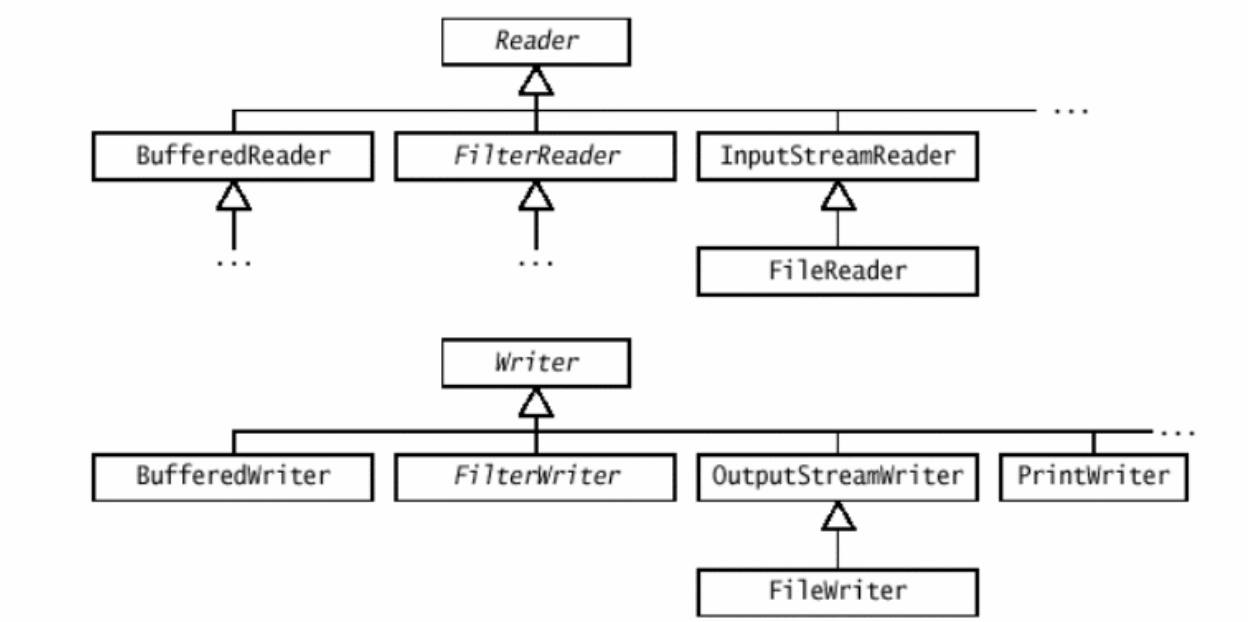


Рисунок 2. Иерархия символьных потоков

До сих пор речь шла только о считывании и записи в поток данных в виде ***byte***. Для работы с другими примитивными типами данных java, определены интерфейсы ***DataInput*** и ***DataOutput***, и существующие их реализации - классы-фильтры ***DataInputStream*** и ***DataOutputStream***.

Интерфейсы DataInput и DataOutput определяют, а классы DataInputStream и DataOutputStream, соответственно реализуют, методы считывания и записи всех примитивных типов данных. При этом происходит конвертация этих данных в ***byte*** и обратно. При этом в поток будут записаны байты, а ответственность за восстановление данных лежит только на разработчике - нужно считывать данные в виде тех же типов, в той же последовательности, как и производилась запись. То есть, можно, конечно,

записать несколько раз int или long, а потом считывать их как short или что-нибудь еще - считывание произойдет корректно и никаких предупреждений о возникшей ошибке не возникнет, но результат будет соответствующий - значения, которые никогда не записывались.

- Запись производится методом ***writeXxx()***, где Xxx – тип данных – int, long, double, byte.
- Для считывания необходимо соблюдать и применять те операторы ***readXxx()***, в котором были записаны типы данных с помощью writeXxx().

Интерфейсы DataInput и DataOutput представляют возможность записи/считывания данных ***примитивных*** типов Java. Для аналогичной работы с объектами определены унаследованные от них интерфейсы ObjectInput и ObjectOutput соответственно.

Основные символьные классы приведены в таблице.

Поточный класс	Значение
BufferedReader	Буферизированный символьный поток ввода
BufferedWriter	Буферизированный символьный поток вывода
CharArrayReader	Поток ввода, которой читает из символьного массива
CharArrayWrite	Выходной поток, который записывает в символьный массив
FileReader	Поток ввода, который читает из файла
FileWriter	Выходной поток, который записывает в файл
FilterReader	Отфильтрованный поток ввода
FilterWriter	Отфильтрованный поток вывода
InputstreamReader	Поток ввода, который переводит байты в символы
LineNumberReader	Поток ввода, который считает строки
OutputStreamWriter	Поток ввода, который переводит символы в байты
PipedReader	Канал ввода
PipedWriter	Канал вывода
PrintWriter	Поток вывода, который поддерживает print() и println()
PushbackReader	Поток ввода, возвращающий символы в поток ввода
Reader	Абстрактный класс, который описывает символьный поток ввода
StringReader	Поток ввода, который читает из строки
StringWriter	Поток вывода, который записывает в строку
Writer	Абстрактный класс, который описывает символьный поток вывода

