Занятие № 10.

Потоки ввода-вывода в Java

Потоки ввода-вывода

Чтение и запись в файл

Сериализация

Потоки ввода-вывода

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.**

Ключевым понятием здесь является понятие **потока**. Хотя понятие "поток" в программировании довольно перегружено и может обозначать множество различных концепций. В данном случае применительно к работе с файлами и вводом-выводом мы 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. У нас может быть определен поток, который связан с файлом и через который мы можем 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 мы будем решать в Java с помощью потоков.

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.

В основе всех классов, управляющих потоками байтов, находятся два абстрактных класса: **InputStream** (представляющий потоки ввода) и **OutputStream** (представляющий потоки вывода)

Но поскольку работать с байтами не очень удобно, то 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.

Основные классы потоков:

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

Потоки байтов

Класс InputStream

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

- `int available()`: возвращает количество байтов, доступных для чтения в потоке
- `void close()`: закрывает поток
- `int read()`: возвращает целочисленное представление следующего байта в потоке. Когда в потоке не останется доступных для чтения байтов, данный метод возвратит число -1
- `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
- `int read(byte[] buffer, int offset, int length)`: считывает некоторое количество байтов, равное length, 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.
- `long skip(long number)`: пропускает в потоке при чтении некоторое количество байт, которое равно number

Класс OutputStream

Класс OutputStream является базовым классом для всех классов, которые работают с бинарными потоками записи. Свою функциональность он реализует через следующие методы:

- `int close()`: закрывает поток
- `void flush()`: очищает буфер вывода, записывая все его содержимое
- `void write(int b)`: записывает в выходной поток один байт, который представлен целочисленным параметром b
- `void write(byte[] buffer)`: записывает в выходной поток массив байтов buffer.
- `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]`.

Абстрактные классы Reader и Writer

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

- `absract void close()`: закрывает поток ввода
- `int read()`: возвращает целочисленное представление следующего символа в потоке. Если таких символов нет, и достигнут конец файла, то возвращается число -1
- `int read(char[] buffer)`: считывает в массив buffer из потока символы, количество которых равно длине массива buffer. Возвращает количество успешно считанных символов. При достижении конца файла возвращает -1
- `int read(CharBuffer buffer)`: считывает в объект CharBuffer из потока символы. Возвращает количество успешно считанных символов. При достижении конца файла возвращает -1
- `absract int read(char[] buffer, int offset, int count)`: считывает в массив buffer, начиная со смещения offset, из потока символы, количество которых равно count
- `long skip(long count)`: пропускает количество символов, равное count. Возвращает число успешно пропущенных символов

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
- `Writer append(CharSequence chars)`: добавляет в конец выходного потока набор символов chars. Возвращает объект Writer
- `abstract void close()`: закрывает поток
- `abstract void flush()`: очищает буферы потока
- `void write(int c)`: записывает в поток один символ, который имеет целочисленное представление
- `void write(char[] buffer)`: записывает в поток массив символов
- `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
- `void write(String str)`: записывает в поток строку
- `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

Функционал, описанный классами Reader и Writer, наследуется непосредственно классами символьных потоков, в частности классами FileReader и FileWriter соответственно, предназначенными для работы с текстовыми файлами.

Закрытие потоков

При завершении работы с потоком его надо закрыть с помощью метода close(), который определен в интерфейсе Closeable. Метод close имеет следующее определение:

void close() throws IOException

Этот интерфейс уже реализуется в классах InputStream и OutputStream, а через них и во всех классах потоков.

При закрытии потока освобождаются все выделенные для него ресурсы, например, файл. В случае, если поток окажется не закрыт, может происходить утечка памяти.

Чтение и запись в файл

Чтение файлов и класс FileInputStream

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

Для создания объекта FileInputStream 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.

Считаем данные из файла и выведем на консоль:

```
import java.io.*;

public class FilesApp {

    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]);
}
```

Запись файлов и класс FileOutputStream

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

Например, 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://SomeDir//notes.txt");
                FileOutputStream fos=new FileOutputStream("C://SomeDir//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.

Сериализация

Сериализация объекта - это способность объекта сохранять полную копию его и любых других объектов на которые он ссылается, используя поток вывода(например, во внешний файл). Таким образом, объект может быть воссоздан из сериализованной(сохраненной) копии немного позже, когда это потребуется.

