ЛАБОРАТОРНАЯ РАБОТА №5

**ПОТОКИ ВВОДА/ВЫВОДА. ПРЕОБРАЗОВАНИЕ ТИПОВ**

Цель: изучить потоки ввода/вывода и работу с ними в Java.

## Понятие потока данных

Часть вычислительной платформы, которая отвечает за обмен данными, – система ввода/вывода. В Java она представлена библиотекой java.io (input/output). Реализация системы ввода/вывода осложняется не только различными типами данных, но еще и необходимостью передачи информации на различные устройства. Информацией можно обмениваться в двоичном представлении, символьном или текстовом, с применением некоторой кодировки, или передавать числа в различных представлениях. Доступ к данным может потребоваться как последовательный (например, считывание HTML-страницы), так и произвольный (работа с несколькими частями одного файла). Зачастую для повышения производительности применяется буферизация.

В Java для описания работы по вводу/выводу используется специальное понятие поток данных (stream). Поток данных связан с некоторым источником или приемником данных, способным получать или предоставлять информацию. Соответственно, потоки делятся на входящие – читающие данные и выходящие – передающие данные. Введение концепции stream позволяет отделить основную логику программы, обменивающейся информацией с любыми устройствами одинаковым образом, от низкоуровневых операций с такими устройствами ввода/вывода.

Три потока определены в классе System статическими полями in, out и err. Их можно использовать без всяких дополнительных определений. Они называются соответственно стандартным вводом (stdin), стандартным выводом (stdout) и стандартным выводом сообщений (stderr). Эти стандартные потоки могут быть соединены с разными конкретными устройствами ввода и вывода.

Потоки out и err – это экземпляры класса PrintStream, организующего выходной поток байтов. Эти экземпляры выводят информацию на консоль методами print(), println() и write(), которых в классе PrintStream имеется около двадцати для разных типов аргументов.

Поток err предназначен для вывода системных сообщений программы: трассировки, сообщений об ошибках или, просто, о выполнении каких-то этапов программы. Такие сведения обычно заносятся в специальные журналы, log-файлы, а не выводятся на консоль. В Java есть средства переназначения потока, например, с консоли в файл.

Поток in – это экземпляр класса InputStream. Он назначен на клавиатурный ввод с консоли методами read(). Класс InputStream абстрактный, поэтому реально используется какой-то из его подклассов.

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.

Существующие стандартные классы помогают решить большинство типичных задач. 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 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 преобразуется в нужный формат.

## Стандартные потоки

Три потока определены в классе System статическими полями in, out и err. Их можно использовать без всяких дополнительных определений. Они называются соответственно стандартным вводом (stdin), стандартным выводом (stdout) и стандартным выводом сообщений (stderr). Эти стандартные потоки могут быть соединены с разными конкретными устройствами ввода и вывода.

Потоки out и err – это экземпляры класса PrintStream, организующего выходной поток байтов. Эти экземпляры выводят информацию на консоль методами print(), println() и write(), которых в классе PrintStream имеется около двадцати для разных типов аргументов.

Поток err предназначен для вывода системных сообщений программы: трассировки, сообщений об ошибках или, просто, о выполнении каких-то этапов программы. Такие сведения обычно заносятся в специальные журналы, log-файлы, а не выводятся на консоль. В Java есть средства переназначения потока, например, с консоли в файл.

Поток in – это экземпляр класса InputStream. Он назначен на клавиатурный ввод с консоли методами read(). Класс InputStream абстрактный, поэтому реально используется какой-то из его подклассов.

## Классы байтовых потоков

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, которые наследуются от InputStream.

Простейшая операция представлена методом read() (без аргументов). Он является абстрактным и, соответственно, должен быть определен в классах-наследниках. Этот метод предназначен для считывания ровно одного байта из потока, однако возвращает при этом значение типа int. В том случае, если считывание произошло успешно, возвращаемое значение лежит в диапазоне от 0 до 255 и представляет собой полученный байт (значение int содержит 4 байта и получается простым дополнением нулями в двоичном представлении).

Если достигнут конец потока, то есть, в нем больше нет информации для чтения, то возвращаемое значение равно -1.

