Урок 3

Средства ввода-вывода

Обзор средств ввода-вывода. Байтовые, символьные, буферизованные потоки. Сетевое взаимодействие, сериализация и десериализация объектов.

Дополнительные материалы

Используемая литература

Общие сведения

Одним из вариантов работы с вводом и выводом данных в Java является применение пакета java.io. Операции ввода-вывода в этом пакете основаны на использовании потоков (Stream) - абстрактных сущностей по которым можно передавать данные в одном из направлений (либо на вход, либо на выход). В зависимости от того, куда направлены эти потоки, вы можете одинаково работать с файлами, сетевыми соединениями, звуковыми устройствами и т.д. Если у вас есть какой-то набор байт, вы можете направить его в исходящий поток данных, если это будет файловый поток, то байты запишутся в файл, если это будет поток для консоли, то байты будут выведены в консоль, если поток используется в качестве точки назначения сетевое соединения, то байты отправятся в сеть. То есть использование потоков позволяет одинаково работать с совершенно разными источниками данных, что очень удобно.

Заметка. Не путайте потоки из многопоточности (Thread) и потоки ввода-вывода (Stream). В этом материале мы говорим именно о потоках ввода-вывода - Stream. В свою очередь Stream не имеет ничего общего с Stream API.

Класс File

При решении многих задач приходится иметь дело с файлами и файловой системой, поэтому давайте начнем разбор пакета java.io именно с класса File. Класс File, согласно документации Oracle, представляет собой абстрактное представление пути к файлу или директории.

Важно! Если мы создадим экземпляр класса File:
File file = new File(“Harry Potter and Philosopher’s Stone.mp4”);
Это ни в коем случае не означает, что мы загрузим этот фильм в память и будем с ним работать, file будет хранить в себе просто пусть к этому файлу.
Также **важно** знать что мы можем указать путь даже на несуществующий файл или каталог, ведь это всего лишь путь.

С помощью класса File мы можем узнать различные свойства файла (время и дата создания, размер, расширение и т.д.) по указанному пути к этому файлу. В таблице приведены наиболее часто применяемые методы класса File.

Название метода	Назначение
boolean exists()	Проверка существования файла по указанному пути
String getName()	Возвращает имя файла или директории
File getParent(), String getParentName()	Возвращает директорию, в которой хранится данный файл, в зависимости от метода будет возвращен объект типа File или String
boolean isDirectory(), boolean isFile()	Проверяет, что объект типа File указывает на директорию, или на файл. (Эта проверка имеет место быть, так как в объект типа File, мы можем записать путь указывающий как на файл, так и на каталог)

long length()	Возвращает размер файла в байтах
File[] listFiles(), String[] list()	Возвращает список файлов в текущей директории, в зависимости от метода будет возвращен или массив File[], или String[]
boolean() delete	Удаляет файл по указанному пути. Если File указывает путь на каталог, то удаление выполнится только если этот каталог пуст.
boolean mkdir(), boolean mkdirs()	mkdir() создает только один каталог по указанному пути, если на пути к нему не будет хватать промежуточных звеньев (каталогов), то операция не выполнится. mkdirs() создает каталог по указанному пути, если на пути к нему не будет хватать промежуточных звеньев, все эти звенья будут также созданы

Важно! В операционной системе и файл, и директория является файлом (попробуйте в файловом менеджере создать файл без расширения, допустим с именем “1”, и рядом создать каталог с таким же именем).

Байтовые и символьные потоки

При работе с пакетом java.io мы будем иметь дело с потоками двух типов: байтовые и символьные. Поскольку все с чем нам придется работать состоит просто из набора байтов, то зачастую придется иметь дело именно с байтовыми потоками данных. Символьные же потоки удобны при работе с текстом.

