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

Операции ввода-вывода в Java выполняются на основе потоков – абстрактных сущностей, которые выдают и получают информацию. За связь потоков с физическими устройствами отвечает система ввода-вывода. Это дает возможность использовать разные устройства, используя одни и те же классы и методы. Например, методы вывода на консоль можно использовать и для записи данных в файл. Для реализации потоков используется иерархия классов, содержащихся в пакете **java.io.**

Класс File

Файлы служат первичными источниками и адресатами данных во многих программах. Большинство классов, определенных в пакете java.io, оперируют потоками ввода-вывода. Но класс **File** взаимодействует с файловой системой и работает непосредственно с файлами. В нем дается описание свойств файлов, но не определяется, как именно данные извлекаются или сохраняются.

Объект класса **File** – это абстрактное представление файла и пути к нему. С его помощью мы узнаем о правах доступа к файлу на диске, о времени, дате, пути к каталогу, а также манипулируем этими сведениями. Посредством класса **File** предоставляет такие виды информации о файле или каталоге, как:

- **canRead()** и **canWrite()** – возможно ли чтение и изменение содержимого файла;
- **exists()** – существует ли файл/каталог на диске;
- **getName()** – возвращает имя файла или директории;
- **getParent(), getParentName()** – возвращают директорию, где файл находится в виде строки названия и объекта File;
- **getPath()** – возвращает путь к файлу;
- **isDirectory(), isFile()** – указывает ли объект на директорию или на файл;
- **isHidden()** – скрытый файл или нет;
- **lastModified()** – время последнего изменения файла;
- **list()** – если объект указывает на каталог, то метод получает массив **String[]**, в котором хранятся имена файлов в этом каталоге;
- **listFiles()** – действует аналогично **list()**, только возвращает массив **File[]**;

Каталог в Java – это тоже объект типа File, который содержит список других файлов и каталогов. Если при создании объекта типа File указать каталог, и вызвать метод **isDirectory()**, этот метод вернет логическое значение true. Тогда для этого объекта можно вызвать метод **list()**, чтобы извлечь список других файлов и подкаталогов, находящихся в нем.

Байтовые и символьные потоки

В актуальных релизах Java определены два типа потоков: байтовые и символьные.

Байтовые потоки подходят для управления вводом и выводом байтов и особенно удобны при работе с файлами. Их можно использовать для чтения и записи двоичных данных.

Символьные потоки предназначены для обмена символьными данными. Благодаря кодировке Unicode их легко интернационализировать. В ряде случаев символьные потоки эффективнее байтовых.

Чтобы поддерживать два типа потоков ввода-вывода, были созданы две иерархии классов – для байтовых и символьных данных. Из-за множества классов система ввода-вывода при первом знакомстве кажется сложной. Но в основном функциональные возможности символьных потоков аналогичны байтовым.

На нижнем уровне все средства ввода-вывода имеют байтовую организацию, а символьные потоки предлагают инструменты, адаптированные для обработки символов.

Работа с байтовыми потоками ввода-вывода

InputStream и OutputStream

InputStream – это базовый абстрактный класс, который описывает базовые методы для чтения байтовых потоков данных. Простейшая операция представлена методом **read()**, который считывает один байт из потока. При этом он возвращает значение типа **int** в диапазоне от 0 до 255 и представляет собой полученный байт. Он не обладает знаком и не принадлежит диапазону от -128 до +127, как примитивный тип **byte** в Java. В конце потока возвращаемое значение равно -1.

Для считывания массива байт используется метод **read(byte[] b)**, при выполнении которого в цикле вызывается абстрактный метод **read()**. Количество байт, считываемое таким образом, равно длине переданного массива. Данные в потоке могут закончиться до того, как будет заполнен весь массив. Поэтому метод возвращает количество прочитанных байт.

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

Класс **OutputStream** – это базовый класс для потоков вывода, в котором так же определяются три метода: **write(), write(byte[] b) и write(byte[] b, int off, int len)**. Метод **write(int)** принимает в качестве параметра **int**, но записывает в поток только **byte**.

Когда работа с потоками ввода-вывода окончена, их необходимо закрыть с помощью метода **close()**, чтобы освободить системные ресурсы.