Если же считать из потока данные не удается из-за каких-то ошибок, или сбоев, будет брошено исключение java.io.IOException. Этот класс наследуется от Exception, т.е. его всегда необходимо обрабатывать явно. Дело в том, что каналы передачи информации, будь то Internet или, например, жесткий диск, могут давать сбои независимо от того, насколько хорошо написана программа. А это означает, что нужно быть готовым к ним, чтобы пользователь не потерял нужные данные.

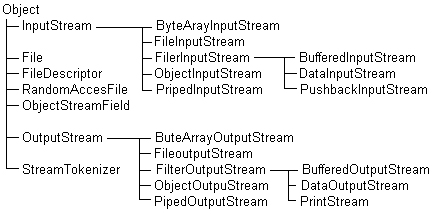
Иерархия классов байтовых потоков представлена на рисунке 6.1.

Рис. 6.1 Иерархия классов байтовых потоков

Метод read() – это абстрактный метод, но именно с соблюдением всех указанных условий он должен быть реализован в классах-наследниках.

На практике обычно приходится считывать не один, а сразу несколько байт – то есть массив байт. Для этого используется метод read(byte[]) (в качестве параметров передается массив byte[]). При выполнении этого метода в цикле производится вызов абстрактного метода read() (определенного без параметров) и результатами заполняется переданный массив. Количество байт, считываемое таким образом, равно длине переданного массива. Но при этом может так получиться, что данные впотоке закончатся еще до того, как будет заполнен весь массив. То есть возможна ситуация, когда впотоке данных (байт) содержится меньше, чем длина массива. Поэтому метод возвращает значение int, указывающее, сколько байт было реально считано. Понятно, что это значение может быть от 0 до величины длины переданного массива.

Если же мы изначально хотим заполнить не весь массив, а только его часть, то для этих целей используется метод read(byte[], int, int), которому, кроме массива byte[], передаются еще два int значения. Первое – это позиция в массиве, с которой следует начать заполнение, второе – количество байт, которое нужно считать. Такой подход, когда для получения данных передается массив и два int числа – offset (смещение) и length (длина), является довольно распространенным и часто встречается не только в пакете java.io.

При вызове методов read() возможно возникновение такой ситуации, когда запрашиваемые данные еще не готовы к считыванию. Например, если мы считываем данные, поступающие из сети, и они еще просто не пришли. В таком случае нельзя сказать, что данных больше нет, но и считать тоже нечего – выполнение останавливается на вызове метода read() и получается "зависание".

Чтобы узнать, сколько байт впотоке готово к считыванию, применяется метод available(). Этот метод возвращает значение типа int, которое показывает, сколько байт впотоке готово к считыванию. При этом не стоит путать количество байт, готовых к считыванию, с тем количеством байт, которые вообще можно будет считать из этогопотока. Метод available() возвращает число – количество байт, именно на данный момент готовых к считыванию.

Когда работа с входнымпотоком данных окончена, его следует закрыть. Для этого вызывается метод close(). Этим вызовом будут освобождены все системные ресурсы, связанные спотоком.

Точно так же, как InputStream – это базовый класс для потоков ввода, класс OutputStream – это базовый класс дляпотоков вывода.

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 ничего не возвращают (void).

Метод write(int) является абстрактным и должен быть реализован в классах-наследниках. Этот метод принимает в качестве параметра int, но реально записывает в поток только byte – младшие 8 бит в двоичном представлении. Остальные 24 бита будут проигнорированы. В случае возникновения ошибки этот метод бросает java.io.IOException, как, впрочем, и большинство методов, связанных с вводом-выводом.

Для записи впоток сразу некоторого количества байт методу write() передается массив байт. Или, если мы хотим записать только часть массива, то передаем массив byte[] и два int-числа – отступ и количество байт для записи. Понятно, что если указать неверные параметры – например, отрицательный отступ, отрицательное количество байт для записи, либо если сумма отступ плюс длина будет больше длины массива, – во всех этих случаях кидается исключение IndexOutOfBoundsException.

Реализация потока может быть такой, что данные записываются не сразу, а хранятся некоторое время в памяти. Например, мы хотим записать в файл какие-то данные, которые получаем порциями по 10 байт, и так 200 раз подряд. В таком случае вместо 200 обращений к файлу удобней будет скопить все эти данные в памяти, а потом одним заходом записать все 2000 байт. То есть класс выходного потока может использовать некоторый внутренний механизм для буферизации (временного хранения перед отправкой) данных. Чтобы убедиться, что данные записаны впоток, а не хранятся в буфере, вызывается метод flush(), определенный в OutputStream. В этом классе его реализация пустая, но если какой-либо из наследников использует буферизацию данных, то этот метод должен быть в нем переопределен.

