БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
МЕХАНИКО-МАТЕМАТИЧЕСКИЙ ФАКУЛЬТЕТ
Кафедра веб-технологий и компьютерного моделирования

А. С. Кравчук, А. И. Кравчук, Е. В. Кремень

ЯЗЫК JAVA

ПОТОКИ ВВОДА-ВЫВОДА. РАБОТА С ФАЙЛАМИ

Учебные материалы
для студентов специальности 1-31 03 08
«Математика и информационные технологии
(по направлениям)»

МИНСК
2023

УДК 004.432.045:004.738.5Java(075.8)
ББК 32.973.2-018.1я73-1
К78

Рекомендовано советом
механико-математического факультета БГУ
26 января 2023 г., протокол № 5

Рецензент
кандидат технических наук, доцент *М. Н. Садовская*

Кравчук, А. С.

К78 Язык Java. Потоки ввода-вывода. Работа с файлами : учеб. материалы для студентов спец. 1-31 03 08 «Математика и информационные технологии (по направлениям)» / А. С. Кравчук, А. И. Кравчук, Е. В. Кремень. – Минск : БГУ, 2023. – 63 с.

Подробно излагается потоковая организация ввода-вывода в Java. В Java фактически определены две полноценные подсистемы ввода-вывода: одна – для обмена байтами, другая – для обмена символами. Ввод-вывод в Java очень обширен и включает множество классов, интерфейсов и методов. Рассматриваются наиболее важные и часто используемые языковые средства ввода-вывода. Особое внимание уделено работе с файлами. Издание ориентировано как на тех, кто не имеет опыта практического программирования на языке Java, так и на тех, кто хотел бы систематизировать и улучшить свои знания. В каждой теме приводится необходимый теоретический материал и код программ, что существенно ускоряет усваивание материала, а также способствует более квалифицированному подходу к программированию.

УДК 004.432.045:004.738.5Java (075.8)
ББК 32.973.2-018.1я73-1

Оглавление

Введение

Подавляющее большинство современных программ не существует изолированно, а требует интеграции с другими. Программа должна иметь возможность, как получить данные извне, так и передать данные кому-то еще. Большинство языков имеют в своем арсенале механизм передачи данных. Это может быть взаимодействие с файловой системой, передача данных по сети или передача данных из одной области памяти в другую в рамках одной программы.

Отличительной чертой многих языков программирования является работа с файлами и потоками. В Java основной функционал работы с потоками сосредоточен в классах из пакета `java.io`. В настоящее время добавлен более современный способ работы с потоками - `Java NIO`. `Java NIO`, или `Java Non-blocking I/O` (иногда переводят как `Java New I/O`) предназначен для реализации высокопроизводительных операций ввода-вывода. Для работы с архивами используются классы из пакета `java.util`.

Ключевым понятием здесь является понятие потока (`stream`). Понятие «поток» в программировании довольно перегружено и может обозначать множество различных концепций. В общем поток (`stream`) можно рассматривать как абстрактное обозначение источника или приемника данных, которые способны обрабатывать информацию. В случае применительно к работе с файлами и вводом-выводом будем говорить о потоке (`stream`), как об абстракции, которая используется для чтения или записи информации (файлов, сокетов, текста консоли и т.д.).

Поток связан с реальным физическим устройством с помощью системы ввода-вывода Java. Может быть определен поток, который связан с файлом и через который можно 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 решаются в Java с помощью потоков.

Большинство классов пакета `java.io` реализуют потоки последовательного доступа.

По направлению движения данных потоки последовательного доступа можно разделить на две группы: поток ввода и поток вывода. 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.

Вторым критерием разделения потоков является тип передаваемых данных. Разделяют два вида потоков ввода/вывода: байтовые и символьные.

В итоге получаем четыре типа потоков. Для каждого из этих типов в Java есть отдельный базовый абстрактный класс. Абстрактными эти классы сделаны потому, что возможна передача данных между файлами, по сети и из одной области памяти в другую. Эту специфику учитывает и реализует уже специальный класс, который расширяет базовый абстрактный класс. При этом базовые функции для всех специальных классов одинаковые, что удобно для программистов. Подчеркнем еще раз, что концептуально все потоки по своей сути представляют одно и тоже. Это придает гибкость и универсальность концепции использования потоков.

Базовые абстрактные типы для четырех основных типов потоков:

- `InputStream`,
- `OutputStream`,
- `Reader`,
- `Writer`.

В основе всех классов, управляющих потоками байтов, находятся два абстрактных класса: `InputStream` (поток ввода) и `OutputStream` (поток 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.

Универсальная схема работы с потоком в упрощенном виде выглядит так:

1. создается экземпляр потока;
2. поток открывается для чтения или записи;
3. производится чтение из потока/запись в поток;
4. поток закрывается.

Замечание.

*При работе с файлами **обязательным** условием является **обработка исключительных ситуаций**. В примерах продемонстрированы два существующих подхода: первый – это обработка исключительной ситуации с помощью блока `try…catch` внутри метода, а второй – перебрасывание исключительной (контролируемой/управляемой) ситуации из метода Java-машине с помощью синтаксической конструкции `throws Исключение`.*

Потоки байтов

Класс InputStream

Класс `InputStream` является базовым для всех классов, управляющих байтовыми потоками ввода. Рассмотрим некоторые его методы:

- `int` **`available`**`()` – оценивает и возвращает количество байтов, которое можно прочитать (или пропустить) из данного входного потока без блокировки.
- `void` **`close`**`()` - закрывает поток и освобождает все системные ресурсы, связанные с потоком. Возможна генерация исключений `IOException`;
- `int` **`read`**`()` – метод является абстрактным, и должен быть определен в классах-наследниках. Предназначен для считывания ровно одного байта из потока, но возвращает при этом значение типа `int`. В случае если считывание произошло успешно, то возвращаемое значение лежит в диапазоне от `0` до `255` и представляет собой полученный байт, дополненный нулями до четырех байтов, которые отводятся под тип `int`. В случае если достигнут конец потока, то возвращаемое значение равно `-1`. Если считать данные из потока не удается из-за каких-либо ошибок или сбоев, то генерируется исключение `IOException`.
- `int` **`read`**`(byte[] buffer)` - считывает байты из потока в массив `buffer`. После чтения возвращает число считанных байтов. Если ни одного байта не было считано, то возвращается число `-1`. Выбрасывает исключение `IOException`, если произошла ошибка ввода-вывода и `NullPointerException`, если `buffer` имеет значение `null`.
- `int` **`read`**`(byte[] buffer, int offset, int length)` - считывает некоторое количество байтов, меньше либо равное `length` (предпринимается попытка прочитать `length` байтов, но реально может быть прочитано меньшее число), из потока в массив `buffer`. При этом считанные байты помещаются в массиве, начиная со смещения `offset`, то есть с элемента `buffer[offset]`. Метод возвращает число успешно прочитанных байтов. Если параметр `length` равен нулю, то байты не считываются, и возвращаемое значение также равно нулю. Если ни одного байта не было считано,

поскольку был достигнут конец файла, то возвращается число -1. Возможна генерация исключений IOException (при ошибках ввода-вывода или если входной поток был закрыт) или NullPointerException (когда buffer равен null), IndexOutOfBoundsException (если offset<0, length<0 или length> buffer.length – offset).

- byte[] **readAllBytes**() считывает все оставшиеся байты из входного потока;
- long **skip**(long number) - пропускает в потоке при чтении некоторое количество байт, которое меньше либо равно number. Метод может по разным причинам пропускать меньшее количество байтов, например 0, если достигнут конец файла до того, как number байтов были пропущены. Методом возвращается фактическое количество пропущенных байтов. Если number отрицательно, метод для класса InputStream всегда возвращает 0, и ни один байт не пропускается. Переопределение данного метода в некоторых подклассах могут по-разному обрабатывать отрицательное значение формального параметра number.

Класс OutputStream

Класс OutputStream является базовым классом для всех классов, которые работают с бинарными потоками записи. Он объявляет три основных метода, необходимых для записи байтов данных в поток, а также имеет методы для закрытия и очистки потоков. Свою функциональность он реализует через следующие методы:

- void **close**() - закрывает поток и освобождает все системные ресурсы, связанные с этим потоком. Закрытый поток не может выполнять операции вывода и не может быть открыт повторно. Возможна генерация исключений IOException;
- void **flush**() - очищает буфер вывода, записывая все его содержимое. Многие операционные системы для повышения производительности буферизируют операции записи. Другими словами, вместо того, чтобы записывать каждый байт по мере его поступления, байты накапливаются в буфере размером от нескольких байтов до нескольких тысяч байтов. Затем, когда буфер заполняется, все данные записываются сразу. Метод flush() принудительно записывает данные независимо от того, заполнен буфер или нет;

- `abstract void` **`write`**`(int b)` - записывает в выходной поток один байт, который представлен целочисленным параметром `b`, метод должен быть определен в дочерних классах. Выбрасывает исключение `IOException`, если произошла ошибка ввода-вывода, в частности, если выходной поток был закрыт;
- `void` **`write`**`(byte[] buffer)` - записывает в выходной поток массив байтов buffer. Возможна генерация исключений `IOException`;
- `void` **`write`**`(byte[] buffer, int offset, int length)` - записывает в выходной поток некоторое число байтов, равное `length`, из массива `buffer`, начиная со смещения `offset`, то есть с элемента `buffer[offset]`. Возможна генерация исключений `IOException`, `NullPointerException`, `IndexOutOfBoundsException`.

Абстрактные классы Reader и Writer

Платформа Java хранит значения символов, используя соглашения `Unicode`. Ввод-вывод потока символов автоматически преобразует этот внутренний формат в локальный набор символов и обратно. В англоязычных регионах локальный набор символов обычно представляет собой 8-битный расширенный набор символов `ASCII`. Таким образом, программа, которая использует потоки символов вместо потоков байтов, автоматически адаптируется к локальному набору символов.

Для обработки символьных потоков в формате `Unicode` применяется отдельная иерархия подклассов абстрактных классов `Reader` и `Writer`, которые почти полностью повторяют функциональность байтовых потоков, но являются более актуальными при передаче текстовой информации.

Абстрактный класс `Reader` предоставляет функционал для чтения текстовой информации. Рассмотрим его основные методы:

- `absract void` **`close`**`()` - закрывает поток ввода и освобождает все связанные с ним системные ресурсы;
- `int` **`read`**`()` - возвращает целочисленное представление следующего символа в потоке. Если таких символов нет, и достигнут конец файла, то возвращается число `-1`. Возможна генерация исключения `IOException`;

- `int` **`read`**`(char[] buffer)` - считывает в массив `buffer` из потока символы, количество которых меньше либо равно длине массива `buffer`. Возвращает количество успешно считанных символов. При достижении конца файла возвращает `-1`. Возможна генерация исключения `IOException`;
- `int` **`read`**`(CharBuffer buffer)` - считывает в объект `CharBuffer` из потока символы. Возвращает количество успешно считанных символов. При достижении конца файла возвращает `-1` Возможна генерация исключения `IOException`;
- `absract int` **`read`**`(char[] buffer, int offset, int count)` - считывает в массив `buffer`, начиная со смещения `offset`, из потока символы, количество которых равно `count` Возможна генерация исключений `IOException` и `IndexOutOfBoundsException`;
- `long` **`skip`**`(long count)` - пропускает количество символов, равное `count`. Возвращает число успешно пропущенных символов. Если поток уже исчерпан до вызова этого метода, символы не пропускаются и возвращается ноль. Возможна генерация исключений `IllegalArgumentException`, если `count` отрицательно и `IOException`, если возникает ошибка ввода/вывода.