ByteArrayInputStream и ByteArrayOutputStream

Класс **ByteArrayInputStream** представляет поток, считывающий данные из массива байт.

```
byte[] arr = {100, 25, 50};
ByteArrayInputStream in = new ByteArrayInputStream(arr);
int x;
while((x = in.read()) != -1) {
    System.out.print(x + " ");
}
```

Результат работы:

```
100 25 50
```

Для записи байт в массив применяется класс **ByteArrayOutputStream**. Он содержит буфер, куда записывает данные при вызове методов **write()**. По завершении записи в поток можно получить содержимое этого буфера с помощью метода **toByteArray()**.

```
ByteArrayOutputStream out = new ByteArrayOutputStream();
out.write(10);
out.write(11);
byte[] arr = out.toByteArray();
```

Эти классы могут быть полезны для проверки данных, записываемых в выходной поток.

FileInputStream и FileOutputStream

Класс **FileInputStream** используют для чтения данных из файла. Его конструктор принимает в качестве параметра название читаемого файла. Для записи в файл применяется класс **FileOutputStream**. При создании объектов этого класса можно не только указать путь файла, но и обозначить, будут ли данные дописываться в конец файла, или он будет перезаписан. В случае отсутствия указанного файла – он будет создан.

Пример:

```
byte[] bw = {10, 20, 30};
byte[] br = new byte[20];
FileOutputStream out = null;
FileInputStream in = null;
try {
    out = new FileOutputStream("12345.txt");
    out.write(bw);
    out.close();
} catch (IOException e) {
    e.printStackTrace();
} finally {
    try {
        out.close();
    } catch (IOException e) {
        e.printStackTrace();
    }
}
try {
    in = new FileInputStream("12345.txt");
    int count = in.read(br);
    System.out.println("Прочитано " + count + " байт");
    in.close();
} catch (Exception e) {
    e.printStackTrace();
} finally {
    try {
        in.close();
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

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 можно обеспечить вызовом метода **connect()** с передачей соответствующего объекта **PipedStream**. Другой вариант – передать этот объект еще при вызове конструктора.

```
PipedInputStream in = null;
PipedOutputStream out = null;
try {
    in = new PipedInputStream();
    out = new PipedOutputStream();
    out.connect(in);
    for (int i = 0; i < 100; i++) {
        out.write(i);
    }
    int x;
    while ((x = in.read()) != -1) {
        System.out.print(x + " ");
    }
    in.close();
    out.close();
} catch (IOException e) {
    e.printStackTrace();
} finally {
    try {
        in.close();
    } catch (IOException e) {
        e.printStackTrace();
    }
    try {
        out.close();
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

SequenceInputStream

Класс **SequenceInputStream** последовательно считывает данные из нескольких входных потоков. 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.

Пример:

```
FileInputStream in1 = null, in2 = null;
SequenceInputStream seq = null;
FileOutputStream out = null;
try {
    in1 = new FileInputStream("1.txt");
    in2 = new FileInputStream("2.txt");
    seq = new SequenceInputStream(in1, in2);
    out = new FileOutputStream("3.txt");
    int rb = seq.read();
    while(rb != -1){
        out.write(rb);
        rb = seq.read();
    }
} catch (IOException e) {
    e.printStackTrace();
} finally {
    try { seq.close(); } catch ( IOException e ) { };
    try { out.close(); } catch ( IOException e ) { };
}
```

В результате выполнения этого кода в файл 3.txt будет записано содержимое файлов 1.txt и 2.txt. Закрытие потоков производится в блоке **finally**. Потоки **in1** и **in2** будут автоматически закрыты объектом **seq**.

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 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 ввода-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()**.

```
try {
    OutputStream out = new BufferedOutputStream(new
FileOutputStream("file.txt"));
    for(int i = 0; i < 1000000; i++)
        out.write(i);
    out.close();
    InputStream in = new BufferedInputStream(new FileInputStream("file.txt"));
    while (in.read() != -1) { }
    in.close();
} catch (IOException e) {
    e.printStackTrace();
}
```

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 static void main(String[] args) {
  try {
    DataOutputStream out = new DataOutputStream(new
FileOutputStream("file.txt"));
    out.writeInt(128);
    out.writeLong(128);
    out.close();
    DataInputStream in = new DataInputStream(new FileInputStream("file.txt"));
    System.out.println(in.readInt());
    System.out.println(in.readLong());
    in.close();
  } catch (Exception e) {
    e.printStackTrace();
  }
}
```

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 объект мог быть сериализован, он должен реализовать интерфейс **Serializable**, который не определяет никаких методов.

Рассмотрим пример записи/чтения объекта в байтовый массив:

