ЛАБОРАТОРНАЯ РАБОТА «ВВОД-ВЫВОД. ПАКЕТ JAVA.IO. СЕРИАЛИЗАЦИЯ/ДЕСЕРИАЛИЗАЦИЯ ОБЪЕКТОВ»

Цель работы.

1.изучить применение классов пакета java.io для организации ввода-вывода данных в приложениях на языке Java;

2.освоить работу с текстовыми и бинарными файлами;

3.изучить и освоить механизм сериализации-десериализации объектов классов.

Порядок выполнения работы:

1.изучить особенности реализации потоков ввода-вывода в Java;

2.изучить способы использования потоков ввода-вывода в Java;

3.создать приложение, демонстрирующее различные свойства потоков ввода-вывода для Java:

– ввод/вывод на консоль в русской кодировке;

– ввод/вывод в файлы в русской кодировке;

– взаимодействие между потоками исполнения через каналы ввода-вывода;

– ввод/вывод объектов разных типов в файлы и потоки.

.

Краткие теоретические сведения

1.1. Общие сведения о потоках ввода/вывода. Базовые классы потоков.

Как известно всем программистам с давних времен, большинство программ не может выполнять свою работу, не имея доступа к внешним данным. Данные извлекаются из источника ***ввода.*** Результат программы направляется в ***вывод.***

На языке Java эти понятия определяются очень широко. Например, источником ввода или местом вывода может служить **сетевое соединение, буфер памяти или дисковый файл** — всеми ими можно манипулировать при помощи классов ввода-вывода Java. Хотя физически они совершенно различны, все эти устройства описываются единой абстракцией — ***потоком.***

Поток - это логическая сущность, которая выдает или получает информацию. Поток присоединен к физическому устройству при помощи системы вводавывода Java. Все потоки ведут себя похоже, даже несмотря на то, что физические устройства, к которым они присоединены, в корне отличаются.

Потоковая система ввода-вывода, используемая пакетом java.io, была частью языка Java начиная с его первого выпуска и широко используется до сих пор. Однако начиная с версии 1.4 в язык Java была добавлена вторая система ввода-вывода. Она называется **NIO** (что первоначально было акронимом от **New I/O** (новый ввод-вывод)).

Система NIO расположена в пакете ***java.nio*** и его внутренних пакетах. С выпуском комплекта JDK 7 возможности системы NIO были существенно расширены.

Преимущества потоков

Потоковый интерфейс ввода-вывода в Java предоставляет чистую абстракцию для сложных и зачастую обременительных задач. Композиция классов фильтрующих потоков позволяет динамически строить собственные настраиваемые потоковые интерфейсы, которые отвечают вашим требованиям к передаче данных.

Программы Java, использующие эти абстрактные высокоуровневые классы — InputStream, OutputStream, Reader и Writer**,** — будут корректно функционировать в будущем, даже когда появятся новые усовершенствованные конкретные потоковые классы. Эта модель работает очень хорошо, когда мы переключаемся от набора потоков на основе файлов к сетевым потокам и потокам сокетов.

И наконец, сериализация объектов играет важную роль в программах Java различных типов. Классы сериализации ввода-вывода Java обеспечивают переносимое решение этой непростой задачи.

Исключения ввода-вывода

Два исключения играют важную роль в обработке ввода-вывода. Первое из них — исключение **IOException** — имеет отношение к большинству классов ввода-вывода, описанных в данной главе, поэтому при ошибке ввода-вывода происходит передача исключения IOException**.** В большинстве случаев, если файл не может быть открыт, передается исключение FileNotFoundException. Класс исключения FileNotFoundException происходит от класса IOException, поэтому оба могут быть обработаны в одном блоке **catch,** предназначенном для обработки исключения IOException. Этот подход используется для краткости в большинстве примеров кода данной работы. Однако в собственных приложениях вам может иметь смысл обрабатывать их по отдельности.

Другой класс исключения, который иногда очень важен при выполнении ввода-вывода, — это класс SecurityException**.** Когда присутствует менеджер безопасности, некоторые классы файлов передают исключение SecurityException при попытке открыть файл с нарушением безопасности. По умолчанию приложения, запущенные при помощи команды **java,** не используют менеджер безопасности. Поэтому приведенные здесь примеры ввода-вывода не отслеживают возможность передачи исключения SecurityException**.** Однако апплеты будут использовать менеджер безопасности, предоставленный браузером, и файловый ввод-вывод, выполняемый апплетом, может передать исключение SecurityException**.** В таком случае вам придется обрабатывать и это исключение.