Давайте посмотрим на классы в пакете java.io: InputStream, OutputStream, FileInputStream, FileOutputStream, InputStreamReader, StandardCharsets, BufferedInputStream, BufferedOutputStream, DataInputStream, DataOutputStream, ByteArrayInputStream, ByteArrayOutputStream, CharArrayReader, CharArrayWriter, PrintStream, PrintWriter, PipedInputStream, PipedOutputStream, PipedReader, PipedWriter, Reader, Writer, FileReader, FileWriter, BufferedReader, BufferedWriter, SequenceInputStream, PushBackReader, RandomAccessFile, ObjectInputStream, ObjectOutputStream (Serializable, Externalizable).

На первый взгляд кажется что в этом всем невозможно разобраться и запомнить что для чего нужно. На самом же деле все классы делятся на группы: входной/выходной поток, байтовы/символьный поток, и плюс к этому каждый класс добавляет немного особенностей. Input/Reader - означает чтение, Output/Write - запись, Stream - работу с байтами, (Reader, Writer) - с символами, Buffered - добавление буферизации, (Byte, Char) - указывает тип, с которым работаем, Object - работаем с объектами (сериализация, десериализация). Теперь даже простое чтение имени класса должно навести на мысль о том, чем этот класс занимается.

Давайте приведем пример рассуждений: если мы видим слово Input, значит будем что-то считывать, если видим слово Stream, значит речь идет о байтах, если добавили слово File, значит будет работать с файлами, если увидели слово Buffered, значит применяется буферизация при чтении, итого:

```
BufferedInputStream in = new BufferedInputStream(new FileInputStream("demo.txt"));
```

Приведенный выше объект in предназначен для чтения байтов из файла “demo.txt”, и чтение будет буферизированным. Детально все особенности разберем далее по тексту.

На самом деле, что бы мы ни использовали, в любом случае будет читать/писать байты, однако же символьные потоки данных предлагаются удобные средства для автоматического преобразования байтов в символы и текст, только всей этой работы не видим.

Работа с байтовыми потоками ввода-вывода

InputStream и OutputStream

InputStream представляет собой базовый абстрактный класс для потоков ввода, описывающий методы для чтения байтовых данных.

Важно! Метод int read() считывает один байт из потока и возвращает int в диапазоне от 0 до 255, который представляет собой полученный байт. То есть мы получим не совсем привычный для Java знаковый байт (в диапазоне -128 до 127), а именно беззнаковый байт в пределах 0 до 255.
Важно! Если мы прочитали весь поток, и данных в нем больше нет, то следующий вызов метода read() вернет значение -1, по которому мы можем определять завершение чтения потока.

Для считывания массива байт используется метод int read(byte[] arr), при выполнении которого в цикле вызывается абстрактный метод read(). Количество байт, считываемое таким образом, равно длине переданного массива. Если данных в потоке осталось меньше, чем длина массива, массив будет записан не до конца.

Важно! Метод int read(byte[] arr) возвращает количество прочитанных байт в виде int значения.

Для заполнения части массива используется метод read(byte[] arr, int off, int len), где off – это позиция в массиве, с которой начнется заполнение, а len – количество байт, которое нужно считать.

Класс **OutputStream** представляет собой базовый абстрактный класс для потоков вывода, описывающий методы для записи байтовых данных. В нем определены три метода: write(int value), write(byte[] arr) и write(byte[] arr, int off, int len). Метод write(int) принимает в качестве параметра int, но записывает в поток только байт (число в пределах 0-255).

По завершению работы с потоками ввода-вывода, их необходимо обязательно закрыть с помощью метода close(), для освобождения системных ресурсов.