Когда работа с потоком закончена, его следует закрыть. Для этого вызывается метод close(). Этот метод сначала освобождает буфер (вызовом метода flush()), после чего поток закрывается и освобождаются все связанные с ним системные ресурсы. Закрытый поток не может выполнять операции вывода и не может быть открыт заново. В классе OutputStream реализация метода close() не производит никаких действий.

Итак, классы InputStream и OutputStream определяют необходимые методы для работы с байтовыми потоками данных. Эти классы являются абстрактными. Их задача – определить общий интерфейс для классов, которые получают данные из различных источников. Такими источниками могут быть, например, массив байт, файл, строка и т.д. Наиболее распространенные будут рассмотрены далее.

## Классы FileInputStream и FileOutputStream

Класс FileInputStream 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 будет брошено java.io.FileNotFoundException. Если же объект успешно создан, то при вызове его методовread()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 6.1 Работа с классами FileInputStream и FileOutputStream

package example;

import java.io.\*;

public class Example {

private String path;

public Example(String path) {

this.path = path;

}

public void read() throws IOException {

FileInputStream inputStream = new FileInputStream(path);

// читаем первый символ с потока байтов

int data = inputStream.read();

char content;

// если dataбудет равна 0 то это значит,

// что файл пуст

while (data != -1) {

// переводим байты в символ

content = (char) data;

// выводим полученный символ

System.out.print(content);

// читаем следующий байты символа

data = inputStream.read();

}

// закрываем поток чтения файла

inputStream.close();

}

public void write(String st) throws IOException {

FileOutputStream outputStream = new FileOutputStream(path);

// записываем данные в файл, но

// пока еще данные не попадут в файл,

// а просто будут в памяти

outputStream.write(st.getBytes());

// только после закрытия потока записи,

// данные попадают в файл

outputStream.close();

}

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

Example examples = new Example("d:\\file.txt");

examples.write(" example ");

examples.read();

}

}

Результатом работы программы будет:

example

## Классы BufferedInputStream и BufferedOutputStream

На практике при считывании с внешних устройств ввод данных почти всегда необходимо буферизировать. Для буферизации данных служат классы BufferedInputStream и BufferedOutputStream. Эти классы используются для ускорения работы и только создают буфер для потоков ввода/вывода.

BufferedInputStream 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 BufferedInputStream добавляет поддержку методов mark() и reset(). Эти методы определены еще в классе InputStream, но там их реализация по умолчанию бросает исключение IOException. 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(). Так же буфер освобождается перед закрытием потока. При этом будет закрыт и надстраиваемый поток (так же поступает BufferedInputStream).

Пример 6.2 Работа с классами BufferedInputStream и BufferedOutputStream

package example;

import java.io.\*;

public class Example {

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

InputStream inStream = null;

BufferedInputStream bis = null;

try {

//открываем поток для чтения данных из файла

inStream = new FileInputStream("d:\\file.txt");

//преобразовываем в буферизированныйпоток

bis = new BufferedInputStream(inStream);

while (bis.available() > 0) {

char c = (char) bis.read();

System.out.println("Char: " + c);

}

} catch (Exception e) {

e.printStackTrace();

} finally {

if (inStream != null) {

inStream.close();

}

if (bis != null) {

bis.close();

}

}

}

}

Результатом работы программы будет:

Char: e

Char: x

Char: a

Char: m

Char: p

Char: l

Char: e

## Классы символьных потоков

Рассмотренные классы – наследники InputStream и OutputStream – работают с байтовыми данными. Если с их помощью записывать или считывать текст, то сначала необходимо сопоставить каждому символу его числовой код. Такое соответствие называется кодировкой.

Известно, что Java использует кодировку Unicode, в которой символы представляются двухбайтовым кодом. Байтовые потоки зачастую работают с текстом упрощенно – они просто отбрасывают старший байт каждого символа. В реальных же приложениях могут использовать различные кодировки. Поэтому в версии Java1.1 появился дополнительный набор классов, основывающийся на типах Reader и Writer.