Базовые классы потоков

Основанный на потоках ввода-вывода, Java построен на базе абстрактных классов: InputStream, OutputStream, Reader и Writer**.** Они используются для создания некоторых конкретных подклассов потоков.

Хотя ваши программы реализуют свои операции ввода-вывода через конкретные подклассы, классы верхнего уровня определяют базовые функциональные возможности, общие для всех потоковых классов.

Классы InputStream и OutputStream предназначены для байтовых потоков, а абстрактные классы Reader и Writer — для символьных. Классы байтовых и символьных потоков формируют отдельные иерархии. В целом классы символьных потоков следует использовать, имея дело с символами строк, а классы байтовых потоков — работая с байтами или другими двоичными объектами.

1.2 Файловый ввод/вывод

Для работы с файлами в Java предусмотрены следующие классы:

- класс **File** – работа с файлом, как с объектом (нет доступа к содержимому файла)
- для чтения/записи двоичных файлов – **FileInputStream** и **FileOutputStream**
- для чтения/записи текстовых файлов – **FileReader** и **FileWriter**

Рассмотрим эти классы ниже.

1.2.1 Класс File

Хотя большинство классов, определенных в пакете java.io**,** работают с потоками, класс File этого не делает. Он имеет дело непосредственно с файлами и файловой системой. То есть **класс** File не указывает, как извлекается и сохраняется информация в файлах; **он описывает свойства самих файлов.**

Объект класса **File** служит для получения информации и манипулирования информацией, ассоциированной с дисковым файлом, такой как права доступа, время, дата и путь к каталогу, а также для навигации по иерархиям подкаталогов.

Интерфейс **Path** и класс **Files,** добавленные в систему NIO комплектом JDK 7, являются серьезной альтернативой классу File во многих случаях.

Класс **File** — первичный источник и место назначения для данных во многих программах. Хотя существует несколько ограничений в части использования файлов в аплетах (из соображений безопасности), тем не менее они продолжают оставаться центральным ресурсом для хранения постоянной и разделяемой информации.

Каталог в Java трактуется как объект **класса** File с единственным дополнительным свойством — списком имен файлов, которые могут быть получены методом list()**.**

Для создания объектов класса File могут быть использованы следующие ***конструкторы***.

```
File(String путьКкатапогу)
File(String путьКкатапогу, String имяфайла)
```

```
File(File объектКаталога, String имяфайла)
```

Здесь ***путьКкаталогу***—это путь к файлу; ***имяфайла*** — имя файла или подкаталога; ***объектКаталога*** — объект класса File**,** указывающий каталог.

В следующем примере создается три файла: **f1, f2** и **f3**. Первый объект класса File создается с путем к каталогу в единственном аргументе. Второй включает два аргумента — путь и имя файла. Третий включает путь, присвоенный файлу f**1,** и имя файла; объект файла **f3** ссылается на тот же файл, что и **f2.**

```
File f1 = new File ("/*');
File f2 = new File("/","autoexec.bat");
File f3 = new File(f1,"autoexec.bat");
```

Класс **File** определяет множество методов, представляющих стандартные свойства объекта класса **File:**

Например, метод **getName** () возвращает имя файла,

метод **getParent** () — имя родительского каталога,

метод **exists** () — значение **true,** если файл (или каталог) существует, и значение **false** — если нет.

Однако класс **File** не симметричен. В нем есть несколько методов, позволяющих ***проверять*** свойства простого файлового объекта, у которых нет дополняющих их методов изменения этих атрибутов.

