Ввод\вывод. Сериализация.

В Java существует две встроенные библиотеки для работы с потоками ввода, вывода. Это java.io и java.nio. Первая это более старая, но по-прежнему актуальная библиотека. Вторая это новая библиотека и это отразилось в его названии New Input Output.

Библиотека java.io следует применять, когда вам необходимо использовать один поток, ведь несколько потоков это проблематика этой библиотеки. Новая же позволяет лучше масштабировать.

Кроме ввода и вывода они представляют собой средства для работы с файлами, директориями, а также с сетями.

Основным достоинством java.io это Сериализация. *Сериализация* - это процесс записи состоянии объекта в файл и дальнейшей *десериализацией*, преобразования записанного состояния в объект.

Библиотека java.io.

Класс File.

Данный класс оперирует действия с файлом и файловой системой. Класс File служит для получения сведения о файле на диске: права доступа, путь каталогу, абсолютные пути, дата, время, манипулированная сведениями и перемещение по иерархии.

Для создания объекта класса File следует воспользоваться одним из конструкторов, где *путь к каталогу* обозначает путь к файлу, а *имя файла* имя конкретного файла; *объект каталог* объект типа File инкапсулирующий путь файлу, а где *объект\_URI* – объект типа URL, описывающий файл:

File (String *путь\_каталога*)

File (String *путь\_каталога*, String *имя\_файла*)

File (File *объект\_каталога*)

File (File *объект\_каталога*, String *имя\_файла*)

File (URL *объект\_URL*)

В классе File определяется ряд методов. Например, где getName() возвращает имя файла, метод getParent() возвращают имя родителя, exists () проверяет, существует ли файл.

Ниже представлен пример использования File:

package myPackage;

import java.io.File;

public class Main{

public static void main(String[] args) {

File f = new File("C:\\Users\\Семья\\IdeaProjects\\" +

"Task14\\out\\production\\Task14\\com\\company\\Main.class");

System.out.println ("Имя файла: "+f.getName ());

System.out.println ("Путь"+f.getPath ());

System.out.println ("Абсолютный путь: "+f.getAbsolutePath ());

System.out.println ("Родительский путь: "+f.getParent ());

System.out.println ("Файл доступен для записи: "+f.canWrite ());

System.out.println ("Файл доступен для чтения: "+f.canRead ());

System.out.println ("Это файл: "+f.isFile ());

System.out.println ("Это каталог: "+f.isDirectory ());

System.out.println ("Размер файла "+f.length () +" байтов");

}

}

Объект File также может представлять каталог. Проверить объект File на каталог можно методом isDirectory() . Методом list() можно извлечь файлы и директории в директории. Полное объявление метода:

String [] list ()

Ниже представлен пример использования list():

package myPackage;

import java.io.File;

public class Main{

public static void main(String[] args) {

String dirname="C:\\Users";

File f = new File(dirname);

if(f.isDirectory()){

System.out.println("Каталог: "+dirname);

String [] paths= f.list();

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

File f1 = new File(dirname+"\\"+paths[i]);

if(f1.isDirectory())

System.out.println (paths[i] +" является каталогом ");

else

System.out.println (paths[i] +" является файлом");

}

}

}

}

Есть вторая форма данного метода, где указывается *фильтр,* по которому будет отбираться файлы/директории. Полное объявление метода:

String [] list (FilenameFilter *фильтр*)

Функциональный интерфейс FilenameFilter определяет функцию accept(). Она вызывается у каждого фала из директории. Полное объявление:

boolean accept (File *каталог*, String *имя\_файла*)

Он возвращает true, если *файл* должен быть включен в список. Он может сортироваться по расширениям, префиксу, равность значению и множество количество вариантов.

В нижней программе класс Filter реализует FilenameFilter. Он осуществляет отбор по расширениям файла/директория. Код:

package myPackage;

import java.io.File;

import java.io.FilenameFilter;

public class Main{

public static void main(String[] args) {

String dirname="C:\\Program Files";

File f = new File(dirname);

FilenameFilter filter= new Filter ("xml");

String [] s = f.list(filter);

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

System.out.println(s[i]);

}

}

}

class Filter implements FilenameFilter {

String postfix;

Filter (String postfix){

this.postfix="."+postfix;

}

@Override

public boolean accept(File dir, String name) {

return name.endsWith(postfix);

}

}

Есть также схожий метод listFiles() его отличие в том, что он не массив строк, а массив объектов File. Полные объявления данного метода:

File [] listFiles ()

File [] listFiles (FilenameFilter *фильтр*)

File [] listFiles (FIleFilter *ff*)

Интерфейс FIleFilter работает схоже с FilenameFilter. Он объявляет метод accept(), который отбирает файлы. Полное объявление метода:

boolean accept (File *файл*)

Создание директорий.

В классе File определенно два метода, которые позволяют создавать директории. Это mkdir(), который создает директорию по уже указанному пути, а метод mkdirs () создает каталог и всех родительских каталогов.

Все они возвращают true если они удачно создали директорию и возвращают false если они не помешало, что-либо. Данные методы не рекомендуется использовать.

Исключения в системе ввода-вывода.

Система ввода-вывода определяет две основные и часто употребляемые исключения. Первый это IOException, который генерируется в момент ошибки ввода-вывода. Второй производный от IOException исключение FileNotFoundException, который кидается когда указанный файл не находится или файл нельзя открыть. Обоих можно обрабатывать в одном блоке catch.

Интерфейсы AutoCloseable,

Closeable и Flushable.

Эти интерфейсы играют важную роль для классов ввода-вывода. Интерфейсы Closeable и Flushable определяются в java.io. Интерфейс Flushable находится java.lang.

Интерфейс AutoCloseable обеспечивает try с ресурсами, который автоматизирует процесс закрытия потока ввода/вывода. Чтоб объект мог использовать try с ресурсами его класс должен реализовывать AutoCloseable. Он определяет один метод close(), который высвобождает ресурсы:

void close() throws Exception

Интерфейс Closeable обеспечивает закрытие потока ввода/вывода методом close. Он реализует интерфейс AutoCloseable.

Все объекты реализующие Flushable могут принудительно вывести буферизованные данные в тот поток ввода, к которому присоединен данный объект. Он определяет метод flush():

void flush() throws Exception

Два способа закрытия потока.

Когда поток ввода-вывода больше не используется его следует закрыть. Если его не закрыть то может произойти утечка памяти.

Традиционный способ заключается в использование метода close в конструкции try-finally. Ниже представлен синтаксис:

try {

// Открываем файл и получить доступ к нему

} catch (*исключение\_ввода\_вывода* ) {

// Обработка исключений

} finally {

// Закрытие файла  
}

В второй форме используется try с ресурсами. Синтаксис try с ресурсами:

try (*спецификация\_ресурса*) {

//Использование ресурса

}

Потоки ввода-вывода.

Потоки ввода-вывода разделяет на байтовый и символьные потоки. За байтовый ввод и вывод отвечают абстрактные классы InputStream и OutputStream, а за символьные – абстрактные классы Reader and Writer, символьные потоки поддерживают кодировку Unicode.

Байтовые потоки следует использовать для низкоуровневых операций, а символьные наоборот. Байтовые потоки удобны при работе с файлами и с ними могут работать разные объекты. Символьные потоки же следует использовать для символов или их строками. Потоки ввода-вывода байтов и потоки ввода-вывода символов образуют две отдельные иерархии.

Байтовые потоки ввода-вывода.

Класс InputStream.