ByteArrayInputStream и ByteArrayOutputStream

Для начала давайте немного поработаем просто с байтовыми массивами в качестве источника данных. И тут же потренируемся с записью. Класс **ByteArrayInputStream** представляет поток, считывающий данные из массива байт, напишем программу, в которой дан массив байт new byte[] { 65, 66, 67}, с помощью ByteArrayInputStream прочитаем из него значения, и выведем результат в консоль.

```
public class StreamDemoApp {
    public static void main(String[] args) {
        byte[] arr = {65, 66, 67};
```

```
        ByteArrayInputStream in = new ByteArrayInputStream(arr);
        int x;
        while ((x = in.read()) != -1) {
            System.out.print(x + " ");
        }
    }
}
// Результат:
// 65 66 67
```

Все вроде бы предельно просто, в массиве были числа 65, 66, 67, вот и в консоли мы увидели 65, 66, 67. А давайте сделаем немного по-другому.

```
public class StreamDemoApp {
    public static void main(String[] args) {
        byte[] arr = {0, 127, -1};
        ByteArrayInputStream in = new ByteArrayInputStream(arr);
        int x;
        while ((x = in.read()) != -1) {
            System.out.print(x + " ");
        }
    }
}
// Результат:
// 0 127 255
```

Возникает вопрос, откуда взялось число 255, ведь в массиве то мы видели -1. Все дело в том, что мы считываем байтовое значение в int, и наш байт “преобразуется в беззнаковый”. Если мы посмотрим как выглядит Java byte -1 в двоичном виде, то получим [11111111], после его чтения, мы получили int, который в двоичном виде тоже выглядит как [11111111], то есть никакой ошибки тут нет.

ByteArrayOutputStream можем использовать для записи в поток, после чего получить результат нашей записи в виде байтового массива с помощью метода toByteArray().

```
public class StreamDemoApp {
    public static void main(String[] args) {
        ByteArrayOutputStream out = new ByteArrayOutputStream();
        out.write(10);
        out.write(11);
        byte[] arr = out.toByteArray();
    }
}
```

Эти классы могут быть полезны для проверки данных, которые мы считываем или записываем откуда- или куда-либо.

FileInputStream и FileOutputStream

Классы **FileInputStream** и **FileOutputStream** применяются для чтения данных из файла, и записи в него. На работе с этими классами желательно остановиться поподробнее, так как все остальные

будут либо надстройками с дополнительными возможностями, либо использовать другие источники данных помимо файлов.

Для начала разберемся что нужно чтобы записать что-нибудь в файл. Нам понадобиться FileOutputStream, а следовательно было бы неплохо посмотреть на его конструкторы.

Конструктор	Описание
FileOutputStream(File file)	Создает файловый поток вывода из объекта типа File, если файл по указанному пути существовал, то при открытии потока он очищен
FileOutputStream(File file, boolean append)	Создает файловый поток вывода из объекта типа File, если файл по указанному пути существовал, то при открытии запись данных продолжится с конца файла
FileOutputStream(String name)	Создает файловый поток вывода из файла по пути, указанному в String name, если файл по указанному пути существовал, то при открытии потока он очищен
FileOutputStream(String name, boolean append)	Создает файловый поток вывода из файла по пути, указанному в String name, если файл по указанному пути существовал, то при открытии запись данных продолжится с конца файла

Какой бы конструктор мы не взяли, нам придется указать путь к файлу, либо в виде объекта типа File, либо в виде строки. Кроме того, мы можем указать надо ли начинать запись с начала, или продолжить запись с конца файла.

Давайте посмотрим на код, который позволит нам записать в файл слово Java. Вначале нам нужно получить байтовое представление слова Java, создадим байтовый массив outData[] и заполним его с помощью метода класса String getBytes(). После того как данные готовы, открываем поток записи FileOutputStream в файл demo.txt, используя try-с-ресурсами, и с помощью метода write() записываем содержимое массива в файл. try-с-ресурсами автоматически закроет наш поток записи и программа завершится.

```
public class StreamDemoApp {
    public static void main(String[] args) {
        byte[] outData = "Java".getBytes();
        try (FileOutputStream out = new FileOutputStream("demo.txt")) {
            out.write(outData);
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Рассмотрим пример побайтовой записи данных в файл.