В следующем примере демонстрируется применение нескольких методов класса File. Здесь подразумевается, что в каталоге приложения **src** существует каталог по имени **java** и что он содержит файл по имени COPYRIGHT**.**

Пример: Работа с объектом-файлом, чтение его свойств

```
// Демонстрация работы с File
import java.io.File;
class FileDemo {
  /*
  * метод для вывода сообщений - фактически замена
  * System.out.println
  */
  static void p(String s) {
     System.out.println (s);
  }
  public static void main(String args[] ) {
      File f1 = new File("src/java/COPYRIGHT");
      p("Имя файла: " + f1.getName());
      p("Путь: " + f1.getPath()) ;
      p("Абсолютный путь: " + f1.getAbsolutePath());
      p("Родительский каталог: " + f1.getParent());
      p(f1.exists() ? "существует" : "не существует");
      p(f1.canWrite() ? "доступен для записи" : "не доступен для
      записи");
```

```
        p(f1.canRead() ? "доступен для чтения" : "не доступен для
        чтения");
        p(f1.isDirectory() ? "является каталогом" : "не является
        каталогом");
        p(f1.isFile() ? "является обычным файлом" : "может быть
        именованным каналом");
        p(f1.isAbsoIute() ? "является абсолютным" : "не является
        абсолютным");
        p("Время модификации: " + f1.lastModifled());
        p("Размер: " + f1.length() + " байт");
    }
}
```

Эта программа выдает нечто вроде следующего.

```
Имя файла: COPYRIGHT
Путь: \java\COPYRIGHT
Абсолютный путь: \java\COPYRIGHT
Родительский каталог: \java
существует
доступен для записи
доступен для чтения
не является каталогом
является обычным файлом
не является абсолютным
Время модификации: 1282832030047
Размер: 695 байт
```

Большинство методов класса File самоочевидно, но **методы** isFile **()** и isAbsolute **()** — нет.

Метод **isFile ()** возвращает значение **true,** если вызывается с файлом, и значение **false** — если с каталогом. Также метод isFile **()** возвращает значение **false** для некоторых специальных файлов, таких как драйверы устройств и именованные каналы, поэтому этот метод может применяться для гарантии того, что данный файл действительно ведет себя как файл.

Метод **isAbsolute ()** возвращает значение **true,** если файл имеет абсолютный путь, и значение **false** —если относительный.

Класс File включает также два полезных служебных метода. Первый из них, метод **renameTo (),** показан ниже:

boolean renameTo(File *новоеИмя)*

Здесь имя файла, указанное в параметре ***новоеИмя***, становится новым именем вызывающего объекта **класса** File**.** Метод возвращает значение **true** в случае успеха и значение **false** — в случае неудачи, если файл не может быть переименован (если вы пытаетесь переименовать файл, указывая имя существующего файла).

Второй служебный метод **— delete(),** который удаляет дисковый файл, представленный путем вызывающего объекта **класса** File**.**

boolean delete()

Также вы можете использовать **метод delete()** для удаления каталога, если он пуст. Метод delete **()** возвращает значение **true,** если ему удается удалить файл, и значение **false,** если файл не может быть удален. В табл. 2 приведено еще несколько методов класса File**,** которые вы наверняка сочтете полезными.

Таблица 2. Полезные методы класса **File**

Метод	Описание
`void deleteOnExit()`	Удаляет файл, ассоциированный с вызывающим объектом, по завершении работы виртуальной машины Java
`long getFreeSpace()`	Возвращает количество свободных байтов хранилища, доступных в разделе, ассоциированном с вызывающим объектом
`long getTotalSpace()`	Возвращает емкость хранилища раздела, ассоциированного с вызывающим объектом

Окончание табл. 19.1

Метод	Описание
`long getUsableSpace()`	Возвращает количество доступных, годных к употреблению свободных байтов, находящихся в разделе, ассоциированном с вызывающим объектом
`boolean isHidden()`	Возвращает значение `true`, если вызывающий файл является скрытым, и значение `false` — в противном случае
`boolean setLastModified(long` *миллисекунд*`)`	Устанавливает временную метку для вызываемого файла в значение *миллисекунд*, которое представляет количество миллисекунд, прошедших с 1 января 1970 г., по UTC
`boolean setReadOnly()`	Делает вызывающий файл доступным только для чтения

Также существуют методы, помечающие файлы как доступные только для чтения, записи или выполнения.

Поскольку класс File реализует интерфейс Comparable**,** также поддерживается метод compareTo ()**.**

.

Path toPath() – Метод возвращает объект **интерфейса** Path**,** который представляет файл, инкапсулируемый вызываемым объектом класса File**.** (Другими словами, метод toPath **()** преобразует **объект класса** File **в объект интерфейса** Path.)

