**ЛАБОРА ТОРНАЯ РАБОТА №2**

**РАБОТА С ФАЙЛАМИ. ПОСТРОЕНИЕ ПЛАНА ТЕСТИРОВАНИЯ**

**Цель работы:** освоить создание консольных приложений; освоить работу с файлами; научиться строить план тестирования приложения.

**Задача 1.**

Во входном текстовом файле хранится числовой массив объёмом до 1 млн значений. Сами значения являются целыми положительными и не превосходят 1000. Возможны повторяющиеся значения. В выходной текстовый файл нужно вывести эти числа в порядке возрастания.

Структура входного файла:

первая строка в файле показывает длину числового массива (от 1 до 1000000).

Далее идет указанное количество числовых значений от 1 до 1000 каждое по одному числу в строке.

Решение должно быть выполнено в виде консольного приложения, результаты выведены в текстовый файл.

Консольное приложение не имеет окна и работает только с файлами. При необходимости диалога с пользователем используются стандартные файлы ввода-вывода. В нашей задаче для ввода и вывода будут использованы только внешние файлы.

**Задача 2.**

Модифицируйте программу, чтобы она выводила данные в порядке убывания.

**Задача 3.**

Модифицируйте программу, чтобы она могла по требованию пользователя выводить данные либо по возрастанию, либо по убыванию. Информация о требуемом виде сортировки должна содержаться в том же входном файле.

**Задача 4.**

Разработайте программу проверки правильности сортировки

**Классы** *FileInputStream* **и** *FileOutputStream*

Класс *FilelnputStream* используется для чтения данных из файла. Конструктор такого класса в качестве параметра принимает название файла, из которого будет производиться считывание. При указании строки имени файла нужно учитывать, что она будет напрямую передана операционной системе, поэтому формат имени файла и пути к нему может различаться на разных платформах. Если при вызове этого конструктора передать строку, указывающую на несуществующий файл или каталог, то отработает *java.io.FileNotFoundException.* Если же объект успешно создан, то при вызове его методом *read()* возвращаемые значения будут считываться из указанного файла.

Для записи байт в файл используется класс *FileOutputStream.* При создании объектов этого класса, то есть при вызовах его конструкторов, кроме имени файла также можно указать, будут ли данные дописываться в конец файла либо файл будет перезаписан. Если указанный файл не существует, то сразу после создания *FileOutputStream* он будет создан. При вызовах методов *write()* передаваемые значения будут записываться в этот файл. По окончании работы необходимо вызвать метод *close(),* чтобы сообщить системе, что работа по записи файла закончена.

Пример 2.1

В примере реализован класс, который записывает информацию в файл *d\\test.txt.*

import java, io.\*;

class File Writer {

public static void main(String args[]){

byte[] bytesToWrite = {1, 2, 3};

byte[] bytesReaded = new byte[10];

String fileName = "d:\\test.txt";

try {

// создать выходной поток

FileOutputStream outFile = new FileOutputStream(fileName);

System.out.printlnf'File is open to write");

// записать массив

out File.write(bytesToWrite);

System.out.println("Written: " + bytesToWrite.length + " byte’’);

// по окончании использования должен быть закрыт outFile.close();

System.out.println("Output stream is closed’’);

// создать входной поток

FilelnputStream inFile = new FilelnputStream(fileName);

System.out.printIn("File is open to read");

// узнать, сколько байт готово к считыванию

int bytesAvailable = inFile.available();

System.out.println("Ready to read: " + bytesAvailable + " byte’’);

// считать в массив

int count = inFile.read(bytesReaded.().bytesAvailable);

System.out.println("Read: " + count + " byte");

for (int i=0;i<count;i++)

System.out.print(bytesReaded[i]+",");

System.out.println();

inFile.close();

System.out.println("Input stream is closed");

} catch (FileNotFoundException e) {

System.out.println("Its impossible to write to file: ’’ + fileName);

} catch (IOException e) {

System.out.println("lnput/output error: " + e.toString());

}

}

}

Результат выполнения программы:

File is open to write

Written 3 byte

Output stream is closed

File is open to read

Ready to read 3 byte

Read 3 byte

1,2,3,

Input stream is closed

При работе с *FilelnputStream* метод *available()* практически наверняка вернет длину файла, то есть число байт, сколько вообще из него можно считать. Но нежелательно использовать его при написании программ, которые должны устойчиво работать на различных платформах: метод *available()* возвращает число байт, которое может быть на данный момент считано без блокирования. Тот факт, что, скорее всего, это число и будет длиной файла, является лишь частным случаем работы на некоторых платформах.