```
public class StreamDemoApp {
    public static void main(String[] args) {
        byte[] outData = new byte[1024 * 1024];
        for (int i = 0; i < outData.length; i++) {
```

```
            outData[i] = 65;
        }
        try (FileOutputStream out = new FileOutputStream("demo.txt")) {
            for (int i = 0; i < outData.length; i++) {
                out.write(outData[i]);
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Второй пример не сильно отличается от первого, но тут есть некоторая особенность. Мы также записываем данные в файл из массива, но делаем это не за один вызов метода write(), а последовательно выполняем write() для каждого байта массива.

Важно! При выполнении каждой операции write(), Java выполняет вызов нативного метода для записи данных, что является “довольно тяжелой” операцией. Выполнить запись байтового массива размером в 1 миллион элементов (1024 * 1024) за одну операцию write() будет выполнено во много-много раз быстрее, чем делать 1 миллион отдельных write(). (При условии что мы не учитываем буферизацию, но об этом позже)

То есть во втором примере запись будет выполняться очень-очень медленно.

Важно! Для ускорения записи либо используйте буферизацию, либо выполняйте запись кусками (массивами) по десятку-сотне-тысяче байт.

Теперь перейдем к вопросу чтения данных из файлов. И для начала посмотрим какие варианты создания FileInputStream у нас есть.

Конструктор	Описание
FileInputStream(File file)	Создает файловый поток ввода из объекта типа File
FileInputStream(String filename)	Создает файловый поток ввода из объекта типа String

Какой бы вариант не выбрали, при создании FileInputStream можно лишь указать путь к файлу.

В первом примере, для чтения мы будем использовать байтовый массив buf длиной 20. Открываем файл demo.txt и читаем данные из файла в байтовый буфер. Перегрузка метода int read(byte[] buf) возвращает количество прочитанных байт, если метод вернул -1, значит файл закончился и можно закрывать поток чтения.

```
public class StreamDemoApp {
    public static void main(String[] args) {
        byte[] buf = new byte[20];
```

```
        try (FileInputStream in = new FileInputStream("demo.txt")) {
            int count;
            while ((count = in.read(buf)) > 0) {
                for (int i = 0; i < count; i++) {
                    System.out.print((char) buf[i]);
                }
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Возможен вариант с побайтовым чтением файла, но без буферизации чтение будет очень медленным за счет большого количества вызовом нативных методов, поэтому код приведен лишь для примера.

```
public class StreamDemoApp {
    public static void main(String[] args) {
        try (FileInputStream in = new FileInputStream("demo.txt")) {
            int x;
            while ((x = in.read()) > -1) {
                System.out.print((char)x);
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

А вот сравните предыдущий пример, и следующий. Приведенный ниже пример содержит в себе **ошибку**, которую можно случайно совершить.

```
public class StreamDemoApp {
    public static void main(String[] args) {
        try (FileInputStream in = new FileInputStream("demo.txt")) {
            while (in.read() > -1) {
                System.out.print((char)in.read());
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Казалось бы, что логика чтения никак не изменилась, мы все также читаем файл и выводим его содержимое в консоль. Однако на самом деле, вы будете выводить каждый второй символ файла (то есть потеряете половину байт). Это связано с тем, что каждый вызов метода read() приводит к чтению одного байта из потока, хоть вы его вызвали в System.out.println(), хоть в while().

С простыми операциями чтения/записи мы ознакомились, и должны помнить что побайтовые операции чтения/записи (без использования массивов) могут быть очень медленными из-за большого количества вызовов нативных методов. Для решения этой проблемы добавим буферизацию.

BufferedInputStream и BufferedOutputStream

BufferedInputStream содержит массив байт, который служит буфером для считываемых данных. При вызове метода read() происходит обращение к операционной системе, и во внутренний буфер читается блок данных (по умолчанию – 8192 байта, при создании BufferedInputStream в конструктор можно в качестве аргумента передавать размер буфера). При следующих вызовах read() данные читаются уже из буфера без обращения к операционной системе. Как только данные в буфере заканчиваются – из потока читается следующий блок.

При использовании объекта класса BufferedOutputStream запись производится без обращения к устройству вывода при записи каждого байта. Сначала данные записываются во внутренний буфер. Непосредственное обращение к устройству вывода и запись происходит только тогда, когда буфер будет полностью заполнен. Принудительное освобождение буфера с последующей записью можно вызвать методом flush() или закрытием потока записи методом close().

Не имеет смысла делать два отдельных примера на чтение/запись, так как они будут практически идентичны вариантам FileInputStream/FileOutputStream, поэтому в одном примере показаны обе операции.