Path — это интерфейс, добавленный в JDK 7. Он находится в пакете **java.nio.file** и является частью системы NIO. Таким образом, **метод** toPath **()** формирует мост между старым классом **File** и новым интерфейсом **Path.**

1.2.2 Классы FileInputStream и FileOutputStream

Класс FileInputStream

Класс FileInputStream создает объект класса InputStream**,** который вы можете использовать для чтения байтов из файла. Так выглядят два его наиболее часто используемых конструктора.

```
FileInputStream(String путьКфайлу)
FileInputStream(File объектФайла)
```

Каждый из них может передать исключение FileNotFoundException**.** Здесь ***путьКфайлу*** — полное путевое имя файла, а ***объектФайла*** — объект класса File, описывающий файл. В следующем примере создается два объекта класса Fileln-putStream**,** использующих один и тот же дисковый файл и оба эти конструктора.

```
/*создаем объект байтовый файловый поток ввода f0 и привязываем
его к файлу autoexec.bat, расположенному в папке src проекта*/

   FileInputStream f0 = new FileInputStream("src/autoexec.bat");

/*

1) создаем объект-файл f и связываем его с файлом autoexec.bat,
расположенном в папке src проекта

2)создаем объект байтовый файловый поток ввода f1 и привязываем
его ранее созданному объекту файла f

*/
   File f = new File("src/autoexec.bat");
   FileInputStream f1 = new FileInputStream(f);
```

Хотя первый конструктор, вероятно, используется чаще, второй позволяет подробно исследовать файл с помощью методов класса **File,** прежде чем присоединять его к входному потоку. При создании объект класса **FileInputStream** открывается также и для чтения.

Класс **FileInputStream** переопределяет шесть методов абстрактного класса **InputStream.** Методы **mark()** и **reset ()** не переопределяются, и все попытки использовать метод reset**()** с объектом класса **FileInputStream** приводят к передаче исключения IOException**.**

*Пример***. Чтение байтов данных из файла посредством FileInputStream**

Следующий пример показывает, как прочесть один байт, массив байтов и диапазон из массива байтов. Также он иллюстрирует использование метода **available** () для определения оставшегося количества байтов, а также метода **skip () —** для пропуска нежелательных байтов. *Программа читает свой собственный исходный файл*, который должен присутствовать в текущем каталоге. Обратите внимание на то, что здесь

используется оператор **try-с-*ресурсами*** для автоматического закрытия файла, когда он больше не нужен.

```
// Демонстрация применения FileInputStream.
// Эта программа использует оператор try-с-ресурсами. Требует
JDK 7 и выше.

package temp;

import java.io.*;

public class FileInputStreamDemo {
  public static void main(String args[]) {
    int size;
// stream is closing with try-with-resources.
    try ( FileInputStream f =new
FileInputStream("src/temp/FileInputStreamDemo.java")) {
      System.out.println("Total Available Bytes: " + (size =
f.available()));
      int n = size/40;
      System.out.println("First " + n +" bytes of the file one
read() at a time");
      for (int i=0; i < n; i++ ) {
        System.out.print((char) f.read());
      }
      System.out.println("\n Still Available: " + f.available());
      System.out.println("Reading the next " + n +" with one
read(b[])");
      byte b[] = new byte[n];
      if (f.read(b) != n) {
        System.err.println("couldn't read " + n + " bytes.");
      }
      System.out.println(new String(b, 0, n));
      System.out.println("\n Still Available: " + (size =
f.available()));
      System.out.println("Skipping half of remaining bytes with
skip()");
      f.skip(size/2);
      System.out.println("Still Available: " + f.available());
      System.out.println("Reading " + n/2 +" into the end of
array");
      if (f.read(b, n/2, n/2) != n/2) {
        System.err.println ("couldn't read " + n/2 + " bytes.");
      }
      System.out.println(new String(b, 0, b.length));
      System.out.println("\n Still Available: " + f.available());
    }
    catch(IOException e) {
      System.out.println("I/O Error: " + e);
    }
  }
}
```

Результат работы программы:

Полужирным выделены сообщения программы, обычным начертанием – вывод прочитанной из файла информации

Total Available Bytes: 1701
First 42 bytes of the file one read() at a time
package temp;

import java.io. *;

pub
Still Available: 1659
Reading the next 42 with one read(b[])
lic class FileInputStreamDemo {
publi

Still Available: 1617
Skipping half of remaining bytes with skip()
Still Available: 809
Reading 21 into the end of array
lic class FileInputSt;
}

Still Available: 788

Этот несколько надуманный пример демонстрирует чтение тремя способами, пропуск ввода и проверку количества доступных данных в потоке.