Сериализация объектов, как новая возможность, введенная в JDK 1.1, предоставляет функцию для преобразования групп или отдельных объектов, в поток битов или массив байтов, для хранения или передаче по сети. И как было сказано, данный поток битов или массив байтов, можно преобразовать обратно в объекты Java. Главным образом это происходит автоматически благодаря классам ObjectInputStream и ObjectOutputStream. Программист может решить реализовать эту функцию, путем реализации интерфейса Serializable при создании класса.

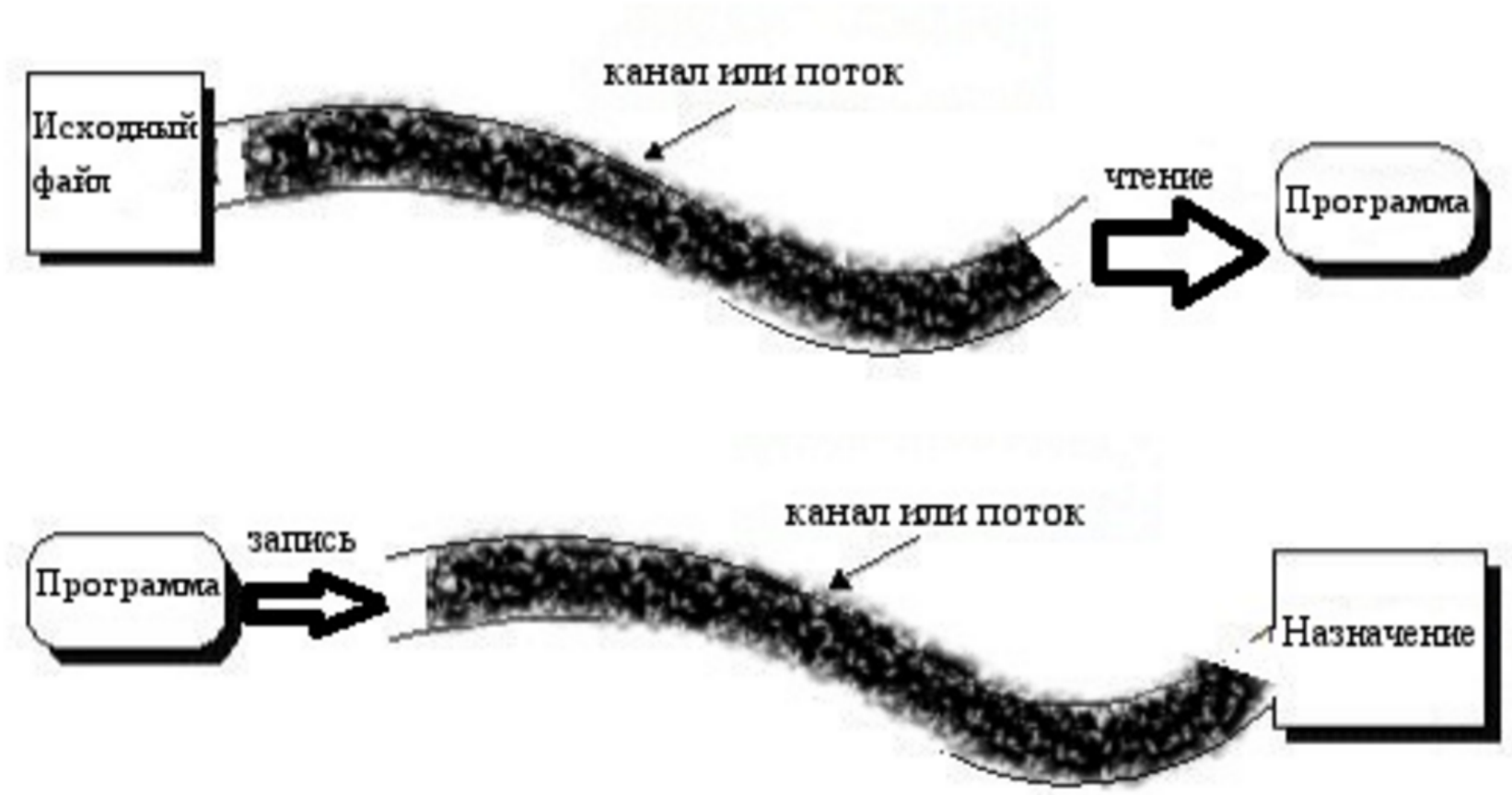
Процесс сериализиции также известен как **маршалинг** объекта, десериализация же известна как **демаршалинг**.