```
public class StreamDemoApp {
    public static void main(String[] args) {
        try (OutputStream out = new BufferedOutputStream(new
FileOutputStream("demo.txt"))) {
            for (int i = 0; i < 1000000; i++) {
                out.write(i);
            }
        } catch (IOException e) {
            e.printStackTrace();
        }

        try (InputStream in = new BufferedInputStream(new
FileInputStream("demo.txt"))) {
            int x;
            while ((x = in.read()) != -1) {
                System.out.print((char)x);
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

При таком подходе, скорость побайтового чтения/записи будет не сильно отличаться от тех же операций с применением массивов.

DataInputStream и DataOutputStream

Не всегда получается работать только с набором байтовых данных – как правило, приходится записывать и другие примитивные типы данных. Для удобной работы с ними определены классы DataInputStream и DataOutputStream. При записи происходит конвертация любых примитивных типов в байты, а при чтении – наоборот. Пример

```
public class StreamDemoApp {
    public static void main(String[] args) {
        try (DataOutputStream out = new DataOutputStream(new
FileOutputStream("demo.txt"))) {
            out.writeInt(128);
            out.writeLong(128L);
        } catch (IOException e) {
            e.printStackTrace();
        }

        try (DataInputStream in = new DataInputStream(new
FileInputStream("demo.txt"))) {
            System.out.println(in.readInt());
            System.out.println(in.readLong());
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

Без применения DataOutputStream для записи в файл long значения пришлось бы разбить его на 8 байт, и только после этого записать их в файл. А при чтении, прочитать 8 байт и собрать из них long. Написать код для выполнения этих действий не составит труда, однако поскольку это довольно частая задача, то она уже решена в Java.

Сериализация

Для чтения и записи объектов предназначены классы ObjectInputStream и ObjectOutputStream. Перед записью необходимо провести сериализацию – преобразовать объект в набор байт. При чтении выполняется десериализация – восстановление объекта из набора байт. Чтобы экземпляр класса мог быть сериализован, класс должен реализовать маркерный интерфейс Serializable.

Рассмотрим пример записи/чтения объекта типа Cat в байтовый массив:

```
public class SerializeDemoApp {
    private static class Cat implements Serializable {
        private String name;

        public Cat(String name) {
            this.name = name;
        }
```

```
        @Override
        public String toString() {
            return "Кот " + name;
        }
    }

    public static void main(String[] args) {
        byte[] byteCat = null;
        try (ByteArrayOutputStream barrOut = new ByteArrayOutputStream();
             ObjectOutputStream objOut = new ObjectOutputStream(barrOut)) {
            Cat catOut = new Cat("Барсик");
            objOut.writeObject(catOut);
            byteCat = barrOut.toByteArray();
            System.out.println("Кот до сериализации: " + catOut);
            System.out.println("Вот так он выглядит в байтовом представлении: "
+ Arrays.toString(byteCat));
        } catch (IOException e) {
            e.printStackTrace();
        }

