



Девятая лекция

Понятие потоков ввода/вывода

Потоком ввода/вывода (I/O Stream) называется произвольный источник или приемник, который способен генерировать либо получать некоторые данные.

Другими словами поток это процесс передачи от источника и/или к приемнику различных файлов, обмен информацией по сети, ввод-вывод в консоли и т. д .

К примеру у нас может быть определен поток, который связан с файлом и через который мы можем вести чтение или запись файла. Это также может быть поток, связанный с сетевым сокетом, с помощью которого можно получить или отправить данные в сети.

Все эти задачи: чтение и запись различных файлов, обмен информацией по сети, ввод-вывод в консоли мы будем решать в Java с помощью потоков.

Все потоки ведут себя одинаковым образом, хотя физические устройства, с которыми они связаны, могут сильно различаться.

Реализация конкретным потоком низкоуровневого способа приема/передачи информации скрыта от программиста

Подсистема ввода/вывода Java

Объект, из которого можно считать данные, называется потоком ввода, а объект, в который можно записывать данные, - потоком вывода. Например, если надо считать содержание файла, то применяется поток ввода, а если надо записать в файл - то поток вывода.

Основная подсистема ввода/вывода Java представлена пакетом **java.io**. В JDK 7 добавлен более современный способ работы с потоками под названием Java NIO или Java New IO эти классы лежат в пакете **java.nio.***.

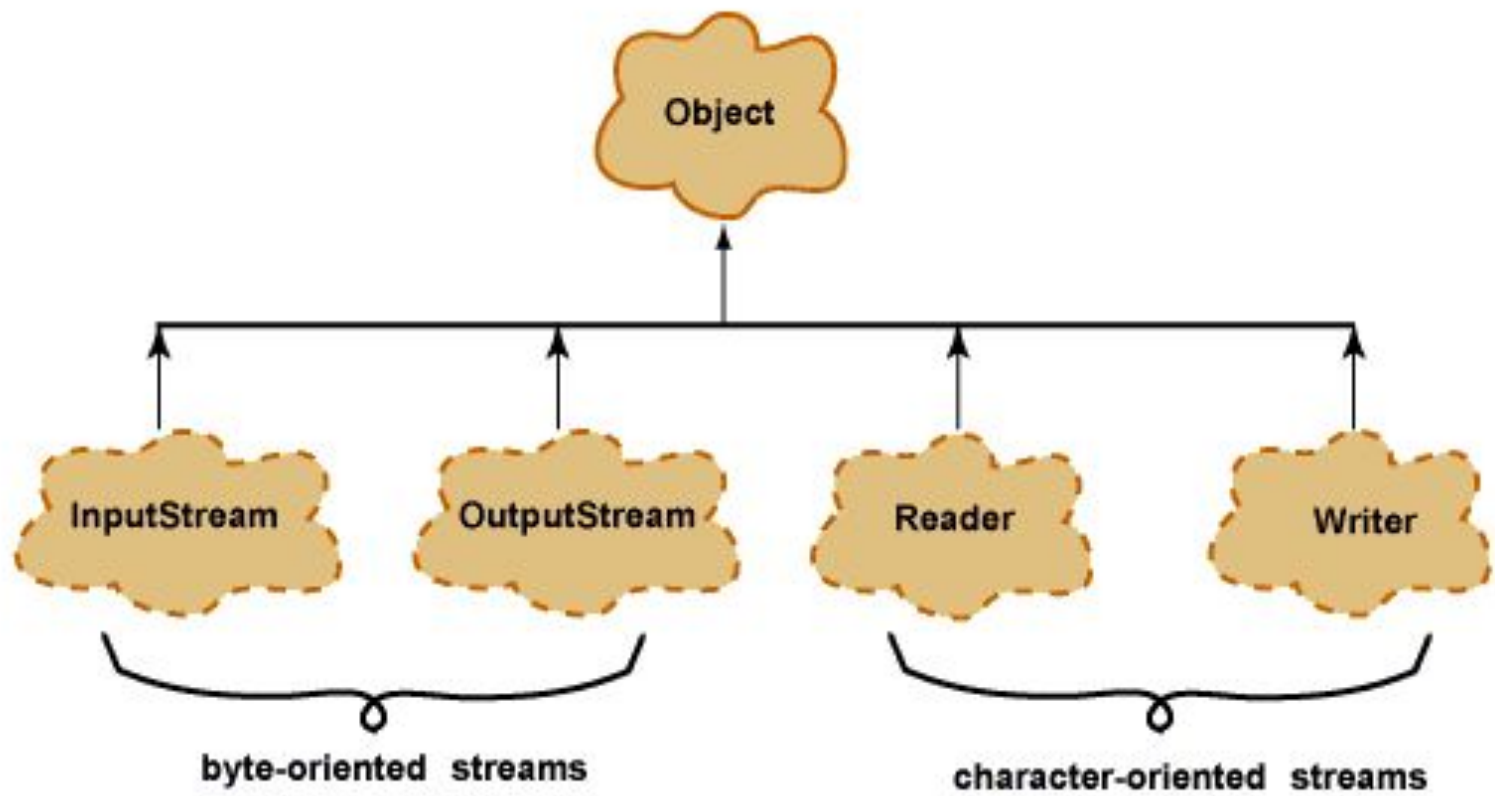
Java поддерживает два типа потоков – символные и байтовые