Иерархия Reader и Writer, представленная на рисунке 6.2, очень схожа с аналогичной для байтовых потоков InputStream и OutputStream. Главное отличие между ними – Reader и Writer работают с потоком символов (char). Только чтение массива символов в Reader описывается методом read(char[]), а запись в Writer – write(char[]).

В [таблице 6.1](http://www.intuit.ru/studies/courses/16/16/lecture/500?page=4#table.15.1) приведены соответствия классов для байтовых и символьных потоков.

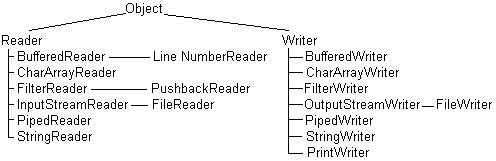


Рис. 6.2 Иерархия классов символьных потоков

Таблица 6.1

Соответствие классов для байтовых и символьных потоков

|  |  |
| --- | --- |
| **Байтовый поток** | **Символьный поток** |
| InputStream | Reader |
| OutputStream | Writer |
| ByteArrayInputStream | CharArrayReader |
| ByteArrayOutputStream | CharArrayWriter |
| Нет аналога | InputStreamReader |
| Нет аналога | OutputStreamWriter |
| FileInputStream | FileReader |
| FileOutputStream | FileWriter |
| FilterInputStream | FilterReader |
| FilterOutputStream | FilterWriter |
| BufferedInputStream | BufferedReader |
| BufferedOutputStream | BufferedWriter |
| PrintStream | PrintWriter |
| DataInputStream | Нет аналога |
| DataOutputStream | Нет аналога |
| ObjectInputStream | Нет аналога |
| ObjectOutputStream | Нет аналога |
| PipedInputStream | PipedReader |
| PipedOutputStream | PipedWriter |
| StringBufferInputStream | StringReader |
| Нет аналога | StringWriter |
| LineNumberInputStream | LineNumberReader |
| PushBackInputStream | PushBackReader |
| SequenceInputStream | Нет аналога |

Как видно из таблицы, различия крайне незначительны и предсказуемы.

Например, конечно же, отсутствует преобразование в символьно епредставление примитивных типов Java и объектов (DataInput/Output, ObjectInput/Output). Добавлены классы-мосты, преобразующие символьные потокив байтовые: InputStreamReader и OutputStreamWriter. Именно на их основе реализованы FileReader и FileWriter. Метод available() класса InputStream в классе Reader отсутствует, он заменен методом ready(), возвращающим булевое значение, – готов ли поток к считыванию (то есть будет ли считывание произведено без блокирования).

В остальном же использование символьныхпотоковидентично работе с байтовымипотоками.

## Классы InputStreamReader и OutputStreamWriter

Классы InputStreamReader и OutputStreamWriter обеспечивают трансляцию между символьными и байтовыми потоками с учетом заданной кодировки символов или кодировки, принятой по умолчанию в конкретной локальной системе. Объекту InputStreamReader в качестве источника передается байтовый поток ввода, и InputStreamReader обеспечивает чтение соответствующих символов Unicode. Объекту OutputStreamWriter в качестве получателя передается байтовый поток вывода, и OutputStreamWriter сохраняет в нем байтовые представления символов Unicode.

Классы-мосты InputStreamReader и OutputStreamWriter при преобразовании символов используют некоторую кодировку. Ее можно задать, передав в конструктор в качестве аргумента ее название. Если оно не будет соответствовать никакой из известных кодировок, будет брошено исключение UnsupportedEncodingException. Вот некоторые из корректных значений этого аргумента (чувствительного к регистру) для распространенных кодировок:"Cp1251","UTF-8","8859\_1"и т.д.

Описания конструкторов классов InputStreamReader и OutputStreamWriter приведены ниже.

public InputStreamReader (InputStream in) – cоздает объект InputStreamReader для ввода данных из указанного потока InputStream с учетом кодировки символов, принятой по умолчанию.

public InputStreamReader(InputStream in, String encoding) throws UnsupportedEncodingException – создает объект InputStreamReader для ввода данных из указанного по тока InputStream с учетом заданной кодировки символов encoding. Если кодировка encoding не поддерживается, выбрасывается исключение типа UnsupportedEncodingException.

public OutputStreamWriter(OutputStream out) – cоздает объект OutputStreamWriter для вывода данных в указанный поток OutputStream с учетом кодировки символов, принятой по умолчанию.

public OutputStreamWriter(OutputStream out, String encoding) throws UnsupportedEncodingException – cоздает объект OutputStreamWriter для вывода данных в указанный поток OutputStream с учетом заданной кодировки символов encoding. Если кодировка encoding не поддерживается, выбрасывается исключение типа UnsupportedEncodingException.