Абстрактный класс InputStream реализует модель ввода байтов в Java. Он реализует интерфейсы AutoCloseable и Closeable. Многие его методы генерирует исключение IOException. Методы данного класса:

* int available (int *осталось*) – Возвращает количество байт, которые остались не прочитанными.
* void close() – Закрывает поток.
* read () – Читает байтовый поток.
* read (byte [] *массив*) – читает в указанные байтовый *массив* .
* read (byte [] *массив*, int *начало*, int *конец*) – читает в байтовый *массив* от *начало* до *конца*.
* long skip(long *пропустить*) – пропускает определенное количество байтов.
* void mark(int *позиция*) – ставит марку на определенной *позиция*.
* void reset() – возвращает на марку.
* boolean markSupported () – проверяет поддерживает ли поток mark () и reset ().
* byte [] readAllBytes () – читает поток, все прочитанное передает в массив.

Класс OutputStream.

Абстрактный класс OutputStream определяет модель поток ввода. Он реализует интерфейсы Closeable и AutoCloseable. Многие методы данного класса генерируют исключение IOException. Методы данного класса:

* void write (int i) – пишет первый 8 бит.
* void write (byte [] b) – пишет весь массив.
* void write (byte [] b, int start, int the\_end) – пишет часть массива.
* void flush () – скидывает буфер.
* void close () – закрывает временно поток.

Класс FileOutputStream

В классе создается объект класса OutputStream служащим для ввода байтов в файл. Он реализует интерфейс Closeable, AutoCloseable и Flushable. Ниже представлены четыре конструктора данного класса:

FileOutputStream (String *путь\_файла*)

FileOutputStream (File *объект\_файла*)

FileOutputStream (String *путь\_файла*, boolean *добавить*)

FileOutputStream (File *объект\_файла*, boolean *добавить*)

Где *путь\_файла* имя пути к файлу, *объект\_файла* объект File описывающий файл. Если параметр *добавить* принимает true файл откроется в режиме ввода данных. Исключение FileNotFoundException будет сгенерировано если открывается файл для чтения. Если указанные путь или объект файла ведут на не существующий файл то автоматически создаться файл с указанный путем.

В нижней программе демонстрируется пример использования FileOutputStream:

package myPackage;

import java.io.FileOutputStream;

import java.io.IOException;

public class Main{

public static void main(String[] args) {

String text = "Class OutputStream writes bytes to a file." +

"\nHe extends the abstract class OutputStream.";

byte [] bytes =text.getBytes();

try(FileOutputStream stream=new FileOutputStream("java1.txt");

FileOutputStream stream1=new FileOutputStream("java2.txt");

FileOutputStream stream2=new FileOutputStream("java3.txt")){

// Записываем массив байтов

stream.write(bytes);

// Записываем каждый второй байт в файл

for (int i = 0; i < bytes.length; i+= 2) {

stream1.write(bytes[i]);

}

// Записываем половину массива

stream2.write(bytes, bytes.length/2, bytes.length/2);

}catch (IOException exc){

System.out.println("Ошибка ввода-вывода: "+exc);

}

}

}

Класс FileInputStream.

Класс FileInputStream создает объект InputStream для вывода байтов из файлов. Ни представлены некоторый конструкторы данного класса:

FileInputStream (String *путь\_файла*)

FileInputStream (File *объект\_файла*)

Здесь *путь\_файла* обозначает путь к конкретному файлу, *объект\_файла* объект описывает файл. Если файл не найден и/или не существует генерируется исключение FileNotFoundException.

Ниже представлен пример использования данного класса:

package myPackage;

import java.io.FileInputStream;

import java.io.FileNotFoundException;

import java.io.FileOutputStream;

import java.io.IOException;

public class Main{

public static void main(String[] args){

try(FileOutputStream stream=new FileOutputStream("java.txt")) {

stream.write("this string was writing to a file java.txt".getBytes());

} catch (FileNotFoundException e) {

System.out.println("Файл в режиме для чтения и/или он не найден.");

} catch (IOException e) {

System.out.println("Ошибка ввода-вывода: "+e.getMessage());

}

long size;

try(FileInputStream stream=new FileInputStream("java.txt")) {

System.out.println("Количество байтов для чтения: "+stream.available());

size = stream.available();

System.out.print("Одна четвертая файла: ");

for (int n=0; n<size/4; n++)

System.out.print((char)stream.read());

stream.skip(2);

System.out.println("Пропускаем 2 байта...");

System.out.print("Остальные байты: ");

for(int i=0; i<stream.available(); i++)

System.out.print((char) stream.read());

} catch (FileNotFoundException e) {

System.out.println("Файл не найден");

} catch (IOException e) {

System.out.println("Ошибка ввода-вывода");

}

}

}

Классы ByteArrayInputStream.

Класс ByteArrayInputStream реализует поток ввода, использующий массив байтов в качестве источника. Конструкторы данного класса:

ByteArrayInputStream (byte *массив*)

ByteArrayInputStream (byte *массив*, int *начало*, int *количество байтов*)

Где *массив* обозначает конкретный байтов. Во второй форме можно указать с какого индекса и сколько байтов должен прочитать поток.

Следует сказать, что его не нужно закрывать методом close() т.е. использоваться try с ресурсами. В нем переопределяется метод mark() и reset(). Если метод mark() не ставит марку, то метод reset() вернется в начало.

Ниже представлен пример использования данного класса:

package myPackage;

import java.io.ByteArrayInputStream;

public class Main{

public static void main(String[] args){

String text="abc";

byte [] bytes=text.getBytes();

ByteArrayInputStream stream=new ByteArrayInputStream(bytes);

for(int i=0; i < 2; i++){

int n;

while ((n=stream.read()) != -1){

if(i==0)

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

else System.out.print(Character.toUpperCase((char) n));

}

System.out.println();

stream.reset();

}

}

}

Класс ByteArrayOutputStream.

Класс ByteArrayOutputStream реализует поток ввода, использующий массив байтов в качестве источника. Конструкторы данного класса:

ByteArrayOutputStream ()

ByteArrayOutputStream ( int *количество\_байтов*)

В первой создается пустой буфер размером 32 байта, а во второй форме создается буфер с размер *количество\_байтов*. Его не нужно закрывать методом close().

Ниже представлен пример использования его:

package myPackage;

import java.io.ByteArrayOutputStream;

import java.io.FileNotFoundException;

import java.io.FileOutputStream;

import java.io.IOException;

public class Main{

public static void main(String[] args) {

String text="ByteArrayOutputStream save a byte\'s array";

byte[] bytes = text.getBytes();

ByteArrayOutputStream stream=new ByteArrayOutputStream();

try {

stream.write(bytes);

} catch (IOException e) {

System.out.println("Ошибка ввода-вывода: "+e);

}

System.out.println("Содержимое в виде строки: "+stream.toString());

byte [] arr=stream.toByteArray();

System.out.print("Содержимое в виде массива: [");

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

System.out.print((char) arr[i]);

System.out.println("]");

try(FileOutputStream stream1=new FileOutputStream("java.txt")) {

stream.writeTo(stream1);

} catch (FileNotFoundException e) {

System.out.println("Файл не найден и/или закрыт");

} catch (IOException exc) {

System.out.println("Ошибка ввода-вывода: "+exc);

}

stream.reset();

for(int i=0; i<=3; i++) stream.write( 'S');

System.out.println("Перезаписанный вариант: "+stream.toString());

}

}

Класс DataInputStream и DataOutputStream.

Эти классы позволяет выводить примитивные данные в поток или выводить примитивы. Они реализуют DataInput и DataOutput соответственно.

Класс DataInputStream расширяет FilterInputStream, который расширяет InputStream. Он реализует интерфейсы Closeable, AutoCloseable и Flushable. Конструктор данного класса:

DataInputStream (InputStream *поток\_вывода*)

У него определяются ряд методов типа readInt(), readLong(), readBoolean() и т.д. Большинство методов генерирует исключение IOException.

Класс DataOutputStream расширяет FilterOutputStream, который расширяет OutputStream. Он реализует интерфейсы Closeable, AutoCloseable и Flushable. Конструктор данного класса:

DataOutputStream (OutputStream *поток\_вывода*)

У него определяются ряд методов типа writeInt(int *значение*), writeLong(long *значение*), writeBoolean(boolean *значение*) и т.д. Большинство методов генерирует исключение IOException.