Сериализация - это механизм, предоставляющий возможность объекту сохранить свою копию и все другие объекты, на которые ссылается данный объект, во внешний файл с помощью класса ObjectOutputStream. Сохранены могут быть структуры данных, диаграммы, объекты класса JFrame или любые другие объекты, независимо от их типа. В то же время, сериализация сохраняет информацию о том, какого типа объект, чтобы в дальнейшем, при десериализации, эта информация использовалась для воссоздания точного типа объекта, которым он был.

Чтобы полностью понять концепцию сериализации, надо иметь четкое понимание других двух концепций — персистентности объектов и потоков.

Потоки:

Каждая программа должна записывать свои данные в место хранения или канал, и каждая программа должна считывать данные из канала или места хранения. В Java, эти каналы, куда программы записывают и откуда программы считывают данные, называются **Потоками(Stream).**



Графическое представление Потоков

Потоки в основном делятся на два типа:

- Байтовые классы-потоки именуемые *Streams
- Символьные классы-потоки именуемые *Reader и *Writer

Каждый поток записи данных, содержит набор методов записи. И каждый поток считывания данных, соответственно имеет подобный набор методов чтения. Как только поток создается, все эти методы должны быть вызваны.

Теперь, поговорим о персистентности.

Персистентность:

Персистентность объекта - это способность объекта жить или по-другому — «пережить» выполнение программы. Это значит, что любой объект, который был создан во времени выполнения, уничтожается сборщиком мусора JVM, всякий раз, когда данный объект далее перестает использоваться.

Но в случае реализации API персистентности, данные объекты не будут уничтожаться сборщиком мусора JVM, вместо чего им будет позволено «жить», что также дает возможность доступа к ним при следующем запуске приложения. Другими словами, персистенция означает существование времени жизни объекта, независимо от времени жизни приложения, которое запущено.

Один из способов реализации персистентности - это хранение объектов где-нибудь во внешнем файле или в базе данных, а затем восстановление их в более позднее время, используя данные файлы или базу данных как источники. Здесь сериализация и вступает в игру. Любой неперсистентный объект существует так долго, как долго работает JVM.

Сериализованные объекты – это просто объекты, преобразованные в потоки, которые затем сохраняются во внешний файл или передаются через сеть для хранения и восстановления.

Реализация интерфейса Serializable

Любой класс должен реализовывать интерфейс java.io.Serializable для сериализации объектов этого класса. Интерфейс Serializable не имеет методов и только маркерует класс, чтобы можно было идентифицировать его как сериализуемый. Только поля объекта сериализованного класса могут быть сохранены. Методы или конструкторы не сохраняются, как части сериализованного потока. Если какой-либо объект действует как ссылка на другой объект, то поля этого объекта также сериализованны, если класс этого объекта реализует интерфейс Serializable. Другими словам, получаемый таким образом граф этого объекта, сериализуем полностью. Граф объекта включает дерево или структуру полей объекта и его подобъектов.

Два главных класса, которые помогают реализовать интерфейс Seriliazable:

- ObjectInputStream
- ObjectOutputStream

Листинг 1. Пример простого класса, чтобы показать сериализацию

```
import java.io.*;
public class RandomClass implements Serializable {
 // Генерация рандомного значения
 private static int r() {
        return (int)(Math.random() * 10);
 }
    private int data[];
    // Конструктор
public RandomClass() {
        datafile = new int[r()];
        for (int i=0; i<datafile.length; i++)
        datafile[i]=r();
 }
    public void printout() {
 System.out.println("This RandomClass has "+datafile.length+" random integers");
 for (int i=0; i<datafile.length; i++) {
        System.out.print(datafile[i]+":");
        System.out.println();
    }
}
```

В приведенном выше коде, создается класс, который является сериализуемым, т.к. «промаркерован» интерфейсом сериализации. Класс создает массив случайных целых чисел, когда создается его экземпляр.

Приведенный ниже код, показывает возможность записи объектов в поток, используя класс ObjectOutputStream. Программа имеет массив целых чисел, но для сериализации мы не должны перебирать ее внутренние объекты.