Пример на запись и считывание символьного потока.

```
String fileName = "d:\\file.txt";
FileWriter fw = null;
BufferedWriter bw = null;
FileReader fr = null;
BufferedReader br = null;

//Строка, которая будет записана в файл
String data = "Some data to be written and readed \n";

try{
fw = new FileWriter(fileName);
bw = new BufferedWriter(fw);
System.out.println("Write some data to file: " + fileName);
// Несколько раз записать строку
for(int i=(int)(Math.random()*10);--i>=0;)bw.write(data);
bw.close();

fr = new FileReader(fileName);
br = new BufferedReader(fr);

String s = null;
int count = 0;
System.out.println("Read data from file: " + fileName);
// Считывать данные, отображая на экран
while((s=br.readLine())!=null)
System.out.println("row " + ++count + " read:" + s);
br.close();
}catch(Exception e){
e.printStackTrace(); }
```

Консольный ввод-вывод

Все программы Java автоматически импортируют пакет java.lang. Этот пакет определяет класс с именем ***System***, инкапсулирующий некоторые аспекты исполнительной среды Java. Класс System содержит также три предопределенные поточные переменные ***in***, ***out*** и ***err***. Эти поля объявлены в System со спецификаторами public и static.

Объект ***System.out*** называют потоком стандартного вывода. По умолчанию с ним связана консоль.

На объект ***System.in*** ссылаются как на стандартный ввод, который по умолчанию связан с клавиатурой.

К объекту ***System.err*** обращаются как к стандартному потоку ошибок, который по умолчанию также связан с консолью.

Пример считывания символов с консоли.

```
// Использует BufferedReader для чтения символов с  консоли,
import java.io.*;
class BRRead
{
public static void main(String args [ ])
throws IOException {
char c;
BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
System.out.println("Введите символы, 'q' - для завершения.");

// чтение символов
do {
c = (char) br.read();
System.out.println(c); }
while(c != 'q');
} }
```

Пример считывания строк с консоли.

```
import  java.io..*;
class TinyEdit {
public static void main(String args[];
throws IOException (

// Создать BufferedReader-объект,используя System.in
BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
String str[] = new String[100];
System.out.println("Введите строки текста.");
System.out.println("Введите 'stop' для завершения.");
for (int i=0; i<100; i++)
{
str[i] = br.readLine();
if(str[i].equals("stop"))
break; }
System.out.println("\n Вот ваш файл:");
// Вывести строки на экран.
for (int i=0; i<100; i++) {
```

```
if(str[i].equals("stop")) break;
System.out.println(str[i]); }
} }
```

Сериализация

В java имеется стандартный механизм превращения объекта в набор байт - ***сериализации***. Для того, что бы объект мог быть сериализован, он должен реализовать интерфейс ***java.io.Serializable*** (соответствующее объявление должно явно присутствовать в классе объекта или, по правилам наследования, неявно в родительском классе вверх по иерархии). Интерфейс java.io.Serializable не определяет никаких методов. Его присутствие только определяет, что объекты этого класса разрешено сериализовывать.

После того, как объект был сериализован, то есть превращен в последовательность байт, его по этой последовательности можно восстановить, при этом восстановление можно проводить на любой машине (вне зависимости от того, где проводилась сериализация), то есть последовательность байт можно передать на другую машину по сети или любым другим образом, и там провести десериализацию. При этом не имеет значения операционная система под которой запущена Java - например, можно создать объект на машине с ОС Windows, превратить его в последовательность байт, после чего передать их по сети на машину с ОС Unix, где восстановить тот же объект.

Для работы по сериализации в java.io определены интерфейсы ***ObjectInput***, ***ObjectOutput*** и реализующие их классы ***ObjectInputStream*** и ***ObjectOutputStream*** соответственно. Для сериализации объекта нужен выходной поток OutputStream, который следует передать при конструировании ObjectOutputStream. После чего вызовом метода writeObject() сериализовать объект и записать его в выходной поток. Например:

```
// сериализация объекта Integer(1)
ByteArrayOutputStream os =new ByteArrayOutputStream();
```

```
Object objSave =new Integer(1);
ObjectOutputStream oos =new ObjectOutputStream(os);
oos.writeObject(objSave);
```

Что бы просмотреть, во что превратился объект objSave, можно просмотреть содержимое массива

byte[] bArray = os.toByteArray();

А чтобы получить этот объект, можно десериализовать его из этого массива:

```
byte[]  bArray =os.toByteArray(); //получение содержимого массива
ByteArrayInputStream is =new ByteArrayInputStream(bArray);
ObjectInputStream ois =new ObjectInputStream(is);
Object objRead =ois.readObject();
```

Сериализация объекта заключается в сохранении и восстановлении состояния объекта. Состояние описывается значением полей. Причем не только описанных в классе, но и унаследованных. При попытке самостоятельно написать механизм восстановления возникли бы следующие проблемы:

- Как установить значения полей, тип которых private
- Объект создается с помощью вызова конструктора. Как восстановить ссылку в этом случае
- Даже если существуют set-методы для полей, как выбрать значения и параметры.