В основе всех классов, управляющих потоками байтов, находятся два абстрактных класса:

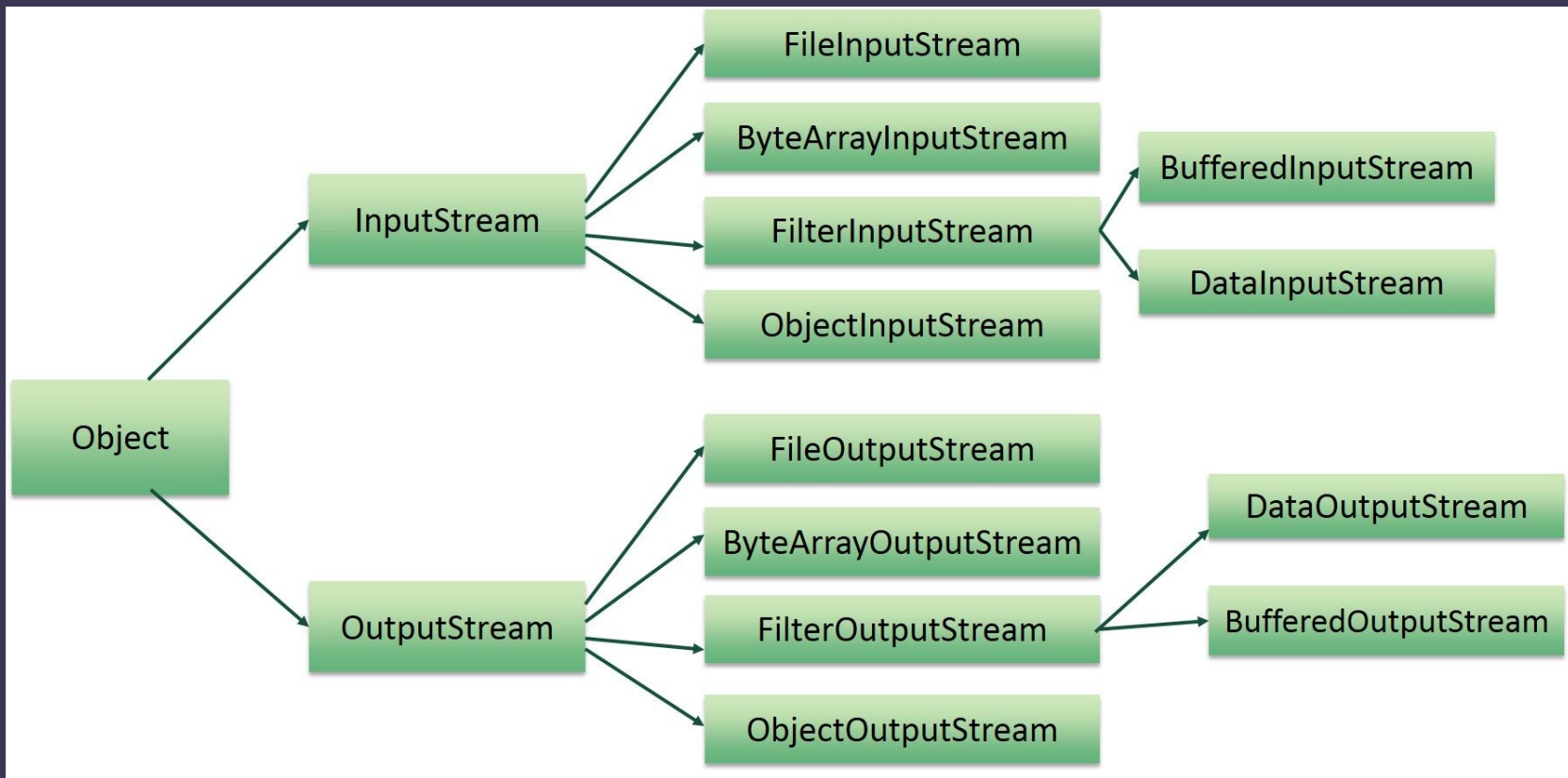
InputStream (представляющий потоки ввода) и
OutputStream (представляющий потоки вывода)