Единственные методы, которые должен реализовать подкласс класса `Reader`, это `read(char[], int, int)` и `close()`. Однако большинство подклассов переопределяют некоторые и некоторые другие методы, чтобы обеспечить более высокую эффективность и/или дополнительную функциональность.

Класс `Writer` определяет функционал для всех символьных потоков вывода. Его основные методы:

- `Writer` **`append`**`(char c)` - добавляет в конец выходного потока символ `c`. Возвращает объект `Writer`. Возможна генерация исключений `IOException`;
- `Writer` **`append`**`(CharSequence chars)` - добавляет в конец выходного потока набор символов `chars`. Возвращает объект `Writer`. Возможна генерация исключений `IOException`;
- `abstract void` **`close`**`()` - закрывает поток, предварительно очищая его. Как только поток будет закрыт,

дальнейшие вызовы `write()` или `flush()` приведут к возникновению исключения `IOException`. Повторное закрытие ранее уже закрытого потока не имеет никакого эффекта;

- `abstract void` **`flush`**`()` - очищает буферы потока Если поток сохранил какие-либо символы из различных методов `write()` в буфере, немедленно запишет их в предполагаемое место назначения. Если предполагаемым местом назначения является абстракция, предоставляемая базовой операционной системой, например файл, то очистка потока гарантирует только то, что байты, ранее записанные в поток, передаются операционной системе для записи, но не гарантирует, что они действительно записываются на физическое устройство, такое как дисковод;
- `void` **`write`**`(int c)` - записывает в поток один символ, который имеет целочисленное представление. Возможна генерация исключений `IOException`;
- `void` **`write`**`(char[] buffer)` - записывает в поток массив символов. Возможна генерация исключений `IOException`;
- `absract void` **`write`**`(char[] buffer, int off, int len)` - записывает в поток только несколько символов из массива `buffer`. Причем количество символов равно `len`, а отбор символов из массива начинается с индекса `off` Возможна генерация исключений `IOException` и `IndexOutOfBoundsException`;
- `void` **`write`**`(String str)` - записывает в поток строку. Возможна генерация исключений `IOException`;
- `void` **`write`**`(String str, int off, int len)` - записывает в поток из строки некоторое количество символов, которое равно `len`, причем отбор символов из строки начинается с индекса `off` Возможна генерация исключений `IOException` и `IndexOutOfBoundsException`.

Методы, которые обязан реализовать подкласс класса `Writer`, это `write(char[], int, int)`, `flush()` и `close()`. Однако большинство подклассов переопределяют и другие методы, чтобы обеспечить более высокую эффективность и/или дополнительную функциональность.

Функционал, описанный классами `Reader` и `Writer`, наследуется непосредственно классами символьных потоков, в частности классами

FileReader и FileWriter соответственно, предназначенными для работы с текстовыми файлами.

Чтение и запись файлов. FileInputStream и FileOutputStream

Запись файлов и класс FileOutputStream

Класс FileOutputStream предназначен для записи потоков необработанных байтов, таких, например, как данные изображения. Он является производным от класса OutputStream, поэтому наследует всю его функциональность.

С помощью конструктора класса FileOutputStream задается файл, в который производится запись. Класс поддерживает несколько конструкторов, в том числе:

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

В данных конструкторах файл задается либо через строковый путь, либо через объект File. Путь к файлу filePath может быть как абсолютным, так и относительным. Если файл не существует, то он будет создан. Если файл существует, то он будет перезаписан. Если не удалось создать или открыть файл для записи, то генерируется исключение FileNotFoundException. Второй параметр - append задает способ записи: если он равен true, то данные дозаписываются в конец файла, а при false - файл полностью перезаписывается.

Пример.

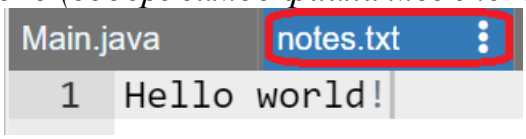
Main.java notes.txt

```
1  import java.io.*;
2  public class Main
3  {
4      public static void main(String[] args) {
5          //строка для записи в файл
6          String text = "Hello world!";
7          //конструкция try-with-resources. Разъяснения в
8          //разделе «закрытие потоков»
9          try(FileOutputStream fos =
10             new FileOutputStream("notes.txt")) {
```

```
11              // перевод строки в байты
12              byte[] buffer = text.getBytes();
13              fos.write(buffer, 0, buffer.length);
14              System.out.println("Файл уже записан");
15          }
16          catch(IOException ex){
17              System.out.println(ex.getMessage());
18          }
19      }
20  }
```

Замечания:

- *При выполнении программы в среде `OnlineGDB` даже если разработчик не создал предварительно файл `notes.txt` самостоятельно, то он будет создан автоматически в новой закладке (содержимое файла можно просмотреть):*



- *При выполнении программы в среде `JDoodle` файл тоже будет создан, но явно просмотреть его содержимое нельзя.*
- *Исключения IOException может возникнуть если в записываемый файл не сможет поместиться записываемая информация, либо файл не будет создан.*

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 конструкции `try…catch`.

При этом необязательно записывать весь массив байтов. Используя перегрузку метода `write()`, можно записать и одиночный байт:

```
fos.write(buffer[0]); // запись первого байта
```

Замечание.

В данном случае в качестве результата в файл будет записан символ `'H'`.

Чтение файлов и класс FileInputStream

`FileInputStream` предназначен для чтения потоков байтов, таких, например, как данные изображения. Он является наследником класса `InputStream` и поэтому реализует все его методы.

Для создания объекта `FileInputStream` можно использовать ряд конструкторов. Наиболее используемая версия конструктора в качестве параметра принимает путь к считываемому файлу:

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

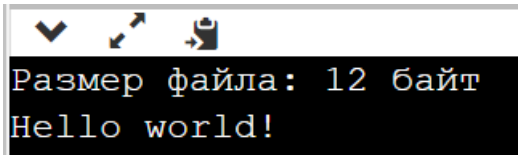
Если файл не может быть открыт, например, по указанному пути такого файла не существует, то генерируется исключение `FileNotFoundException`.

Пример.

Main.java notes.txt

```
1  import java.io.*;
2  public class Main
3  {
4      public static void main(String[] args) {
5          //конструкция try-with-resources. Разъяснения в
6          //разделе «закрытие потоков»
7          try(FileInputStream fin =
8                      new FileInputStream("notes.txt"))
9          {
10             String result = "";
11             System.out.printf("Размер файла: %d байт \n",
12                                 fin.available());
13             int i = -1;
14             while((i = fin.read()) != -1){
15                 result = result + (char) i;
16             }
17             System.out.print(result);
18         }
19         catch(IOException ex){
20             System.out.println(ex.getMessage());
21         }
22     }
23 }
```

Результат работы программы:



В данном случае в программе считывается каждый отдельный байт в переменную `i` с помощью инструкции:

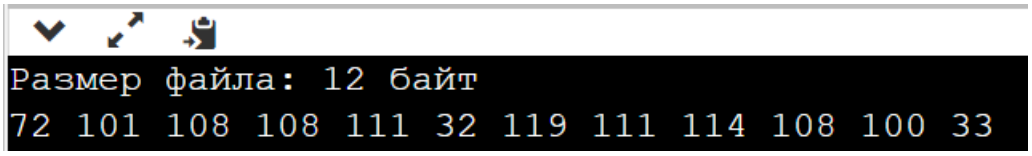
```
while((i = fin.read())!=-1){
```

Когда в потоке больше нет данных для чтения, метод `read()` возвращает число `-1`.

Затем каждый считанный байт конвертируется в объект типа `char` и выводится на консоль. Если явное преобразование типов не выполнять, используя код цикла в виде:

```
while((i = fin.read()) != -1) {
                System.out.print(i + " ");
},
```

то на экране отобразятся коды символов:



Можно также считать данные в массив байтов и затем производить с ним манипуляции:

```
byte[] buffer = new byte[fin.available()];
// считаем файл в буфер
fin.read(buffer, 0, fin.available());
System.out.println("Содержимое файла:");
for(int i = 0; i < buffer.length; i++){

    System.out.print((char) buffer[i]);
}
```

Объединим оба примера и выполним чтение из одного (`notes.txt`) и запись в другой (`notes_new.txt`) файл.

Пример.

Main.java | notes.txt | notes_new.txt

```
1  import java.io.*;
2  public class Main
3  {
4     public static void main(String[] args) {
5        //конструкция try-with-resources. Разъяснения в
6        //разделе «закрытие потоков»
7        try(FileInputStream fin =
8                  new FileInputStream("notes.txt");
9            FileOutputStream fos =
10                 new FileOutputStream("notes_new.txt"))
11       {
12          byte[] buffer = new byte[fin.available()];
13          //считываем буфер
14          fin.read(buffer, 0, buffer.length);
15          //записываем из буфера в файл
16          fos.write(buffer, 0, buffer.length);
17       }
18       catch(IOException ex){
19          System.out.println(ex.getMessage());
20       }
21    }
22 }
```

Замечание.

При выполнении программы в среде `OnlineGDB` даже если разработчик не создал предварительно файл `notes_new.txt` самостоятельно, то он будет создан автоматически в новой закладке (содержимое файла можно просмотреть):

Main.java | notes.txt | notes_new.txt

```
1  Hello world!
```

Классы `FileInputStream` и `FileOutputStream` предназначены прежде всего для записи двоичных файлов, то есть для записи и чтения байтов. И хотя (как показывают примеры) они также могут использоваться для работы с текстовыми файлами, но все же для этой задачи больше подходят другие классы.

Закрытие потоков

В Java существуют средства автоматического управления памятью. Поиском и освобождением ненужных участков в памяти в JVM занимается специальный процесс, который называется `garbage collector` (`GC`). Он работает в фоновом режиме. Программисту на Java, не нужно беспокоиться о таких проблемах, как уничтожение объектов, если они больше не используются. Однако несмотря на то, что в Java этот процесс выполняется автоматически, он не дает 100% гарантии. Не зная, как устроен сборщик мусора и память Java, программист может создать объекты, которые не будут удаляться `GC`, даже если они больше не используются.

В частности при работе с файловыми потоками может произойти так называемая «утечка ресурса». Если программа использует файл, то по окончании работы с ним, поток должен быть закрыт с помощью метода `close()` при этом файл автоматически помечается в операционной системе как свободный, чтобы другие программы могли обращаться к этому ресурсу. Если не сделать этого, то ресурс так и останется помеченный как занятый и операционная система будет считать его используемым и никто другой не сможет во время работы программы обратиться к этому ресурсу.

Таким образом при завершении работы с потоком его надо закрыть с помощью метода `close()`, который определен в интерфейсе `Closeable`. Метод `close()` имеет следующей контракт:

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

Интерфейс `Closeable` 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.

Существует *два способа* закрытия файла. *Первый* традиционный заключается в использовании блока `try…catch…finally`.

Пример.