Ниже представлен пример использования данных классов:

package myPackage;

import java.io.\*;

public class Main{

public static void main(String[] args) {

try(DataOutputStream stream=new DataOutputStream(new FileOutputStream("java.txt"))) {

stream.writeLong(24l);

stream.writeInt(23);

stream.writeBoolean(true);

} catch (FileNotFoundException e) {

System.out.println();

} catch (IOException e) {

System.out.println(e);

}

try(DataInputStream stream=new DataInputStream(new FileInputStream("java.txt"))) {

System.out.printf("Содержимое файла: %d %d %b", stream.readLong(), stream.readInt(), stream.readBoolean());

} catch (FileNotFoundException e) {

System.out.println(e);

} catch (IOException e) {

System.out.println(e);

}

}

}

Класс SequenceInputStream.

Данный класс соединяет два потока в один. Достигается это тем, что в процессе поток SequenceInputStream сначала выполняет запросы одного потока, а потом другого потока. Конструкторы данного класса:

SequenceInputStream (InputStream *поток\_1*, InputStream *поток\_2*)

SequenceInputStream (Enumeration <E extends InputStream> *перечисление*)

В первой склеивается *поток\_1* и *поток\_2*, а во второй поочередно все элементы *перечисления*. Ниже представлен пример использования данного класса:

package myPackage;

import java.io.\*;

import java.util.Enumeration;

import java.util.Vector;

public class Main{

public static void main(String[] args) {

Vector<String> files=new Vector<>();

files.addElement("java.txt");

files.addElement("java1.txt");

files.addElement("java2.txt");

try(FileOutputStream stream=new FileOutputStream("java.txt");

FileOutputStream stream1=new FileOutputStream("java1.txt");

FileOutputStream stream2=new FileOutputStream(“java2.txt”)) {

stream.write(“text to java.txt\n”.getBytes());

stream1.write(“text to java1.txt\n”.getBytes());

stream2.write(“text to java2.txt\n”.getBytes());

} catch (FileNotFoundException e) {

System.out.println(e);

} catch (IOException e) {

System.out.println(e);

}

try(InputEnumeration enumStream=new InputEnumeration(files);

InputStream stream=new SequenceInputStream(enumStream)){

int n;

while ((n=stream.read()) != -1)

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

} catch (IOException e) {

System.out.println(“Ошибка ввода-вывода”);

}

}

}

class InputEnumeration implements Enumeration<FileInputStream>, AutoCloseable{

private Enumeration<String> files;

public InputEnumeration(Vector<String> files){

this.files=files.elements();

}

@Override

public boolean hasMoreElements() {

return files.hasMoreElements();

}

@Override

public FileInputStream nextElement() {

try {

return new FileInputStream(files.nextElement().toString());

} catch (FileNotFoundException e) {

return null;

}

}

@Override

public void close() {

}

}

Класс PrintStream.

Класс PrintStream выводит данные по дескриптору файла System.out тип System. Он реализует Flushable, AutoCloseable, Closeable и Appendable. Благодаря этому он является наиболее употребительным классом.

Класс BufferedInputStream.

Класс BufferedInputStream позволяет заключить объект класса InputStream и добиться высокой производительности благодаря буферизации ввода-вывода. Конструкторы данного класса:

BufferedInputStream (InputStream *поток*)

BufferedInputStream (InputStream *поток*, int *размер\_буфера*)

Где *поток* из него будет создан буферизованный поток вывода. Во второй форме можно также указать *размер буфера*. Размер буфера может зависть от машины, объем доступа памяти и т.д. поэтому лучше установить размер буфера 8192.

Кроме метода read() в нем переопределяется mark(), reset() и markSupported(). Ниже представлен пример использования данного класса:

package myPackage;

import java.io.BufferedInputStream;

import java.io.ByteArrayInputStream;

import java.io.IOException;

public class Main{

public static void main(String[] args) {

String text="ABCDE";

byte [] bytes=text.getBytes();

ByteArrayInputStream instr=new ByteArrayInputStream(bytes);

try(BufferedInputStream bis=

new BufferedInputStream(

instr, 8192)) {

System.out.println("Char : "+(char)bis.read());

System.out.println("Char : "+(char)bis.read());

System.out.println("Char : "+(char)bis.read());

bis.mark(0);

System.out.println("Char : "+(char)bis.read());

System.out.println("reset() invoked");

System.out.println("char : "+(char)bis.read());

bis.reset();

System.out.println("char : "+(char)bis.read());

} catch (IOException e) {

System.out.println(e);

}

}

}

Класс BufferedOutputStream.

Данный класс создает буферизованный поток ввода. Его особенностью может считаться метод flush(), который обеспечивает запись в буферизируемый поток ввода. По конструкторам он схож с BufferedInputStream. Ниже представлен пример использования данного класса:

package myPackage;

import java.io.\*;

public class Main{

public static void main(String[] args) {

String text = "Hello World!";

try(BufferedOutputStream stream =

new BufferedOutputStream(

new FileOutputStream("java.txt"))){

stream.write(text.getBytes(), 0, text.length());

}catch (IOException exc){

System.out.println(exc);

}

}

}

Класс PushbackInputStream.

Одним из нововведений в буферизацию потоков ввода-вывода является технология *возврата в поток вывода*. Он обеспечивает при чтения байтов – читающий байт возвращается обратно в поток вывода. Благодаря его можно быстро пробежаться по потоку вывода и выяснить что из него возвращается фактически не извлекая данные.

Ниже перечислены конструкторы данного класса, где *поток\_вывода* обозначает поток данные чьи будут использоваться. Во второй форме можно дополнительно указать размер буфера.

PushbackInputStream (InputStream *поток\_вывода*)

PushbackInputStream (InputStream *поток\_вывода*, int *размер\_буфера*)

В нем кроме основных методов в нем определяется метод unread(), ниже приведены три формы данного метода:

void unread(int *d*)

void unread (byte [] *буфер*)

void unread (byte [] *буфер*, int *смещение*, int *размер*)

В первой форме в поток возвратится младший байт *d*. Он возвратится в следующем вызове метода read(). Во второй форме байты возвращаются из *буфера*.

Ниже представлен пример использования данного класса:

package myPackage;

import java.io.\*;

public class Main {

public static void main(String[] args) {

String text="if(y == a) y = 0;\n";

byte [] buf=text.getBytes();

try(PushbackInputStream stream =

new PushbackInputStream(

new ByteArrayInputStream(buf))){

int n;

while ((n=stream.read()) != -1){

switch (n){

case '=' :

if((n=stream.read()) == '=')

System.out.print(".eq.");

else {

System.out.print("<-");

stream.unread(n);

}break;

default:

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

break;

}

}

}catch (IOException exc){

System.out.println(exc);

}

}

}

Класс RandomAccessFile.

Данный класс позволяет инкапсулирует *файл произвольного доступа*. Он не расширяет OutputStream и InputStream, но реализует интерфейсы DataInputStream, DataOutputStream и Closeable, AutoCloseable. Он позволяет установить позицию указателя файла. Ниже перечислены конструкторы:

RandomAccessFile (String *путь\_файла*, String *режим*)

RandomAccessFile (File *объект\_файла*, String *режим*)

Здесь *режим* может принимать “r” режим чтения, “rw” читать и писать, “rws” все изменения стразу вносить на физический файл.