В приведенном примере для наглядности закрытие потоков производилось сразу же после окончания их использования в основном блоке. Однако лучше закрывать потоки в *finally-блоке.*

Пример 2.2

…

} finally {

try{ inFile.close();} catch(IOException e) {};

}

Такой подход гарантирует, что поток будет закрыт и будут освобождены все связанные с ним системные ресурсы.

*BufferedlnputStream* и *BufferedOutputStream*

На практике при считывании с внешних устройств ввод данных почти всегда необходимо буферизировать. Для буферизации данных служат классы *BufferedlnputStream* и *BufferedOutputStream.*

*BufferedlnputStream* содержит массив байт, который служит буфером для считываемых данных. То есть когда байты из потока считываются либо пропускаются (метод *skip()),* сначала заполняется буферный массив, причем из надстраиваемого потока загружается сразу много байт, чтобы не требовалось обращаться к нему при каждой операции *read* или *skip.* Также класс *BufferedlnputStream* добавляет поддержку методов *mark()* и *reset().* Эти методы определены еще в классе *InputStream,* но там их реализация по умолчанию бросает исключение *IOЕхсерtiоп.* Метод *mark()* запоминает точку во входном потоке, а вызов метода *reset()* приводит к тому, что все байты, полученные после последнего вызова *mark(),* будут считываться повторно, прежде чем новые байты начнут поступать из надстроенного входного потока.

*BufferedOutputStream* предоставляет возможность производить многократную запись небольших блоков данных без обращения к устройству вывода при записи каждого из них. Сначала данные записываются во внутренний буфер. Непосредственное обращение к устройству вывода и

соответственно запись в него произойдет, когда буфер заполнится. Инициировать передачу содержимого буфера на устройство вывода можно и явным образом, вызвав метод *flush().* Также буфер освобождается перед закрытием потока. При этом будет закрыт и надстраиваемый поток (так же поступает *BufferedlnputStream).*

Следующий пример наглядно демонстрирует повышение скорости считывания данных из файла с использованием буфера.

Пример 2.3

import java.util.\*;

import java.io.\*;

class DemoBufferedIOStream

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

try {

String fileName = "d:\\filel";

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);

}

long time = System.currentTimeMillis() - timeStart;

System.out.println("Writing time: " + time + " millisec");

outStream.close();

// Определить время считывания без буферизации timeStart = System.currentTimeMillis();

inStream = new FileInputStream(fileName);

while(inStream.read()!=-l){

}

time = System.currentTimeMillis() - timeStart;

inStream.close();

System.out.println( "Direct read time: " + (time) + " millisec"):

// Теперь применим буферизацию

timeStart = System.currentTimeMillis();

inStream = new FilelnputStream( fileName);

inStream = new BufferedInputStream( inStream);

while(inStream.read() !=-1){

}

time = System.currentTime.Millis() - timeStart;

inStream.close();

System.out.println("Buffered read time: " + (time) + " millisec");

}

catch (lOException e) {

System.out.println("IOException: " + e.toString());

e.printStackTrace();

}

}

}

Результат выполнения программы:

Writing time: 359 millisec

Direct read time: 6546 millisec

Buffered read time: 250 millisec

В данном случае не производилось никаких дополнительных вычислений, занимающих процессорное время, только запись и считывание из файла. При этом считывание с использованием буфера заняло в 10 (!) раз меньше времени, чем аналогичное без буферизации. Для более быстрого выполнения программы запись в файл производилась с буферизацией, однако ее влияние на скорость записи нетрудно проверить, убрав из программы строку, создающую *BufferedOutputStream..*

***DataInputStream и DataOutputStream***

Для работы с примитивными типами данных *Java* определены интерфейсы *Datalnput* и *DataOutput* и их реализации - классы-фильтры *DatalnputStream* и *DataOutputStream.*

Интерфейсы *Datalnput* и *DataOutput* определяют, а классы *DatalnputStream* и *DataOutputStream* соответственно реализуют методы считывания и записи значений всех примитивных типов. При этом происходит конвертация этих данных в набор *byte* и обратно. Чтение необходимо организовать так, чтобы данные запрашивались в виде тех же типов, в той же последовательности, как и производилась запись. Если записать, например, *int* и *long,* а потом считывать их как *short,* чтение будет выполнено корректно, без исключительных ситуаций, но числа будут получены совсем другие.