Класс FileOutputStream

Класс FileOutputStream создает объект класса OutputStream**,** который вы можете использовать для записи байтов в файл. Он реализует интерфейсы AutoCloseable, Closeable **и** Flushable. Вот четыре его наиболее часто используемых конструктора.

```
FileOutputStream(String путькфайлу)
FileOutputStream(File объектФайла)
FileOutputStream(String путькфайлу, boolean добавить)
FileOutputStreamfFile объектФайла, boolean добавить)
```

Они могут передать исключение FileNotFoundException**.** Здесь ***путькфайлу***— полное путевое имя файла, а ***объектФайла*** — объект класса **File,** описывающий файл. Если параметр *добавить* содержит значение **true,** файл открывается в режиме добавления.

Создание объекта класса FileOutputStream не зависит от того, существует ли указанный файл. Класс FileOutputStream создает его перед открытием, когда вы создаете объект. В случае попытки открытия файла, доступного только для чтения, будет передано исключение.

***Пример*. Запись байтов данных в файл посредством FileOutputStream**

В следующем примере создается буфер байтов. Сначала создается объект класса **String,** а затем используется метод getBytes**()** для извлечения его эквивалента в виде байтового массива. Затем создается три файла. Первый — **file1.txt -—** будет содержать каждый второй байт примера. Второй **-file2.txt-—** полный набор байтов. Третий — **file3.txt** — будет содержать только последнюю четверть.

```
package temp;

// Демонстрация применения FileOutputStream.
// Эта программа использует традиционный подход закрытия файла.
import java.io. *;
class FileOutputStreamDemo {
    public static void main(String args[]) {
        String source = "Now is the time for all good men\n" +
                " to come to the aid of their country\n" +
                " and pay their due taxes.";
        byte buf[] = source.getBytes();
        FileOutputStream f0 = null;
        FileOutputStream f1 = null;
        FileOutputStream f2 = null;
        try {
            f0 = new FileOutputStream("file1.txt");
            f1 = new FileOutputStream("file2.txt");
            f2 = new FileOutputStream("file3.txt");
// запись в первый файл по отдельным байтам (байты с номерами 0, 2,
4 и т.д.)
            for (int i = 0; i < buf.length; i += 2)
               f0.write (buf [i] );

// запись во второй файл содержимого буфера
            f1.write(buf);
// запись в третий файл последней четверти содержимого буфера
            f2.write(buf, buf.length-buf.length/4, buf.length/4);
        }
        catch(IOException e) {
            System.out.println("An I/O Error Occurred");
        }
        finally {
            try {
                if (f0 != null) f0.close();
            } catch(IOException e) {
                System.out.println("Error Closing filel.txt");
            }
            try {
                if(f1 != null) f1.close();
            } catch (IOException e) {
                System.out.println("Error Closing file2.txt");
            }
            try {
                if (f2 != null) f2.close() ;
```

```
            } catch(IOException e) {
                System.out.println("Error Closing file3.txt");
            }
        }
    }
}
```

Так будет выглядеть содержимое каждого файла после выполнения этой программы. Сначала будет представлен файл **file1.txt.**

```
Nwi h iefralgo e
t oet h i ftercuty n a hi u ae.
```

Затем файл **file2.txt**.

```
Now is the time for all good men
to come to the aid o£ their country
and pay their due taxes.
```

И наконец, файл **file3.txt.**

```
nd pay their due taxes.
```

Как видно из комментариев вверху, приведенная выше программа демонстрирует пример использования традиционного подхода закрытия файла, когда он больше не нужен. Этот подход применялся всеми версиями Java до JDK 7 и широко используется в устаревшем коде. Как можно заметить, здесь есть немного довольно неуклюжего кода, требуемого для явного вызова метода close**(),** поскольку каждый его вызов может передать исключение IOException**,** если операция закрытия потерпит неудачу.