Для работы с потоками символов были добавлены абстрактные классы **Reader** (для чтения потоков символов) и **Writer** (для записи потоков символов).

Все остальные классы, работающие с потоками, являются наследниками этих абстрактных классов.



ОСНОВНЫЕ КЛАССЫ БАЙТОВЫХ ПОТОКОВ:



Класс InputStream

Абстрактный класс **InputStream** предоставляет минимальный набор методов для работы с входным потоком **байтов**:

int available() - Возвращает количество еще доступных байт потока

int read() - Возвращает очередной байт. Значения от 0 до 255. Если чтение невозможно, возвращает -1

int read(byte[] buf, int offset, int count) - Вводит байты в массив. Возвращает количество реально введенных байтов

long skip(long n) - Пропускает n байтов потока

void close() - Закрывает поток и освобождает занятые системные ресурсы

Все методы класса предназначены для чтения байт, при возникновении ошибки они возбуждают исключение `IOException`.

Потомки класса `InputStream`

`ObjectInputStream` - поток объектов. Создается при сохранении объектов системными средствами.

`DataInputStream` - Форматированное чтение из памяти.

`BufferedInputStream` - накапливает вводимые данные в специальном буфере без постоянного обращения к устройству ввода.

`ByteArrayInputStream` - использует массив байтов как источник данных

`FileInputStream` - Класс `FileInputStream` создаёт объект класса `InputStream`, который можно использовать для чтения байтов из файла.

`FilterInputStream` - абстрактный класс надстройки, которые добавляют к существующим потокам полезные свойства. Объект `FilterInputStream` получает ввод от другого объекта `InputStream`, некоторым образом обрабатывает(фильтрует) байты и возвращает результат. Фильтрующие потоки могут объединяться в последовательности, при этом несколько фильтров превращаются в один сквозной фильтр.

Класс `OutputStream`

Абстрактный класс `OutputStream` предоставляет минимальный набор методов для работы с выходным потоком **байтов**

`void write(int b)` - Абстрактный метод записи в поток одного байта

`void write(byte[] buf, int offset, int count)` - Запись в поток массива байтов или его части

`void flush()` - Форсированная выгрузка буфера для буферизированных потоков. Если получателем служит другой поток, его буфер тоже сбрасывается

`void close()` - Закрытие потока и высвобождение системных ресурсов

Потомки класса OutputStream

ObjectOutputStream - поток двоичных представлений объектов. Создается при сериализации

BufferedOutputStream - накапливает выводимые байты без постоянного обращения к устройству. И когда буфер заполнен, производится запись данных.

ByteArrayOutputStream - использует массив байтов как приемник данных

DataOutputStream - Форматированное чтение в память

FileOutputStream - Класс FileOutputStream создаёт объект класса OutputStream, который можно использовать для записи байтов в файл. Создание нового объекта не зависит от того, существует ли заданный файл, так как он создаёт его перед открытием. В случае попытки открытия файла, доступного только для чтения, будет передано исключение.

FilterOutputStream - абстрактный класс надстройки над классом OutputStream, которые добавляют к существующим потокам полезные свойства.

Существует множество классов и методов для чтения и записи файлов. Наиболее распространённые из них — классы `FileInputStream` и `FileOutputStream`, которые создают байтовые потоки, связанные с файлами. Чтобы открыть файл, нужно создать объект одного из этих классов, указав имя файла в качестве аргумента конструктора.

`FileInputStream(String filename)` throws `FileNotFoundException`
`FileOutputStream(String filename)` throws `FileNotFoundException`

В `filename` нужно указать имя файла, который вы хотите открыть. Если при создании входного потока файл не существует, передаётся исключение `FileNotFoundException`. Аналогично для выходных потоков, если файл не может быть открыт или создан, также передаётся исключение.

```
try {
    InputStream from = new FileInputStream("testic");
    OutputStream to = new FileOutputStream("testic2");
    while (from.available() > 0) {
        int s1 = from.read();
        to.write(s1);
    }
    from.close();
    to.close();
} catch (IOException e) {
    //Обрабатываем ошибки
}
```

Надстройки над потоками

Классы-настройки это классы которые добавляют к существующим потокам полезные дополнительные свойства.