Интерфейс Seriliazable заботится об этом автоматически.

Листинг 2. Простой пример сериализации объектов для вывода в файл

```
import java.io.*;
import java.util.*;
public class OutSerialize {
    public static void main (String args[]) throws IOException {
        RandomClass rc1 = new RandomClass();
        RandomClass rc2 = new RandomClass();
//создание цепи потоков с потоком вывода объекта в конце
ObjectOutputStream out = new ObjectOutputStream(new FileOutputStream("objects.dat"));
        Date now = new Date(System.currentTimeMillis());
//java.util.* был импортирован для использования класса Date
        out.writeObject(now);
        out.writeObject(rc1);
        out.writeObject(rc2);
out.close();
        System.out.println("I have written:");
System.out.println("A Date object: "+now);
        System.out.println("Two Group of randoms");
rc1.printout();
rc2.printout();
 }
}
```

Код ниже демонстрирует возможности класса ObjectInputStream, который считывает сериализованные данные с внешнего файла, в программу. Заметьте, что объекты считываются в том же порядке, в котором были записаны в файл.

Листинг 3. Чтение сериализованных объектов или Десериализация

```
import java.io.*;
import java.util.*;
public class InSerialize {
 public static void main (String args[]) throws  IOException, ClassNotFoundException {
    ObjectInputStream in =  new ObjectInputStream (new FileInputStream("objects.dat"));
 Date d1 = (Date)in.readObject();
 RandomClass rc1 = (RandomClass)in.readObject();
    RandomClass rc2 = (RandomClass)in.readObject();
    System.out.println("I have read:");
    System.out.println("A Date object: "+d1);
    System.out.println("Two Group of randoms");
    rc1.printout();
rc2.printout();
 }
}
```

Почти все классы Java могут быть сериализованны, включая классы AWT. Фрейм, который является окном, содержит набор графических компонентов. Если фрейм сериализован, механизм сериализации заботится об этом и сериализует все его компоненты и данные(позицию, содержание и т.д.). Некоторые объекты классов Java, не могут быть сериализованы, потому что содержат данные, что ссылаются на кратковременные ресурсы операционных систем. Например классы java.io.FileInputStream и java.lang.Thread. Если объект содержит ссылки на несериализуемые элементы, вся операция сериализации потерпит неудачу и будет выброшено исключение NotSerializableException. Если какой-либо объект ссылается на ссылку несериализованного объекта, то его можно сериализовать используя ключевое слово transient.

Листинг 4. Создание сериализуемых объектов используя ключево слово transient

```
public class Sclass implements Serializable{
public transient Thread newThread;
//помните, что поток(поток параллельного исполнения) по умолчанию не сериализуемый класс
    private String studentID;
    private int sum;
}
```

Безопасность в Сериализации

Сериализация класса в Java подразумевает передачу всех его данных во внешний файл или базу данных через поток. Мы можем ограничить данные, которые будут сериализованны, когда того пожелаем.

Есть два способа сделать это:

- Каждый параметр класса объявленный как transient, не сериализуются(по умолчанию все параметры класса сериализуются)
- Или каждый параметр класса, который мы хотим сериализовать, помечается тегом Externalizable(по умолчанию никакие параметры не сериализуются).

Поле данных не будет сериализовано с помощью ObjectOutputStream, когда оно будет вызвано для объекта, если это поле данных, данного объекта помечено как transient. Например – private transient String password.

С другой стороны, для явного объявления данных объекта как сериализуемых, мы должны промаркировать класс как Externalizable, и реализовать методы writeExternal и readExteranl для записи и чтения данных этого объекта явно.

Например:

```
import java.io.Externalizable;
import java.io.IOException;
import java.io.ObjectInput;
import java.io.ObjectOutput;

public class MessageToSerialize implements Externalizable {
    private int a = 13;

    public MessageToSerialize(){
    }

    public MessageToSerialize(int a){
        this.a = a;
    }

    public int getA(){
        return this.a;
    }

    @Override
    public void writeExternal(ObjectOutput out) throws IOException {
        out.write(this.a);

    }

    @Override
    public void readExternal(ObjectInput in) throws IOException, ClassNotFoundException {
        this.a = in.readInt();
    }
}
```