Ключевым методом класса RandomAccessFile является seek(), который позволяет установить *новую позицию*. Полное объявление:

void seek(long *новая\_позиция*) throws IOException

Ниже представлен пример использования данного класса:

package myPackage;

import java.io.IOException;

import java.io.RandomAccessFile;

public class Main {

public static void main(String[] args) {

try(RandomAccessFile raf=

new RandomAccessFile("java.txt", "rws")) {

int n;

raf.seek(0);

raf.writeUTF("Hello World from object RandomAccessFile.\n");

raf.seek(8);

while ((n=raf.read()) != -1)

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

raf.seek(0);

while ((n=raf.read()) != -1)

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

System.out.println("Размер: "+raf.length()+" байтов.");

} catch (IOException e) {

e.printStackTrace();

}

}

}

Фильтруемые потоки.

*Фильтруемые потоки ввода-вывода* представляют собой оболочку какому-то потоку. К ним относят FilterInputStream и FilterOutputStream, которые содержат соответствующие потоки ввода-вывода байтов. Он содержит основные методы. Данные классы служат суперклассами для всех фильтрующих потоков.

Символьные потоки ввода-вывода.

Класс Reader.

Данный класс является абстрактный и определяет потоковый вывод символов. Он реализует Closeable, AutoCloseable и Readable. Методы данного класса.

* int read () – Возвращает целочисленное представление следующего массива. Когда поток заканчивается возвращается -1.
* int read (char [] *буфер*) – Возвращает результат чтения в указанный *буфер*. При окончания чтения возвращает -1.
* int read (char [] *буфер*, int *смещение*, int *длина*) – Возвращает результат чтения в *буфер* в определенные границы.
* long skip(long *количество\_символов*) – Пропускает указанное *количество символов*.
* boolean ready () – Проверяет будет ли ждать следующий запрос на вывод.
* void mark (int *позиция*) – Ставит марку на *позиции*.
* void reset() – Возвращается к последней поставленной марке.
* boolean markSupported () – Проверяет можно ли использовать mark() и reset().

Класс Writer.

Класс Writer является абстрактным и определяет потоковый вывод символов в Java. Он реализует Closeable, AutoCloseable, Flushable и Appendable. Методы:

* void writer (int *символ*) – Записывает в поток указанный *символ*.
* void writer (String *строка*) – Читает сразу строку.
* void writer (char [] *буфер*) – Читает весь массив.
* void writer (char [] *буфер*, int *смещение*, int *размер*) – Записывает часть *буфер*.
* void writer (String *строки*, int *смещение*, int *размер*) – Записывает часть *строки*.
* abstract void flush () – Очищает буфер.
* abstract void close () – Закрывает поток.
* Writer append (char ) – Добавляет указанный *символ* в конец потока ввода. Возвращает ссылку на вызывающий поток ввода.
* Writer append (CharSequence *последовательность\_символов*) – Добавляет указанную *последовательность символов* в конец потока.
* Writer append (CharSequence *последовательность\_символов*, int *начало*, int *конец*) – Добавляет часть указанной *последовательности символов* в конец потока.

Классы FileWriter и FileReader.

Класс FIleWriter создает поток типа Writer записывающий символы в файлы. Ниже представлен конструкторы данного класса:

FileWriter (String *путь\_файла*)

FileWriter (String *путь\_файла*, boolean *добавить*)

FileWriter (File *объект\_файла*)

FileWriter (File *объект\_файла*, boolean *добавить*)

Здесь *путь файла* обозначает полный путь к определенному файла, а если его нет он автоматически создается. Где *объект файла* объект описывающий файл. Если в *добавить* принять true добавленные данные запишутся в конец.

Класс FileReader создает поток типа Reader и служит для чтения содержимым данных файла. Конструкторы данного класса:

FileReader (String *путь\_файла*)

FileReader (File *объект\_файла*)

Здесь *путь файла* обозначает полный путь к конкретному файлу. Где *объект файла* обозначает объект описывающий файл. Если файл был не найден или закрыт генерируется исключение FileNotFoundException.

Ниже представлен пример использования выше описанных классов:

package myPackage;

import java.io.\*;

public class Main {

public static void main(String[] args) {

String source= "Hello World\n";

char [] buf=source.toCharArray();

try(FileWriter f=new FileWriter("java.txt");

FileWriter f1=new FileWriter("java1.txt");

FileWriter f2=new FileWriter("java2.txt")) {

// Записываем стразу строку.

f.write(source);

for (int i = 0; i < buf.length; i += 2) {

f1.write(buf[i]);

}

f2.write(source, source.length()/2, source.length()/2);

} catch (IOException e) {

System.out.println(e);

}

File f=new File("java.txt");

try(FileReader reader=new FileReader(f);

FileReader reader1=new FileReader("java1.txt")){

char [] result=new char[(int) f.length()];

reader.read(result);

System.out.print("Содержимое java.txt: ");

for(char c : result)

System.out.print(c);

System.out.print("Содержимое java1.txt: ");

int n;

while ((n=reader1.read()) != -1)

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

System.out.println();

}catch (IOException exc){

System.out.println("Ошибка ввода-вывода: "+exc);

}

}

}

Классы CharArrayReader и CharArrayWriter.

Класс CharArrayReader реализует поток вывода, где источником является *массив символов.* Ниже представлены два конструктора данного класса:

CharArrayReader (char [] *массив*)

CharArrayReader (char [] *массив*, int *начало*, int *длина*)

Класс CharArrayWriter реализует поток ввода, использующий массив символов в качестве адресата. У него два конструктора, где *длина буфера* обозначает размер буфера где будут хранится вводимые данные:

CharArrayWriter ()

CharArrayWriter (int *размер\_буфера*)

Выше описанные класса не требуют вызов метод close(). В приведенной ниже программе сначала создается поток CharArrayWriter, в который вводится строка text. После в try с ресурсами создается два потока вывода CharArrayReader: первый читает всю строку, а второй половину строки. Далее из потока ввода все данные перекачиваются в файл. Код:

package myPackage;

import java.io.\*;

public class Main {

public static void main(String[] args) {

String text="Quite silly when a stupid film gains 98 percent of ratings.\n";

CharArrayWriter writer=new CharArrayWriter();

try {

writer.write(text);

} catch (IOException e) {

System.out.println("Исключение ввода-вывода: "+e);

}

System.out.println("Содержимое writer: \n"+writer.toString());

try(CharArrayReader reader=new CharArrayReader(text.toCharArray());

CharArrayReader reader1=new CharArrayReader(text.toCharArray(), 0, text.length()/2)){

System.out.println("Содержимое reader: ");

int n;

while ((n=reader.read()) != -1){

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

}

System.out.println("Содержимое reader1: ");

int c;

while ((c=reader1.read()) != -1){

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

}

}catch (IOException exc){

System.out.println("Исключение ввода-вывода: "+exc);

}

try(FileWriter writer1=new FileWriter("java.txt")){

writer.writeTo(writer1);

}catch (IOException exc){

System.out.println("Исключение ввода-вывода: "+exc);

}

}

}

Классы BufferedReader и BufferedWriter.

Класс BufferedReader является производным от Reader и буферизует поток вывода. У него определены два конструктора:

BufferedReader (Reader *поток\_вывода*)

BufferedReader (Reader *поток\_вывода*, int *размер\_буфера*)

В первой конструкторе на базе *поток вывода* будет создан буферизированный поток. Во второй форме также можно дополнительно указать размер буфера, в котором будут хранится выводимые данные.

Данный класс дополняется двумя методами readLine() и lines (). Метод readLine () работает схоже как и read() за исключением того, что он читает сразу строками. При окончании чтения возвращает null. Методом lines () можно получить Stream параметризованный String.

Класс BufferedWriter отвечает за буферизацию вводимых данных и буферизирует поток ввода. Ниже представлены конструкторы данного класса:

BufferedWriter (Writer *поток\_ввода*)

BufferedWriter (Writer *поток\_ввода*, int *длина\_буфера*)

В первой форме создается буферизированный поток ввода на основе *потока ввода*. Во второй форме можно дополнительно указать размер буфера где будет хранится вводные данные.

Кроме стандартных методов он дополняется еще одним методом. Методом newLine() можно в текущем моменте получить пустую линию.