Пример 2.4

try {

ByteArrayOutputStream out = new ByteArrayOutputStream();

DataOutputStream outData = new DataOutputStream(out);

outData.writeByte( 128);

// этот метод принимает аргумент int, но записывает

// лишь младший байт

outData.writelnt(128);

outData.writeLong(128);

outData.writeDouble(128);

outData.close();

byte[] bytes = out.toByteArray();

InputStream in = new ByteArrayInputStream(bytes);

DatalnputStream inData = new DataInputStream(in);

System.out.println("Correct sequence reading:");

System.out.println("readByte: " + inData.readByte());

System.out.println("readlnt:" + inData.readInt());

System.out.println("readLong: " + inData.readLong());

System.out.println("readDouble: " + inData.readDouble());

inData.close();

System.out.println("Inverted sequence reading:");

in = new ByteArrayInputStream(bytes);

inData = new DatalnputStream(in);

System.out.println("readlnt: " + inData.readInt());

System.out.println("readDouble: " + inData.readDouble());

System.out.println("readLong: " + inData.readLong());

inData.close();

} catch (Exception e) {

System.out.println("Impossible IOException occurs: " +

e.toString());

e.printStackTrace();

}

Результат выполнения программы:

Correct sequence reading:

readByte: -128

readlnt: 128

readLong: 128

readDouble: 128.0

Inverted sequence reading:

read Int: -2147483648

readDouble: -0.0

readLong: -9205252085229027328

***Класс File***

Если классы потоков осуществляют реальную запись и чтение данных, то класс *File -* это вспомогательный инструмент, призванный обеспечить работу с файлами и каталогами.

Объект класса *File* является абстрактным представлением файла и пути к нему. Он устанавливает только соответствие с ним, при этом для создания объекта неважно, существует ли такой файл на диске. После создания можно выполнить проверку, вызвав метод *exists,* который возвращает значение *true,* если файл существует. Создание или удаление объекта класса *File* никоим образом не отображается на реальных файлах. Для работы с содержимым файла можно получить экземпляры *FileI/OStream.*

Объект *File* может указывать на каталог (узнать это можно путем вызова метода *isDirectory()).* Метод *list* возвращает список имен (массив *String)* содержащихся в нем файлов (если объект *File* не указывает на каталог, будет возвращен *null).* Следующий пример демонстрирует использование объектов класса *File..*

Пример 2.5

import java.io.\*;

public class FileDemo {

public static void findFiles(File file, FileFilter filter,

PrintStream output) throws IOException{

if (file.isDirectory()) {

File[] list = file.listFiles();

for (int i=list.length; --i>=0;) {

findFiles(list[i], filter, output);

}

} else {

if (filter. accept(file))

output.println("\t" + file.getCanonicalPath());

}

}

public static void main(String[] args) {

class NameFilter implements FileFilter {

private String mask;

NameFilter(String mask) {

this.mask = mask;

}

public boolean accept(File file){

return (file.getName().indexOf(mask)!=-l)?true:false;

}

}

File pathFile = new File(".");

String filterString = ".java";

try {

FileFilter filter = new NameFilter(filterString);

findFiles(pathFile, filter, System.out);

} catch(Exception e) {

e.printStackTrace();

}

System.out.println("work finished");

}

}

При выполнении этой программы на экран будут выведены названия (в каноническом виде) всех файлов с расширением .java, содержащихся в текущем каталоге и всех его подкаталогах.

Для определения того, что файл имеет расширение .*java,* использовался интерфейс *FileFilter* с реализацией в виде внутреннего класса *NameFilter.* Интерфейс *FileFilter* определяет только один метод *accept,* возвращающий значение, определяющее, попадает ли переданный файл в условия фильтрации. Помимо этого интерфейса, существует еще одна разновидность интерфейса фильтра - *FilenameFilter,* где метод *accept* определен несколько иначе: он принимает не объект файла к проверке, а объект *File,* указывающий на каталог, где находится файл для проверки, и строку его названия. Для проверки совпадения с учетом регулярных выражений нужно соответствующим образом реализовать метод *accept.* В конкретном приведенном примере можно было обойтись и без использования интерфейсов *FileFilter* или *FilenameFilter.* На практике их можно использовать для вызова методов *list* объектов *File* - в этих случаях будут возвращены файлы с учетом фильтра.

Также класс *File* предоставляет возможность получения некоторой информации о файле:

* методы *canRead* и *canWrite -* возвращается *boolean*-значение*,* можно ли будет приложению производить чтение и изменение содержимого из файла соответственно;
* *getName* - возвращает строку - имя файла (или каталога);
* *getParent, getParentName —* возвращают каталог, где файл находится в виде строки названия и объекта *File* соответственно;
* *getPath -* возвращает путь к файлу (при этом в строку преобразуется абстрактный путь, на который указывает объект *File);*
* *isAbsolutely* - возвращает *boolean* значение, является ли абсолютным путь, которым указан файл. Определение, является ли путь абсолютным, зависит от системы, где запущена Java-машина. Гак, для *Windows* абсолютный путь начинается с указания диска либо символом \. Для *Unix* абсолютный путь начинается символом /;
* *isDirectory, isFile -* возвращает *boolean* значение, указывает ли объект на каталог либо файл соответственно;
* *isHidden -* возвращает *boolean* значение, указывает ли объект на скрытый файл;
* *lastModified -* дата последнего изменения;
* *length* - длина файла в байтах.