Методы read класса InputStreamReader обеспечивают ввод байтов из заданного потока InputStream и преобразование их в символы с использованием соответствующей кодировки. Аналогично, методы write класса OutputStreamWriter получают переданные символы, преобразуют их в байты, используя соответствующую кодировку, и выводят в заданный поток OutputStream.

## Классы BufferedReader и BufferedWriter

Класс BufferedReader увеличивает производительность за счет буферизации ввода. Для этого создается буфер для потока ввода.

У данного класса имеются два конструктора.

BufferedReader(Reader входной\_поток)

BufferedReader(Reader входной\_поток, int размер\_буфера)

Первая форма создает буферизованный символьный поток, используя размер буфера по умолчанию. Во второй форме размер буфера задает размер\_буфера.

Закрытие объекта класса BufferedReader приводит к закрытию также внутреннего потока, определенного аргументом входной\_поток.

Как и в случае с байтовым потоком, буферизованный символьный входной поток также обеспечивает фундамент для поддержки перемещения обратно по потоку в пределах доступного буфера. Для обеспечения этого класс BufferedReader реализует методы mark() и reset(), а метод BufferedReader.markSupported() возвращает значение true.

Класс BufferedWriter – это класс, производный от класса Writer, который буферизует вывод. Благодаря применению класса BufferedWriter, можно повысить производительность за счет снижения количества операций физической записи в выходное устройство. Он создает буфер для потоков вывода.

Класс BufferedWriter имеет следующие два конструктора.

BufferedWriter(Writer выходной\_поток)

BufferedWriter(Writer выходной\_поток, int размер\_буфера)

Первая форма создает буферизованный поток, использующий буфер с размером по умолчанию. Во второй размер буфера передается в параметре размер\_буфера.

Пример 6.3 Работа с классами BufferedReader и BufferedWriter

package example;

import java.io.\*;

public class Example {

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

InputStream is = null;

InputStreamReader isr = null;

BufferedReader br = null;

try {

// открываем поток для чтения

is = new FileInputStream("d:\\file.txt");

// создаем новый поток для чтения

isr = new InputStreamReader(is);

// создаем буферизированный поток

br = new BufferedReader(isr);

int value = 0;

while ((value = br.read()) != -1) {

char c = (char) value;

System.out.println(c);

}

} catch (Exception e) {

e.printStackTrace();

} finally {

if (is != null) {

is.close();

}

if (isr != null) {

isr.close();

}

if (br != null) {

br.close();

}

}

}

}

Результатом работы программы будет:

e

x

a

m

p

l

e

## Классы FileReader и FileWriter

Класс FileReader – производный от класса Reader, который можно использовать для чтения содержимого файла. Два наиболее часто используемых его конструктора выглядят так.

FileReader (String путь\_к\_файлу)

FileReader(File объект\_файла)

Оба могут передать исключение FileNotFoundException. Здесь путь\_к\_файлу – полное путевое имя файла, а объект\_файла – объект класса File, описывающий файл.

Этот класс создает объект класса, производного от класса Writer, который вы можете применять для записи файла. Рассмотрим его наиболее часто используемые конструкторы.

FileWriter(String путь\_к\_файлу)

FileWriter(String путь\_к\_файлу, boolean добавить)

FileWriter(File объект\_файла)

FileWriter(File объект\_файла, boolean добавить)

Все они могут передавать исключение IOExeption. Здесь путь\_к\_файлу – полное путевое имя файла, а объект\_файла – объект класса File, описывающий файл. Если добавить равно true, то вывод добавляется в конец файла.

Создание объекта класса FileWriter не зависит от того, существует ли файл. Класс FileWriter создаст файл перед его открытием для вывода, когда вы создаете объект. В случае попытки открытия файла, доступного только для чтения, передается исключение IOException.

