

Потоки ввода-вывода. Работа с файлами. Сериализация



Григорий
Вахмистров



Григорий Вахмистров

Backend Developer в Tennisi.bet

План занятия

1. [Файлы и каталоги](#)
2. [Потоки ввода-вывода](#)
3. [Побайтовое чтение и запись файлов](#)
4. [Закрытие потоков](#)
5. [Буферизация байтовых потоков](#)
6. [Посимвольное чтение и запись](#)
7. [Буферизация символьных потоков](#)
8. [ZIP архивы](#)
9. [Сериализация](#)
10. [Итоги](#)
11. [Домашнее задание](#)



Файлы и каталоги

Класс File

Класс **File**, определенный в пакете **java.io**, управляет информацией о файлах и каталогах.

На уровне операционной системы файлы и каталоги отличаются, но в Java они описываются одним классом **File**.

В зависимости от того, что должен представлять объект **File** - файл или каталог, можно использовать один из его конструкторов:

```
File(String dirPath)
File(String dirPath, String fileName)
File(File dirPath, String fileName)
```

Создадим объекты **File** для файлов и каталогов:

```
// создаем объект File в качестве каталога
File dir1 = new File("C://SomeDir");
// создаем объект File для файла, находящегося в каталоге
File file1 = new File("C://SomeDir", "Hello.txt");
// создаем объект File для файла, находящегося в каталоге dir1
File file2 = new File(dir1, "Hello2.txt");
```

Работа с каталогами

Создание экземпляра класса **File** не приводит к созданию каталога на жестком диске компьютера.

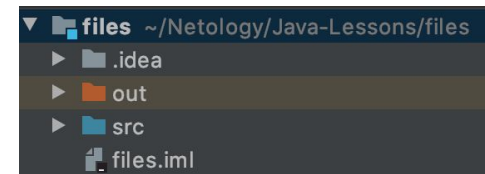
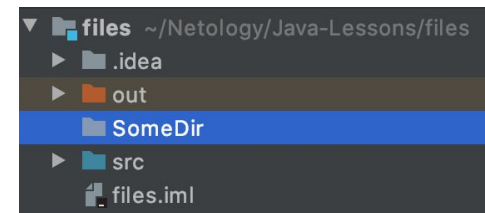
Для работы с каталогами необходимо вызывать методы у объекта **File**, например:

- создание - **mkdir()**;
- переименование - **renameTo(File newDir)**;
- удаление - **delete()**.

```
// определяем объект для каталога
File dir = new File("SomeDir");
// пробуем создать каталог
if (dir.mkdir())
    System.out.println("Каталог создан");
```

```
// определяем новый объект для каталога
File newDir = new File("NewDir");
// пробуем переименовать каталог
dir.renameTo(newDir);
```

```
// пробуем удалить каталог
if (newDir.delete())
    System.out.println("Каталог удален");
```



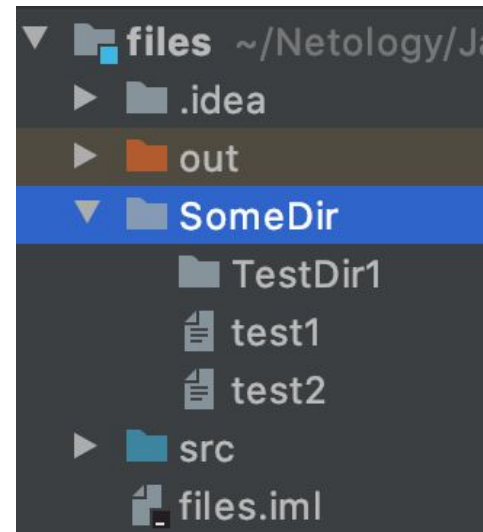
Список объектов каталога

Если объект **File** представляет каталог, то метод **isDirectory()** возвращает **true**.

Для каталога можно получить его вложенные подкаталоги и файлы с помощью методов **list()** и **listFiles()**.