Ниже представлен пример использования данных классов:

package myPackage;

import java.io.\*;

public class Main {

public static void main(String[] args) {

try(BufferedWriter writer=

new BufferedWriter(

new FileWriter(

"java.txt"))){

String text="Writing is one of the " +

"greatest inventions of mankind.";

writer.write(text);

writer.newLine();

writer.append("Размер строки: "+text.length());

}catch (IOException exc){

System.out.println("Ошибка ввода-вывода: "+exc);

}

try(BufferedReader reader=

new BufferedReader(

new FileReader(

"java.txt"))){

long count = reader.

lines().

count();

System.out.println("Количество строк: "+count);

System.out.println("Данные с файла: ");

String s;

while ((s=reader.readLine()) != null)

System.out.println(s);

}catch (IOException exc){

System.out.println(exc);

}

}

}

Класс PushbackReader.

Класс PushbackReader реализует механизм возврата в поток ввода. Он описан в описании класса PushbackInputStream. Ниже перечислены конструкторы данного класса:

PushbackReader (Reader *поток\_вывода*)

PushbackReader (Reader *поток\_вывода*, int *размер\_буфера*)

В первой форме создается буферизованный поток на основе *потока вывода* ,в который можно возвратить один символ. Во второй форме можно указать дополнительно размер буфера. При закрытия потока PushbackReader закрывается также базовый поток.

Методом unread () можно возвратить символ или несколько символов в поток PushbackReader. Ниже перечислены конструкторы данного класса:

void unread (int *символ*)

void unread (char [] *массив*) throw IOException

void unread (char [] *массив*, int *начало*, int *длина*) throw IOException

Ниже представлен пример использования данного класса:

package myPackage;

import java.io.\*;

public class Main {

public static void main(String[] args) {

try(PushbackReader reader=

new PushbackReader(

new CharArrayReader(

("if (m == 0) \n " +

" m = 4; ").toCharArray()))) {

int n;

while ((n=reader.read()) != -1) {

switch (n) {

case '=' :

if((n=reader.read()) == '=')

System.out.print(" .eq. ");

else {

System.out.print("<-");

reader.unread(n);

} break;

default :

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

break;

}

}

}catch (IOException exc){

System.out.println("Ошибка ввода-вывода: "+exc);

}

}

}

Класс PrintWriter.

У класса PrintStream есть свой символьный аналог PrintWriter, реализующий интерфейсы Appendable, Flushable , Closeable и AutoCloseable. Конструкторы данного класса, которые используют другие потоки:

PrintWriter (OutputStream *поток\_ввода*)

PrintWriter (OutputStream *поток\_ввода*, boolean *добавить*)

PrintWriter (Writer *поток\_ввода*)

PrintWriter (Writer *поток\_ввода*, boolean *добавить*)

Где *поток ввода* обозначает конкретный поток на котором будет построен далее поток. Если в параметры *добавить* передать true то будет происходить автоматическая очистка буфера. По умолчанию принимается false. Ниже перечислены конструкторы с использованием файлов:

PrintWriter (String *путь\_файла*) throws FileNotFoundException

Print Writer (String *путь\_файла*, String *набор\_символов*)

throws FileNotFoundException, UnsupportedEncodingException

PrintWriter (File *объект\_файла*) throws FileNotFoundException

PrintWriter (File *объект\_файла*, String *набор\_символов*)

throws FileNotFoundException, UnsupportedEncodingException

В любом случае файл создается автоматически. Конкретную кодировку символов можно указать параметром *набор\_символов*.

Он представляет такие методы как println(), print(), format(), append() и printf(). Ниже представлен пример использования данного класса:

package myPackage;

import java.io.\*;

public class Main {

public static void main(String[] args) {

try(PrintWriter writer=new PrintWriter(new File("java.txt"))) {

writer.println("adc");

writer.print("Число: "+12);

writer.append("3");

writer.printf("\nИз метода printf() %d %s %.2f", 235 , "Number\_\_\_", 346.64);

} catch (FileNotFoundException e) {

System.out.println(e);

}

try(PrintWriter writer=new PrintWriter(System.out)){

writer.println("Из PrintWriter на основе System.out");

}

}

}

Сериализация.

*Сериализация* это процесс записи состояния объекта в файл. Она требуется когда нужно сохранить состояние объекта в место постоянного хранения - файл. После сериализации следует применить процесс десериализации.

Если на вершине иерархия класс сериализуется, то рекурсивным путем сериализуются и его подклассы, а после восстановятся в процессе десериализации.

Интерфейс Serializable.

Сериализоваться может только тот объект, который реализует интерфейс Serializable. В нем не определяется членов и он служит маркером. Переменный типа transient и статические поля не могут сериализоваться и десериализоваться.

Интерфейс Externalizable.

Процесс сериализации и десериализации происходят автоматически, но если есть нужда то можно управлять самостоятельно процессов сериализации и десериализации. Его удобно к примеру можно использовать если нужно использовать алгоритм сжатия и/или шифрования данных.

В нем определяется два метода:

void readExternal (ObjectInput *поток\_ввода*) throws IOException, ClassNotFoundException

void writeExternal (ObjectOutput *поток\_вывода*) throws IOException, ClassNotFoundException.

Иерархия классов и интерфейсов.

Интерфейс ObjectOutput отвечает за сериализацию объекта. Он расширяет интерфейсы AutoCloseable и DataOutput. Среди его методов стоит отметить writeObject(), который записывает *объект* в поток ввода т.е. сериализует. Полное объявление:

void writeObject (Object *объект*)

Класс ObjectOutputStream расширяет класс OutputStream и реализует ObjectOutput. Этот класс отвечает за ввод объекта. Конструктор данного класса:

ObjectOutputStream ( OutputStream *поток\_ввода*) throw IOException

Опять здесь стоит отметить метод writeObject(), но стоит отметить, что он реализует методы writeUTF(), writeInt () и другие. Но их можно использовать и записывать в один файл два объекта и число свободно.

Интерфейс ObjectInput отвечает за десериализацию объекта. Он расширяет интерфейсы AutoCloseable и DataInputStream. Среди его методов следует отметить метод readObject(), который восстанавливает объект. Полное объявление метода:

Object readObject ()

Класс ObjectInputStream расширяет класс InputStream и реализует ObjectInput. Этот класс отвечает за десериализацию объекта. Конструктор класса:

ObjectInputStream (InputStream *поток\_вывода*)

Среди методов также стоит отметить метод readObject(), хотя у него также есть методы типа readUTF(), readShort(), которые работают также как в ObjectInputStream.

Пример сериализации.

В нижнее приведен пример использования сериализации и десериализации. Класс User содержит 3 параметра: int – уровень, String – имя, int – hashCode(). Стоит отметить если класс, который будет сериализован не будет реализовывать интерфейс Serializable сгенерируется исключение NotSerializableException.

Сначала создается объект user класса User. Поток out типа FileOutputStream, ссылается на файл objFile, в который будет записано состояние объекта user. После out передаются аргументов в конструктор потока objOutStr типа ObjectOutputStream. Здесь user записывается в objOutStr методом writeObject().

Далее создается поток in типа FileInputStream, ссылающийся на файл objFile. Поток in передается в параметры в поток ObjectInputStream objInStr. Методом readObject () происходит десериализация объекта user.

package myPackage;

import java.io.\*;

public class Main {

public static void main(String[] args) {

User user=new User(43, "Петя");

try(FileOutputStream out = new FileOutputStream("objFile");

ObjectOutputStream objOutStr=new ObjectOutputStream(out)) {

objOutStr.writeObject(user);

} catch (FileNotFoundException e) {

System.out.println("Файл не найден и/или закрыт");

} catch (IOException e) {

System.out.println("Ошибка ввода-вывода: "+e);

}

try(FileInputStream in=new FileInputStream("objFile");

ObjectInputStream objInStr=new ObjectInputStream(in)) {

User user1=(User) objInStr.readObject();

System.out.println(user1);

} catch (FileNotFoundException e) {

System.out.println("Файл не найден и/или закрыт");

} catch (IOException e) {

System.out.println("Ошибка ввода-вывода: "+e);

} catch (ClassNotFoundException e) {

System.out.println("Класс не найден");

}

}

}

Класс User:

package myPackage;

import java.io.\*;

public class User implements Serializable {

int level;

String name;

int hashCode = this.hashCode();

public User(int level, String name) {

this.level = level;

this.name = name;

}

@Override

public String toString() {

return "Персонаж: " + name + ", уровень" + level + ". @" + hashCode;

}

}

Библиотека java.nio.