Работать с байтовыми потоками не очень удобно. Необходимо считывать информацию в виде набора байт, а после преобразовывать байты в нужный тип. В Java имеются, так называемые, класс-фильтры DataInputStream (для чтения) и DataOutputStream (для записи), которые имеют методы для чтения и записи данных в нужном типе. Для использования достаточно упаковать в них исходные потоки.

Пример. Чтение и запись данных примитивных типов в файл

В файл file1.dat записать 10 целых случайных чисел из диапазона (-500; 500). Переписать в новый файл file2.dat все нечетные числа исходного файла.

В следующем примере создается поток для работы с примитивными типами данных. Байты считываются (записываются) из файла и преобразуются к нужному типу посредством вызова соответствующих методов. Сначала данные записываются в файл **file1.dat**, посредством класса DataOutputStream (для записи целых чисел вызывается метод writeInt(). Затем нечетные числа из файла file1.dat записываются в файл **-file2.dat.**

//Замечание. В примере создаются двоичные файлы. Поэтому просмотр их содержания в текстовых редакторах невозможен.

```
//запись данных в file1.dat
try ( FileOutputStream fout = new FileOutputStream("file1.dat"));
      DataOutputStream dos =  new DataOutputStream (fout)
    )
{
    System.out.println("File content: ");
    for (int i = 0; i < 10; i++) {
        int x = (new java.util.Random()).nextInt(1000) - 500;
        System.out.print(x + " ");
        dos.writeInt(x); //Правильная запись числа типа int в файл

//fout.write(x); /* если записывать в поток fout напрямую, то в файл
будет записано число x, обрезанное до 1 байта */
    }
    System.out.println();
} catch (IOException e) {
    System.out.println("Error - " + e);
}

//чтение информации из файла file1.dat и запись четных чисел в
file2.dat

try (DataInputStream dis  = new DataInputStream(new
FileInputStream("file1.dat"));
   DataOutputStream dos = new DataOutputStream(new
FileOutputStream("file2.dat"));
) {
    while (dis.available()>0) {
      int x = dis.readInt();
      if (x % 2 == 0)
          dos.writeInt(x);
    }
} catch (IOException e) {
    System.out.println("Error - " + e);
}
```

1.2.3 Классы FileReader и FileWriter

Класс FileReader

Класс FileReader создает объект класса Reader, который вы можете использовать для чтения символов из файла. Так выглядят два его наиболее часто используемых конструктора.

```
FileReader(String путьКфайлу)
```

```
FileReader(File объектФайла)
```

Каждый из них может передать исключение FileNotFoundException**.** Здесь ***путьКфайлу*** — полное путевое имя файла, а ***объектФайла*** — объект класса File, описывающий файл. В следующем примере создается два объекта класса FileReader**,** использующих один и тот же дисковый файл и оба эти конструктора.

```
/*создаем объект символьный файловый поток ввода f0 и привязываем
его к файлу autoexec.bat, расположенному в папке src проекта*/

   FileReader f0 = new FileReader("src/autoexec.bat");

/*

1) создаем объект-файл f и связываем его с файлом autoexec.bat,
расположенном в папке src проекта

2)создаем объект символьный файловый поток ввода f1 и привязываем
его ранее созданному объекту файла f

*/
   File f = new File("src/autoexec.bat");
   FileReader f1 = new FileReader(f);
```

Хотя первый конструктор, вероятно, используется чаще, второй позволяет подробно исследовать файл с помощью методов класса **File,** прежде чем присоединять его к входному потоку. При создании объект класса **FileReader** открывается также и для чтения.

Класс **FileReader** переопределяет методы абстрактного класса **Reader**. Для ускорения выполнения операций чтения и записи используется буферизованный ввод/вывод. Для этого исходный поток помещается во внутрь буферизованного потока.