Также можно изменить некоторые свойства файла - методы *setReadOnly, setLastModified.* назначение которых очевидно из названия. Если нужно создать файл на диске, это позволяют сделать методы *createNewFile, mkDir, mkDirs.* Соответственно *createNewFile* создает пустой файл (если таковой еще не существует), *mkDir* создает каталог, если для него все родительские уже существуют, a *mkDirs* создает каталог вместе со всеми необходимыми родительскими.

Файл можно и удалить - для этого предназначены методы *delete* и *deleteOnExit.* При вызове метода *delete* файл будет удален сразу же, а при вызове *deleteOnExit* по окончании работы Java-машины (только при корректном завершении работы) отменить запрос уже невозможно.

Таким образом, класс *File* дает возможность достаточно полного управления файловой системой.

***Класс Random Access Fite***

Этот класс реализует сразу два интерфейса - *Datalnput* и *DataOutput -* следовательно, может производить запись и чтение всех примитивных типов *Java.* Эти операции, как следует из названия, производятся с файлом. При этом их можно производить поочередно, произвольным образом перемещаясь по файлу с помощью вызова метода *seek(long)* (переводит на указанную позицию в

файле). Узнать текущее положение указателя в файле можно вызовом метода *getFilePointer.*

При создании объекта этого класса конструктору в качестве параметров нужно передать два параметра: файл и режим работы. Файл, с которым будет проводиться работа, указывается либо с помощью *String -* название файла, либо объектом *File,* ему соответствующим. Режим работы *(mode) -* представляет собой строку либо «r» (только чтение), либо «rw» (чтение и запись). Попытка открыть несуществующий файл только на чтение приведет к исключению *FileNotFoundException.* При открытии на чтение и запись он будет незамедлительно создан (или же будет брошено исключение *FileNotFoundException,* если это невозможно осуществить).

После создания объекта *RandomAccessFile* можно воспользоваться методами интерфейсов *Datalnput* и *DataOutput* для проведения с файлом операций считывания и записи. По окончании работы с файлом его следует закрыть, вызвав метод *close.*

В следующем примере создается объект типа *String,* инициализируется текстовой строкой, определяется количество гласных, пробелов и общее количество букв. Реализован также механизм записи информации в файл и ее считывания из файла.

Пример 2.6

import java.io.\*;

public class Lab1

{

private String data;

private String filename;

private String choice;

private RandomAccessFile fio;

private BufferedReader in=

new BufferedReader(new InputStreamReader(System.in));

public void runConsol(){

while(true){

try

System.out.println("\nEnter your choice:");

System.out.println('"1.Read text from file"):

System.out.println("2.Type text");

System.out.println("3.Exit");

//чтение выбора пользователя

choice=in.readLine();

if (choice.compareTo("1")==0){

System.out.println("Type your filename");

filename=in.readLine();

fio = new RandomAccessFile(new File(filename), "r");

data=fio.readLine();

fio.close();

System.out.println("\nFile Input:\n"+data);

int spaces=0, glas=0, lett=0;

char ch;

for(int i=0;i<data.length();i++)

{

ch=Character.toLowerCase(data.charAt(i));

if(Character.isWhitespace(ch))

spaces++;

if((ch=='a')||(ch='e')||(ch=='i')||(ch=='o')||(ch='u')||(ch=='y'))

glas++;

lett++;

}

System.out.println("\nspaces - "+spaces+"\nvowels - "+glas+"\nletters -

"+(lett-spaces));

}

else if (choice.compareTo("2")==0){

System.out.println("Type your text");

data=in.readLine();

System.out.println("Type your filename");

filename=in.readLine();

fio = new RandomAccessFile(new File(filename), "rw");

fio.writeBytes(data);

fio.close();

System.out.println("Your text was saved");

}

else if (choice.compareTo("3")==0){

return;

}

}

catch(FileNotFoundException e){

System.out.println("File not found");

}

catch(IOException e){

System.out.println("Error1");

}

catch( Exception e){

System.out.println("Error2");

}

}

}

}

Результат выполнения программы:

Enter your choice:

1. Read text from file
2. Type test
3. Exit

Выбрав 2-й вариант, введем строку и имя файла, где необходимо сохранить строку. После этого будет предложено снова осуществить выбор.

Теперь выберем 1-й вариант. Введем имя нашего файла. В результате выведется исходная строка и итоги ее анализа: количество пробелов, гласных и общее количество букв. И снова будет предложен выбор.

Теперь выберем 3-й вариант. После этого программа завершит свою работу.