Устройство.

Новая система ввода-вывода опирается на буфера и каналы. Основное преимущество является возможность масштабирования потоков, возможность отсутствия простоев в выполнение.

Буфера определяются в пакете java.nio. Все буфера являются подклассами, производными от класса Buffer, с основными возможностями. К ним относят:

* *Текущая позиция* - индекс в буфере, с которого в следующий раз начнется операция чтения или записи данных.
* *Предел* - индекс за позицией последней доступной ячейки в буфере.
* *Емкость* - количество элементов, которые можно хранить в буфере.

У класса Buffer определяют ряд методов:

final int capacity() - Возвращает количество элементов, которые можно хранить

в вызывающем буфере

final Buffer clear() - Очищает вызывающий буфер и возвращает ссылку на него

final Buffer flip() - Задает текущую позицию в качестве предела для вызывающего буфера и затем устанавливает текущую позицию в нуль. Возвращает ссылку на буфер.

final boolean hasRemaining()- Проверяет на наличие байтов в буфере.

abstract boolean isDirector() – Проверят на директорию.

abstract boolean isReadOnly() – Проверяет на возможность буфера к чтению.

final int limit() – Возвращает лимит буфера.

final Buffer limit(int *лимит*) – задает лимит для буфера.

От класса Buffer происходят приведенные ниже классы конкретных буферов, где тип хранимых данных можно определить по их именам. Класс MappedByteBuffer является производным от класса ByteBuffer и используется для сопоставления файла с буфером. Каждый буфер представляет свои версии методов put() и get().

Каналы.

*Канал* - представляет открытое соединение с источником или адресатом ввода-вывода. Следует сказать, что для каналов можно использовать try с ресурсами.

Один из способов получить канал это метод getChannel() для объекта, который поддерживает канал. Конкретный тип возвращаемого канала зависит от типа объекта, для которого вызывается метод getChannel(). Например, когда метод getChannel() вызывается для объекта типа FileInputStream, FileOutputStream или RandomAccessFile, он возвращает канал типа FileChannel.

Еще способ получить канал использовать статистический метод класса Files newByteChannel().Он возвращает канал типа SeekableByteChannel, т.е. интерфейса, реализуемого классом FileChannel.

В каналах типа FileChannel и SocketChannel поддерживаются различные методы read() и write(), которые позволяют выполнять операции ввода-вывода через канал. Также в классе FileChannel определен ряд методов, который помогают управлять каналом. К ним относят:

* open() – открывает файл и возвращает для него канал.
* map() – помогает сопоставить файл с каналом.

Набор символов и селекторы.

*Набор символов* определяет способ сопоставления байтов с символами. С помощью *кодера* можно закодировать последовательность символов в виде байтов. Процесс декодирования производится с помощью *декодера*. Кодеры и декодеры используется по умолчанию.

*Селектор* - обеспечивает возможность многоканального ввода-вывода по ключам, не прибегая к блокировке. Иными словами, с помощью селекторов можно выполнять операции ввода-вывода через несколько каналов.

Чтение из файла с помощью java.nio.

Пример чтение из файла:

package com.company;

import java.io.IOException;

import java.nio.ByteBuffer;

import java.nio.channels.FileChannel;

import java.nio.channels.SeekableByteChannel;

import java.nio.file.\*;

public class Main {

public static void main(String[] args) {

int count;

try (SeekableByteChannel fChan = Files.newByteChannel(Paths.get("test.txt"))){

ByteBuffer mBuf = ByteBuffer.allocate(128);

do {

count = fChan.read(mBuf);

if(count != -1) {

mBuf.rewind();

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

System.out.print((char)mBuf.get());

}

} while(count != -1);

System.out.println();

} catch (IOException e) {

System.out.println("Ошибка ввода-вывода " + e.toString());

}

}

}

В начала создается объект, который служит как канал. В нашем случае используется класс SeekableByteChannel. Статистический метод newByteChannel(), в который мы передаем ссылку на файл. Все выше описание действует в try с ресурсами:

try (SeekableByteChannel fChan = Files.newByteChannel(Paths.get("test.txt"))){

После этого нам нужен буфер. Во время его созданию мы задаем емкость 128. Следует учитывать, что метод read также возвращает -1 при конце чтения. В коде используется цикл do-while. В нем переменной count присваивается результат чтения файла к буферу:

count = fChan.read(mBuf);

Дальше при условии count != -1 мы возвращаем содержимое буферу по штучно. Следует обратить внимание на метод rewind() мы подготавливаемся к чтению.

Дальше обрабатываются исключения.

Есть также возможность еще сопоставить. Для этого нужно использовать класс MappedByteBuffer и вызвать fChan метод map(). Следует учитывать при сопоставление используется FileChannel. Код:

package com.company;

import java.io.IOException;

import java.nio.MappedByteBuffer;

import java.nio.channels.FileChannel;

import java.nio.file.\*;

public class Main {

public static void main(String[] args) {

// получить канал к файлу в блоке оператора try с ресурсами

try (FileChannel fChan =

(FileChannel) Files.newByteChannel(Paths.get("test.txt"),

StandardOpenOption.CREATE\_NEW,

StandardOpenOption.READ) ) {

long fSize = fChan.size();

MappedByteBuffer mBuf =

fChan.map(FileChannel.MapMode.READ\_ONLY, 0, fSize);

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

System.out.print((char)mBuf.get());

System.out.println();

} catch(InvalidPathException e) {

System.out.println("Ошибка указания пути " + e);

} catch (IOException e) {

System.out.println("Ошибка ввода-вывода " + e);

}

}

}

В аргументы map() передаются атрибуты файла, индекс старта чтения и размер.

Запись в файл

Ниже представлен пример записи в файл, а после чтения из него:

package myPackage;

import java.io.IOException;

import java.nio.\*;

import java.nio.channels.FileChannel;

import java.nio.channels.SeekableByteChannel;

import java.nio.file.Files;

import java.nio.file.Paths;

import java.nio.file.StandardOpenOption;

public class Main{

public static void main(String [] args){

try(FileChannel channel=(FileChannel) Files.newByteChannel(

Paths.get("test.txt"),

StandardOpenOption.CREATE,

StandardOpenOption.WRITE)){

ByteBuffer buffer=ByteBuffer.allocate(26);

for(char ch=0; ch<26; ch++){

buffer.put((byte) ('A'+ch));

}

buffer.rewind();

channel.write(buffer);1

buffer.clear();

}catch (IOException exc){

System.out.println(exc);

}

try(SeekableByteChannel channelRead=Files.newByteChannel(Paths.get("test.txt"))){

ByteBuffer buffer=ByteBuffer.allocate(28);

int count;

do{

count=channelRead.read(buffer);

if(count != -1){

buffer.rewind();

for(int i=0; i<26; i++){

System.out.println("Символ "+i+": "+(char)buffer.get());

}

}

}while (count != -1);

}catch (IOException exc){

System.out.println(exc);

System.exit(1);

}

}

}

Здесь используется FileChannel. Так как Files.newByteChannel() возвращает SeekableByteChannel то мы, приводим результат к FileChannel. В параметры мы передаем режим StandardOpenOption.WRITE, указывая, что канал должен только записывать.

