Конспект лекции
Пакет java.io и работа с ресурсами

Цель и задачи лекции

Цель – изучить работу с потоками данных.

Задачи:

1. Научиться читать и писать данные из различных потоков
2. Дать понимание сериализации

План занятия

1. Потоки данных
2. Классы пакета java.io
3. Сериализация

Потоки данных

Очень часто приходится получать какой-то поток данных, а потом как-то их обрабатывать и отправлять дальше. Например, пользователь ввел логин и пароль, программа в свою очередь должна получить эти данные, обработать и сохранить в файл. В Java библиотека IO API находится в пакете java.io и для того, чтобы начать её использовать, достаточно импортировать данную библиотеку в класс.

Классы InputStream и OutputStream

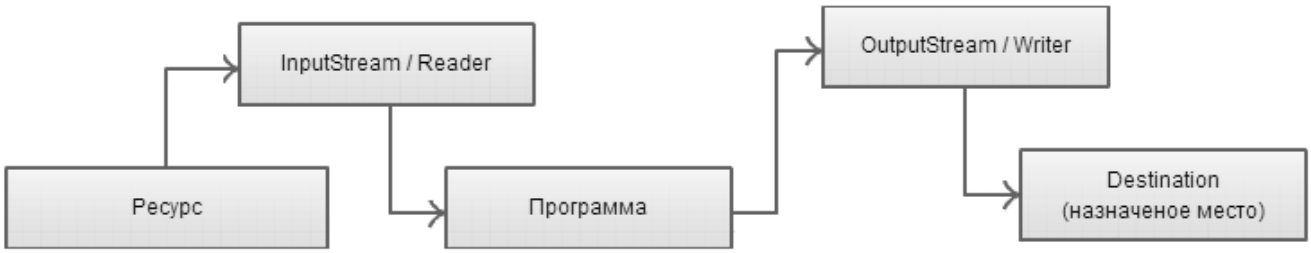
В java.io существуют так называемые потоки ввода и вывода (InputStream and OutputStream).

В основном java.io предназначен для чтения и записи данных в ресурс:

1. файл;
2. при работе с сетевым подключением;
3. System.err, System.in, System.out;
4. канал (pipe): данные помещаются с одного конца и извлекаются с другого;
5. при работе с буфером;
6. другие источники (например, подключение к интернету).

Процесс чтения данных из ресурса и запись их в назначенное место описан схемой.

Программа, которая должна считать данные с потока и записать в поток показана на рисунке ниже.



Как видите в этом случае Ресурс связан с InputStream /Reader, а он в свою очередь связан с программой и через него программа получает поток данных.

Для работы с указанными источниками используются подклассы базового класса InputStream.

Методы класса InputStream:

- int available() - возвращает количество байтов ввода, доступные в данный момент для чтения
- close() - закрывает источник ввода. Следующие попытки чтения передадут исключение IOException
- void mark(int readlimit) - помещает метку в текущую точку входного потока, которая остаётся корректной до тех пор, пока не будет прочитано readlimint байт
- boolean markSupported() - возвращает true, если методы mark() и reset() поддерживаются потоком
- int read() - возвращает целочисленное представление следующего доступного байта в потоке. При достижении конца файла возвращается значение -1
- int read(byte[] buffer) - пытается читать байты в буфер, возвращая количество прочитанных байтов. По достижении конца файла возвращает значение -1
- int read(byte[] buffer, int byteOffset, int byteCount) - пытается читать до byteCount байт в buffer, начиная с смещения byteOffset. По достижении конца файла возвращает -1
- reset() - сбрасывает входной указатель в ранее установленную метку
- long skip(long byteCount) - пропускает byteCount байт ввода, возвращая количество проигнорированных байтов

Класс OutputStream – это абстрактный класс, определяющий байтовый потоковый вывода. Наследники данного класса определяют куда направлять данные: в массив байтов, в файл или канал. Из массива байт можно создать текстовую строку String.