Получим содержимое определенного каталога:

```
// определяем объект для каталога
File dir = new File("SomeDir");
// если объект представляет каталог
if (dir.isDirectory()) {
    // получаем все вложенные объекты в каталоге
    for (File item : dir.listFiles()) {
        // проверим, является ли объект каталогом
        if (item.isDirectory()) {
            System.out.println(item.getName() + " - каталог");
        } else {
            System.out.println(item.getName() + " - файл");
        }
    }
}
```



Вывод консоли:

```
TestDir1 - каталог
test1 - файл
test2 - файл
```

Работа с файлами

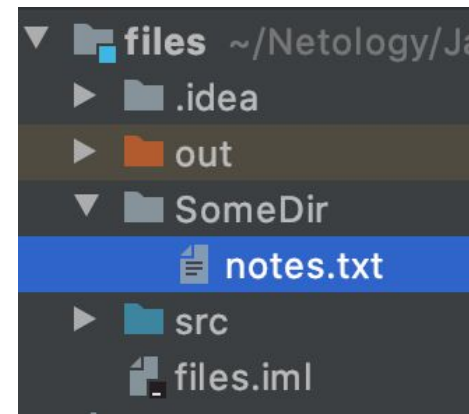
Работа с файлами аналогична работе с каталогами.

Создание экземпляра класса **File** не приводит к созданию файла, для этого так же необходимо вызвать соответствующий метод.

Например, создадим файл с помощью вызова метода **createNewFile()**:

```
// определяем объект для файла
File myFile = new File("SomeDir//notes.txt");

// создадим новый файл
try {
    if (myFile.createNewFile())
        System.out.println("Файл был создан");
} catch (IOException ex) {
    System.out.println(ex.getMessage());
}
```



Блок **try-catch** необходим, так как метод **createNewFile()** может пробросить исключение, если путь к файлу будет некорректен.

Получим информацию о файле

Получим некоторую информацию о ранее созданном файле **notes.txt**:

```
System.out.println("Имя файла: " + myFile.getName());
System.out.println("Родительский каталог: " + myFile.getParent());
System.out.println("Размер файла: " + myFile.length());

if (myFile.exists())
    System.out.println("Файл существует");
else
    System.out.println("Файл не был найден");

if (myFile.canRead())
    System.out.println("Файл может быть прочитан");
else
    System.out.println("Файл не может быть прочитан");

if (myFile.canWrite())
    System.out.println("Файл может быть записан");
else
    System.out.println("Файл не может быть записан");
```

Вывод консоли:

```
Родительский каталог: SomeDir
Размер файла: 0
Файл существует
Файл может быть прочитан
Файл может быть записан
```



Потоки ввода-вывода

Понятие потока

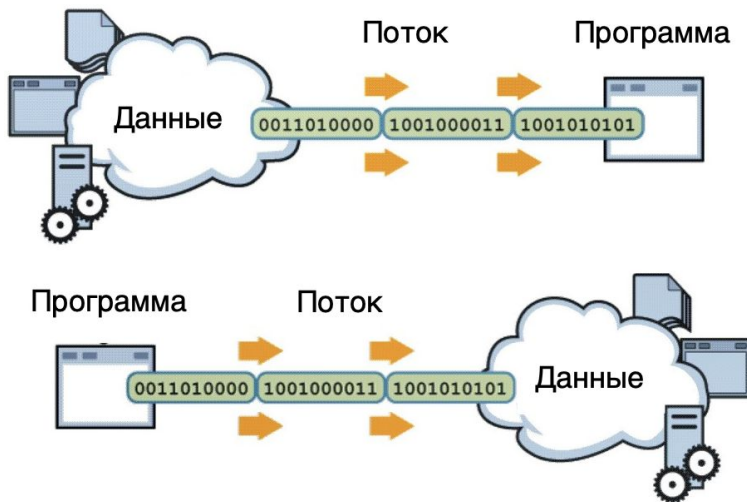
Программы интегрируются за счет передачи данных одним из способов:

- Через запись и чтение файлов в одной файловой системе.
- Через сетевое взаимодействие.
- Передача информации из одной области памяти в другую.