        try (ByteArrayInputStream barrIn = new ByteArrayInputStream(byteCat);
             ObjectInputStream objIn = new ObjectInputStream(barrIn)) {
            Cat catIn = (Cat) objIn.readObject();
            System.out.println("А вот такого кота мы восстановили из набора
байтов: " + catIn);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
// Результат
// Кот до сериализации: Кот Барсик
// Вот так он выглядит в байтовом представлении: [-84, -19, 0, 5, 115, 114, 0,
32, 99, 111, 109, 46, 103, 101, 101, 107, 98, 114, 97, 105, 110, 115, 46, 83,
116, 114, 101, 97, 109, 68, 101, 109, 111, 65, 112, 112, 36, 67, 97, 116, 36,
-41, -54, 26, -127, 71, -65, 68, 2, 0, 1, 76, 0, 4, 110, 97, 109, 101, 116, 0,
18, 76, 106, 97, 118, 97, 47, 108, 97, 110, 103, 47, 83, 116, 114, 105, 110,
103, 59, 120, 112, 116, 0, 12, -48, -111, -48, -80, -47, -128, -47, -127, -48,
-72, -48, -70]
// А вот такого кота мы восстановили из набора байтов: Кот Барсик
```

Из примера выше видно, что десериализованный кот равен сериализованному ранее коту. Несмотря на то, что код для сериализации довольно прост, сама логика работы с сериализаций имеет довольно много особенностей.

При сериализации объект может хранить ссылки на другие объекты, в свою очередь тоже хранящие ссылки. Все они должны быть восстановлены при десериализации. Если несколько ссылок указывают на один объект, то при восстановлении они должны сохранить эти указания. Далее большой пример кода.

```
public class SerializeDemoApp {
    private static class Book implements Serializable {
        private String title;

        public Book(String title) {
            this.title = title;
        }
    }

    private static class Student implements Serializable {
        private int id;
        private String name;
        private int score;
        private Book book;

        public Student(int id, String name, int score) {
            System.out.println("Конструктор класса Student");
            this.id = id;
            this.name = name;
            this.score = score;
        }

        public void info() {
            System.out.println(id + " " + name + " " + score);
        }
    }

    public static void main(String[] args) {
        Student studentOneOut = new Student(1, "Боб", 80);
        Student studentTwoOut = new Student(2, "Билл", 70);
        Book jungleBook = new Book("Jungle Book");
        studentOneOut.book = jungleBook;
        studentTwoOut.book = jungleBook;
        try (ObjectOutputStream objOut = new ObjectOutputStream(new
FileOutputStream("students.ser"))) {
            objOut.writeObject(studentOneOut);
            objOut.writeObject(studentTwoOut);
        } catch (IOException e) {
            e.printStackTrace();
        }

        Student studentOneIn = null;
        Student studentTwoIn = null;
        try (ObjectInputStream objIn = new ObjectInputStream(new
FileInputStream("students.ser"))) {
            studentOneIn = (Student) objIn.readObject();
            studentTwoIn = (Student) objIn.readObject();
            studentOneIn.info();
            studentTwoIn.info();
        } catch (Exception e) {
            e.printStackTrace();
        }
```

```
        System.out.println(studentOneIn.book);
        System.out.println(studentTwoIn.book);
    }
}
```

При сериализации, объект помечается как записанный в файл, и даже если вы внесете в него изменения и попробуете повторно сериализвать, то ничего не выйдет, так как Java строит граф объектов для сериализации и помечает уже записанные объекты. Такой подход крайне важен при работе со связными объектами, чтобы не получалось “кольцевой” записи, когда два объекта ссылаются друг на друга и пытаются по кругу сериализовать друг друга.

Если класс содержит в качестве полей ссылочные типы данных, то они также подлежат сериализации, и поэтому тоже должны реализовать интерфейс Serializable.

При десериализации конструкторы не вызываются: объект просто восстанавливается в том виде, в каком он был. Обратим внимание на то, что происходит с состоянием объекта, унаследованным от суперкласса. Ведь оно определяется не только значениями полей, определенными в нем самом, но также и унаследованными от суперкласса. Сериализуемый подтип берет на себя такую ответственность, если у суперкласса определен конструктор по умолчанию, который будет вызван при десериализации. В противоположном случае будет получено исключение InvalidClassException.

В процессе десериализации поля родительских классов, не реализующих интерфейс Serializable, инициируются вызовом конструктора без параметров. Поля сериализуемого класса будут восстановлены из потока.

Для того, чтобы исключить поле из процесса сериализации, необходимо добавить в его объявление модификатор transient, например, чтобы не сохранять объект, десериализация которого не имеет смысла (загрузка из файла сетевого соединения). При восстановлении такого поля объекта, оно получит значение по умолчанию.