PrintStream – Класс `PrintStream` - это именно тот класс, который используется для вывода на консоль. Когда мы выводим на консоль некоторую информацию с помощью вызова `System.out.println()`, то тем самым мы задействует **PrintStream**, так как переменная `out` в классе `System` как раз и представляет объект класса `PrintStream`, а метод `println()` - это метод класса `PrintStream`.

BufferedOutput/InputStream – буферизированный выходной поток. Ускоряет вывод. Это класс часто используются в сочетании с файловыми потоками — работа с файлом на диске происходит сравнительно медленно, и буферизация позволяет сократить количество обращений к физическому носителю. При создании буферизованного потока можно явно задать размер буфера или положиться на значение, принятое по умолчанию. Буферизованный поток использует массив тип `byte` для промежуточного хранения байтов, проходящих через поток.

```
OutputStream out = new FileOutputStream(path);  
BufferedOutputStream return bout = new BufferedOutputStream(out);
```

DataOutput/inputStream - поток для вывода значений простых типов. Имеет такие методы как `writeBoolean()`, `writeInt()`, `writeLong()`, `writeFloat()` и т. п. Для успешного чтения таких данных из потока `DataInputStream` они должны быть предварительно записаны с помощью соответствующих методов `DataOutputStream` в том же порядке.

Пример использования PrintStream

PrintStream полезен не только для вывода на консоль. Мы можем использовать данный класс для записи информации в поток вывода.

В качестве потока вывода используется объект FileOutputStream. С помощью метода println() производится запись информации в выходной поток - то есть в объект FileOutputStream.

```
import java.io.*;

public class FilesApp {
    public static void main(String[] args) {
        String text = "Привет мир - !!!"; // строка для записи
        try{
            OutputStream fos=new FileOutputStream("/home/ansash/1/testic");
            PrintStream printStream = new PrintStream(fos);
            printStream.println(text);
            System.out.println("Запись в файл произведена");
            printStream.close();
            fos.close();
        }catch(IOException ex){
            System.out.println(ex.getMessage());
        }
    }
}
```

Буферизированный ввод/вывод

```
public class FileCopy {
    public static void main(String[] args) {
        try {
            BufferedInputStream bin = new BufferedInputStream(new
            FileInputStream("one.jpg"));
            BufferedOutputStream bout = new BufferedOutputStream(new
            FileOutputStream("two.jpg"));

            int c = 0;
            while (true) {
                c = bin.read();
                if (c != -1)
                    bout.write(c);
                else
                    break;
            }
            bin.close();
            bout.flush(); //освобождаем буфер (принудительно записываем
            содержимое буфера в файл)
            bout.close(); //закрываем поток записи (обязательно!)
        }
        catch (java.io.IOException e) {
            System.out.println(e.toString());
        }
    }
}
```

При завершении работы с потоком его надо закрыть с помощью метода `close()`. Этот метод уже реализуется в классах **InputStream** и **OutputStream**, а через них и во всех классах потоков.

При закрытии потока освобождаются все выделенные для него ресурсы, например, файл. Поскольку при открытии или считывании файла может произойти ошибка ввода-вывода, то код считывания помещается в блок `try`. И чтобы быть уверенным, что поток в любом случае закроется, даже если при работе с ним возникнет ошибка, вызов метода `close()` помещается в блок `finally`. И, так как метод `close()` также в случае ошибки может генерировать исключение `IOException`, то его вызов также помещается во вложенный блок `try..catch`

```
finally{
    try{
        fin.close();
    }
    catch(IOException ex){

        System.out.println(ex.getMessage());
    }
}
```