Передача данных — процесс “пересылки” некоторого количества байт информации.

Одна программа (отправитель) “отправляет” байты, а другая (потребитель) “потребляет” эти байты.

Байты идут друг за другом от отправителя к потребителю в виде **ПОТОКА** байтов.





Классы для работы с потоками ввода-вывода

Для работы с потоками байтов* используются абстрактные классы:

- InputStream - последовательное чтение из него байт (byte)
- OutputStream - последовательную запись в него байт (byte)

Для работы с потоками символов** используются абстрактные классы:

- Reader - последовательное чтение из него символов (char)
- Writer - последовательную запись в него символов (char)

* Поток байтов - последовательность, состоящая из байтов (byte).


** Поток символов - последовательность, состоящая из символов (char).

Наследники основных классов

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

Основные классы потоков:

| InputStream | OutputStream | Reader | Writer |
|----------------------|-----------------------|-----------------|-----------------|
| FileInputStream | FileOutputStream | FileReader | FileWriter |
| BufferedInputStream | BufferedOutputStream | BufferedReader | BufferedWriter |
| ByteArrayInputStream | ByteArrayOutputStream | CharArrayReader | CharArrayWriter |
| FilterInputStream | FilterOutputStream | FilterReader | FilterWriter |
| DataInputStream | DataOutputStream | | |
| ObjectInputStream | ObjectOutputStream | | |



Побайтовое чтение и запись файлов

Запись файлов и класс `FileOutputStream`

Класс **`FileOutputStream`** предназначен для записи байтов в файл. Он является производным от класса **`OutputStream`**, поэтому наследует всю его функциональность.

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`**.

Параметр **`append`** задает способ записи: если **`true`** - данные **дозаписываются** в конец файла, а при **`false`** - файл полностью **перезаписывается**.

Запись файлов и класс `FileOutputStream`

Например, запишем в файл строку:

```
String text = "Hello world!";  
// откроем байтовый поток записи в файл  
try (FileOutputStream fos = new FileOutputStream("notes.txt")) {  
    // перевод строки в массив байтов  
    byte[] bytes = text.getBytes();  
    // запись байтов в файл  
    fos.write(bytes, 0, bytes.length);  
} catch (IOException ex) {  
    System.out.println(ex.getMessage());  
}
```

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.

Для записи в файл используем метод **`write()`**, который принимает преобразованную в массив байт строку.

Для автоматического закрытия файла и освобождения ресурса объект **`FileOutputStream`** создается с помощью конструкции **`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`** можно использовать ряд конструкторов. Наиболее используемый в качестве параметра принимает путь к считываемому файлу:

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

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

Чтение файлов и класс `FileInputStream`

Считаем данные из ранее записанного файла и выведем на консоль:

```
// откроем байтовый поток для чтения файла
try (FileInputStream fin = new FileInputStream("notes.txt")) {
    int i;
    // считываем файл пока есть доступные байты
    while ((i = fin.read()) != -1) {
        System.out.print((char) i);
    }
} catch (IOException ex) {
    System.out.println(ex.getMessage());
}
```

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, речь о которых пойдет далее.



Заккрытие потоков

Заккрытие потоков

При завершении работы с потоком его необходимо закрыть с помощью метода **close()**, который определен в интерфейсе **Closeable**.

Метод **close()** имеет следующее определение:

```
void close() throws IOException
```

Интерфейс **Closable** 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. В случае, если поток окажется не закрыт, может произойти утечка памяти*.

* Ут́ечка па́мяти (англ. memory leak) — процесс неконтролируемого уменьшения объёма свободной оперативной или виртуальной памяти компьютера, связанный с ошибками в работающих программах, вовремя не освобождающих ненужные уже участки памяти, или с ошибками системных служб контроля памяти.

Блок Try..Catch..Finally

Первый способ закрыть поток и освободить ресурсы заключается в использовании комбинаций блоков **try-catch-finally**.