```
public class Account implements Serializable {
    private String name;
    private String login;
    private transient String password;
}
```

При необходимости управлять ходом сериализации используется интерфейс Externalizable. В этом случае в поток автоматически записывается только идентификация класса. Сохранить и восстановить всю информацию должен сам класс – через методы writeExternal() и readExternal().

При сериализации объект первым делом проверяется на поддержку интерфейса Externalizable. Если проверка пройдена, вызывается метод writeExternal(). Если объект не поддерживает Externalizable, но реализует Serializable, используется стандартная сериализация. При восстановлении Externalizable объекта экземпляр создается через вызов public конструктора без аргументов. Затем вызывается метод readExternal(). Объекты Serializable восстанавливаются посредством считывания из потока ObjectInputStream.

Версии классов

За время хранения сериализованного объекта в класс могут быть внесены изменения, которые сделают процесс десериализации невозможным. Например, если сериализовать объект класса Person:

```
public class Person implements Serializable {
    private String name;
}
```

После чего заменить поле name на два поля:

```
public class Person implements Serializable {
    protected String firstName;
    protected String lastName;
}
```

При попытке десериализации будет брошено исключение InvalidClassException. Этого не произошло бы при таком изменении:

```
public class Person implements Serializable {
    private String name;
    String lastName;
}
```

Для отслеживания таких ситуаций каждому классу присваивается его идентификатор (ID) версии. Это число long, полученное при помощи хэш-функции. Для вычисления используются имена классов, всех реализуемых интерфейсов, методов и полей класса. При десериализации объекта идентификаторы класса и идентификатор, взятый из потока, сравниваются.

Изменения, проводимые с классом, можно разбить на две группы:

- ***совместимые***, которые можно производить в классе и поддерживать совместимость с ранними версиями; Это добавление поля к классу, добавление или удаление суперкласса, изменение модификаторов доступа полей, удаление у полей модификаторов static или transient, изменение кода методов, инициализаторов, конструкторов;
- ***несовместимые*** – изменения, нарушающие совместимость. Это удаление поля, изменение название пакета класса, изменение типа поля, добавление к полю экземпляра ключевого слова static или transient, реализация Serializable вместо Externalizable или наоборот.

Важно сохранять возможность восстановить именно те поля, которые были записаны в поток при сериализации.

PipedInputStream и PipedOutputStream

Объекты классов **PipedInputStream** и **PipedOutputStream** всегда используются в паре. Данные, записанные в объект PipedOutputStream, могут быть считаны в соединенном объекте PipedInputStream. Соединение обеспечивается за счет указания парного объекта в качестве аргумента конструктора или метода connect(). Данная пара классов используется очень редко, поэтому ограничимся примером кода.

```
public class StreamDemoApp {
    public static void main(String[] args) {
        try (PipedInputStream in = new PipedInputStream();
             PipedOutputStream out = new PipedOutputStream(in)) {
            for (int i = 0; i < 10; i++) {
                out.write(i);
            }

            int x;
            while ((x = in.read()) != -1) {
                System.out.print(x + " ");
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

SequenceInputStream

Класс SequenceInputStream последовательно считывает данные из нескольких входных потоков, как будто мы работаем с одним потоком. Конец потока SequenceInputStream будет достигнут только тогда, когда подойдет к окончанию последний в списке поток.