Сериализуемый объект может хранить ссылки на другие объекты, которые в свою очередь так же могут хранить ссылки на другие объекты. И все они тоже должны быть восстановлены при десериализации. При этом, важно, что если несколько ссылок указывают на один и тот же объект, то в восстановленных объектах эти ссылки так же указывали на один и тот же объект. Если класс содержит в качестве полей другие объекты, то эти объекты так же будут сериализовываться и поэтому тоже должны быть сериализуемы. В свою очередь, сериализуемы должны быть и все объекты, содержащиеся в этих сериализуемых объектах и т.д. Полный путь ссылок объекта по всем объектным ссылкам, имеющимся у него и у всех объектов на

которые у него имеются ссылки, и т.д. - называется графом исходного объекта.

Однако, вопрос, на который следует обратить внимание - что происходит с состоянием объекта, унаследованным от суперкласса. Ведь состояние объекта определяется не только значениями полей, определенными в нем самом, но так же и таковыми, унаследованными от супепркласса. Сериализуемый подтип берет на себя такую ответственность, но только в том случае, если у суперкласса определен конструктор по умолчанию, объявленный с модификатором доступа таким, что будет доступен для вызова из рассматриваемого наследника. Этот конструктор будет вызван при десериализации. В противном случае, во время выполнения будет брошено исключение java.io.InvalidClassException.

В процессе десериализации, поля НЕ сериализуемых классов (родительских классов, НЕ реализующих интерфейс Serializable) инициируются вызовом конструктора без параметров. Такой конструктор должен быть доступен из сериализуемого их подкласса. Поля сериализуемого класса будут восстановлены из потока.

Пример сериализации и десериализации объекта.

```
public class Parent{
  public String firstname, lastname;
    public parent(){ firstname='old_first';   lastname='old_last';}
}
public class Child extends Parent implements Serializable{
    private int age;
    public Child (int age){ this.age=age;}
}….
FileOutputStream fos=new FileOutputStream("output.bin");
ObjectOutputStream oos=new ObjectOutputStream(fos);
Child c= new Child(2);
oos.writeObject(c); oos.close();

FileInputStream fis=new FileInputStream("output.bin");
ObjectInputStream ois=new ObjectInputStream(fis);
Ois.readObject();
ois.close();
```

4. Порядок выполнения работы

- изучить предлагаемый теоретический материал;
- реализуйте в виде программ на языке Java следующие задачи:

1. осуществить запись в файл большого произвольного массива данных, используя бинарные потоки, двумя способами: запись **без** и запись **с** использованием классов-фильтров, позволяющих выполнить буферизацию. Произвести замеры времени выполнения записи в файл и сравнить полученное ускорение. Указание: для отсчета времени в миллисекундах необходимо использовать системную функцию следующим образом: long timeStart =System.currentTimeMillis(). То же самое проделать для операции считывания данных из файла.
2. Расширить программу (апплет, на котором размещены кнопки для создания экземпляров Rectangle, ColoredRect, DrawableRect и реализована функция по перетаскиванию объектов мышью), созданную на прошлом лабораторном занятии, следующим образом. Добавить на апплет 2 кнопки – LoadFromFile и SaveToFile, одна из которых сериализует созданные объекты прямоугольников и сохраняет их в файл, другая – считывает из файла и продолжает работу программы с того момента, когда была осуществлена сериализация. Если программа была закрыта сразу после сериализации, то при повторном запуске программы и десериализации объектов, объекты должны располагаться на том же месте (обладать тем же состоянием), что и до закрытия программы. Программа должна позволять неограниченное число раз считывать объекты из файла, не удаляя при этом уже существующие.

5. Содержание отчета

В отчете следует указать:

1. цель работы;
2. введение;
3. программно-аппаратные средства, используемые при выполнении работы;
4. основную часть (описание самой работы), выполненную согласно требованиям к результатам выполнения лабораторного практикума;
5. заключение (описание результатов и выводы);
6. список используемой литературы.

6. Литература

1. Вязовик, Н.А. Программирование на Java. [электронный ресурс] http://www.intuit.ru/department/pl/javapl (8.11.2009).
2. Хорстманн К.С., Корнелл Г. Библиотека профессионала. JAVA 2. Том 1. Основы. 8-е издание. Пер. с англ. – М.: ООО Издательский дом “Вильямс”, 2008. – 816 с.
3. Эккель Б. Философия JAVA. – СПб.: Питер, 2003. – 971с.