Например, считаем данные из файла:

```
FileInputStream fin = null;
try {
    fin = new FileInputStream("notes.txt");
    int i;
    while ((i = fin.read()) != -1) {
        System.out.print((char) i);
    }
} catch (c ex) {
    System.out.println(ex.getMessage());
} finally {
    try {
        if (fin != null) {
            fin.close();
        }
    } catch (IOException ex) {
        System.out.println(ex.getMessage());
    }
}
```

При открытии или считывании файла может произойти исключение **IOException**, поэтому код считывания помещается в блок **try**.

Чтобы закрыть поток в любом случае, даже если при работе с ним возникнет ошибка, вызов метода **close()** помещается в блок **finally**.

Так как метод **close()** также в случае ошибки может генерировать исключение **IOException**, то его вызов также помещается во вложенный блок **try-catch**.



Try с ресурсами

Начиная с Java 7, можно использовать другой способ закрытия потоков, который автоматически вызывает метод **close()**. Этот способ заключается в использовании конструкции **try с ресурсами**.

Данная конструкция работает с классами потоков, реализующими интерфейс **Closeable**, который наследуется от интерфейса **AutoCloseable**.

Try с ресурсами

Перепишем предыдущий пример с использованием конструкции **try с ресурсами**:

```
try (FileInputStream fin = new FileInputStream("notes.txt")) {  
    int i;  
    while ((i = fin.read()) != -1) {  
        System.out.print((char) i);  
    }  
} catch (IOException ex) {  
    System.out.println(ex.getMessage());  
}
```

После окончания работы в блоке **try** у ресурса (в данном случае у объекта **FileInputStream**) автоматически вызывается метод **close()**.

Try с множественными ресурсами

Если требуется использовать несколько потоков, которые после выполнения следует закрыть, то можно указать объекты потоков через точку с запятой:

```
try (FileInputStream fin = new FileInputStream("notes.txt");
    FileOutputStream fos = new FileOutputStream("new_notes.txt")) {

    // работаем с файлами

} catch (IOException e) {
    e.printStackTrace();
}
```

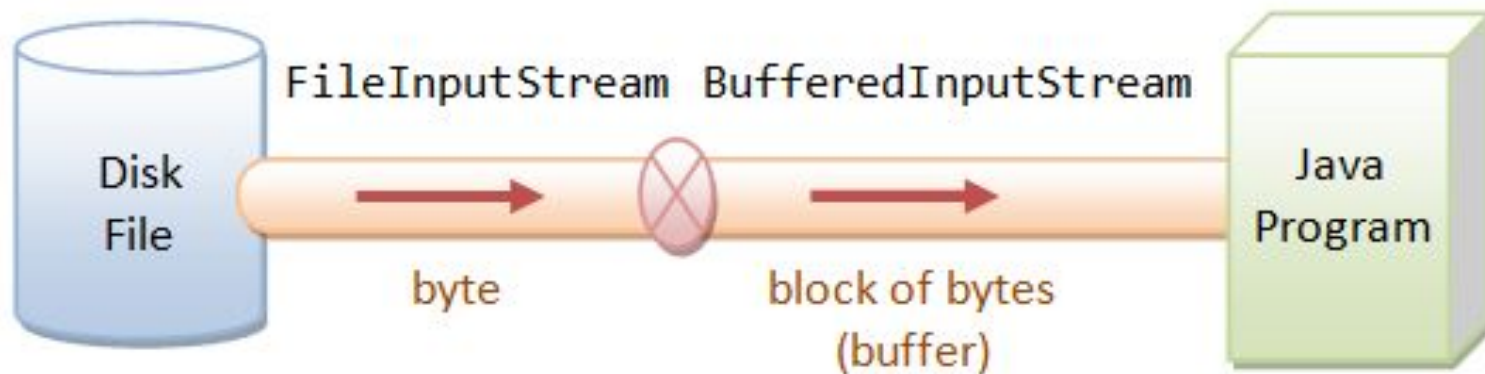



Буферизация байтовых ПОТОКОВ

Буферизованные потоки

Для оптимизации операций ввода-вывода используются буферизованные потоки.