Main.java

```
1  import java.io.*;
2  public class Main
3  {
4      public static void main(String[] args) {
5          FileInputStream fin = null;
```

```
6           try {
7               fin = new FileInputStream("notes.txt");
8               int i=-1;
9               while((i = fin.read()) != -1) {
10                  System.out.print((char)i);
11              }
12          }
13          catch(IOException ex) {
14              System.out.println(ex.getMessage());
15          }
16          finally {
17              try {
18                  if(fin != null) fin.close();
19              }
20              catch(IOException ex) {
21                  System.out.println(ex.getMessage());
22              }
23          }
24      }
25  }
```

Результат работы программы:

```
notes.txt (No such file or directory)
```

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

Замечание.

В примере выше метод `close()` (строка 18) может сгенерировать исключение. Если же при этом ранее в основном коде работы с ресурсом тоже возникнет исключение, то последнее перезатрется исключением из `close()`. Информация об исходной ошибке пропадет и уже будет невозможно узнать, что было причиной исходного исключения.

В настоящее время можно использовать еще один способ закрытия файла, который автоматически вызывает метод `close()`. Этот способ заключается в использовании конструкции `try-with-resources` (try-с-ресурсами). Данная конструкция работает с объектами, которые реализуют интерфейс `AutoCloseable`. Так как все классы потоков реализуют интерфейс `Closeable`, который в свою очередь наследуется от `AutoCloseable`, то их также можно использовать в данной конструкции

Перепишем предыдущий пример с использованием конструкции `try-with-resources`.

Пример.

Main.java

```
1  import java.io.*;
2  public class Main {
3      public static void main(String[] args) {
4          //конструкция try-with-resources
5          try(FileInputStream fin = new
6                  FileInputStream("notes.txt"))
7          {
8              int i = -1;
9              while((i = fin.read()) != -1) {
10                 System.out.print((char) i);
11             }
12         }
13         catch(IOException ex){
14             System.out.println(ex.getMessage());
15         }
16     }
17 }
```

Результат работы программы совпадает с предыдущим.

Формат конструкции `try-with-resources`, следующий:

```
try(ИмяКласса имяОбъекта = КонструкторКласса(~~парам~~));
```

Отличительной особенностью является использование круглых скобок у ключевого слова `try`. В круглых скобках может быть перечислено через точку с запятой несколько операторов, открывающих разные потоки.

Подобная конструкция использовалась, в одном из примеров ранее.

Пример.

```
try(FileInputStream fin =
          new FileInputStream("notes.txt");
    FileOutputStream fos =
          new FileOutputStream("notes_new.txt"))
{
   //блок try
}
```

Данная конструкция также не исключает использования блоков `catch`. После окончания работы в блоке `try` у указанного в круглых скобках ресурса или перечисленных ресурсов (в данном случае у объекта `FileInputStream`) автоматически вызывается метод `close()`.

Замечание.

Если исключение будет выброшено в основном коде и в методе `close()`, то приоритетнее будет первое исключение, а второе исключение будет подавлено, но информация о нем сохранится.

Оператор `try`-с-ресурсами упрощает процесс освобождения ресурсов, исключая даже возможность забыть освободить используемый ресурс по небрежности. Поэтому такой способ освобождения ресурсов рекомендуется применять при всякой возможности в процессе разработки нового кода.

Класс Scanner

Рассмотрим подробно класс `Scanner`, хотя им уже многократно пользовались для ввода данных. Он предназначен для извлечения информации практически из любых источников.

`Scanner` разбивает входные данные на токены (лексемы), используя шаблон разделителя, который по умолчанию соответствует пробелу. Полученные токены могут быть затем преобразованы в значения различных типов с использованием различных `next`-методов.

Некоторые конструкторы класса:

- `Scanner(String source)`
- `Scanner(InputStream source)`
- `Scanner(InputStream source, String charsetName)`
- `Scanner(InputStream source, Charset charset)`

- `Scanner(File source)`
- `Scanner(File source, String charsetName)`
- `Scanner(File source, Charset charset)`
- `Scanner(Readable source)`
- `Scanner(Path source)`
- `Scanner(Path source, String charset)`
- `Scanner(Path source, Charset charset)`

где `source` - источник входных данных, `charset` - кодировка источника, а `charsetName` тип кодировки, используемый для преобразования байтов из потока в символы для сканирования.

Объект класса `Scanner` читает наборы символов (токены или лексемы) с набором разделителей из источника, указанного в конструкторе. «По умолчанию» набором разделителей являются пробельные символы, а для строк – символ перевода на следующую строку.

Формат создания объекта класса `Scanner`:

```
Scanner ОбъектСканера = new Scanner(ИницОбъект);
```

Пример.

```
Scanner console = new Scanner(System.in);
```

`Scanner` поддерживает чтение токенов типа `String`, всех примитивных типов языка Java (кроме `char`), а также `BigInteger` и `BigDecimal`. Кроме того, числовые значения могут использовать разделители тысяч. Таким образом, в US-локали будет правильно читается строка "1 237 712" как представляющая целочисленное значение.

Методов у класса `Scanner` очень много, но можно по частоте использования в учебных программах выделить следующие две группы методов.

Первую группу составляют методы, сначала пропускающие все входные данные, соответствующие шаблону разделителя, а затем считывающие очередной токен из потока или объекта `ИницОбъект`, с которым связан объект сканера `ОбъектСканера`, и преобразующие его в данное определенного типа:

- **`nextLine`**`()` – метод считывает и возвращает оставшуюся часть текущей строки из `ИницОбъект`, не включая разделитель строк в конце. Позиция устанавливается в начале следующей строки;
- **`nextBoolean`**`()` - метод, считывающий булевское значение из `ИницОбъект`;

- **nextByte**() - метод, считывающий значение типа byte из ИницОбъект;
- **nextDouble**() - метод, считывающий значение типа double из ИницОбъект;
- **nextFloat**() - метод, считывающий значение типа float из ИницОбъект;
- **nextInt**() - метод, считывающий значение типа int из ИницОбъект;
- **nextLong**() - метод, считывающий значение типа long из ИницОбъект;
- **nextShort**() - метод, считывающий значение типа short из ИницОбъект.

Следующей группой методов является служебные методы для анализа следующего токена, возвращающие true, если следующий токен во входных данных сканера может быть интерпретирован как значение определенного типа:

- **hasNextLine**() - метод, который возвращает значение true или false, определяя является ли порция данных, которую возвратил метод nextLine() строкой;
- **hasNextInt**() - метод проверяет, является ли следующая порция введенных данных целым числом, или нет (возвращает, соответственно, true или false);
- **hasNextByte**(), **hasNextShort**(), **hasNextLong**(), **hasNextFloat**(), **hasNextDouble**() - все эти методы делают то же для остальных типов данных.

Использование методов типа hasNexType() перед вызовом методов типа nextType() позволяют избежать возникновения исключительных ситуаций и часто используется, поскольку обработка исключительных ситуаций достаточно затратна в плане ресурсов.

Сканер по умолчанию интерпретирует числа как десятичные, но с помощью метода useRadix(int) можно установить другое основание системы счисления для вводимых числовых данных. Метод reset() устанавливает основание системы счисления сканера «по умолчанию» равное 10.

Приведем менее привычный пример использования конструктора Scaner-а.

Пример.

Main.java

```
1  import java.io.*;
2  import java.util.Scanner;
3  public class Main
4  {
5      public static void main(String[] args) {
6          Scanner scanner =
7              new Scanner("Люблю тебя, Петра творенье,\n" +
8                  "Люблю твой строгий, стройный вид,\n" +
9                  "Невы державное теченье,\n" +
10                 "Береговой ее гранит");
11         while (scanner.hasNextLine()) {
12             System.out.println(scanner.nextLine());
13         }
14         scanner.close();
15     }
16 }
```

Результат работы программы:

```
Люблю тебя, Петра творенье,
Люблю твой строгий, стройный вид,
Невы державное теченье,
Береговой ее гранит
```

В этом же примере есть еще один метод, на который нужно обязательно обратить внимание - `close()`. Как и любой объект, работающий с потоками ввода-вывода, сканер должен быть закрыт по завершении своей работы, чтобы больше не потреблять ресурсы компьютера.

Замечание.

При закрытии объекта сканера, если его базовый читаемый объект реализует интерфейс `Closeable`*, также вызывается метод* `close()` *и для него.*

Далее для проверки наличия произвольной лексемы (значимого набора символов) объект `Scanner`-а использует метод `boolean hasNext()`. Произвольная лексема считывается методом

`String next()`. После извлечения любой лексемы текущий указатель устанавливается перед следующей лексемой.

Проверка конкретного типа производится с помощью одного из методов группы `boolean hasNextТип()` или `boolean hasNextТип (int radix)`, где `radix` - основание системы счисления.

Объект класса `Scanner` определяет границы лексемы, основываясь на наборе разделителей. Можно задавать разделители с помощью метода `useDelimiter(Pattern pattern)` или `useDelimiter(String regex)`, где `pattern` и `regex` содержит набор разделителей в виде регулярного выражения.

Пример.

Main.java

```
1  import java.io.*;
2  import java.util.Scanner;
3  public class Main
4  {
5      public static void main(String[] args) {
6          String init = "1.3;2.0;8.5;4.8; 9.0;   1;";
7          Scanner scanner =
8                  new Scanner(init).useDelimiter(";\\s*");
9          while (scanner.hasNext()) {
10             System.out.print(scanner.next() + " ");
11         }
12         scanner.close();
13     }
14 }
```

Результат работы программы:

```
1.3 2.0 8.5 4.8 9.0 1
```

В методе `useDelimiter()` указано, что разделителем является знак `';'` и регулярное выражение `\\s*` указывающее, что после точки с запятой может быть произвольное (от нуля и больше) число пробелов.

Можно вместо строки «`;\\s*`», указать строку «`\\s*;\\s*`». Это будет означать, что произвольное число пробелов может быть до и после символа-разделителя `';'`.

Замечание.

Работа с регулярными выражениями не входит в данный курс и приведен только с целью демонстрации работы метода `useDelimiter()`.

Работа с `FileWriter` и `FileReader`

Отметим, что работа с классом `FileWriter` ничем не отличается от работы с классом `FileOutputStream`. Все можно свести к замене наименования классов.

Однако с подклассом `FileReader` такое утверждение, к сожалению, не верно. Рассмотрим более подробно правила работы с подклассом `FileReader`.

Первое что необходимо отметить, что при работе с `FileReader` необходимо объявить не только объект этого класса, но еще и объект класса `Scaner`. При этом конструктор `Scaner`-а должен быть проинициализирован уже созданным объектом `FileReader`-а. Формат:

```
FileReader Объект = new FileReader("Путь+имяФайла");
Scanner ОбъектСканера = new Scanner(Объект);
```

Далее необходимо использовать методы класса Scanner, уже обсужденные выше.

Пример.

```
FileReader fr = new FileReader("file1.txt");
Scanner scan = new Scanner(fr);
while (scan.hasNextLine()) {
    System.out.println(scan.nextLine());
}
```

Замечание.

Работа с методами `nextLine()` *и* `hasNextLine()` *полностью аналогична использованию итераторов в коллекциях*

Переделаем последний пример предыдущего пункта на использование объектов подкласса `FileWriter` суперкласса `Writer` и подкласса `FileReader` суперкласса `Reader`. В отличие от предыдущего примера, выполним запись, а затем чтение из одного и того же файла.

Отличительной особенностью исполнения данного примера является разделение работы с файлами на методы.

