Java
Файловый ввод / вывод

Лекция #7

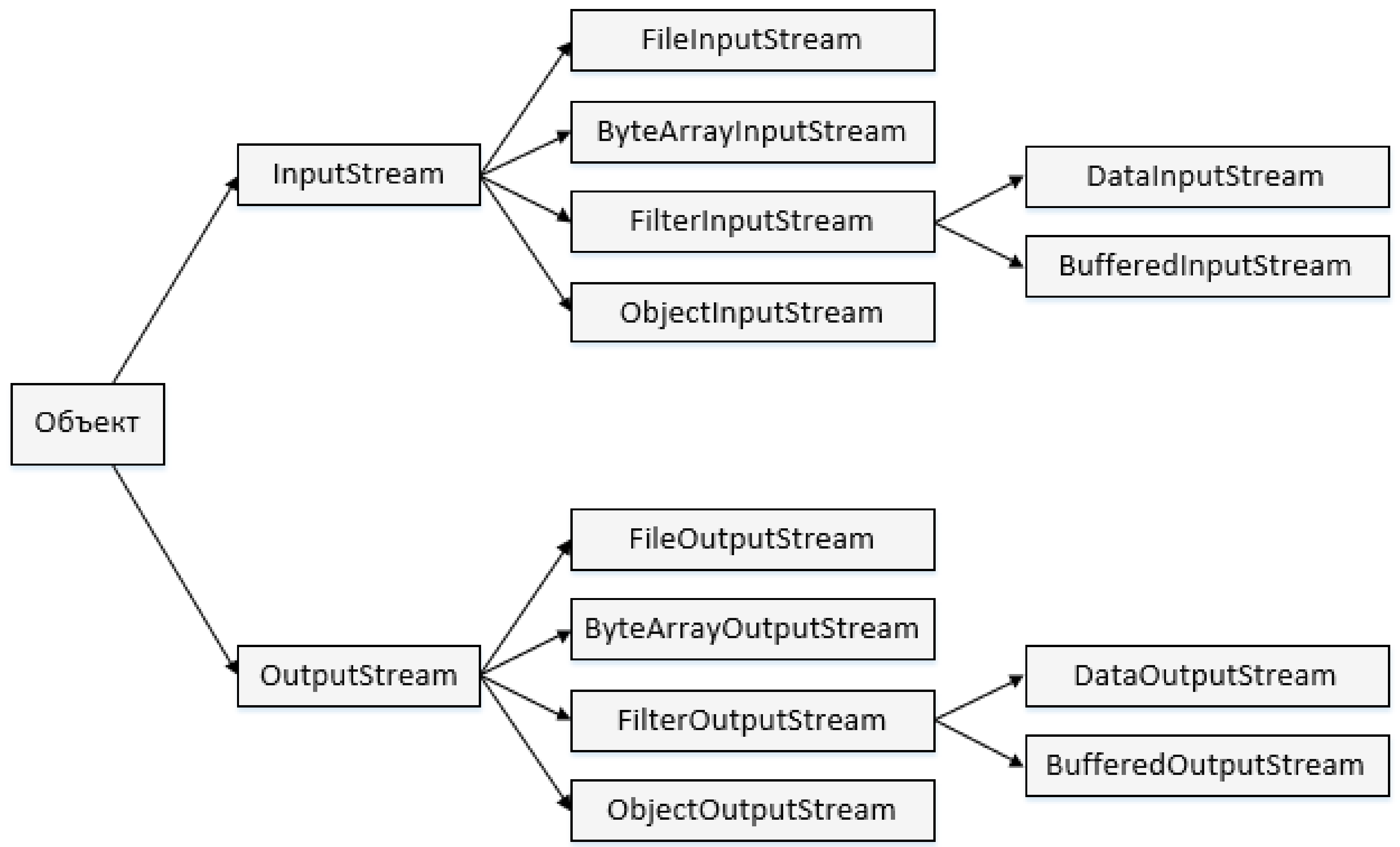
Пустовалова О.Г.
доцент. каф. мат.мод.
ИММиКН ЮФУ

Содержание

- Двоичные или бинарные файлы
- Байтовый поток
- Запись файлов и класс FileOutputStream
- Чтение файлов и класс FileInputStream
- Закрытие потоков

Файловый ввод и вывод для байтовых потоков

Иерархия классов для управления потоками Ввода и Вывода



Бинарный файл

Двоичный (бинарный) файл — в широком смысле: последовательность произвольных байтов. Название связано с тем, что байты состоят из бит, то есть двоичных (англ. binary) цифр.

В узком смысле слова **двоичные** файлы противопоставляются **текстовым** файлам.

При этом с точки зрения технической реализации на уровне аппаратуры, текстовые файлы являются частным случаем двоичных файлов, и, таким образом, в широком значении слова под определение «двоичный файл» подходит любой файл.

Байтовый поток

Для файлового ввода/вывода создаются байтовые потоки с помощью классов

FileInputstream и **FileOutputstream**

Это особенно удобно для **бинарных** файлов, хранящих байт-коды, архивы, изображения, звук.

Запись файлов и класс FileOutputStream

Через конструктор класса **FileOutputStream** задается файл, в который производится запись.