Или...

```
try(FileInputStream fin=new FileInputStream("C://SomeDir//Hello.txt");
    FileOutputStream fos = new FileOutputStream("C://SomeDir//Hello2.txt"))
{
    //.....
}
```

```
public class FilesApp {
    public static void main(String[] args) {
        Person tom = new Person("Tom", 35, 1.75, true);
        try(DataOutputStream dos = new DataOutputStream(new FileOutputStream("data.bin")))
        {
            dos.writeUTF(tom.name);
            dos.writeInt(tom.age);
            dos.writeDouble(tom.height);
            dos.writeBoolean(tom.married);
            System.out.println("Запись в файл произведена");
        }
        catch(IOException ex){
            System.out.println(ex.getMessage());
        }
        try(DataInputStream dis = new DataInputStream(new FileInputStream("data.bin")))
        {
            String name = dis.readUTF();
            int age = dis.readInt();
            double height = dis.readDouble();
            boolean married = dis.readBoolean();
            System.out.printf("Человека зовут: %s , его возраст: %d , его рост: %f метров,
женат/замужем: %b",
                name, age, height, married);
        }
        catch(IOException ex){

            System.out.println(ex.getMessage());
        }
    }
}
```

СИМВОЛЬНЫЕ ПОТОКИ

Хотя с помощью ранее рассмотренных классов можно записывать текст в файлы, однако все же их возможностей для полноценной работы с текстовыми файлами недостаточно. Для этой цели служат совсем другие классы, которые являются наследниками абстрактных классов **Reader** и **Writer**.

Если известно, что байты представляют собой только символы в некоторой кодировке, можно использовать специальные классы наследники базовых классов – **Reader** и **Writer**

Reader содержит абстрактные методы `read(...)` и `close()`.
Дополнительные методы объявлены в потомках этого класса

Writer содержит абстрактные методы `write(...)`, `flush()` и `close()`

Потомки класса Reader

BufferedReader - буферизированный вводный поток символов

CharArrayReader - позволяет читать символы из массива как из потока.

StringReader - то же из строки

InputStreamReader – при помощи методов класса Reader читает байты из потока InputStream и превращает их в символы. В процессе превращения использует разные системы кодирования

FileReader - поток для чтения символов из файла

FilterReader – служит для создания надстроек

Потомки класса `Writer`

`BufferedWriter` - буферизированный выводной поток. Размер буфера можно менять, хотя размер, принятый по умолчанию, пригоден для большинства задач.

`CharArrayWriter` - позволяет выводить символы в массив как в поток.

`StringWriter` - позволяет выводить символы в изменяемую строку как в поток.

`PrintWriter` - можно использовать как для вывода информации на консоль, так и в файл или в любой другой поток вывода..

`OutputStreamWriter` – мост между классом `OutputStream` и классом `Writer`. Символы, записанные в этот поток, превращаются в байты. При этом можно выбирать способ кодирования символов.

`FileWriter` - поток для записи символов в файл.

`FilterWriter` – служит для быстрого создания пользовательских надстроек

Пример программы

Вводить строки с клавиатуры и записывать их в файл на диске.

```
try {  
    // Создаем буферизованный символьный входной поток  
    BufferedReader in = new BufferedReader(  
        new InputStreamReader(System.in));  
  
    // Используем класс PrintWriter для вывода  
    PrintWriter out = new PrintWriter (new FileWriter("data.txt"));  
  
    // Записываем строки, пока не введем строку "stop"  
    while (true) {  
        String s = in.readLine();  
        if (s.equals("stop"))  
            break;  
        out.println(s);  
    }  
    out.close();  
} catch (IOException ex) {  
    // Обработать исключение  
}
```

Класс `RandomAccessFile`

`RandomAccessFile` применяется для работы с файлами произвольного доступа, он позволяет перемещаться по файлу, читать из него или писать в него, как угодно.

Для перемещения по файлу в `RandomAccessFile` применяется метод `seek()`.

`RandomAccessFile(String name, String mode)`

`name` – имя файла, зависящее от системы.

`mode` – режим открытия файла, может принимать

значения `"r"`, `"rw"`, `"rws"`, `"rwd"`.

`"r"` Открывает файл только по чтению. Запуск любых методов записи данных приведет к выбросу исключения `IOException`.

`"rw"` Открывает файл по чтению и записи. Если файл еще не создан, то осуществляется попытка создать его.

`"rws"` Открывает файл по чтению и записи подобно `"rw"`, и также требует системе при каждом изменении содержимого файла или метаданных синхронно записывать эти изменения на основной носит

Пример работы с RandomAccessFile

Создать файл прямого доступа, выполнить запись в файл и чтение из файла

```
RandomAccessFile rf = new RandomAccessFile("rtest.dat", "rw");
```

```
// Записать в файл 10 чисел и закрыть файл
```

```
for(int i = 0; i < 10; i++)  
    rf.writeDouble(i * 1.414);  
rf.close();
```

```
// Открыть файл, записать в него еще одно число и снова закрыть
```

```
rf = new RandomAccessFile("rtest.dat", "rw");  
rf.seek(5 * 8);  
rf.writeDouble(47.0001);  
rf.close();
```

```
// Открыть файл с возможностью только чтения "r"  
rf = new RandomAccessFile("rtest.dat", "r");
```

```
// Прочитать 10 чисел и показать их на экране
```

```
for(int i = 0; i < 10; i++)  
    System.out.println("Value " + i + ": " + rf.readDouble());  
rf.close();
```