Кроме того, в данном примере исключительная ситуация будет пробрасываться через цепочку вызовов методами Java-машине через конструкцию `throws Исключение`.

Пример.

Main.java file.txt

```
1  import java.io.*;
2  import java.util.Scanner;
3  public class Main
4  {
5      private static void createFile(String str,
6                                     String fileName) throws
7                                     IOException {
8          FileWriter fw = new FileWriter(fileName);
9          fw.write(str);
10         fw.close();
11     }
12     private static String readFile(String fileName) throws
13                                    IOException {
14         FileReader fr = new FileReader(fileName);
15         Scanner scan = new Scanner(fr);
16         String str = scan.nextLine();
17         scan.close();
18         fr.close();
19         return str;
20     }
21     public static void main(String[] args) throws
22                                           IOException {
23         createFile("Привет, Мир", "file.txt");
24         String result = readFile("file.txt");
25         System.out.println(result);
26     }
27 }
```

Результат работы программы:

Main.java file.txt

```
1  Привет, Мир
```

```
Привет, Мир
```

Буферизованные потоки

Базовые потоки ввода-вывода используют небуферизованный ввод-вывод. Это означает, что каждый запрос на чтение или запись обрабатывается непосредственно базовой операционной системой. А поскольку каждый такой запрос часто инициирует доступ к диску, сетевую активность или какую-либо другую относительно дорогостоящую операцию, то использование таких потоков не слишком эффективно.

Буферизация позволяет избежать необходимости обращения к источнику или приемнику при выполнении каждой отдельной операции чтения или записи.

Буферизованные входные потоки считывают данные из области памяти, называемой буфером, а собственный `API` (`Application Programming Interface`) ввода вызывается только тогда, когда буфер пуст. Точно так же буферизованные потоки вывода записывают данные в буфер, а собственный `API` вывода вызывается только при заполнении буфера или принудительной его очистке.

Рассмотрим подробно алгоритм работы буферизированных потоков ввода. Для ввода (чтения) из потока вызывается его метод `read()`. Если при выполнении `read()` *буферизированного* потока выясняется, что буфер потока пуст, то:

1. вызывается метод `read()` основного потока-*источника* данных;
2. буфер заполняется максимально возможной порцией данных (это или размер буфера, или объем данных источника);

После заполнения буфера управление передается методу `read()` *буферизированного* потока и данные будут извлекаться из буфера, пока его содержимое не будет исчерпано, после чего объект буферизированного потока вновь будет вынужден вызвать метод `read()` потока-*источника* и т.д.

Совершенно аналогично можно описать алгоритм работы буферизированных потоков вывода. Запись осуществляется методом `write()` буферизированного потока в буфер. Когда буфер данных заполняется, вызывается `write()` основного потока-*приемника*, который освобождает буфер и т.д. Такой механизм позволяет превратить последовательность запросов на вывод (или запись) небольших порций данных в поток-буфер в единственный вызов метода `write()` потока-*приемника*.

Метод `flush()` в данном случае осуществляет очистку буфера, т.е. вывод (запись) всех данных из буфера в поток-*приемник* вызовом его метода `write()`.

Существует четыре класса буферизованных потоков, используемых для обертки небуферизованных потоков: `BufferedInputStream` и `BufferedOutputStream` для создания буферизованных потоков байтов, `BufferedReader` и `BufferedWriter` для создания буферизованных потоков символов.

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.

`BufferedOutputStream` определяет два конструктора:

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

Первый параметр (`outputStream`) - это поток вывода, который унаследован от `OutputStream`, а второй параметр - размер буфера.

Рассмотрим на примере записи в файл.

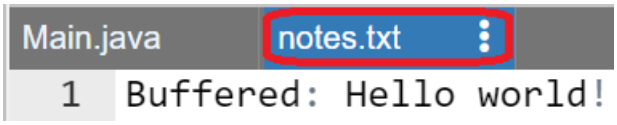
Пример.

Main.java notes.txt

```
1  import java.io.*;
2  public class Main
3  {
4      public static void main(String[] args) {
5          //строка для записи
6          String text = "Buffered: Hello world!";
7          //создаем BufferedOutputStream с размером буфера
8          //«по умолчанию» 16384 bytes = 16 KB.
9          try(FileOutputStream out =
10                 new FileOutputStream("notes.txt");
11             BufferedOutputStream bos =
12                     new BufferedOutputStream(out))
13         {
14             //перевод строки в байты
15             byte[] buffer = text.getBytes();
```

```
16              bos.write(buffer, 0, buffer.length);
17          }
18          catch(IOException ex){
19              System.out.println(ex.getMessage());
20          }
21      }
22  }
```

Результат работы программы:



Класс `BufferedOutputStream` в конструкторе принимает в качестве параметра объект `OutputStream` - в данном случае это файловый поток вывода `FileOutputStream`. И также производится запись в файл. Опять же `BufferedOutputStream` не добавляет много новой функциональности, он просто оптимизирует действия потока вывода.

Дадим пояснения по использованному в примере методу `void write(byte[] buf, int off, int len)`. Он записывает `len` байт из массива байтов `buf`, начиная со смещения `off`, в буферизованный выходной поток. Обычно этот метод сохраняет байты из данного массива в буфер потока, при необходимости сбрасывая буфер в базовый выходной поток. Однако если запрошенная длина больше либо равна размеру буфера потока, то этот метод очищает буфер и записывает байты непосредственно в базовый выходной поток.

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 - размер буфера.

Например, буферизируем считывание данных из потока `FileInputStream`.

Пример.

Main.java notes.txt

```
1  import java.io.*;
2  public class Main
3  {
4      public static void main(String[] args) {
5          //Из FileInputStream создаем
6          //BufferedInputStream с размером буфера
7          //16384 (16 KB).
8          try(FileInputStream fin =
9                      new FileInputStream("notes.txt");
10             BufferedInputStream br =
11                  new BufferedInputStream(fin, 16384))
12         { //блок try начало
13             System.out.printf("Размер файла: %d байт \n",
14                                br.available());
15             int i = -1;
16             while((i = br.read()) != -1) {
17             System.out.print((char) i);
18             }
19         } //блок try конец
20         catch(IOException ex) {
21             System.out.println(ex.getMessage());
22         }
23     }
24 }
```

Результат работы программы:

```
Размер файла: 25 байт
Buffered: Hello world!
```

Класс `BufferedInputStream` в конструкторе принимает объект `fin`. В данном случае таким объектом является экземпляр класса `FileInputStream`.

Как и все потоки ввода `BufferedInputStream` обладает методом `read()`, который считывает данные. Соответственно здесь также считываются данные с помощью метода `read()`.

Класс `BufferedInputStream` просто оптимизирует производительность при работе с потоком `FileInputStream`.

Естественно, вместо `FileInputStream` может использоваться любой другой класс, который унаследован от `InputStream`.

Буферизация символьных потоков. `BufferedReader` 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 каждый из конструкторов принимает поток вывода (`out`), в который надо осуществить запись. Второй параметр (`sz`) указывает на размер буфера.

Замечание.

Так как `BufferedWriter` потомок класса `Writer`, то он может использовать все те методы для чтения из потока, которые определены в `Reader`.

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

Класс `BufferedReader` имеет аналогичные конструкторы:

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

Замечание.

Так как `BufferedReader` наследуется от класса Reader, то он может использовать все те методы для чтения из потока, которые определены в `Reader`, а также `BufferedReader` определяет свой собственный метод `readLine()`, который позволяет считывать из потока построчно.

Очевидно, что применительно к записи/чтению в/из файла классы должны работать в паре:

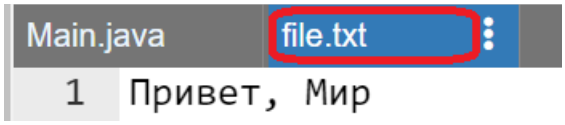
- `FileWriter` и `BufferedWriter`;
- `FileReader` и `BufferedReader`.

Пример.

Main.java file.txt

```
1  import java.io.*;
2  import java.util.Scanner;
3  public class Main
4  {
5      private static void createFile(String str,
6                                     String fileName)
7                                     throws
8                                     IOException {
9          FileWriter fw = new FileWriter(fileName);
10         BufferedWriter bw =
11                 new BufferedWriter(fw, 16384);
12         bw.write(str);
13         bw.close();
14         fw.close();
15     }
16
17     private static String readFile(String fileName)
18                                    throws
19                                    IOException {
20         FileReader fr = new FileReader(fileName);
21         BufferedReader br =
22                 new BufferedReader(fr, 16384);
23         Scanner scan = new Scanner(br);
24         String str = scan.nextLine();
25         scan.close();
26         br.close();
27         fr.close();
28         return str;
29     }
30
31     public static void main(String[] args)
32                                 throws IOException {
33         createFile("Привет, Мир", "file.txt");
34         String result = readFile("file.txt");
35         System.out.println(result);
36     }
37 }
```

Результат работы программы:



Сериализация/десериализация

Сериализация (в программировании) - процесс перевода какой-либо структуры данных в последовательность битов для хранения. Обратной к операции сериализации является операция десериализации (структуризации) - восстановление начального состояния структуры данных из битовой последовательности.

Сериализация предшествует записи состояния объекта в поток, соответственно десериализация проводится в процессе извлечения или восстановления состояния объекта из потока.

Предположим, у разработчика есть сложная иерархия классов, в каждом из которых с десяток полей. Некоторые поля ссылаются на объекты других классов или, того хуже, содержат коллекции объектов. При использовании сериализатора, вам достаточно одной инструкции для сохранения словаря, содержащего объекты из этой иерархии, в файл.

Сериализация при работе с файлами – это простейший пример. Она необходима при передаче сложно-структурированных данных куда-либо (т.е. использования любых потоков).

Если же *не* пользоваться сериализацией, то будет необходимо писать длинный и сложный код для ручного сохранения всего этого изобилия в файл, то в итоге получится объемный и сложный код.

Поэтому сериализация не только очень удобна, но и позволяет значительно упростить разработку программы, когда идет работа со сложными объектами.

Дополнительным плюсом сериализации является сохранение кроссплатформенности. Не зависимо от типа операционной системы, сериализация переводит объект в поток байтов. Поэтому можно сериализовать объект на компьютере с одной ОС, а восстановить его, или десериализовать, совершенно на другом компьютере с другой ОС. Кроме того, если необходимо передать объект по сети, то можно сериализовать объект, сохранить его в файл и передать по сети получателю. А получатель сможет восстановить полученный объект.

Интерфейс `Serializable`

Сразу надо сказать, что сериализовать можно только те объекты, которые реализуют интерфейс `Serializable`. Этот интерфейс не определяет никаких методов, просто он служит указателем системе, что объект класса, реализующий его, может быть сериализован.

Сериализация. Класс `ObjectOutputStream`

Для сериализации объектов в поток используется класс `ObjectOutputStream`. Он записывает данные в поток.