Пример. Распечатка текстового файла

Следующий пример показывает, как прочитать и вывести на экран содержимое текстового файла:

```
//способ 1
/*создаем файловый поток, упаковываем его в поток BufferedReader для
ускорения доступа. Теперь для чтения из файла можно применять методы
класса BufferedReader
*/
    try (FileReader fileReader = new FileReader("file1.txt");
    BufferedReader br = new BufferedReader(fileReader)){
        while (br.ready()){ // пока есть непрочитанные данные
            String st = br.readLine();//считываем строку
            System.out.println(st); // выводим на экран
        }
```

```
    }catch (IOException e){
        System.out.println("Error - "+e);
    }
}
```

Чтение из файла с помощью класса Scanner

Чтение данных посредством класса Scanner значительно облегчает работу с данными. В классе Scanner есть как методы для чтения строк, так и методы для чтения данных примитивных типов. Благодаря чему, не требуются дополнительные операции по преобразованию информации из строки в другие типы.

***Пример*. Чтение из текстового файла посредством Scanner**

В текстовом файле записаны целые числа. Требуется вывести на экран четные числа файла.

Сравните:

```
try (FileReader fileReader = new
FileReader("fileNumbers.txt");
 BufferedReader br = new
BufferedReader(fileReader)) {
  while (br.ready()) {
   String st = br.readLine();
   String n[] = st.split(" ");
   for (String x : n) {
    if (Integer.parseInt(x)% 2==0)
    System.out.print(x + " ");
   }
  }
} catch (IOException e) {
  System.out.printf("Error-%s",e);
}
```

```
File file =
 new File("fileNumbers.txt");

try (Scanner sc = new
Scanner(file)) {
   while (sc.hasNextInt()) {
     int x = sc.nextInt();
     if (x % 2 == 0)
     System.out.print(x +" ");
   }
} catch (IOException e) {
   System.out.println("Error:
"+ e);
}
```

Класс FileWriter

Класс FileWriter создает объект класса Writer**,** который вы можете использовать для записи символов в файл. Так выглядят два его наиболее часто используемых конструктора.

```
FileWriter(String путьКфайлу)
FileWriter(File объектФайла)
```

Каждый из них может передать исключение FileNotFoundException**.** Здесь ***путьКфайлу*** — полное путевое имя файла, а ***объектФайла*** — объект класса File, описывающий файл.

Пример. Работа с текстовыми файлами

В текстовом файле file1.txt записана информация о книгах в следующем формате:

<Автор>;<Название>;<Год>

Примерное содержание файла

```
Пушкин;Барышня-крестьянка;2005
Лермонтов;Мцыри;1987
Гоголь;Мертвые души;2018
Пушкин;Полтава;2020
Пушкин;Евгений Онегин;1999
```

Переписать в файл file2.txt книги Пушкина в том же формате.

```
File file1 = new File("file1.txt");
//Если файл file1.txt не существует, то создать его и записать в
него данные
if (!file1.exists()) {
    try (FileWriter fw1 = new FileWriter(file1);
         ) {
        Scanner sc = new Scanner(System.in);
        for (int i=0; i<5; i++){
            System.out.print("Author:   ");
             String author = sc.nextLine();
            System.out.print("Title :   ");
             String title = sc.nextLine();
            System.out.print("Year  :   ");
             String year = sc.nextLine();
            fw1.write(author+" "+ title+" "+year+"\n");
        }
    } catch (IOException e) {
        System.out.println("Something wrong with filework");
    }
}

//Создаем поток для чтения из file1.txt и поток для записи в файл
file2.txt

File file2 = new File("file2.txt");
try(BufferedReader br = new BufferedReader(new FileReader(file1));
    FileWriter fw2 = new FileWriter(file2);){

    /*1*/ String auth =new String("Пушкин".getBytes("cp1251"),"utf-
8");

    /*2*/ //String auth1 = "Пушкин";

    System.out.println("Введите фамилию искомого автора: ");
    auth = (new Scanner(System.in, "utf-8")).nextLine();
```

```
    while (br.ready()) {
        String st = br.readLine(); //считываем одну строку из файла
        String data[] = st.split(";"); // разбиваем по полям
        if (data[0].equals(auth)) {// если поле Автор совпадает с
auth
            //формируем строку для записи в file2.txt
            st = data[0] + " " + data[1] + " " + data[2] + "\n";
            fw2.write(st);//запись в файл file2.txt
        }
    }

}catch (IOException e) {
    System.out.println("Something wrong with filework");
}
```

Замечание:

По умолчанию кодировка для русских символов в редакторе кода совпадает с системной кодировкой. Т.е. для ОС Windows - по умолчанию - windows-1251. Кодировка файлов, создаваемых программой на Java по умолчанию - utf-8.