Класс File

В отличие от большинства классов ввода/вывода, класс **File** работает не с потоками, а непосредственно с файлами. Данный класс позволяет получить информацию о файле: права доступа, время и дата создания, путь к каталогу. А также осуществлять навигацию по иерархиям подкаталогов.

Каждый объект **File** представляет абстрактный файл или каталог, возможно и не существующий

Абсолютный путь - это путь, который указывает на одно и то же место в файловой системе, вне зависимости от текущей директории. Полный путь всегда начинается с корневого каталога.

Относительный путь - это путь по отношению к текущему рабочему каталогу.

Префикс выглядит по-разному в различных операционных системах: символ устройства "C:", "D:" в системе Windows, символ корневого каталога "/" в системе UNIX, символы "\\\" в UNC и т.д. Каждое имя последовательности является именем каталога, а последнее имя может быть именем каталога или файла

Конструкторы класса File

File(String filePath), где filePath – имя файла на диске

File(String dirPath, String filePath), здесь параметры dirPath и filePath вместе задают то же, что один параметр в предыдущем конструкторе

File(File dirObj, String fileName), вместо имени каталога выступает другой объект File

Методы класса File

boolean createNewFile(): создает новый файл по пути, который передан в конструктор. В случае удачного создания возвращает true, иначе false

boolean delete(): удаляет каталог или файл по пути, который передан в конструктор. При удачном удалении возвращает true.

boolean exists(): проверяет, существует ли по указанному в конструкторе пути файл или каталог. И если файл или каталог существует, то возвращает true, иначе возвращает false

String getAbsolutePath(): возвращает абсолютный путь для пути, переданного в конструктор объекта

String getName(): возвращает краткое имя файла или каталога

String getParent(): возвращает имя родительского каталога

boolean isDirectory(): возвращает значение true, если по указанному пути располагается каталог

boolean isFile(): возвращает значение true, если по указанному пути находится файл

boolean isHidden(): возвращает значение true, если каталог или файл являются скрытыми

long length(): возвращает размер файла в байтах

long lastModified(): возвращает время последнего изменения файла или каталога. Значение представляет количество миллисекунд, прошедших с начала эпохи Unix

String[] list(): возвращает массив файлов и подкаталогов, которые находятся в определенном каталоге

File[] listFiles(): возвращает массив файлов и подкаталогов, которые находятся в определенном каталоге

boolean mkdir(): создает новый каталог и при удачном создании возвращает значение true

boolean renameTo(File dest): переименовывает файл или каталог

Каталоги

Каталог – это особый файл, который содержит в себе список других файлов и каталогов

Для каталога метод **isDirectory()** возвращает **true**

Метод **File[] listFiles()** возвращает список подкаталогов и файлов данного каталога

Пример: получить массив файлов и каталогов, которые находятся в рабочем (или текущем) каталоге

```
File path = new File(".");
File[] list = path.listFiles();
for(int i = 0; i < list.length; i++)
    System.out.println(list[i].getName());
```

Фильтры (интерфейс `FileFilter`)

Интерфейс `FileFilter` применяется для проверки, подпадает ли объект `File` под некоторое условие

Метод `boolean accept(File file)` возвращает истину, если аргумент удовлетворяет условию

Метод `listFiles(FileFilter filter)` В отличие от одноименного метода, но без параметра, отбирает не все файлы данного каталога, а только те, которые удовлетворяют определенному условию. Параметр `filter` предназначен для задания этого условия.

Метод `listFiles` будет вызывать метод `accept` для каждого файла в каталоге, и те, для которых `accept` вернет `true`, будут включены в результирующий список. Остальные будут проигнорированы.

Пример работы с фильтрами

Для использования возможностей FileFilter нам нужно построить класс, удовлетворяющий интерфейсу FileFilter , и определить в нем соответствующий метод accept .

```
public class MyFilter implements FilenameFilter{
String end;
public MyFilter(String end){
    this.end = end;
}
@Override
public boolean accept(File dir,String name){
    return name.endsWith(end); }
}
```

```
File path = new File("/home/ansash/1");
File files[] = path.listFiles(new MyFilter(".txt"));
System.out.println("Сортировка" + Arrays.toString(files));
```

Использование интерфейса Path

7 JDK предоставляет множество полезных классов, например Files и Paths, предназначенных для работы с файлами и путями к ним.