Буферизованные потоки добавляют к стандартным потокам специальный буфер в памяти, с помощью которого повышается производительность при чтении и записи потоков.



Класс `BufferedInputStream`

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

Класс **`BufferedInputStream`** определяет два конструктора:

```
BufferedInputStream(InputStream inputStream)
BufferedInputStream(InputStream inputStream, int bufSize)
```

Первый параметр **`InputStream`** - это поток ввода, с которого данные будут считываться в буфер. Второй параметр **`int`** - размер буфера.

Класс `BufferedInputStream`

Например, оптимизируем производительность считывания данных из потока `ByteArrayInputStream` за счет буферизации класса **`BufferedInputStream`**:

```
String text = "Hello world!";
byte[] buffer = text.getBytes();

// создаем входной байтовый поток
// и передаем его в входной буферизированный поток
try (ByteArrayInputStream in = new ByteArrayInputStream(buffer);
     BufferedInputStream bis = new BufferedInputStream(in)) {
    int c;
    while ((c = bis.read()) != -1) {
        System.out.print((char) c);
    }
} catch (Exception e) {
    System.out.println(e.getMessage());
}
```

Класс **`BufferedInputStream`** в конструкторе принимает объект **`InputStream`**, а конкретно - экземпляр класса **`ByteArrayInputStream`**. Метод **`read()`** считывает данные побайтно из массива **`buffer`**.

Класс **BufferedOutputStream**

Класс **BufferedOutputStream** создает буфер для потоков вывода. Этот буфер накапливает выводимые байты без постоянного обращения к устройству. И когда буфер заполнен, производится запись данных.

Класс **BufferedOutputStream** определяет два конструктора:

```
BufferedOutputStream(OutputStream outputStream)
BufferedOutputStream(OutputStream outputStream, int bufSize)
```

Первый параметр **OutputStream** - это поток вывода, а второй параметр **int** - размер буфера.


Класс `BufferedOutputStream`

Оптимизируем действие потока вывода на примере записи в файл с использованием класса **`BufferedOutputStream`**:

```
String text = "Hello world!";
byte[] buffer = text.getBytes();

// создаем выходной байтовый поток
// и передаем его в выходной буферизированный поток
try (FileOutputStream out = new FileOutputStream("notes.txt");
    BufferedOutputStream bos = new BufferedOutputStream(out)) {
    // производим запись от 0 до последнего байта из массива
    bos.write(buffer, 0, buffer.length);
} catch (IOException ex) {
    System.out.println(ex.getMessage());
}
```

Класс **`BufferedOutputStream`** в конструкторе принимает в качестве параметра объект **`OutputStream`** - в данном случае это файловый поток вывода **`FileOutputStream`**.



Посимвольное чтение и запись



Чтение и запись текстовых файлов

Рассмотренные ранее классы могут записывать текст в файлы, однако они предназначены прежде всего для работы с бинарными потоками данных, и их возможностей для полноценной работы с текстовыми файлами недостаточно.

Для этой цели служат другие классы, которые являются наследниками абстрактных классов **Reader** и **Writer**.

Запись файлов. Класс `FileWriter`

Класс **`FileWriter`** является производным от класса **`Writer`**. Он используется для записи текстовых файлов.

Чтобы создать объект **`FileWriter`**, можно использовать один из следующих конструкторов:

```
FileWriter(File file)
FileWriter(File file, boolean append)
FileWriter(FileDescriptor fd)
FileWriter(String fileName)
FileWriter(String fileName, boolean append)
```

Так, в конструктор передается либо путь к файлу в виде строки, либо объект **`File`**, который ссылается на конкретный текстовый файл.

Параметр **`append`** указывает, должны ли данные дозаписываться в конец файла (если параметр **`true`**), либо файл должен перезаписываться (**`false`**).