```
public static void main(String[] args) {
  try {
    ByteArrayOutputStream os = new ByteArrayOutputStream();
    ObjectOutputStream oos = new ObjectOutputStream(os);
    Integer integerSave = new Integer(155);
    oos.writeObject(integerSave);
    byte[] arr = os.toByteArray();
    os.close();
    oos.close();
    ByteArrayInputStream is = new ByteArrayInputStream(os.toByteArray());
    ObjectInputStream ois = new ObjectInputStream(is);
    Integer integerRead = (Integer)ois.readObject();
    is.close();
    ois.close();
    System.out.println("Writed: " + integerSave + ", Readed: " + integerRead);
  } catch (Exception e) {
    e.printStackTrace();
  }
}
```

Восстановленный объект будет равен сериализованому. 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. Пример:

```
public class Zachetka implements Serializable {
}
public class Student implements Serializable {
    private int id;
    private String name;
    private int score;
    private Zachetka z;
    public Student(int id, String name, int score) {
        System.out.println("Student constructor");
        this.id = id;
        this.name = name;
        this.score = score;
        this.z = new Zachetka();
    }
    public void info() {
        System.out.println(id + " " + name + " " + score);
    }
}
public class MainClass {
    public static void main(String[] args) {
        Student s = new Student(1, "Bob", 40);
        try (ObjectOutputStream out = new ObjectOutputStream(new
FileOutputStream("out.txt"))) {
            out.writeObject(s);
        } catch (IOException e) {
            e.printStackTrace();
        }
        Student s2;
        try (ObjectInputStream in = new ObjectInputStream(new
FileInputStream("out.txt"))) {
            s2 = (Student) in.readObject();
            s2.info();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

Если объект был сериализован, а затем изменен и снова записан, то изменения не будут добавлены в файл. Механизм сериализации помечает объект, который уже был записан в граф. Когда в очередной раз попадется ссылка на него, она указывает на уже сериализованный объект. Такой механизм необходим для записи связанных объектов.

Если класс содержит в качестве полей другие объекты, то они тоже подлежат сериализации, и поэтому тоже должны быть сериализуемыми. Это касается и их вложенных объектов. Полный путь ссылок объекта по всем объектным ссылкам называется графом исходного объекта.

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.

Если необходимо управлять ходом сериализации и восстановления объекта, 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()** интерфейса **Externalizable**.

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**.

При управлении процессом сериализации не всегда имеет смысл обращаться к реализации интерфейса Externalizable. Если нежелательно сохранять и восстанавливать поле, то достаточно объявить его с модификатором **transient**. Аналогичным образом можно действовать, чтобы не пропустить сохранение объекта, десериализация которого все равно не будет иметь смысла – например, сетевое соединение.

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

Когда объект восстанавливается, таким полям выставляется значение по умолчанию. Для объектов это **null**.

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.

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:

```
BufferedWriter bw = null;
BufferedReader br = null;
try{
    bw = new BufferedWriter(new FileWriter("input.txt"));
    for (int i=0; i < 20; i++) bw.write("Java");
    bw.close();
    br = new BufferedReader(new FileReader("input.txt"));
    String str;
    while((str = br.readLine()) != null)
        System.out.println(str);
    br.close();
}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).

```
// Содержимое файла 1.txt: "123456789"
try (RandomAccessFile raf = new RandomAccessFile("1.txt", "r")) {
    raf.seek(2);
    System.out.println((char)raf.read());
} catch (IOException e) {
    e.printStackTrace();
}
```

```
Результат: 3
```

Домашнее задание

1. Прочитать файл (около 50 байт) в байтовый массив и вывести этот массив в консоль;
2. Последовательно сшить 5 файлов в один (файлы примерно 100 байт). Может пригодиться следующая конструкция:

   ```
   ArrayList<InputStream> al = new ArrayList<>();
   ...
   Enumeration<InputStream> e = Collections.enumeration(al);
   ```

3. Написать консольное приложение, которое умеет постранично читать текстовые файлы (размером > 10 mb). Вводим страницу (за страницу можно принять 1800 символов), программа выводит ее в консоль. Контролируем время выполнения: программа не должна загружаться дольше 10 секунд, а чтение – занимать свыше 5 секунд.

Чтобы не было проблем с кодировкой, используйте латинские буквы.

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.