Для создания объекта `ObjectOutputStream` в *конструктор* передается поток, в который производится запись:

```
ObjectOutputStream(OutputStream out)
```

Для записи данных `ObjectOutputStream` использует ряд методов, среди которых можно выделить следующие:

- `void` **`close`**`()` - закрывает поток;
- `void` **`flush`**`()` - очищает буфер и сбрасывает его содержимое в выходной поток;
- `void` **`write`**`(byte[] buf)` - записывает в поток массив байтов;
- `void` **`write`**`(int val)` - записывает в поток один младший байт из `val`;
- `void` **`writeBoolean`**`(boolean val)` - записывает в поток значение `boolean`;
- `void` **`writeByte`**`(int val)` - записывает в поток один младший байт из `val`;
- `void` **`writeChar`**`(int val)` - записывает в поток значение типа `char`, представленное целочисленным значением;
- `void` **`writeDouble`**`(double val)` - записывает в поток значение типа `double`;
- `void` **`writeFloat`**`(float val)` - записывает в поток значение типа `float`;
- `void` **`writeInt`**`(int val)` - записывает целочисленное значение `int`;
- `void` **`writeLong`**`(long val)` - записывает значение типа `long`;

- `void` **`writeShort`**`(int val)` - записывает значение типа `short`;
- `void` **`writeUTF`**`(String str)` - записывает в поток строку в кодировке `UTF-8`;
- `void` **`writeObject`**`(Object obj)` - записывает в поток отдельный объект.

Эти методы охватывают весь спектр данных, которые можно сериализовать.

Пример.

Main.java | person.dat

```
1  import java.io.*;
2  class Person implements Serializable {
3      private String name;
4      private int age;
5      //методы
6      Person(String n, int a){
7          name = n;
8          age = a;
9      }
10     String getName() { return name; }
11     int getAge()     { return age; }
12 }
13
14 public class Main
15 {
16     public static void main(String[] args) {
17         try( ObjectOutputStream oos =
18                         new ObjectOutputStream(
19                         new FileOutputStream("person.dat")))
20         {
21             Person p = new Person("Sam", 33);
22             oos.writeObject(p);
23         }
24         catch(Exception ex){
25             System.out.println(ex.getMessage());
26         }
27     }
28 }
```

Результат работы программы:

Main.java | person.dat

```
1  ¬í··sr··PersonæÕÂo%-·7···I··ageL··namet··Ljava/lang/String;xp···!t··Sam
```

Десериализация. Класс `ObjectInputStream`

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

```
ObjectInputStream(InputStream in)
```

Функционал `ObjectInputStream` сосредоточен в методах, предназначенных для чтения различных типов данных. Рассмотрим основные методы этого класса:

`void` **`close`**`()` - закрывает поток;

`int` **`skipBytes`**`(int len)` - пропускает при чтении несколько байт, количество которых равно `len`;

`int` **`available`**`()` - возвращает количество байт, доступных для чтения;

`int` **`read`**`()` - считывает из потока один байт и возвращает его целочисленное представление;

`boolean` **`readBoolean`**`()` - считывает из потока одно значение `boolean`;

`byte` **`readByte`**`()` - считывает из потока один байт;

`char` **`readChar`**`()` - считывает из потока один символ `char`;

`double` **`readDouble`**`()` - считывает значение типа `double`;

`float` **`readFloat`**`()` - считывает из потока значение типа `float`;

`int` **`readInt`**`()` - считывает целочисленное значение `int`;

`long` **`readLong`**`()` - считывает значение типа `long`;

`short` **`readShort`**`()` - считывает значение типа `short`;

`String` **`readUTF`**`()` - считывает строку в кодировке `UTF-8`;

`Object` **`readObject`**`()` - считывает из потока объект.

Пример.

Main.java | person.dat

```
1  import java.io.*;
2  class Person implements Serializable {
3      private String name;
4      private int age;
5      //методы
6      Person(String n, int a){
7          name = n;
```

```
        age = a;
    }
    String getName() { return name; }
    int getAge()     { return age; }
}

public class Main
{
    public static void main(String[] args) {
        try( ObjectInputStream ois =
                new ObjectInputStream(
                new FileInputStream("person.dat")))
        {
            Person p = (Person) ois.readObject();
            System.out.printf("Name: %s \t Age: %d \n",
                                p.getName(), p.getAge());
        }
        catch(Exception ex){
            System.out.println(ex.getMessage());
        }
    }
}
```

Результат работы программы:

Теперь совместим сохранение и восстановление из файла на примере списка объектов.

Пример.

Main.java people.dat

```
import java.io.*;
import java.util.*;
class Person implements Serializable {
    private String name;
    private int age;
    //методы
    Person(String n, int a){
        name = n;
        age = a;
    }
```

```
    String getName() { return name; }
    int getAge()     { return age; }
}

public class Main
{
    public static void main(String[] args)
                                throws IOException,
                                ClassNotFoundException {
        String fileName = "people.dat";
        //создадим список объектов, которые будем записывать
        ArrayList<Person> writeList = new ArrayList();
        writeList.add(new Person("Tom", 30));
        writeList.add(new Person("Sam", 33));

        ObjectOutputStream oos = new ObjectOutputStream(
                            new FileOutputStream(fileName));
        oos.writeObject(writeList);
        System.out.println("Файл уже записан!");
        oos.close();
        // десериализация в новый список
        ArrayList<Person> readArray = new ArrayList();
        ObjectInputStream ois = new ObjectInputStream(
                                new FileInputStream(fileName));
        readArray = ((ArrayList<Person>) ois.readObject());
        ois.close();

        for(var p : readArray)
            System.out.printf("Name: %s \t Age: %d \n",
                                p.getName(), p.getAge());
    }
}
```

Результат работы программы:

```
Файл уже записан!
Name: Tom         Age: 30
Name: Sam         Age: 33
```

Исключение данных из сериализации

По умолчанию сериализуются все переменные объекта.

Но следует учитывать, что статические члены класса принадлежат классу, а не объекту, и сериализации не подлежат. Поэтому поля,

помеченные спецификатором `static` при десериализации, если в области видимости уже существуют объекты того же типа, получают значение, которое поле имеет на момент десериализации в существующем объекте. В случае отсутствия в области видимости объектов такого типа - получают значение «по умолчанию».

Кроме этого, иногда хотелось бы, чтобы некоторые поля были исключены из сериализации. Это может происходить в тех случаях:

- Если значения полей вычисляются программно. Например, класс, описывающий заказ в интернет-магазине. Каждый заказ, состоит из списка товаров и итоговой стоимости. Общая стоимость заказа складывается из суммарной стоимости каждого товара и вычисляется программно, как сумма стоимости всех товаров. Поэтому итоговую стоимость хранить нет смысла.
- Если поля содержат конфиденциальную (в определенном смысле секретную) информацию. Например, пароли в целях безопасности также не сериализуют, чтобы не допускать утечки приватной информации за пределы JVM.
- Если поля не реализуют интерфейс `Serializable`. Например, иногда класс содержит поля, являющимися ссылками на объекты других классов, которые не реализуют интерфейс `Serializable`. Примеры таких полей логгеры, потоки ввода-вывода, объекты, которые хранят соединения с базой данных и прочие служебные классы. Если попытаться сериализовать объект, который содержит несериализуемые поля, возникнет ошибка `java.io.NotSerializableException`.
- Если поля не являются частью состояния объекта. Сериализовать следует только поля с информацией о состоянии объекта. Поля, добавленные для отладки или для выполнения какой-то служебной функции, которые не несут информации о состоянии объекта, сериализации не подлежат.

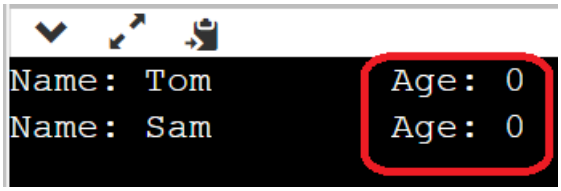
Все поля, которые не должны быть сериализованы помечаются спецификатором `transient`. Например, исключим из сериализации объекта `Person` переменную `age`.

Пример.

```
class Person implements Serializable {
    private String name;
    private transient int age;
```

```
    //методы
    Person(String n, int a) {
        name = n;
        age = a;
    }
    String getName() {return name;}
    int getAge(){ return age;}
}
```

Если данное определение класса со словом `transient` вставить в предыдущий пример, то после работы программы можно получить следующий результат:



После десериализации поле, помеченное спецификатором `transient` получает значение «по умолчанию», соответствующее его типу. Объектный тип «по умолчанию» инициализируется значением `null`. Но в данном случае тип поля `int`, и оно получает значение «по умолчанию» `0`.

Ввод/вывод сложных объектов

Методы `writeObject()` и `readObject()` просты в использовании, но содержат достаточно сложную логику управления объектами. Если поля класса являются примитивными значениями, то никаких дополнительных усилий при их использовании не понадобится. На практике очень часто многие объекты содержат в качестве полей ссылки на другие объекты. Если с помощью методов `readObject()` и `readObject()` нужно восстановить или записать объект из или в поток, они должны быть в состоянии восстановить или записать все объекты, на которые ссылается исходный объект. Эти дополнительные объекты могут иметь свои собственные ссылки и т.д. В этой ситуации `readObject()` (или `writeObject()`) проходит через всю сеть ссылок на объекты и восстанавливает (или записывает) все объекты в этой сети в поток.

Таким образом, один вызов метода `writeObject()` может привести к записи в поток большого количества объектов.

Это показано на следующем рисунке 1, где `writeObject()` вызывается для записи одного объекта с именем `a`. Этот объект содержит ссылки на объекты `b` и `c`, а `b` содержит ссылки на `d` и `e` . При вызове `writeObject(a)` записывается не только `a`, но и все объекты, необходимые для воссоздания `a`, поэтому остальные четыре объекта также записываются. Когда `a` считывается обратно с помощью `readObject()`, остальные четыре объекта также считываются, и все исходные ссылки на объекты сохраняются.

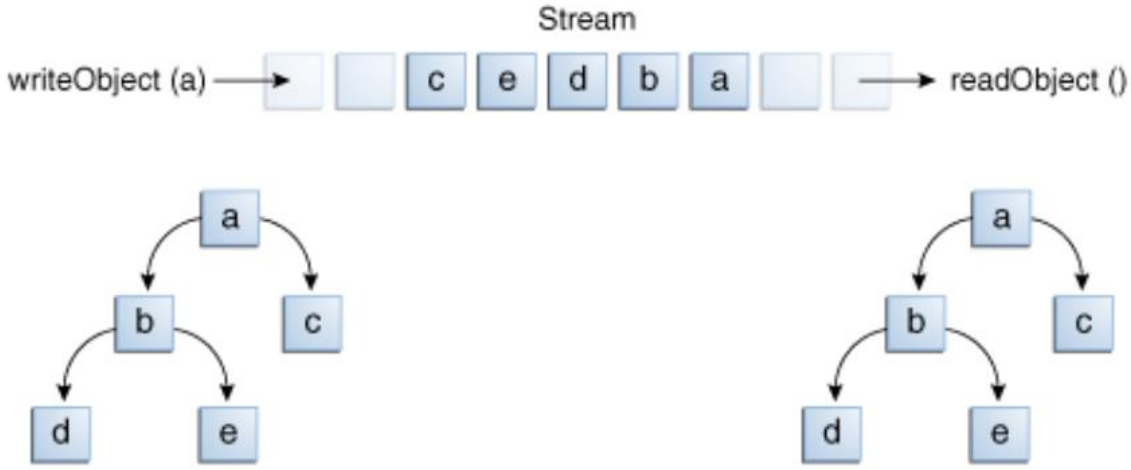


Рисунок 1 - Схема сериализации/десериализации объекта, водержащего ссылки на другие объекты

Замечание.