Запись файлов. Класс FileWriter

Запишем в файл какой-нибудь текст:

```
String text = "Hello World";

try (FileWriter writer = new FileWriter("notes.txt", false)) {
    // запись всей строки
    writer.write(text);
    // запись по символам
    writer.append('\n');
    writer.append('!');
    // дозаписываем и очищаем буфер
    writer.flush();
} catch (IOException ex) {
    System.out.println(ex.getMessage());
}
```

В конструкторе использовался параметр **append** со значением **false**, то есть файл будет перезаписываться. Затем с помощью методов **write()** и **append()**, определенных в базовом классе **Writer**, производится запись данных. Метод **flush()** дозаписывает буфер и очищает его.

Чтение файлов. Класс `FileReader`

Класс **`FileReader`** наследуется от абстрактного класса **`Reader`** и предоставляет функциональность для чтения текстовых файлов.

Для создания объекта **`FileReader`** можно использовать один из его конструкторов:

```
FileReader(String fileName)
FileReader(File file)
FileReader(FileDescriptor fd)
```

А используя методы, определенные в базовом классе **`Reader`**, можно произвести чтение файла:

```
try (FileReader reader = new FileReader("notes.txt")) {
    // читаем посимвольно
    int c;
    while ((c = reader.read()) != -1) {
        System.out.print((char) c);
    }
} catch (IOException ex) {
    System.out.println(ex.getMessage());
}
```



Буферизация символьных ПОТОКОВ

Запись текста через буфер и **BufferedWriter**

Класс **BufferedWriter** записывает текст в поток, предварительно буферизируя записываемые символы, тем самым снижая количество обращений к физическому носителю для записи данных.

Класс **BufferedWriter** имеет следующие конструкторы:

```
BufferedWriter(Writer out)
BufferedWriter(Writer out, int sz)
```

В качестве параметра он принимает поток вывода, в который надо осуществить запись. Второй параметр указывает на размер буфера.

Например, осуществим запись в файл:

```
try (BufferedWriter bw = new BufferedWriter(new FileWriter("notes.txt"))) {
    String text = "Hello World!";
    bw.write(text);
} catch (IOException ex) {
    System.out.println(ex.getMessage());
}
```

Чтение текста и `BufferedReader`

Класс **`BufferedReader`** считывает текст из символьного потока ввода, буферизируя прочитанные символы. Использование буфера призвано увеличить производительность чтения данных из потока.

Класс **`BufferedReader`** имеет следующие конструкторы:

```
BufferedReader(Reader in)  
BufferedReader(Reader in, int sz)
```

Второй конструктор, кроме потока ввода, из которого производится чтение, также определяет размер буфера, в который будут считываться символы.

Чтение текста и `BufferedReader`

Так как **`BufferedReader`** наследуется от класса **`Reader`**, то он может использовать все те методы для чтения из потока, которые определены в **`Reader`**.

И также **`BufferedReader`** определяет свой собственный метод **`readLine()`**, который позволяет считывать из потока построчно.

Рассмотрим применение **`BufferedReader`**:

```
try (BufferedReader br = new BufferedReader(new FileReader("notes.txt"))) {  
    //чтение построчно  
    String s;  
    while ((s = br.readLine()) != null) {  
        System.out.println(s);  
    }  
} catch (IOException ex) {  
    System.out.println(ex.getMessage());  
}
```

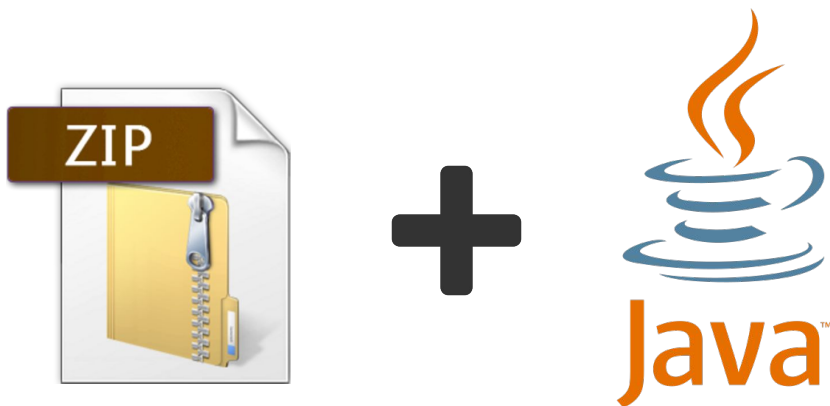


ZIP-архивы

Работа с ZIP-архивами

Для работы с **zip-архивами** в Java используются два класса из пакета `java.util.zip`:

- `ZipInputStream`
- `ZipOutputStream`



ZipOutputStream. Запись архивов

Для создания архива используется класс **ZipOutputStream**. Для этого в его конструктор передается поток вывода:

```
ZipOutputStream(OutputStream out)
```

Для записи файлов в архив для каждого файла создается объект **ZipEntry**, в конструктор которого передается имя архивируемого файла. Чтобы добавить каждый объект **ZipEntry** в архив, применяется метод **putNextEntry()**.

ZipOutputStream. Запись архивов

Создадим архив:

```
try (ZipOutputStream zout = new ZipOutputStream(new
FileOutputStream("zip_output.zip"));
    FileInputStream fis = new FileInputStream("notes.txt")) {
    ZipEntry entry = new ZipEntry("packed_notes.txt");
    zout.putNextEntry(entry);
    // считываем содержимое файла в массив byte
    byte[] buffer = new byte[fis.available()];
    fis.read(buffer);
    // добавляем содержимое к архиву
    zout.write(buffer);
    // закрываем текущую запись для новой записи
    zout.closeEntry();
} catch (Exception ex) {
    System.out.println(ex.getMessage());
}
```



ZipOutputStream. Запись архивов

После добавления объекта **ZipEntry** в поток необходимо добавить в него и содержимое файла. Для этого используется метод **write()**, записывающий в поток массив байтов **buffer**.

В конце записи требуется закрыть **ZipEntry** с помощью метода **closeEntry()**. После этого можно добавлять в архив новые файлы - в этом случае все вышеописанные действия для каждого нового файла повторяются.

Чтение архивов. **ZipInputStream**

Для чтения архивов применяется класс **ZipInputStream**. В конструкторе он принимает поток, указывающий на zip-архив:

```
ZipInputStream(InputStream in)
```

Для считывания файлов из архива **ZipInputStream** использует метод **getNextEntry()**, который возвращает объект **ZipEntry**, который представляет отдельную запись в zip-архиве.

Чтение архивов. ZipInputStream

Например, считаем какой-нибудь архив:

```
try (ZipInputStream zin = new ZipInputStream(new
FileInputStream("output.zip"))) {
    ZipEntry entry;
    String name;
    while ((entry = zin.getNextEntry()) != null) {
        name = entry.getName(); // получим название файла
        // распаковка
        FileOutputStream fout = new FileOutputStream(name);
        for (int c = zin.read(); c != -1; c = zin.read()) {
            fout.write(c);
        }
        fout.flush();
        zin.closeEntry();
        fout.close();
    }
} catch (Exception ex) {
    System.out.println(ex.getMessage());
}
```



Чтение архивов. `ZipInputStream`

`ZipInputStream` в конструкторе получает ссылку на поток ввода. И затем в цикле выводятся все файлы и их размер в байтах, которые находятся в данном архиве.

Затем данные извлекаются из архива и сохраняются в новые файлы, которые находятся в той же папке и которые начинаются с "new".

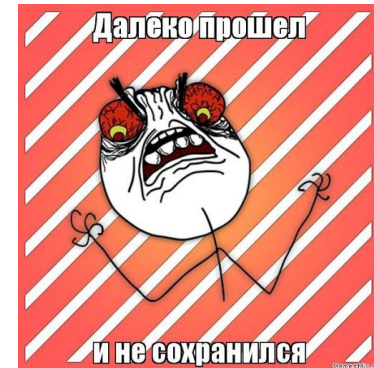


Сериализация

Назначение

Играли ли Вы в старые игровые приставки, где не было сохранений?