try(FileChannel channel=(FileChannel) Files.newByteChannel(

Paths.get("test.txt"),

StandardOpenOption.CREATE,

StandardOpenOption.WRITE)){

В буфер мы складываем все буквы английского алфавита. А после подготавливаем буфер к записи метод rewind().

buffer.rewind();

Методом write() мы записываем в канал буфер.

channel.write(buffer);

После записи всегда следует снова подготовить буфер к чтению:

buffer.rewind();

Далее чистится буфер методом clear():

buffer.clear();

Ниже представлен пример сопоставления с буфером:

package myPackage;

import java.io.IOException;

import java.nio.\*;

import java.nio.channels.FileChannel;

import java.nio.file.Files;

import java.nio.file.Paths;

import java.nio.file.StandardOpenOption;

public class Main{

public static void main(String [] args){

String str="Java.nio 2";

byte [] bytes=str.getBytes();

try(FileChannel channel=(FileChannel) Files.newByteChannel(

Paths.get("test2.txt"),

StandardOpenOption.WRITE,

StandardOpenOption.READ,

StandardOpenOption.CREATE)){

MappedByteBuffer byteBuffer=

channel.map(FileChannel.MapMode.READ\_WRITE,

0, 10);

byteBuffer.put(bytes);

}catch (IOException exc){

System.out.println(exc);

}

}

}

Пример записи и чтения.

Ниже представлен пример записи и чтения файла с помощью системы java.nio:

package com.company;

import java.io.IOException;

import java.nio.ByteBuffer;

import java.nio.channels.FileChannel;

import java.nio.channels.SeekableByteChannel;

import java.nio.file.\*;

public class Main {

public static void main(String[] args) {

int count;

ByteBuffer mBuf = ByteBuffer.allocate(26);

// получить канал к файлу в блоке оператора try с ресурсами

try (FileChannel fChan = (FileChannel)

Files.newByteChannel(Paths.get("abc.txt"),

StandardOpenOption.WRITE, StandardOpenOption.CREATE)) {

for (int i = 0; i < 26; i++) {

mBuf.put((byte) ('A' + i));

}

mBuf.rewind();

// записать данные из буфера в выходной файл

fChan.write(mBuf);

} catch (InvalidPathException e) {

System.out.println("Ошибка указания пути: " + e);

} catch (IOException e) {

System.out.println("Ошибка ввода-вывода: " + e);

System.exit(1);

}

try (FileChannel channelReadABC = (FileChannel) Files.newByteChannel(Paths.get("abc.txt"))) {

do {

count = channelReadABC.read(mBuf);

if (count != -1) {

mBuf.rewind();

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

System.out.print((char) mBuf.get());

}

}

} while (count != -1);

} catch (InvalidPathException exc) {

System.out.println("Ошибка указания пути: " + exc);

} catch (IOException exc) {

System.out.println("Ошибка ввода-вывода: " + exc);

}

try {

Path source = Paths.get("abc.txt");

Path target = Paths.get("abc2.txt");

Files.copy(source, target);

} catch (InvalidPathException exc) {

System.out.println();

} catch (IOException exc) {

}

int count1;

ByteBuffer buffer = ByteBuffer.allocate(128);

try (FileChannel channel3 = (FileChannel) Files.newByteChannel(Paths.get("text.txt"),

StandardOpenOption.CREATE\_NEW,

StandardOpenOption.WRITE)) {

String text = "Text";

byte[] bytes = text.getBytes();

buffer.put(bytes);

buffer.rewind();

channel3.write(buffer);

} catch (IOException exc) {

System.out.println();

} catch (InvalidPathException exc) {

System.out.println();

}

int countd;

try(FileChannel channelReader=(FileChannel)Files.newByteChannel(

Paths.get("text.txt"))){

do{

countd=channelReader.read(buffer);

if(countd != -1){

buffer.rewind();

for(int i=0; i< countd; i++){

System.out.print((char)buffer.get(i));

}

}

} while (countd != -1);

}catch (IOException exc){

System.out.println(exc);

}

}

}

Копирование файла

А теперь было бы неплохо скопировать файл верхний программы. Для этих целей используется статистический метод copy() из класса Files. Полное объявление метода:

static Path copy(Path *откуда*, Path *куда*, CopyOption … *способ*)

Где *способ* определяется способ копирования файла. Способы:

StandardCopyOption.COPY\_ATTRIBUTES - Запросить копирование атрибутов файла.

StandardCopyOption.NOFOLLOW\_LINKS – Не следовать по символическим ссылкам.

StandardCopyOption.REPLACE\_EXISTING - Перезаписать прежний файл.

Ниже представлен пример переделанной выше программы под копирование:

package myPackage;

import java.io.IOException;

import java.nio.\*;

import java.nio.channels.FileChannel;

import java.nio.file.\*;

public class Main{

public static void main(String [] args){

String str="Java.nio 2";

byte [] bytes=str.getBytes();

try(FileChannel channel=(FileChannel) Files.newByteChannel(

Paths.get("test2.txt"),

StandardOpenOption.WRITE,

StandardOpenOption.READ,

StandardOpenOption.CREATE)){

MappedByteBuffer byteBuffer=

channel.map(FileChannel.MapMode.READ\_WRITE,

0, 10);

byteBuffer.put(bytes);

}catch (IOException exc){

System.out.println(exc);

}

try{

Path source=Paths.get("test2.txt");

Path target=Paths.get("test2Copy.txt");

Files.copy(source, target, StandardCopyOption.REPLACE\_EXISTING);

}catch (InvalidPathException exc){

exc.printStackTrace();

} catch (IOException e) {

e.printStackTrace();

}

}

Перемещение файла

Для того чтоб переместить файл следует воспользоваться статическим методом move() из класса Files. Полное объявление:

static Path move(Path *source*, Path *target*, CopyOption … *способ*)

Где *способ* такие же значения, как в копирование. Ниже представлен вариант использования метода move():

package myPackage;

import java.io.IOException;

import java.nio.\*;

import java.nio.channels.FileChannel;

import java.nio.file.\*;

public class Main {

public static void main(String[] args) {

Path source=Paths.get("java.txt");

Path target=Paths.get("newJava.txt");

try(FileChannel channel=

(FileChannel) Files.newByteChannel(source, StandardOpenOption.WRITE, StandardOpenOption.CREATE\_NEW)){

ByteBuffer buffer=ByteBuffer.allocate(26);

for(int i=0; i<26; i++) {

buffer.put((byte) ('A' +i));

}

buffer.rewind();

channel.write(buffer);

buffer.clear();

} catch (IOException e) {

e.printStackTrace();

}

try{

Files.move(source, target, StandardCopyOption.REPLACE\_EXISTING);

System.out.println(target.getFileName());

}catch (IOException exc){

}

}

}

Удаление файла.