Графом называется система объектов произвольной природы (вершин) и парных связей (ребер), соединяющих некоторые пары этих объектов (вершин).

Как видно из рисунка 1, в процессе сериализации вместе с сериализуемым объектом сохраняется его граф объектов. Т.е. все связанные с этим объектом, объекты других классов так же будут `сериализованы` вместе с ним. А значит, все связанные с этим объектом, объекты других классов, которые будут сериализованы должны так же, реализовывать интерфейс `Serializable`.

Пример.

Main.java | people.dat | person.out

```
1  import java.io.*;
2
3  class Address implements Serializable {
4      private String home;
5
6      public Address(String home) {
7          this.home = home;
8      }
9      public String getHome() {
10         return home;
11     }
12      @Override
13      public String toString() {
14          return "Address{" +
15                  "home='" + home
16                  + '}';
17      }
18 }
19
20  class Person implements Serializable {
21     private String name;
22     private int age;
23     //ссылка на сериализуемый объект
24     private Address address;
25
26     public Person(String name, int age, Address address) {
27         this.name = name;
28         this.age = age;
29         this.address = address;
30     }
31     @Override
32     public String toString() {
33         return "Person{" +
34                 "name='" + name + '\'' +
35                 ", age=" + age +
36                 ", address=" + address +
37                 '}';
38     }
39 }
40
41 public class Main {
42     public static void main(String[] args){
43     Address address1 = new Address("Tolstogo 1-28");
44     Address address2 = new Address("Filimonova 27-2");
45     Person igor = new Person("Igor", 21, address1);
46     Person sveta = new Person("Sveta", 20, address1);
47     Person oleg = new Person("Oleg", 22, address2);
48         //Сериализация в файл с помощью
```

```
49          //класса ObjectOutputStream
50          try{
51              ObjectOutputStream objectOutputStream =
52                  new ObjectOutputStream(
53                      new FileOutputStream("person.out"));
54              objectOutputStream.writeObject(igor);
55              objectOutputStream.writeObject(sveta);
56              objectOutputStream.writeObject(oleg);
57              objectOutputStream.close();
58          }
59          catch (IOException e) {
60              e.printStackTrace();
61          }
62
63          Person igorRestored = null;
64          Person svetaRestored = null;
65          Person OlegRestored = null;
66          try{
67              //Востановление из файла с помощью
68              //класса ObjectInputStream
69              ObjectInputStream objectInputStream =
70                  new ObjectInputStream(
71                      new FileInputStream("person.out"));
72              igorRestored =
73                      (Person) objectInputStream.readObject();
74              svetaRestored =
75                      (Person) objectInputStream.readObject();
76              OlegRestored =
77                      (Person) objectInputStream.readObject();
78              objectInputStream.close();
79          }
80          catch (IOException | ClassNotFoundException e) {
81              e.printStackTrace();
82          }
83          System.out.println("Перед сериализацией: " + "\n" +
84                             igor + "\n" + sveta + "\n" + oleg);
85          System.out.println("После сериализации: " + "\n" +
86                             igorRestored + "\n" + svetaRestored +
87                             "\n" + OlegRestored);
88      }
89  }
```

Результат работы программы:

Main.java | people.dat | person.out

```
1  ¬í··sr··java.util.ArrayListx·Ò··Ça····I··sizexp····w·····sr··Person·
2  ··j²·ö···L··namet··Ljava/lang/String;xpt··Tomsq·~··t··Samx
```

Main.java | people.dat | person.out

```
1  ¬í··sr··Person|þ·¡úI5(···I··ageL··addresst··LAddress;L··namet··Ljava/lang/String;xp····sr··AddressÜj·z3.··
2  Tolstogo 1-28t··Igorsq·~·····q·~··t··Svetasq·~·····sq·~··t··Filimonova 27-2t··Oleg
```

```
input
Перед сериализацией:
Person{name='Igor', age=21, address=Address{home='Tolstogo 1-28}}
Person{name='Sveta', age=20, address=Address{home='Tolstogo 1-28}}
Person{name='Oleg', age=22, address=Address{home='Filimonova 27-2}}
После сериализации:
Person{name='Igor', age=21, address=Address{home='Tolstogo 1-28}}
Person{name='Sveta', age=20, address=Address{home='Tolstogo 1-28}}
Person{name='Oleg', age=22, address=Address{home='Filimonova 27-2}}
```

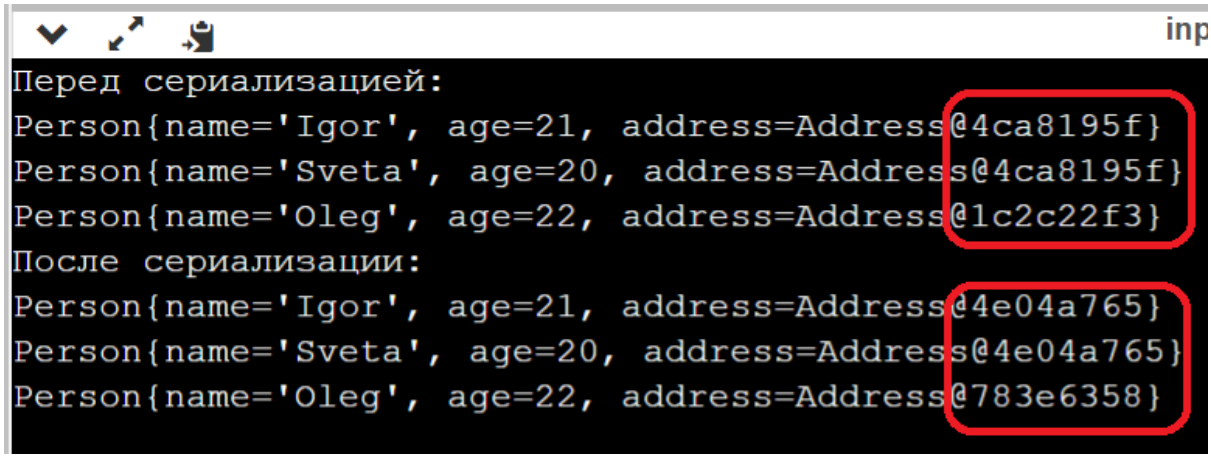
В данном случае в класс `Person` добавлено поле `address`, являющее ссылкой на объект класса `Address`. Оба класса и `Person`, и `Address` реализуют интерфейс `Serializable`. Объекты обоих классов сериализуются, т. е. ссылки на другие объекты (напомним, что за исключением `transient` или `static` полей) также вызывают запись этих объектов.

Немного изменим предыдущий пример, закомментировав метод `toString()` для класса `Address`:

```
class Address implements Serializable {
    private String home;

    public Address(String home) {
        this.home = home;
    }
    public String getHome() {
        return home;
    }
    /* @Override
    public String toString() {
        return "Address{" +
                "home='" + home + '}';
    }
    */
}
```

В этом случае, при выводе будет вызываться метод `toString()` класса `Object`, который выводит адрес выделенного под объект участка памяти. Результат работы программы:



Из результатов выполнения программы видно, что при восстановлении объектов, у которых до сериализации была ссылка на один и тот же объект, после сериализации объекты также содержат ссылку на тот же объект. Это видно по одинаковым ссылкам в объектах до и после восстановления.

Замечание.

Отметим, что, объекты дочерних классов, которые не являются сериализуемыми, но базовые классы которых сериализуемы, могут быть сериализуемыми. В этом случае несериализуемый класс должен иметь конструктор без параметров, чтобы можно было инициализировать его поля.

Совместимость версий объекта при сериализации

Код программ часто меняется, особенно в процессе разработки. Не исключена такая ситуация, когда программист создал класс, затем создал его экземпляр и, сериализовав его, записал, например, в файл. Этот разложенный на байты объект какое-то время находился в файловой системе. Тем временем программист изменил код класса, например, добавил в него новое поле, а затем попытался прочесть объект класса из файла. В этом случае десериализация приведет к возникновению исключительной ситуации `InvalidClassException`.

Дело в том, среда выполнения сериализации связывает с каждым сериализуемым классом уникальный идентификатор версии, называемый `serialVersionUID`, который используется во время десериализации для проверки того, что отправитель и получатель сериализованного объекта загрузили классы для этого объекта, совместимые с сериализацией.

Уникальный идентификатор учитывает всю информацию о классе: число полей, порядок следования полей, их имена, тип полей,

модификаторы доступа включая поля вашего класса, интерфейсы, которые реализует класс, и даже различные реализации компилятора, а также методы их порядок следования. Любые изменения в классе или использование другого компилятора может привести к изменению `SerialVersionUID`, который в конечном итоге не позволит выполнить десериализацию данных.

Несмотря на то, что в механизме сериализации Java есть автоматическая неявная генерация `SerialVersionUID`, рекомендуется явно определять `SerialVersionUID`. Поскольку вычисление `serialVersionUID` «по умолчанию» основывается на сведениях о классе, которые могут различаться в зависимости от реализации компилятора, то использование собственного `serialVersionUID` позволит как минимум избежать зависимости от реализации компилятора.

Для явного определения программистом `serialVersionUID` в `Serializable`-классе рекомендуется использовать конструкцию:

```
static final long serialVersionUID;
```

Также рекомендуется, чтобы в явных объявлениях `serialVersionUID` использовался модификатор `private`.

Это *единственный пример*, когда `static`-поле сериализуется. При десериализации значение этого поля сравнивается с имеющимся у класса в виртуальной машине. Если значения не совпадают, генерируется исключение `java.io.InvalidClassException`.

Можно использовать утилиту, входящую в состав JDK, которая называется serialver, чтобы посмотреть какой код будет присвоен полю `SerialVersionUID` «по умолчанию» (это просто `hash`-код объекта «по умолчанию»). Существуют также специальные программы-генераторы `UID` (`User identifier` - идентификатор пользователя в операционной системе или на сайте). Можно самостоятельно присваивать значение и следить за его изменением при модификации класса.

Пример.

```
class Person implements Serializable {
    private String name;
    private int age;
    //ссылка на сериализуемый объект
    private Address address;
    private static final long serialVersionUID = 2L;
```

```
    public Person(String name, int age, Address address) {
        this.name = name;
        this.age = age;
        this.address = address;
    }
    @Override
    public String toString() {
        return "Person{" +
                "name='" + name + '\'' +
                ", age=" + age +
                ", address=" + address +
                '}';
    }
}
```

Класс `File`

Класс `File`, определенный в пакете `java.io`, не работает напрямую с потоками. Его задачей является управление информацией о файлах и каталогах. Хотя 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 конструкторов для создания объекта:

- `File(String путьККаталогу)`,
- `File(String путККаталогу, String имяФайла)`,
- `File(File каталог, String имяФайла)`.

Пример.

```
// создаем объект File для каталога
File dir1 = new File("C://SomeDir");
//создаем объекты для файлов, которые находятся в каталоге
File file1 = new File("C://SomeDir", "Hello.txt");
File file2 = new File(dir1, "Hello2.txt");
```

Класс `File` имеет ряд методов, которые позволяют управлять файлами и каталогами. Рассмотрим некоторые из них:

- `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.

Работа с каталогами

Если объект `File` представляет каталог, то его метод `isDirectory()` возвращает `true`. И поэтому можно получить его содержимое - вложенные подкаталоги и файлы с помощью методов `list()` и `listFiles()`. Получим все подкаталоги и файлы в определенном каталоге.