Для создания объекта Path существует вспомогательный класс `java.nio.file.Paths`, который содержит метод получения пути `Paths.get`

```
Path path = Paths.get("C:\test.txt");
```

Есть так же возможность получить объект Path из объекта файла `java.io.File` с помощью метода `toPath`

```
File file = new File("C:\test.txt");  
Path testFilePath = file.toPath() или наоборот path.toFile();
```

```
System.out.println("Printing file information: ");  
System.out.println("file name: " + testFilePath.getFileName());  
System.out.println("root of the path: " + testFilePath.getRoot());  
System.out.println("parent of the target: " + testFilePath.getParent());  
System.out.println("It's URI is: " + testFilePath.toUri());  
System.out.println("It's absolute path is: " + testFilePath.toAbsolutePath());  
System.out.println("It's normalized path is: " + testFilePath.normalize());
```

Интерфейс Path содержит два метода для сравнения объектов Path: `equals()` and `compareTo()`.

Использование класса Files

Класс Files (введён в Java 7, находится в пакете `java.nio.file`), который можно использовать для выполнения различных операций с файлами и каталогами. В этом классе находится множество методов для выполнения различных действий. Рассмотрим некоторые из них.

```
byte[] data = Files.readAllBytes(path);  
String content = new String(data, StandardCharsets.UTF_8);
```

С помощью метода `Files.readAllBytes` считываем содержимое файла в виде байт и затем преобразуем данное содержимое в строку.

Копирование файла/директории. Для этого используем метод `Files.copy()`.

```
Files.copy(pathSource, pathDestination);
```

Метод для перемещения файла очень похож на метод для копирования:

```
Files.move(pathSource, pathDestination,  
StandardCopyOption.REPLACE_EXISTING);
```

```
Files.delete(pathSource); - удаление файла;
```

Обход дерева файлов

При работе с файловой системой может возникнуть необходимость обхода дерева файлов, например при поиске файла или копировании каталога со всем его содержимым. Класс `Files` содержит два метода, позволяющих обходить дерево файлов. Их сигнатуры приведены ниже:

`Path walkFileTree(Path start, FileVisitor visitor)`

`Path walkFileTree(Path start, Set options, int maxDepth, FileVisitor visitor)`

`FileVisitor` — это интерфейс, содержащий следующие методы:

`FileVisitResult preVisitDirectory(T dir, BasicFileAttributes attrs)` — выполняется перед доступом к элементам каталога.

`FileVisitResult visitFile(T file, BasicFileAttributes attrs)` — выполняется при доступе к файлу.

`FileVisitResult postVisitDirectory(T dir, IOException exc)` — выполняется, когда все элементы директории пройдены .

`FileVisitResult visitFileFailed(T file, IOException exc)` — выполняется, если к файлу нет доступа.

Вам необходимо реализовать интерфейс `FileVisitor`, чтобы передать соответствующий объект в метод `walkFileTree()`. Но если необходимости реализовывать все четыре метода этого интерфейса нет, то можно просто расширить реализацию класса `SimpleFileVisitor`, переопределив лишь необходимые методы.

```

import java.io.IOException;
import java.nio.file.*;
import java.nio.file.attribute.BasicFileAttributes;

class MyFileVisitor extends SimpleFileVisitor<Path> {
    public FileVisitResult visitFile(Path path,
        BasicFileAttributes fileAttributes) {
        System.out.println("file name:" + path.getFileName());
        return FileVisitResult.CONTINUE;
    }

    public FileVisitResult preVisitDirectory(Path path,
        BasicFileAttributes fileAttributes) {
        System.out.println("Directory name:" + path);
        return FileVisitResult.CONTINUE;
    }
}

public class Test11 {
    public static void main(String[] args) {

        Path pathSource = Paths.get("Введите сюда путь к какому-либо каталогу, содержащему другие каталоги и файлы");
        try {
            Files.walkFileTree(pathSource, new MyFileVisitor());
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}

```

Из этих четырёх методов были переопределены только два для вывода имён каталогов и файлов. Можно контролировать поток обхода с помощью возвращаемых этими методами значений. Их четыре:

CONTINUE: указывает на то, что обход дерева следует продолжить.