Методы класса OutputStream:

- void write(int b) записывает один байт в выходной поток. Аргумент этого метода имеет тип int, что позволяет вызывать write, передавая ему выражение, при этом не нужно выполнять приведение его типа к byte.
- void write(byte b[]) записывает в выходной поток весь указанный массив байтов.
- void write(byte b[], int off, int len) записывает в поток часть массива len байтов, начиная с элемента b[off].
- void flush() очищает любые выходные буферы, завершая операцию вывода.
- void close() закрывает выходной поток. Последующие попытки записи в этот поток будут возбуждать IOException.

ByteArrayInputStream и ByteArrayOutputStream

Классы ByteArrayInputStream и ByteArrayOutputStream реализуют потоки для чтения и записи в массив байт.

Класс ByteArrayInputStream представляет входной поток, использующий в качестве источника данных массив байтов. Он имеет следующие конструкторы:

- ByteArrayInputStream(byte[] buf)
- ByteArrayInputStream(byte[] buf, int offset, int length)

В качестве параметров конструкторы используют массив байтов buf, из которого производится считывание, смещение относительно начала массива offset и количество считываемых символов length.

Считаем массив байтов и выведем его на экран:

```
import java.io.*;

public class Main {

    public static void main(String[] args) {

        byte[] array1 = new byte[]{1, 3, 5, 7};
        ByteArrayInputStream byteStream1 = new ByteArrayInputStream(array1);
        int b;
        while( (b=byteStream1.read())!=-1 ){

            System.out.println(b);
        }

        String text = "Hello world!";
        byte[] array2 = text.getBytes();
        ByteArrayInputStream byteStream2 = new ByteArrayInputStream(array2, 0, 5);
        int c;
        while((c=byteStream2.read())!=-1){

            System.out.println((char)c);
        }
    }
}
```

В отличие от других классов потоков для закрытия объекта ByteArrayInputStream не требуется вызывать метод close.

Класс ByteArrayOutputStream представляет поток вывода, использующий массив байтов в качестве места вывода. Чтобы создать объект данного класса, можно использовать один из его конструкторов:

- ByteArrayOutputStream()
- ByteArrayOutputStream(int size)

Первый конструктор создает массив данных для хранения байтов длиной в 32 байта, а второй конструктор создает массив длиной size.

Примеры использования класса ByteArrayOutputStream:

```
public class Main {
    public static void main(String[] args) {
        ByteArrayOutputStream bos = new ByteArrayOutputStream();
        String text = "Hello World!";

        byte[] buffer = text.getBytes();
        try {
            bos.write(buffer);
        } catch(Exception e) {
            System.out.println(e.getMessage());
        }

        // Преобразование массива байтов в строку
        System.out.println(bos.toString());

        // Вывод в консоль посимвольно
        byte[] array = bos.toByteArray();
        for (byte b: array) {
            System.out.print((char)b);
        }
        System.out.println();
    }
}
```

В классе ByteArrayOutputStream метод write записывает в поток некоторые данные (массив байтов). Этот массив байтов записывается в объекте ByteArrayOutputStream в защищенное поле buf, которое представляет также массив байтов (protected byte[] buf). Так как метод write может вызвать исключение, то вызов этого метода помещается в блок try-catch.

Используя методы toString() и toByteArray(), можно получить массив байтов buf в виде текста или непосредственно в виде массива байт. С помощью метода writeTo() можно перенаправить массив байт в другой поток. Данный метод в качестве параметра принимает объект OutputStream, в который производится запись массива байт. Для ByteArrayOutputStream не надо явным образом закрывать поток с помощью метода close().

FileInputStream и FileOutputStream

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

Для создания объекта FileInputStream мы можем использовать ряд конструкторов. Наиболее используемая версия конструктора 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.