Замечание.

Для проверки работоспособности программы необходимо использовать интегрированную среду JDoodle (https://www.jdoodle.com/online-java-compiler-ide/) в режиме `Advanced IDE`. В этом режиме в левом окне, определяющем структуру проекта Java, необходимо создать несколько пустых папок и файлов (обведено красным, Рисунок 2). После этого запустить программу, приведенную ниже рисунка.

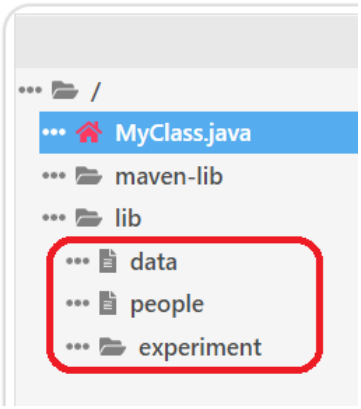


Рисунок 2 - Структура каталогов и файлов проекта

Пример.

/MyClass.java

```
1  import java.io.File;
2  public class MyClass
3  {
4      public static void main(String[] args) {
5          //определяем объект для каталога lib
6          File dir = new File("lib");
7          //если объект представляет каталог
8          if(dir.isDirectory())
9          {
10             //получаем все вложенные объекты в каталоге
11             for(File item : dir.listFiles()) {
12                 if(item.isDirectory()) {
13                     System.out.println(item.getName() +
14                                        "  \t каталог");
15                 }
16             else {
17                System.out.println(item.getName() +
18                "\t файл");
19                 }
20             }
21         }
22     }
23 }
```

Результат работы программы:

```
people    файл
data      файл
experiment        каталог
```

Замечание.

Если разработчик использует инсталлируемое программное обеспечение для выполнения программ на Java, то следует не забывать, что строка, указывающая путь должна иметь определенный синтаксис, например: "C://SomeDir//NewDir".

Теперь выполним еще ряд операций с каталогами, как удаление, переименование и создание.

Пример.

/MyClass.java

```
1  import java.io.File;
2  public class MyClass
3  {
4     public static void main(String[] args) {
5        // определяем объект для корневого каталога
6        File dir = new File("new");
7        //создадим подкаталог
8        if(dir.mkdir())
9           System.out.println("Каталог создан");
10       //переименуем подкаталог
11       File newDir = new File("old");
12       if(dir.renameTo(newDir))
13             System.out.println("Каталог переименован");
14       //удалим каталог
15       boolean deleted = newDir.delete();
16       if(deleted)
17             System.out.println("Каталог удален");
18    }
19 }
```

Результат работы программы:

```
Каталог создан
Каталог переименован
```

```
Каталог удален
```

Замечание.

К сожалению, при использовании онлайн интегрированных сред физически продемонстрировать результаты работы с каталогами не представляется возможным.

Работа с файлами

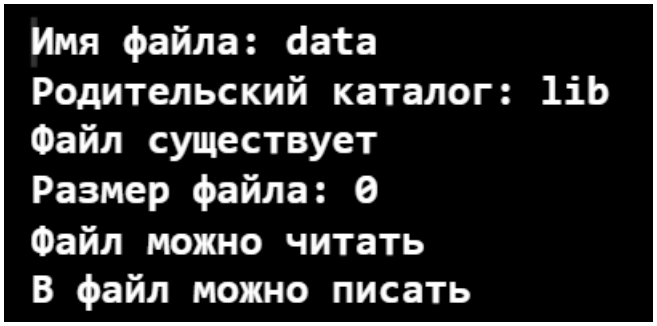
Работа с файлами аналогична работе с каталогами. Например, получим данные по одному из файлов (файл `data`, Рисунок 2).

Пример.

/MyClass.java

```
1  import java.io.File;
2  import java.io.IOException;
3  public class MyClass
4  {
5      public static void main(String[] args) {
6          // определяем объект для каталога
7          File myFile = new File("lib//data");
8          System.out.println("Имя файла: " +
9                             myFile.getName());
10         System.out.println("Родительский каталог: " +
11                            myFile.getParent());
12         if( myFile.exists() )
13             System.out.println("Файл существует");
14         else
15             System.out.println("Файла нет");
16         System.out.println("Размер файла: " +
17                            myFile.length());
18         if( myFile.canRead() )
19             System.out.println("Файл можно читать");
20         else
21             System.out.println("Файл нельзя читать");
22         if( myFile.canWrite() )
23             System.out.println("В файл можно писать");
24         else
25             System.out.println("В файл нельзя писать");
26     }
27 }
```

Результат работы программы:



Класс Files и интерфейс Path, класс Paths

Следует дополнительно остановиться на управлении файлами - их создании, переименовании и т.д. Ранее все подобные операции проводились с помощью класса `File`. Но в настоящее время создатели языка решили изменить работу с файлами и каталогами. Это произошло из-за того, что у класса `File` был ряд недостатков. Например, в нем не было метода `copy()`, который позволил бы скопировать файл из одного места в другое.

Кроме того, в классе `File` было достаточно много методов, которые возвращали `boolean`-значения. При ошибке такой метод возвращает `false`, а не выбрасывает исключение, что делает диагностику ошибок и установление их причин достаточно сложной.

Вместо единого класса `File` появились целых два класса и один интерфейс: `Files`, `Paths`, `Path` (интерфейс).

Класс Paths

`Paths` — это совсем простой класс с единственным статическим методом `get()`. Его создали исключительно для того, чтобы из переданной строки или `URI` получить объект типа `Path`. Другой функциональности у него нет.

Замечание.

`URI` (`Uniform Resource Identifier`) — унифицированный (единообразный) идентификатор ресурса. URI — последовательность символов, идентифицирующая абстрактный или физический ресурс. Представляет из себя символьную строку, позволяющую идентифицировать какой-либо ресурс: документ, изображение, файл, службу, ящик электронной почты и т. д. Однако прежде всего, речь идет о ресурсах сети Интернет.

Пример.

Main.java

```
1  import java.nio.file.Path;
2  import java.nio.file.Paths;
3
4  public class Main
5  {
6      public static void main(String[] args) {
7          Path testFilePath = Paths.get("C:\\Users\\Desktop\\testFile.txt");
8      }
9  }
```

Интерфейс `Path` аналог класса `File`

Замечание.

В общепринятом восприятии, интерфейс обычно должен содержать абстрактные методы и аналогом класса не может быть, т.к. вызвать абстрактный метод невозможно. Однако если методы интерфейса являются исключительно дефолтными и/или статическими, то интерфейс приобретает черты класса.

`Path` – это интерфейс, переработанный аналог класса `File`. Работать с ним значительно проще, чем с `File`:

- во-первых, из него убрали многие статические методы, и перенесли их в класс `Files`;
- во-вторых, в `Path` были упорядочены возвращаемые значения методов. В классе `File` методы возвращали то `String`, то `boolean`. Таким образом в `File` разобраться было непросто.

Например, в классе `File` существует метод `getParent()`, который возвращает путь для текущего файла в виде строки. Но при этом существует еще метод `getParentFile()`, который возвращал то же самое, но в виде объекта класса `File`. Это явно избыточно. Поэтому в интерфейсе `Path` метод `getParent()` и другие методы работы с файлами возвращают просто объект `Path`.

Создание ссылки типа `Path`

Во многих изданиях и лекциях путают `Path` с классом, но это интерфейс и с помощью его можно создать только ссылку, которой будет присваиваться адрес объекта.

К сожалению, возникает терминологическая путаница, т.к. ссылке типа интерфейса необходимо присвоить адрес объекта класса реализующего данный интерфейс, а такого класса нет.

Замечание.

Нельзя создать объект типа `Path` *с помощью кода вида* `new Path()`.

Возникает естественный вопрос, а как объявить и проинициализировать ссылку типа интерфейс в данном случае, т.к. вызвать какой-либо конструктор для инициализации объекта нельзя.

В случае создания ссылки типа `Path`, нужно воспользоваться его статическим методом `of()`. Формат создания ссылки:

```
Path имя = Path.of(путь);
```

где `имя` — это имя переменной типа `Path`; `путь` — это путь к файлу (или директории) вместе с именем файла (или директории) (Таблица 1).

Таблица 1 – Примеры использования метода `of()`

Код	Описание результата
`Path file = Path.of("c:\\projects\\note.txt");`	Создание ссылки `file` типа `Path` и ее инициализация путем к файлу.
`Path directory = Path.of("c:\\projects\\");`	Создание ссылки `directory` типа `Path` и ее инициализация директорией

Но получить ссылку на интерфейс `Path` можно не только используя метод самого интерфейса `Path`. Метод `public Path toPath()` класса `File` также возвращает ссылку на интерфейс `Path`:

```
File file = new File("data/in.txt");
Path path = file.toPath();
```

Есть и еще другие способы, но уже используя методы других классов.

Некоторые методы интерфейса `Path`

Перечислим некоторые методы интерфейса `Path`:

- `getFileName()` — возвращает имя файла из пути (Таблица 2);

Таблица 2 – Особенности использования метода `getFileName()`

Код	Результат
`String str = "c:\\windows\\note.txt";` `Path path = Path.of(str).getFileName();`	Строка `"note.txt"`
`String str = "c:\\windows\\projects\\";` `Path path = Path.of(str).getFileName();`	Строка `"projects"`
`String str = "c:\\";` `Path path = Path.of(str).getFileName();`	`null`

- `getParent()` — возвращает «родительскую» директорию по отношению к текущему пути (то есть ту директорию, которая находится выше по дереву каталогов) (Таблица 3);

Таблица 3 – Особенности использования метода `getParent()`

Код	Результат
`tring str = "c:\\windows\\projects\\note.txt";` `Path path = Path.of(str).getParent();`	Строка `"c:\\windows\\projects\\"`
`String str = "c:\\windows\\projects\\";` `Path path = Path.of(str).getParent();`	Строка `"c:\\windows\\"`
`String str = "c:\\";` `Path path = Path.of(str).getParent();`	`null`

- `getRoot()` — возвращает «корневую» директорию; то есть ту, которая находится на вершине дерева каталогов (Таблица 4);

Таблица 4 – Пример кода использующего метод `getRoot()` и результат его примерния

Код	Результат
`tring str = "c:\\windows\\projects\\";` `Path path = Path.of(str).getRoot();`	Строка `"c:\\"`

- `startsWith()`, `endsWith()` — проверяют, начинается/заканчивается ли путь с переданного пути.

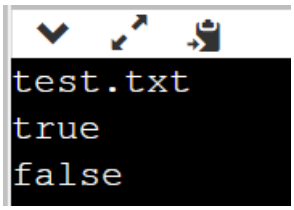
Пример.

Main.java

```
1  import java.nio.file.Path;
2  import java.nio.file.Paths;
3
```

```
4  public class Main
5  {
6     public static void main(String[] args) {
7
8        Path testFilePath = Paths.get("test.txt");
9
10       Path fileName = testFilePath.getFileName();
11       System.out.println(fileName);
12
13       boolean endWithTxt = testFilePath.endsWith("test.txt");
14       System.out.println(endWithTxt);
15
16       boolean startsWithLalala = testFilePath.startsWith("world");
17       System.out.println(startsWithLalala);
18    }
19 }
```

Результаты работы программы:



Замечание.

В метод `endsWith()` нужно передавать именно полноценный путь, а не просто набор символов: в противном случае результатом всегда будет `false`.