TERMINATE: указывает, что обход нужно немедленно прекратить.

SKIP_SUBTREE: указывает, что подкаталоги должны быть пропущены для обхода.

SKIP_SIBLINGS: указывает на то, что обход должен быть остановлен в текущем каталоге и каталогах одного уровня с ним. Если это значение возвращается из preVisitDirectory(), то вложенные файлы/каталоги не обходятся и postVisitDirectory() не срабатывает. Если это значение возвращается из visitFile (), то остальные файлы каталога не обходятся. Если он возвращается из postVisitDirectory (), то остальные каталоги того же уровня не будут обходиться.

НОВЫЙ ВВОД/ВЫВОД

Библиотека нового ввода-вывода появилась в версии JDK 1.4

Ее цель – увеличение производительности и обеспечения безопасности при одновременном конкурентном доступе к данным из нескольких потоков.

Основными понятиями нового ввода/вывода являются

Buffers

Более функциональная и удобная замена массивов. Используется для хранения считанной информации и в качестве источника для записи.

Channels

Вместо Stream'ов, в NIO используются каналы (Channel), которые могут объединять функциональность InputStream и OutputStream.

Selector — своеобразный слушатель, который сообщает, когда с каналом можно совершить какое-то действие.

Копирование файлов с использованием FileChannel

В этом классе есть очень полезный метод `transferFrom()`, который также очень часто используется для копирования файлов. Согласно документации, этот способ копирования файла работает быстрее, чем при использовании потоков (`InputStream`, `OutputStream`).

```
private static void copyFileUsingChannel(File source, File dest)
try (FileChannel sourceChannel = new FileInputStream("in").getChannel();
    FileChannel destChannel = new FileOutputStream("out").getChannel())
{
    destChannel.transferFrom(sourceChannel, 0,
sourceChannel.size());
} catch (IOException e) {
    e.printStackTrace();
}
}
```

Сериализация

В программе мы создаем объекты, которые существуют до тех пор, пока существует, по крайней мере, одна ссылка на данный объект. Если все ссылки на объект уничтожены, то объект, хотя и существует какое-то время, пока его не утилизировал сборщик мусора, но для нас он безвозвратно потерян. По завершении программы все созданные в программе объекты уничтожаются.

Сериализация это процесс сохранения состояния объекта в последовательность байт; десериализация это процесс восстановления объекта, из этих байт.

Интерфейс `Serializable`

Чтобы обладать способностью к сериализации, класс должен реализовать интерфейс-метку `Serializable`

Интерфейс `Serializable` не содержит никаких методов. Он просто служит индикатором того, что класс может быть сериализован

Для того, чтобы значения полей объекта могли быть восстановлены в процессе десериализации, к ним должен быть доступ посредством стандартного конструктора без параметров, который, в принципе, может не содержать никакого кода

```
public class MyClass implements Serializable{  
    ...  
}
```

Запись-чтение объектов

Сериализованные объекты можно записывать и считывать при помощи классов **ObjectOutputStream** и **ObjectInputStream**.

Они также реализуют интерфейсы `DataInput` / `DataOutput`, что дает возможность записывать в поток не только объекты, но и простые типы данных.

wirteObject(Object obj) – запись объекта (класс `ObjectOutputStream`)

Object readObject() – чтение объекта (класс `ObjectInputStream`).

Метод `readObject` может генерировать исключение `java.lang.ClassNotFoundException`

При десериализации объекта, он возвращается в виде объекта класса **Object** - верхнего класса всей иерархии классов Java. Для того, чтобы использовать десериализованный класс, необходимо произвести явное преобразование его к необходимому типу

Пример сериализации объектов

```
public class Point implements Serializable {
    private int x=0, y = 0;
    public Point() {}
    public Point(int x, int y) {
        this.x = x; this.y = y;
    }
    public String toString() { return "("+x+","+y+")"; }
}
// Сериализация
ObjectOutputStream out = null;
try {
    out = new ObjectOutputStream(new BufferedOutputStream(
        new FileOutputStream("A.ser")));
    out.writeObject(new Point(5,6));
    out.flush();
} catch ( IOException ex ) {
    ex.printStackTrace();
}
// Десериализация
ObjectInputStream in = null;
Point restObj = null;
try {
    in = new ObjectInputStream(new BufferedInputStream(
        new FileInputStream("A.ser")));
    restObj = (Point) in.readObject();
} catch ( IOException ex ) {
    ex.printStackTrace();
}
```