Приведем пример, считаем данные из файла и выведем на консоль:

```
import java.io.*;

public class Main {

    public static void main(String[] args) {

        try(FileInputStream fin=new FileInputStream("C://SomeDir//note.txt")) {
            System.out.println("Размер файла: " + fin.available() + " байт(а)");

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

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

Подобным образом можно 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]);
}
```

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

Приведем пример, запишем в файл строку:

```
import java.io.*;

public class FilesApp {

    public static void main(String[] args) {

        String text = "Hello world!"; // строка для записи
        try(FileOutputStream fos=new FileOutputStream("C://SomeDir//notes.txt"))
        {
            // перевод строки в байты
            byte[] buffer = text.getBytes();

            fos.write(buffer, 0, buffer.length);
```

```
        }
        catch(IOException ex){

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

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

Необязательно записывать весь массив байтов. Используя перегрузку метода write(), можно записать и одиночный байт:

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

Совместим оба класса и выполним чтение из одного и запись в другой файл:

```
import java.io.*;

public class FilesApp {

    public static void main(String[] args) {

        try(FileInputStream fin=new FileInputStream("C://temp//notes.txt");
            FileOutputStream fos=new FileOutputStream("C://temp//notes_new.txt"))
        {
            byte[] buffer = new byte[fin.available()];
            // считываем буфер
            fin.read(buffer, 0, buffer.length);
            // записываем из буфера в файл
            fos.write(buffer, 0, buffer.length);
        }
        catch(IOException ex){

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

Классы FileInputStream и FileOutputStream предназначены прежде всего для записи двоичных файлов. И хотя они также могут использоваться для работы с текстовыми файлами, но все же для этой задачи больше подходят другие классы.

BufferedInputStream и BufferedOutputStream

Классы BufferedInputStream и BufferedOutputStream служат для буферизации потока информации.

Класс BufferedInputStream накапливает вводимые данные в специальном буфере без постоянного обращения к устройству ввода:

```
public class FilesApp {

    public static void main(String[] args) {

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

```
        ByteArrayInputStream in = new ByteArrayInputStream(buffer);

        try(BufferedInputStream bis = new BufferedInputStream(in)){

            int c;
            while((c=bis.read())!=-1){

                System.out.print((char)c);
            }
        }
        catch(Exception e){

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

Класс BufferedInputStream в конструкторе принимает объект InputStream. В данном случае таким объектом является экземпляр классаByteArrayInputStream.

Как и все потоки ввода BufferedInputStream обладает методом read(), который считывает данные. И здесь мы считываем с помощью метода read каждый байт из массива buffer.

Фактические все то же самое можно было сделать и с помощью одного ByteArrayInputStream, не прибегая к буферизированному потоку. Класс BufferedInputStream просто оптимизирует производительность при работе с потоком ByteArrayInputStream.

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

```
import java.io.*;

public class FilesApp {

    public static void main(String[] args) {

        String text = "Hello world!"; // строка для записи
        try(FileOutputStream out=new FileOutputStream("notes.txt");
            BufferedOutputStream bos = new BufferedOutputStream(out))
        {
            // перевод строки в байты
            byte[] buffer = text.getBytes();
            bos.write(buffer, 0, buffer.length);
        }
        catch(IOException ex){

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

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

DataInputStream и DataOutputStream

Классы DataInputStream и DataOutputStream реализуют потоки для работы с остальными примитивными типами и со String.

Класс DataOutputStream представляет поток вывода и предназначен для записи данных примитивных типов, таких, как int, double и т.д. Для записи каждого из примитивных типов предназначен свой метод:

- writeBoolean(boolean v) : записывает в поток булевое однобайтовое значение
- writeByte(int v): записывает в поток 1 байт, которые представлен в виде целочисленного значения
- writeChar(int v): записывает 2-байтовое значение char
- writeDouble(double v): записывает в поток 8-байтовое значение double
- writeFloat(float v): записывает в поток 4-байтовое значение float
- writeInt(int v): записывает в поток целочисленное значение int
- writeLong(long v): записывает в поток значение long
- writeShort(int v): записывает в поток значение short
- writeUTF(String str): записывает в поток строку в кодировке UTF-8

Класс DataInputStream действует противоположным образом - он считывает из потока данные примитивных типов. Соответственно для каждого примитивного типа определен свой метод для считывания:

- boolean readBoolean(): считывает из потока булевое однобайтовое значение
- byte readByte(): считывает из потока 1 байт
- char readChar(): считывает из потока значение char
- double readDouble(): считывает из потока 8-байтовое значение double
- float readFloat(): считывает из потока 4-байтовое значение float
- int readInt(): считывает из потока целочисленное значение int
- long readLong(): считывает из потока значение long
- short readShort(): считывает значение short
- String readUTF(): считывает из потока строку в кодировке UTF-8
- int skipBytes(int n): пропускает при чтении из потока n байтов

Рассмотрим применение классов на примере:

```
class Person
{
    public String name;
    public int age;
    public double height;
    public boolean married;

    public Person(String n, int a, double h, boolean m) {
        this.name=n;
```

```
        this.height=h;
        this.age=a;
        this.married=m;
    }
}


public class FilesApp {

    public static void main(String[] args) {

        Person tom = new Person("Tom", 35, 1.75, true);
        // запись в файл
        try(DataOutputStream dos = new DataOutputStream(
                        new FileOutputStream("data.bin"))) {
            // записываем значения
            dos.writeUTF(tom.name);
            dos.writeInt(tom.age);
            dos.writeDouble(tom.height);
            dos.writeBoolean(tom.married);
            System.out.println("Запись в файл произведена");
        } catch(IOException ex) {
            System.out.println(ex.getMessage());
        }

        // обратное считывание из файла
        try(DataInputStream dos = new DataInputStream(
                        new FileInputStream("data.bin"))) {
            // записываем значения
            String name = dos.readUTF();
            int age = dos.readInt();
            double height = dos.readDouble();
            boolean married = dos.readBoolean();
            System.out.printf("Человека зовут: %s , его возраст: %d , его рост: %f
метров, женат/замужем: %b",
                        name, age, height, married);
        } catch(IOException ex){

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

}
```

Здесь последовательно записывается в файл данные объекта Person.

Объект DataOutputStream в конструкторе принимает поток вывода: DataOutputStream (OutputStream out). В данном случае в качестве потока вывода используется объект FileOutputStream, поэтому вывод будет происходить в файл. И с помощью выше рассмотренных методов типа writeUTF() производится запись значений в бинарный файл.

Затем происходит чтение ранее записанных данных. Объект DataInputStream в конструкторе принимает поток для чтения: DataInputStream(InputStream in). Здесь таким потоком выступает объект FileInputStream

ObjectInputStream и ObjectOutputStream и сериализация

ObjectInputStream и ObjectOutputStream позволяют работать с потоками для чтения/записи объектов с использованием механизма сериализации.

Сериализация представляет процесс записи состояния объекта в поток, соответственно процесс извлечения или восстановления состояния объекта из потока называется десериализацией. Сериализация очень удобна, когда идет работа со сложными объектами.

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

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.

Например, сохраним в файл один объект класса Person:

```
class Person implements Serializable {

    public String name;
    public int age;
    public double height;
    public boolean married;

    Person(String n, int a, double h, boolean m){
```

```
        name=n;
        age=a;
        height=h;
        married=m;
    }
}


import java.io.*;

public class FilesApp {

    public static void main(String[] args) {

        try(ObjectOutputStream oos = new ObjectOutputStream(new
FileOutputStream("person.dat")))
        {
            Person p = new Person("Джон", 33, 178, true);
            oos.writeObject(p);
        }
        catch(Exception ex){

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

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

Например, извлечем выше сохраненный объект Person из файла:

```
public class FilesApp {

    public static void main(String[] args) {

        try(ObjectInputStream ois = new ObjectInputStream(
                new FileInputStream("person.dat")))  {
            Person p = (Person)ois.readObject();
            System.out.printf("Имя: %s \t Возраст: %d \n", p.name, p.age);
        }
        catch(Exception ex){

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

    }
}
```

Сериализация

Сериализация - это процесс записи состояния объектов в поток вывода байтов. Она оказывается удобной в том случае, когда требуется сохранить состояние прикладной программы в таком месте постоянного хранения, как файл. В дальнейшем эти объекты можно восстановить в процессе десериализации.