Пример 6.4 Работа с классами FileReader и FileWriter

package example;

import java.io.\*;

class Example {

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

String source = "example\n";

char buffer[] = new char[source.length()];

source.getChars(0, source.length(), buffer, 0);

try (FileWriter f = new FileWriter("file.txt");) {

// записьвфайл

f.write(source);

} catch (IOException e) {

System.out.println("An I/O Error Occurred");

}

try (FileReader fr = new FileReader("file.txt")) {

int readBuffer;

// Читает и отображает файл.

while ((readBuffer = fr.read()) != -1) {

System.out.print((char) readBuffer);

}

} catch (IOException e) {

System.out.println("I/O Error: " + e);

}

}

}

Результатом работы программы будет:

example

## Работа с файловой системой. Класс File

Если классыпотоковосуществляют 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 6.5 Работа с классом File

package example;

import java.io.\*;

import java.util.Arrays;

public class Example {

//записываем информацию в файл document.doc

public static void InputInFile() throws IOException {

File file = new File("document.doc");

System.out.println("Файлсоздан");

file.deleteOnExit();//файл удалится после завершения работы виртуальной машины Java

//поток для записи в файл

FileWriter writer;

writer = new FileWriter(file, true);

writer.append(" example ");

writer.flush();

writer.close();

System.out.println("Информациязаписана в файл");

}

//статический метод вывода информации из файла

public static void OutputOfFile() throws IOException {

File file = new File("document.doc");

FileReader reader;

char buffer[];

int numb;

buffer = new char[1];

reader = new FileReader(file);

System.out.println("Читаеминформациюизфайла");

do {

numb = reader.read(buffer);

System.out.print(buffer[0]);

} while (numb == 1);

System.out.println("\nФайлпрочитан");

reader.close();

}

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

InputInFile();

OutputOfFile();

}

}

Результатом работы программы будет:

Файл создан

Информация записана в файл

Читаем информацию из файла

example

Файл прочитан

Пример 6.6 Работа с классом File

package example;

import java.io.\*;

public class Example {

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) != -1) ? 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");

}

}

Результатом работы программы будет (пример):

C:\Users\1\Documents\NetBeansProjects\example\src\example\Exaple.java

work finished

При выполнении этой программы на экран будут выведены названия 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 объекта и строки названия 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. Так, 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, а mkDirs создаст 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.

# Преобразование типов

Java является строго типизированным языком, а это означает, что каждое выражение и каждая переменная имеет строго определенный тип уже на момент компиляции. Тип устанавливается на основе структуры применяемых выражений и типов литералов, переменных и методов, используемых в этих выражениях.

Рассмотрим основные примеры возможных преобразований.

*Преобразование строки*

Преобразование типа String в byte, short, int, long, float, double, boolean может осуществляться с помощью методов valueOf и parseByte, parseShort, parseInt, parseLong, parseFloat, parseDouble, parseBoolean классов Byte, Short, Integer, Long, Float, Double, Boolean соответственно. Обратите внимание, что при преобразовании строки в логический тип boolean, любая строка, не соответствующая true, будет преобразована в логическое значение false.

Пример 6.7 Использование метода valueOf

Stringstr1 = “1451”;

try {

Integer i2 = Integer.valueOf(str1);

System.out.println(i2);

}catch (NumberFormatExceptione) {

System.err.println(Неверный формат строки!);

}

Пример 6.8 Использование метода parseFloat класса Float:

float f3 = 0;

String str2 = “36.6”;

try {

f3 = Float.parseFloat(str2);

System.out.println(f3);

catch (NumberFormatException e) {

System.err.println(Неверныйформатстроки!);

}

Преобразование String в String[]

Для того чтобы разбить строку на слова, разделенные пробелами, символами табуляций '\t', перевода строки '\n' и возврата каретки '\r' нужно подключить библиотеку java.util.StringTokenizer и создать объект класса StringTokenizer. Этот класс предназначен для выделения из строки отдельных токенов (слов).