При работе с файлами следует также знать следующую терминологию. Путь может быть одним из двух типов:

- абсолютный путь - начинается с корневой директории, например, для Windows это может быть папка `c:\`, а для Linux - это директория `/`;
- относительный путь – путь определяемый относительно текущего рабочего каталога. Если это путь к файлу, расположенному во вложенных папках текущего рабочего каталога, то это как бы конец пути, но только без начала. Относительный путь можно превратить в абсолютный и наоборот.

Для работы с путями существует несколько методов, в частности рассмотрим следующие:

- `boolean isAbsolute()` - метод проверяет, является ли текущий путь абсолютным;

- `Path toAbsolutePath()` – метод превращает путь в абсолютный, если нужно — добавляет к нему текущую рабочую директорию;
- `Path normalize()` – метод позволяет в указанном пути вместо имени директории (папки) писать «`..`», и это будет означать вернуться на одну директорию (папку) назад;
- `Path relativize(Path other)` - метод позволяет вычислить «разницу путей»: один путь относительно другого;
- `Path resolve(Path other)` - метод выполняет операцию, обратную `relativize()` (из абсолютного и относительного пути он строит новый абсолютный путь).

Кроме того существует метод `File toFile()`, возвращающий объект `File`, который хранит тот же путь к файлу, что и объект `Path`.

Класс Files

`Files` — это утилитный класс, куда были вынесены статические методы из класса `File`. Он примерно то же, что и `Arrays` или `Collections`, только работает он с файлами, а не с массивами и коллекциями. Он сосредоточен на управлении файлами и директориями. Используя статические методы `Files`, можно создавать, удалять и перемещать файлы и директории. Для этих операций используются методы `createFile()` (для директорий — `createDirectory()`), `move()` и `delete()`.

В программе, приведенной ниже для примера, сперва создается файл `testFile111.txt` с помощью метода `Files.createFile()` в папке «по умолчанию» on-line интегрированной среды, далее создаем там же папку `testDirectory` с помощью метода `Files.createDirectory()`. После этого перемещаем созданный файл `testFile111.txt` с помощью метода `Files.move()` из папки «по умолчанию» в новую папку `testDirectory`, а в конце — удаляем файл методом `Files.delete()`.

Пример.

Main.java

```
1  import java.io.IOException;
2  import java.nio.file.Files;
3  import java.nio.file.Path;
4  import java.nio.file.Paths;
5  import static java.nio.file.StandardCopyOption.REPLACE_EXISTING;
6
7  public class Main
8  {
```

```
    public static void main(String[] args) throws IOException {
        //создание файла
        Path testFile1 = Files.createFile(Paths.get("testFile111.txt"));
        System.out.println("Был ли файл успешно создан?");
        System.out.println(Files.exists(Paths.get("testFile111.txt")));
        //создание директории
        Path testDirectory = Files.createDirectory(Paths.get("\\testDirectory"));
        System.out.println("Была ли папка (директория) успешно создана?");
        System.out.println(Files.exists(Paths.get("\\testDirectory")));
        //перемещаем файл из папки (директории) "по умолчанию" в
        //папку (директорию) testDirectory
        testFile1 = Files.move(testFile1,
                Paths.get("\\testDirectory\\testFile111.txt"), REPLACE_EXISTING);
        System.out.println("Остался ли наш файл папке (директории)" +
                           "\"по умолчанию\"?");
        System.out.println(Files.exists(Paths.get("testFile111.txt")));
        System.out.println("Был ли наш файл перемещен в testDirectory?");
        System.out.println(Files.exists(
                                Paths.get("\\testDirectory\\testFile111.txt")));
        //удаление файла
        Files.delete(testFile1);
        System.out.println("Файл все еще существует?");
        System.out.println(Files.exists(
                                Paths.get("testDirectory\\testFile111.txt")));
    }
}
```

Результат работы программы:

```
Был ли файл успешно создан?
true
Была ли папка (директория) успешно создана?
true
Остался ли наш файл папке (директории)"по умолчанию"?
false
Был ли наш файл перемещен в testDirectory?
true
Файл все еще существует?
false
```

Замечание.

Большинство методов класса `Files` возвращают в качестве значения и принимают на вход объекты `Path`.

Программное копирование файлов

Рассмотрим такую возможность класса `Files`, как программное копирование файлов с помощью метода `copy()`.

Пример.

Main.java | \testDirectory2\test... | testFile111.txt

```
1  import java.io.IOException;
2  import java.nio.file.Files;
3  import java.nio.file.Path;
4  import java.nio.file.Paths;
5
6  import static java.nio.file.StandardCopyOption.REPLACE_EXISTING;
7
8  public class Main
9  {
10     public static void main(String[] args) throws IOException {
11         //создание файла
12         Path testFile1 = Files.createFile(Paths.get("testFile111.txt"));
13         System.out.println("Был ли файл успешно создан?");
14         System.out.println(Files.exists(Paths.get("testFile111.txt")));
15         //создание директории
16         Path testDirectory2 =
17                 Files.createDirectory(Paths.get("\\testDirectory2"));
18         System.out.println("Была ли директория успешно создана?");
19         System.out.println(Files.exists(Paths.get("\\testDirectory2")));
20         //копируем файл из директории "по умолчанию" в директорию testDirectory2
21         testFile1 = Files.copy(testFile1,
22               Paths.get("\\testDirectory2\\testFile111.txt"), REPLACE_EXISTING);
23         System.out.println("Остался ли наш файл в директорию по \"умолчанию\"?");
24         System.out.println(Files.exists(Paths.get("\\testFile111.txt")));
25         System.out.println("Был ли наш файл скопирован в testDirectory?");
26         System.out.println(
27                 Files.exists(Paths.get("\\testDirectory2\\testFile111.txt")));
28     }
29 }
```

Результат работы программы:

```
Был ли файл успешно создан?
true
Была ли директория успешно создана?
true
Остался ли наш файл в директорию по "умолчанию"?
false
Был ли наш файл скопирован в testDirectory?
true
```

Работа с содержимым файлов

Но класс `Files` позволяет не только управлять самими файлами, но и работать с их содержимым. Для записи данных в файл у него есть метод `write()`, а для чтения методы `read()`, `readAllBytes()` и `readAllLines()`. Подробно остановимся на последнем, т.к. у него тип

возвращаемого значения `List<String>`, т.е. он возвращает список строк файла. Это делает работу с содержимым очень удобной, ведь весь файл, строку за строкой, можно, например, вывести в консоль в обычном цикле `for()`.

Пример.

```
//Пример требует предварительного создания файла
//text.txt
```

Main.java | text.txt

```
1  Hellow world!
2  Илон Маск
3  Билл Гейтс
```

```
//Собственно пример
```

Main.java | text.txt

```
1  import java.io.IOException;
2  import java.nio.file.Files;
3  import java.nio.file.Paths;
4  import java.util.List;
5
6  public class Main
7  {
8      public static void main(String[] args) throws IOException {
9          List<String> lines = Files.readAllLines(Paths.get("text.txt"));
10         for (String s: lines) {
11             System.out.println(s);
12         }
13     }
14 }
```

Результат работы программы:

```
Hellow world!
Илон Маск
Билл Гейтс
```

Пусть необходимо найти в файле все строки, которые начинаются со слова «Как», привести их к `UPPER CASE` и вывести в консоль.

Пример.

```
//Пример требует предварительного создания файла
//text.txt
```

Main.java | text.txt

```
1  Newton
2  Как Ломоносов, а Черчилль
3  сам по себе
```

//Собственно пример

Main.java | text.txt

```
1  import java.io.IOException;
2  import java.nio.file.Files;
3  import java.nio.file.Paths;
4  import java.util.ArrayList;
5  import java.util.List;
6
7  public class Main
8  {
9      public static void main(String[] args) throws IOException {
10         List<String> lines = Files.readAllLines(Paths.get("text.txt"));
11         List<String> result = new ArrayList<>();
12         for (String s: lines) {
13             if (s.startsWith("Как")) {
14                 String upper = s.toUpperCase();
15                 result.add(upper);
16             }
17         }
18         for (String s: result) {
19             System.out.println(s);
20         }
21     }
22 }
```

Результат работы программы:

```
КАК ЛОМОНОСОВ, А ЧЕРЧИЛЛЬ
```

Замечание о работе с ZIP- и JAR-архивами

Кроме общего функционала для работы с файлами Java предоставляет функциональность для работы с таким видом файлов как `zip`-архивы. Для этого в пакете `java.util.zip` определены два класса - `ZipInputStream` и `ZipOutputStream`.

Замечание.

Обойтись в данном случае онлайн-интегрированными средами не получиться.

Для создания архива используется класс `ZipOutputStream`. Для создания объекта `ZipOutputStream` в его конструктор передается поток вывода.

Для чтения архивов применяется класс `ZipInputStream`. В конструкторе он принимает поток ввода, указывающий на `zip`-архив. И затем в цикле необходимо вывести все файлы, которые находятся в данном архиве.

Более подробно с процессов добавления и/или извлечения файлов из архива можно ознакомиться на сайте METANIT.COM (https://metanit.com/java/tutorial/6.12.php).

Кроме этого файлы классов и пакеты можно помещать в специальные `JAR`-архивы, содержимое которых сжимается с помощью алгоритма `zip`. Помимо классов, эти архивы могут содержать и файлы других типов, например, изображения. Такие файлы называются ресурсами. Преимущество `JAR`-архива состоит в том, что он представляет собой всего один файл небольшого размера вместо целой иерархии папок и файлов пакета.

Пакет `java.util.jar` аналогичен пакету `java.util.zip`, за исключением реализации конструкторов и метода `void putNextEntry(ZipEntry e)` класса `JarOutputStream`. Отличие `jar`-файлов от `zip`-файлов только в том, что в `jar`-архивы автоматически включается каталог `META-INF`, содержащий несколько файлов с информацией об иерархии упакованных в архив папок и местоположении файлов.

Литература

1. Руководство по языку программирования Java/ METANIT.COM - [Электронный ресурс], URL: https://metanit.com/java/tutorial/ (дата обращения: 08.06.22)
2. JavaRush [Электронный ресурс], URL: https://javarush.ru/ (дата обращения: 08.06.22)
3. Блинов, И.Н. Java from EPAM / И.Н. Блинов, В.С. Романчик. - Минск: Четыре четверти, 2020. - 560 с.

Учебное издание

Кравчук Александр Степанович
Кравчук Анжелика Ивановна
Кремень Елена Васильевна

Язык Java.
Потоки ввода-вывода.
Работа с файлами

Учебные материалы
для студентов специальности 1-31 03 08
«Математика и информационные технологии
(по направлениям)»

В авторской редакции

Ответственный за выпуск *Е. В. Кремень*

Подписано в печать 16.02.2023. Формат 60×84/16. Бумага офсетная.
Усл. печ. л. 3,72. Уч.- изд. л. 3,64. Тираж 50 экз. Заказ

Белорусский государственный университет.
Свидетельство о государственной регистрации издателя, изготовителя,
распространителя печатных изданий № 1/270 от 03.04.2014.
Пр. Независимости 4, 220030, Минск.

Отпечатано с оригинал-макета заказчика
на копировально-множительной технике
механико-математического факультета
Белорусского государственного университета.
Пр. Независимости 4, 220030, Минск.