Сейчас же в любой игре можно сохранить состояние игры в некий файл, который будет содержать информацию о здоровье, количестве боеприпасов и игровом прогрессе.



Для этого как раз и используются механизмы **сериализации** и **десериализации**:

- Сериализация — это процесс сохранения состояния объекта в последовательность байт.
- Десериализация — это процесс восстановления объекта из этих байт.

Интерфейс Serializable

Сериализовать можно объекты, реализующие интерфейс **Serializable**.

Например, создадим класс, который будет хранить состояние какой-нибудь игры, и имплементируем интерфейс **Serializable**:

```
public class GameProgress implements Serializable {
    private static final long serialVersionUID = 1L;

    private int health;
    private int weapons;
    private int lvl;
    private double distance;

    public GameProgress(
        int health,
        int weapons,
        int lvl,
        double distance) {
        this.health = health;
        this.weapons = weapons;
        this.lvl = lvl;
        this.distance = distance;
    }
}
```

Интерфейс **Serializable** не определяет никаких методов, просто служит указателем системе, что объект, реализующий его, может быть сериализован.

Константа **serialVersionUID** - уникальный идентификатор версии сериализованного класса.

Сериализация. Класс ObjectOutputStream

Класс **ObjectInputStream** предназначен для преобразования объектов в поток байтов.

Например, сохраним в файл один объект класса **GameProgress**:

```
// создадим экземпляр класса сохраненной игры
GameProgress gameProgress =
    new GameProgress(94, 10, 2, 254.32);

// откроем выходной поток для записи в файл
try (FileOutputStream fos = new FileOutputStream("save.dat");
    ObjectOutputStream oos = new ObjectOutputStream(fos)) {
    // запишем экземпляр класса в файл
    oos.writeObject(gameProgress);
} catch (Exception ex) {
    System.out.println(ex.getMessage());
}
```

В корне программы появился файл с именем **"save.dat"** и следующим содержимым:

```
^srcom.company.GameProgressDdistanceIhealthIvlIweaponsxp@o=p
```

Десериализация. Класс `ObjectInputStream`

Класс **`ObjectInputStream`** отвечает за обратный процесс - чтение ранее сериализованных данных из потока. В конструкторе он принимает ссылку на поток ввода:

```
ObjectInputStream(in)
```

Например, извлечем ранее сохраненный объект **`GameProgress`** из файла:

```
GameProgress gameProgress = null;

// откроем входной поток для чтения файла
try (FileInputStream fis = new FileInputStream("save.dat");
     ObjectInputStream ois = new ObjectInputStream(fis)) {
    // десериализуем объект и скастим его в класс
    gameProgress = (GameProgress) ois.readObject();
} catch (Exception ex) {
    System.out.println(ex.getMessage());
}

System.out.println(gameProgress);
```

Вывод консоли:

```
GameProgress{health=94, weapons=10, lvl=2, distance=254.32}
```



Итоги

Итоги

- Для работы с файлами и директориями файловой системы используется класс **File**.
- Чтение или запись данных происходит с использованием потока данных.
- Для работы с потоками байтов используются абстрактные классы **InputStream** и **OutputStream**.
- Для работы с потоками символов используются абстрактные классы **Writer** и **Reader**.
- При завершении работы с потоком его необходимо закрыть методом **close()** или использовать блок **try с ресурсами**.
- Для оптимизации операций ввода-вывода используются буферизованные потоки.
- Java поддерживает работу с **ZIP-архивами**.
- Сериализация — это процесс сохранения состояния объекта в последовательность байт. Требуется реализация интерфейса **Serializable**.



Домашнее задание

Давайте посмотрим ваше [домашнее задание](#).

- Вопросы по домашней работе задавайте **в чате** мессенджера Slack.
- Задачи можно сдавать **по частям**.
- Зачёт по домашней работе проставляется после того, как **приняты все задачи**.

⌘ нетология

**Задавайте вопросы и
пишите отзыв о лекции!**

Григорий Вахмистров