Пример 6.9 Разбиение строки на слова:

String analizing Str="word1 word2 word3";

StringTokenizer tokenizer=new StringTokenizer(analizingStr);

int n = tokenizer.countTokens();

String[] tokens=new String[n];

for(inti=0;i<n;i++)

{

tokens[i]=tokenizer.nextToken();

}

*Преобразование чисел в строку*

Пример 6.10 Преобразование int в String (аналогично для типов double, float, long):

int i = 35;

String str = Integer.toString(i);

System.out.println(str);

*Преобразования символа char*

Пример 6.11 Преобразование char в String c использованием класса Character:

char ch = 'S';

String charToString = Character.toString(ch);

System.out.println(charToString);

Пример 6.12Преобразованиеchar в String с использованием операции добавления класса String:

char ch = 'S';

String str=+ch;

System.out.println(str);

Пример6.13Преобразованиеchar в String с использованием массива:

charch = 'S';

String fromChar = new String(new char[] { ch });

System.out.println(fromChar);

Пример 6.14 Преобразование char в String с использованием метода valueOf классаString:

char ch = 'S';

String valueOfchar = String.valueOf(ch);

System.out.println(valueOfchar);

Пример 6.15 Преобразование char в int c использованием метода getNumericValue класса Character:

char ch = '9';

int i1 = Character.getNumericValue(ch);

System.out.println(i1);

Пример6.16 Преобразование char в int c использованием метода digit класса Character:

char ch = '9';

int i2 = Character.digit(ch,10);

System.out.println(i2);

*Преобразования чисел*

Пример 6.17 Преобразование int в long:

int i = 2015;

long l = (long) (i);

System.out.println(l);

Пример 6.18 Преобразование int в float:

int i = 2015;

float f = (float) (i);

System.out.println(f);

Пример 6.19 Преобразование long в int:

long l = 214748364;

inti = (int) l;

System.out.println(i);

Пример 6.20 Преобразование double в int:

double d = 3.14;

int i = (int) d;

System.out.println(i);

Обратите внимание, что после преобразования double в int отбрасывается дробная часть и остается только целое число.

# Пример выполнения индивидуального задания

Требуется ввести информацию с клавиатуры с помощью потоков ввода, вывести информацию в файл, предусмотреть статический метод чтения из файла и вывода прочитанной информации на экран.

Состав класса Drugstore (аптека): фамилия владельца, наличие лицензии, месячная прибыль.

Пример 6.21 Пример выполнения индивидуального задания

package drugstore;

import java.io.\*;

import java.util.Arrays;

public class Drugstore {

String name;

double profit;

boolean license;

Drugstore () throws IOException{

String bolv;

//открываем символьный поток ввода

BufferedReader input=new BufferedReader (new InputStreamReader(System.in,"Cp1251"));

System.out.print("\nВведите фамилию: ");

this.name=input.readLine();

System.out.print("\nВведите прибыль: ");

this.profit=Double.parseDouble(input.readLine());

while(true){

System.out.print("\nНаличие лицензии(Да/Нет): ");

bolv=input.readLine();

if("Да".equals(bolv)){

this.license=true;

break;

}

if("Нет".equals(bolv)){

this.license=false;

break;

}

System.out.print("\nОшибка! Повторите ввод");

}

}

//записываем информацию в файл document.doc

public void InputInFile()throws IOException{

File file = new File("document.doc");

file.deleteOnExit(); //файл удалится после завершения работы виртуальной машины Java

//поток для записи в файл

FileWriter writer;

writer = new FileWriter(file,true);

writer.append("\nВладелец аптеки "+this.name+"; прибыль: "+this.profit+"; ");

if (this.license)

writer.append("есть лицензия.\n");

else writer.append("нет лицензии.\n");

writer.flush();

writer.close();

}

//статический метод вывода информации из файла

public static void OutputOfFile()throws IOException{

File file = new File("document.doc");

//поток для вывода информации

FileReader reader;

char buffer[];

int numb;

buffer=new char[1];

reader = new FileReader(file);

do{

numb=reader.read(buffer);

System.out.print(buffer[0]);

}while(numb==1);

reader.close();

}

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

Drugstore[] drugstores;

drugstores=new Drugstore[3];

for (int i=0; i<3; i++)

drugstores[i]=new Drugstore();

for(int i=0;i<3;i++)

drugstores[i].InputInFile();

OutputOfFile();