При создании объекта этого класса в конструктор в качестве параметров передаются объекты InputStream. Когда вызывается метод read(), SequenceInputStream пытается считать байт из текущего входного потока. Если в нем больше нет данных, у этого входного потока вызывается метод close(), и следующий входной поток становится текущим. Так до тех пор, пока это не произойдет с последним входным потоком, и из него не будут считаны все данные. Вызов метода close() у SequenceInputStream закрывает этот поток, предварительно завершив все содержащиеся в нем входные потоки.

```
public class StreamDemoApp {
   public static void main(String[] args) {
       try (SequenceInputStream seq = new SequenceInputStream(new
FileInputStream("1.txt"), new FileInputStream("2.txt"));
            FileOutputStream out = new FileOutputStream("out.txt")) {
            int x;
            while ((x = seq.read()) != -1) {
                out.write(x);
            }
       } catch (IOException e) {
            e.printStackTrace();
       }
   }
}
```

В результате выполнения этого кода в файл out.txt будет записано содержимое файлов 1.txt и 2.txt. Закрытие seq после выхода из try, а потоки FileInputStream будут автоматически закрыты самим объектом seq.

Работа с символьными потоками ввода-вывода

Классы Reader и Writer

Наследники InputStream и OutputStream работают с байтовыми данными. Для использования символов в операциях ввода-вывода предназначены наследники классов Reader и Writer. Пример:

```
public class WriterAndReaderDemoApp {
    public static void main(String[] args) {
        try (BufferedWriter writer = new BufferedWriter(new
FileWriter("demo.txt"))) {
            for (int i = 0; i < 20; i++) {
                writer.write("Java\n");
            }
        } catch (IOException e) {
            e.printStackTrace();
        }

        try (BufferedReader reader = new BufferedReader(new
FileReader("demo.txt"))) {
            String str;
            while ((str = reader.readLine()) != null) {
                System.out.println(str);
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Классы InputStreamReader и OutputStreamWriter могут производить преобразование символов, используя различные кодировки, которые задаются при конструировании потока.

RandomAccessFile

Мы рассмотрели работу с последовательными файлами, содержимое которых вводилось и выводилось побайтово, строго по порядку. Но в Java можно обращаться к хранящимся в файле данным и в произвольном порядке.

Для этого существует класс RandomAccessFile, инкапсулирующий файл с произвольным доступом. Этот класс не является производным от InputStream или OutputStream. Вместо этого он реализует интерфейсы DataInput и DataOutput, в которых объявлены основные методы ввода-вывода. Он также поддерживает запросы с позиционированием, то есть позволяет произвольным образом, вызывая метод seek(), задавать положение указателя файла.

При создании объекта этого класса конструктору передаются два параметра: файл (путь в виде строки или объект класса File), и режим работы(«r» – только чтение,«rw» – чтение и запись).

```
public class RandomAccessFileDemoApp {
    public static void main(String[] args) {
        try (RandomAccessFile raf = new RandomAccessFile("demo.txt", "r")) {
            raf.seek(2);
            System.out.println((char) raf.read());
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}

// Содержимое файла demo.txt: "123456789"
// Результат: 3
```

Практическое задание

1. Добавить в сетевой чат запись локальной истории в текстовый файл на клиенте. Для каждой учетной записи файл с историей должен называться history_[login].txt. (Например, history_login1.txt, history_user111.txt)
2. ** После загрузки клиента показывать ему последние 100 строк истории чата.

Дополнительные материалы

1. Кей С. Хорстманн, Гари Корнелл. Java. Библиотека профессионала. Том 1. Основы;
2. Стив Макконнелл. Совершенный код;
3. Брюс Эккель. Философия Java;
4. Герберт Шилдт. Java 8: Полное руководство;
5. Герберт Шилдт. Java 8: Руководство для начинающих.

Используемая литература

Для подготовки данного методического пособия были использованы следующие ресурсы:

1. Герберт Шилдт. Java. Полное руководство // 8-е изд.: Пер. с англ. – М.: Вильямс, 2012. – 1376 с.
2. Герберт Шилдт. Java 8: Руководство для начинающих. // 6-е изд.: Пер. с англ. – М.: Вильямс, 2015. – 720 с.