Сериализация также требуется для реализации удаленного вызова методов (RMI - Remote Method Invocation). Механизм RMI позволяет объекту Java на одной машине обращаться к методу объекта java на другой машине. Объект может быть предоставлен в виде аргумента этого удаленного метода. Передающая машина сериализует и посылает объект, а принимающая машина десериализует его.

Интерфейс Serializable

Средствами сериализации может быть сохранен и восстановлен только объект класса, реализующего интерфейс Serializable. В интерфейсе Serializable не определяется никаких членов. Он служит лишь для того, чтобы указать, что класс может быть сериализирован. Если класс сериализируется, то сериализируются и все его подклассы. Переменные, объявленные как transient, не сохраняются средствами сериализации. Не сохраняются и статические переменные.

Метаданные Java-сериализации - информация, включенная в двоичный формат сериализации, - имеют сложную структуру и решают многие проблемы, досаждающие ранним разработчикам промежуточного ПО. Но всех проблем они тоже не решают.

При Java-сериализации используется свойство serialVersionUID, которое помогает справиться с разными версиями объектов в сценарии сериализации. Вам не нужно декларировать это свойство для своих объектов; по умолчанию платформы Java использует алгоритм, который вычисляет его значение на основе атрибутов класса, его имени и положения в локальном кластере. В большинстве случаев это работает отлично. Но при добавлении или удалении атрибутов это динамически сгенерированное значение изменится, и среда исполнения Java выдаст исключение InvalidClassException.

Чтобы избежать этого, нужно взять в привычку явное объявление serialVersionUID:

```
import java.io.Serializable;
public class Person implements Serializable {
    private static final long serialVersionUID = 20100515;
    // ...
}
```

Интерфейс Externalizable

Средства Java для сериализации и десериализации разработаны таким образом, чтобы большая часть операций сохранения и восстановления состояния объекта выполнялась автоматически. Но иногда требуется управлять этим процессом вручную, например, чтобы воспользоваться алгоритмами сжатия и шифрования данных. Именно для таких случаев и предназначен интерфейс Externalizable.

В интерфейсе Externalizable определяются следующие методы:

```
void readExternal (ObjectInput поток_ввода)
      throws IOException, ClassNotFoundException


void writeExternal (ObjectOutput поток_вывода) throws IOException
```

В этих методах параметр поток ввода обозначает поток байтов, из которого может быть введен объект, а параметр поток_ выв ода - поток байтов, куда это объект может быть выведен.

Преимущества потоков ввода-вывода

Потоковый интерфейс ввода-вывода в Java предоставляет чистую абстракцию для решения сложных и зачастую обременительных задач. Структура классов фильтрующих потоков позволяет динамически строить собственные настраиваемые потоковые интерфейсы, отвечающие требованиям передачи данных. Те программы на Java, где применяются эти классы с высоким уровнем абстракции, в том числе InputStream, OutputStream, Reader и Writer, будут правильно функционировать и впредь - даже в том случае, если появятся новые и усовершенствованные конкретные классы потоков ввода-вывода. Эта модель оказывается вполне работоспособной при переходе от ряда потоков ввода-вывода в файлы к потокам ввода-вывода через сеть и сокеты. И наконец, сериализация объектов играет важную роль в самых разных программах на Java. Классы сериализации ввода-вывода в Java обеспечивают переносимое решение этой порой не совсем простой задачи.

Литература и ссылки

1. Спецификация языка Java https://docs.oracle.com/javase/specs/jls/se8/html/index.html
2. Шилдт Г. Java 8. Полное руководство. 9-е издание. — М.: Вильямс, 2012. — 1377 с.
3. Эккель Б. Философия Java. 4-е полное изд. — СПб.: Питер, 2015. — 1168 с.

Вопросы для самоконтроля

1. Перечислите основные методы класс InputStream?
2. Что такое граф сериализации?
3. Какой паттерн проектирования использует java.io?
4. В чем назначение serialVersionUID?