Класс поддерживает несколько конструкторов:

- FileOutputStream(String filePath)
- FileOutputStream(File fileObj)
- FileOutputStream(String filePath, **boolean** append)
- FileOutputStream(File fileObj, **boolean** append)

Файл задается либо через строковый путь, либо через объект File. Второй параметр - append задает способ записи: если он равен **true**, то данные дозаписываются в конец файла, а при **false** - файл полностью перезаписывается.

Запись строки в файл. Класс FileOutputStream

```
import java.io.*;
public class Main {
    public static void main(String... args) {

        String text = "Hello world!"; // строка для записи

        try(FileOutputStream   fos=new FileOutputStream("d:\\notes.txt"))
        {
            // перевод строки в байты
            byte[] buffer = text.getBytes();

fos.write(buffer, 0, buffer.length);
fos.write(buffer[0]); // запись первого байта
        }

        catch(IOException ex){

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


        }



        System.out.println("The file has been written");    }}
```

Для создания объекта FileOutputStream используется конструктор, принимающий в качестве параметра путь к файлу для записи. Если такого файла нет, то он автоматически создается при записи. Так как здесь записываем строку, то ее надо сначала перевести в массив байтов. И с помощью метода write строка записывается в файл.

Для автоматического закрытия файла и освобождения ресурса объект FileOutputStream создается с помощью конструктции try...catch.

Чтение файлов и класс FileInputStream

Для считывания данных из файла предназначен класс **FileInputStream**, который является наследником класса **InputStream** и поэтому реализует все его методы.

Для создания объекта FileInputStream мы можем использовать ряд конструкторов.

Наиболее используемая версия конструктора:

```
FileInputStream(String fileName) throws FileNotFoundException
```

Если файл не может быть открыт, например, по указанному пути такого файла не существует, то генерируется исключение **FileNotFoundException**.

Считывание данных из файла и вывод на консоль по 1 байту

```
import java.io.*;

public class Main {
    public static void main(String... args) {

        try(FileInputStream fin=new FileInputStream("d:\\notes.txt"))

        {
            System.out.printf("File size: %d bytes \n", fin.available());

            int i=-1;

            while((i=fin.read())!=-1){

                System.out.print((char)i);
            }
        }
        catch(IOException ex){

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

В данном случае мы считываем каждый отдельный байт в переменную i:.

Когда в потоке больше нет данных для чтения, метод возвращает число -1.

Затем каждый считанный байт конвертируется в объект типа char и выводится на консоль.

Считывание данных в массив из файла и вывод на консоль

В данном случае данные считываются данные в массив байт

```
import java.io.*;


public class Main {
    public static void main(String... args) {


        try (FileInputStream fin = new FileInputStream("d:\\notes.txt")) {
            byte[] buffer = new byte[fin.available()];

             // считаем файл в буфер
              fin.read(buffer, 0, fin.available());

            System.out.println("File data:");
            for (int i = 0; i < buffer.length; i++) {

                System.out.print((char) buffer[i]);
            }
        } catch (IOException ex) {

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

Чтение из одного и запись в другой файл

```
import java.io.*;

public class Main {
    public static void main(String... args) {

        try(FileInputStream fin=new FileInputStream("d:\\notes.txt");

            FileOutputStream fos=new FileOutputStream("d:\\notes_new.txt"))
        {
            byte[] buffer = new byte[fin.available()];
            // считываем буфер

            fin.read(buffer, 0, buffer.length);
            // записываем из буфера в файл
            fos.write(buffer, 0, buffer.length);
        }
        catch(IOException ex){

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

Классы FileInputStream и FileOutputStream

Классы **FileInputStream** и **FileOutputStream** предназначены прежде всего для записи **двоичных** файлов, то есть для записи и чтения байтов.

И хотя они также могут использоваться для работы с текстовыми файлами, но все же для этой задачи больше подходят другие классы.

Закрытие потоков

Закрытие потоков

При завершении работы с потоком его надо закрыть с помощью метода **close()**

При закрытии потока освобождаются все выделенные для него ресурсы, например, файл. В случае, если поток окажется не закрыт, может происходить утечка памяти.

Есть два способа закрытия файла. Первый традиционный заключается в использовании блока try..catch..finally.

Закрытие потоков. Первый способ. try..catch..finally

```
import java.io.*;

public class Main {
    public static void main(String... args) {

        FileInputStream fin = null;
        try {
            fin = new FileInputStream("d:\\notes.txt");

            int i = -1;
            while ((i = fin.read()) != -1) {

                System.out.print((char) i);
            }
        } catch (IOException ex) {

            System.out.println(ex.getMessage());
        }
 finally {
        try {

            if (fin != null)
                fin.close();
        } catch (IOException ex) {

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

 }   }}
```

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.

Закрытие потоков. Второй способ. try-with-resources

```
import java.io.*;

public class Main {
    public static void main(String... args) {

        try(FileInputStream fin=new FileInputStream("d:\\notes.txt"))


        {
            int i=-1;

            while((i=fin.read())!=-1){

                System.out.print((char)i);
            }
        }

        catch(IOException ex){

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

Начиная с Java 7 можно использовать еще один способ, который автоматически вызывает метод close.

Этот способ заключается в использовании конструкции **try-with-resources (try-с-ресурсами)**.

Спасибо за внимание!