}

}

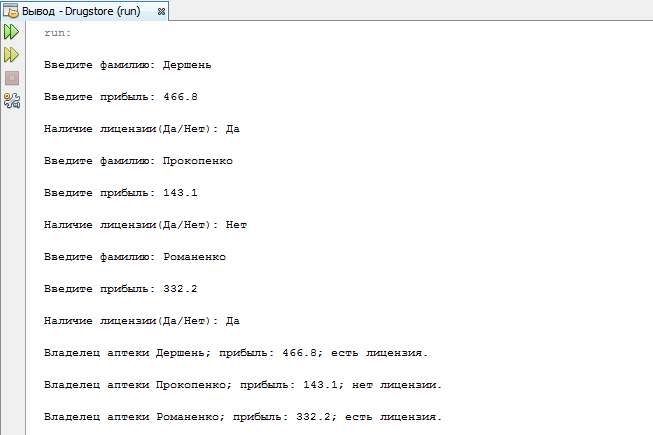


Рис. 6.3 Результат работы примера выполнения индивидуального задания

**Задания для самостоятельного выполнения**

Выполнить задание в соответствии с вариантом. Предусмотреть ввод информации с клавиатуры с помощью потоков ввода, вывод информации в файл, статический метод чтения из файла и вывода прочитанной информации на экран. **Запрещается** использовать класс *Scanner*.

Вариант 1. Состав класса Stock (склад) наименование товара, его стоимость, наличие товара на складе. Подсчитать общую стоимость товаров и количество товаров в наличии.

Вариант 2. Состав класса Student (студент) фамилия, возраст, наличие задолженностей. Подсчитать средний возраст студентов и количество студентов с задолженностями.

Вариант 3. Состав класса Music (музыкальное произведение) название произведения, количество проданных копий, возможность скачать mp3 в интернете. Подсчитать общее количество проданных копий и количество произведений, которые можно скачать mp3 в интернете.

Вариант 4. Состав класса Product (продукт) название продукта, наличие в магазине, цена. Подсчитать общую стоимость продуктов и количество продуктов в наличии.

Вариант 5. Состав класса Book (книга) название, цена, наличие иллюстраций. Подсчитать общую стоимость книг и количество книг с иллюстрациями.

Вариант 6. Состав класса Race (гонки) дата соревнования в виде символьной строки (год, месяц и день), количество участников, наличие приза. Подсчитать общее количество участников и количество соревновании призом.

Вариант7. Состав класса Ward (больничная палата) количество мест, фамилия врача, наличие свободных коек. Подсчитать общее количество мест и количество палат со свободными койками.

Вариант8. Состав класса Tribe (племя) название, численность, знакомство с огнем. Подсчитать общую численность и количество племен знакомых с огнем.

Вариант9. Состав класса Patient (пациент) фамилия, пол, возраст. Подсчитать средний возраст и количество женщин.

Вариант10. Состав класса Employee (служащий) фамилия, зарплата, наличие детей. Подсчитать среднюю зарплату и количество служащих с детьми.

Вариант 11. Состав класса Manager (управляющий) фамилия, возраст, справляется ли с обязанностями. Подсчитать средний возраст управляющих и количество управляющих, которые справляются с обязанностями.

Вариант 12. Состав класса Icecream (мороженое) название, наличие шоколада, процент жирности. Подсчитать средний процент жирности и количество мороженого с шоколадом.

Вариант 13. Состав класса Person (индивидуум) фамилия, возраст, пол. Подсчитать средний возраст и количество мужчин.

Вариант 14. Состав класса Butter (масло) название, наличие растительных добавок, цена. Подсчитать общую стоимость масла и количество видов масла с растительными добавками.

Вариант 15. Состав класса City (город) название, население, площадь. Подсчитать общую площадь и количество городов миллионеров.

Вариант 16. Состав класса Milk (молоко) название, жирность в процентах, наличие на складе. Подсчитать средний процент жирности и количество видов молока на складе.

Вариант 17. Состав класса Comicstore (магазин комиксов): фамилия владельца, наличие лицензии, количество наименований продукции. Подсчитать среднее количество наименований продукции по всем магазинам и количество магазинов с лицензией.

**Контрольные вопросы**

1. Какой абстрактный класс является суперклассом для всех классов, используемых для чтения байтов?

2. Какие источники могут быть использованы классами стандартных входных потоков Java в качестве источника данных?

3. Для чего используются классы BufferedInputStream и BufferedOutputStream?

4. Расскажите об особенностях классов Readerи Writer.

5. Какой класс используется для работы с файлами и каталогами?

6. Назовите методы, с помощью которых можно преобразовать строки.