Для удаления файла можно использовать статистический метод delete(), с полным объявлением:

static void delete(Path *путь*)

Если же есть возможность того что файла не будет можно воспользоваться методом deleteIfExists(), с полным объявлением:

boolean deleteIfExists(Path *путь*)

Ниже представлен пример удаления файла:

package myPackage;

import java.io.IOException;;

import java.nio.ByteBuffer;

import java.nio.channels.FileChannel;

import java.nio.file.\*;

public class Main {

public static void main(String[] args){

Path source=Paths.get("java2.txt");

try(FileChannel channel=

(FileChannel) Files.newByteChannel(

source,

StandardOpenOption.WRITE,

StandardOpenOption.CREATE\_NEW)){

ByteBuffer buffer= ByteBuffer.allocate(26);

for(int i=0; i<26; i++) {

buffer.put((byte) ('A' +i));

}

buffer.rewind();

channel.write(buffer);

buffer.clear();

} catch (IOException e) {

e.printStackTrace();

}

if(Files.exists(source))

System.out.println("Файл существует");

try{

Files.delete(source);

System.out.println("Файл удален");

}catch (IOException exc){

System.out.println(exc);

}

}

}

Применение java.nio для операций с файлами

Получения информации о файле

Для операции сведений о файле в классе Files определен ряд статических методов. Пример:

package myPackage;

import java.io.IOException;

import java.nio.\*;

import java.nio.channels.FileChannel;

import java.nio.file.\*;

import java.nio.file.attribute.BasicFileAttributes;

public class Main{

public static void main(String [] args){

String str="Java.nio 2";

byte [] bytes=str.getBytes();

try(FileChannel channel=(FileChannel) Files.newByteChannel(

Paths.get("test2.txt"),

StandardOpenOption.WRITE,

StandardOpenOption.READ,

StandardOpenOption.CREATE)){

MappedByteBuffer byteBuffer=

channel.map(FileChannel.MapMode.READ\_WRITE,

0, 10);

byteBuffer.put(bytes);

}catch (IOException exc){

System.out.println(exc);

}

Path filePath=Paths.get("test2.txt");

System.out.println("Имя: "+filePath.getFileName());

System.out.println("Путь: "+filePath);

System.out.println("Полный путь: "+filePath.toAbsolutePath());

if(Files.exists(filePath))

System.out.println("Файл существует.");

else

System.out.println("Файл не существует");

try {

if(Files.isHidden(filePath))

System.out.println("Файл скрыт");

else

System.out.println("Файл открытый");

} catch (IOException e) {

e.printStackTrace();

}

if(Files.isReadable(filePath))

System.out.println("Файл доступен для чтения");

if (Files.isWritable(filePath))

System.out.println("Файл доступен для записи");

try {

BasicFileAttributes attributes =

Files.readAttributes(filePath, BasicFileAttributes.class);

if (attributes.isDirectory())

System.out.println("Это каталог");

else

System.out.println("Это не каталог");

if (attributes.isRegularFile())

System.out.println("Это обычный файл");

else

System.out.println("Это не обычный файл");

if (attributes.isSymbolicLink())

System.out.println("Это символическая ссылка");

else

System.out.println("Это не символическая ссылка");

System.out.println("Время последней модификации: "+attributes.lastModifiedTime());

System.out.println("Размер файла: " + attributes.size() + "байтов");

}catch (IOException exc){

System.out.println("Ошибка чтения атрибутов: "+exc);

}

}

}

Обход каталога

Чтоб получить содержимое каталога следует создать объект типа DirectoryStream<Path> и дальше его обойти циклом for-each. Чтоб получить его следует вызвать статистический метод newDirectoryStream из класса Files. Полное объявление данного метода:

static DirectoryStream<Path> newDirectoryStream(Path *путь\_к\_каталогу*) throws IOException

Здесь *путь\_к\_каталогу* обозначает путь к конкретному каталогу, а если каталог не является, каталогом генерируется NotDirectoryStream.

Пример обхода каталога:

package myPackage;

import java.io.IOException;

import java.nio.file.\*;

import java.nio.file.attribute.BasicFileAttributes;

public class Main{

public static void main(String [] args){

String dirPath= "\\Users";

try(DirectoryStream<Path> dirStream=

Files.newDirectoryStream(Paths.get(dirPath))){

System.out.println("Каталог "+dirPath);

for (Path path : dirStream) {

BasicFileAttributes attributes =

Files.readAttributes(

path, BasicFileAttributes.class);

if(attributes.isDirectory())

System.out.print("<DIR> ");

else

System.out.print(" ");

System.out.println(path.getName(1));

}

} catch (IOException e) {

e.printStackTrace();

}

}

}

Чтоб профильтровать директорию можно использовать два способа. Первый заключается в использовании метода newDirectoryStream() в котором передают регулярное выражение:

static DirectoryStream<Path> newDirectoryStream(Path *каталог*, String *шаблон*)

Здесь файлы будут фильтроваться по *шаблон*. Второй же заключается в использования варианта метода newDirectoryStream() в котором передают интерфейс DirectoryStream.Filter:

static DirectoryStream<Path> newDirectoryStream(

Path *каталог*, DirectoryStream. Filter<? super Path> *фильтр\_файлов*)

В данном интерфейсе объявляется метод accept ():

public boolean accept(T *элемент*)

Плюс второго способа в том, что фильтровать можно по размеру, содержимому, модификаторам и т.п. Ниже представлена переделанная верхняя программа:

package myPackage;

import java.io.IOException;

import java.nio.file.DirectoryStream;

import java.nio.file.Files;

import java.nio.file.Path;

import java.nio.file.Paths;

import java.nio.file.attribute.BasicFileAttributes;

public class Main{

public static void main(String[] args) {

DirectoryStream.Filter<Path> dirFilter = (Path entry) -> {

if(Files.isWritable(entry)) return true;

return false;

};

String dirPath= "\\Users";

try(DirectoryStream<Path> dirStream=

Files.newDirectoryStream(Paths.get(dirPath), dirFilter)){

System.out.println("Каталог "+dirPath);

for (Path path : dirStream) {

BasicFileAttributes attributes =

Files.readAttributes(path, BasicFileAttributes.class);

if(attributes.isDirectory())

System.out.print("<DIR> ");

else

System.out.print(" ");

System.out.println(path.getName(1));

}

} catch (IOException e) {

e.printStackTrace();

}

}

}

Обход дерева методом walkFileTree().

Для обхода сразу дерева каталогов в классе Files определен статистический метод walkFileTree(), с полным объявлением:

static Path walkFileTree(Path *корень*, FileVisitors <? extends Path> *посетитель*)

Где *корень* – это исходная точка обхода. А где *посетитель* экземпляр интерфейс FileVisitors. В FileVisitors определен ряд методов, а FileVisitResult ряд значений. Но определять все методы не следует. А где *посетитель* может быть объект класса, который расширяет SimpleFileVisit и определяет метод visitFile().

Ниже представлен пример обхода дерева:

package myPackage;

import java.io.IOException;

import java.nio.file.\*;

import java.nio.file.attribute.BasicFileAttributes;

public class Main{

public static void main(String[] args){

System.out.println("Дерево с Users: “);

try{

Files.walkFileTree(Paths.get(“\\Users”), new MyFileVisitors());

}catch (IOException exc){

System.out.println(exc);

}

}

}

class MyFileVisitors extends SimpleFileVisitor<Path>{

@Override

public FileVisitResult visitFile(Path file, BasicFileAttributes attrs)

throws IOException {

System.out.println(file);

return FileVisitResult.CONTINUE;

}

}

Совместимость с java.io

В java nio 2 использовать для открытия потоков ввода-вывода. Сначала нужно получить Path, а после вызвав статические методы newOutputStream() или newInputStream() из класса Files. Полные объявления данных методов:

static InputStream newInputStream(Paths *путь*, OpenOption … *способ*)

static OutputStream newOutputStream(Paths *путь*, OpenOption … *способ*)

Где *способ* одно из значений StandardOpenOption, которые применимы для java.io. Если он *способ* не указан то выбирается StandardOpenOption.READ.

Ниже представлен пример записи и чтения из файла:

package myPackage;

import java.io.\*;

import java.nio.file.Files;

import java.nio.file.Paths;

public class Main{

public static void main(String[] args){

try(OutputStream stream=

new BufferedOutputStream(

Files.newOutputStream(

Paths.get("java.txt")))) {

for(int i=0; i<26; i++)

stream.write((byte) ('A'+ i));

} catch (IOException e) {

e.printStackTrace();

}

try(InputStream stream = new BufferedInputStream(

Files.newInputStream(

Paths.get("java.txt")))) {

int count;

System.out.println("Алфавит: ");

do{

count = stream.read();

if (count != -1) {

System.out.print((char) count+" ");

}

}while (count != -1);

} catch (IOException e) {

e.printStackTrace();

}

}

}