Если не изменить кодировку текста, данные не будут корректно отображаться.

Так в примере выше описаны две строки (auth и auth1), в которые записана фамилия "Пушкин". В тексте кода строковым переменным присваиваются одинаковые строки ("Пушкин"). Но после запуска программы, они имеют следующее содержимое:

1) строка auth имеет правильное значение **"Пушкин"**, т.к. ее кодировка преобразована в utf-8 (кодировка по умолчанию)

2) строка auth1 содержит текст **"РџСѓС€РєРёPS"**.

Очевидно, что при попытке сравнить авторов со строкой auth1, результат будет false. Т.е. программа не найдет книги Пушкина, хотя они в исходном файле имеются!!!

1.3. Сериализация

Сериализация — это процесс записи состояния объекта в байтовый поток. Она удобна, когда нужно сохранить состояние вашей программы в области постоянного хранения, такой как файл. Позднее вы можете восстановить эти объекты, используя процесс десериализации.

Сериализация также необходима в реализации *дистанционного вызова методов* **(Remote Method Invocation — RMI).**

RMI позволяет объекту Java на одной машине обращаться к методу объекта Java на другой машине. Объект может быть применен как аргумент этого дистанционного метода. Посылающая машина сериализует объект и передает его. Принимающая машина десериализует его.

Предположим, что объект, подлежащий сериализации, ссылается на другие объекты, которые, в свою очередь, имеют ссылки на еще какие-то объекты. Такой набор объектов и отношений между ними формирует ориентированный граф. В этом графе могут присутствовать и циклические ссылки. Иными словами, объект X может содержать ссылку на объект Y, а объект Y — обратную ссылку на X. Объекты также могут содержать ссылки на самих себя.

Средства сериализации и десериализации объектов устроены так, что могут корректно работать во всех этих сценариях. Если вы попытаетесь сериализовать объект, находящийся на вершине такого графа объектов, то все прочие объекты, на которые имеются ссылки, также будут рекурсивно найдены и сериализованы. Аналогично во время процесса десериализации все эти объекты и их ссылки корректно восстанавливаются. Ниже приведен обзор интерфейсов и классов, поддерживающих сериализацию.

Интерфейс Serializable

Только объект, реализующий интерфейс Serializable**,** может быть сохранен и восстановлен средствами сериализации. Интерфейс Serializabl**e** не содержит никаких членов. Он просто используется для того, чтобы указать, что класс может быть сериализован. Если класс является сериализуемым, все его подклассы также сериализуемы.

Переменные, объявленные как **transient,** не сохраняются средствами сериализации. Не сохраняются также статические переменные.

Интерфейс Externalizable

Средства Java для сериализации и десериализации спроектированы так, что большая часть работы по сохранению и восстановлению состояния объекта выполняется автоматически. Однако бывают случаи, когда программисту нужно управлять этим процессом. Например, может оказаться желательным

использовать технологии сжатия и шифрования. Интерфейс Externalizable предназначен именно для таких ситуаций.

Интерфейс Externalizable определяет следующие два метода.

void readExternal(ObjectInput *входнойПоток)*
throws IOException, ClassNotFoundException void
writeExternal(ObjectOutput *выходнойПоток)*
throws IOException

В этих методах ***входнойПоток*** — это байтовый поток, из которого объект может быть прочитан, а ***выходнойПоток*** — байтовый поток, куда он записывается.

Интерфейс ObjectOutput

Интерфейс ObjectOutput расширяет интерфейсы AutoCloseable и DataOutput**,** поддерживает сериализацию объектов. Он определяет методы, показанные в табл. 2. Особо отметим метод writeObject **().** Он вызывается для сериализации объекта. В случае ошибок все методы этого интерфейса передают исключение IOException**.**

Таблица 2. Методы, определенные в интерфейсе **ObjectOutput**

Метод	Описание
`void close()`	Закрывает вызывающий поток. Последующие попытки записи передадут исключение `IOException`
`void flush()`	Финализирует выходное состояние, чтобы очистить все буферы. То есть все выходные буферы сбрасываются

Окончание табл. 19.7

Метод	Описание
`void write(byte буфер[])`	Записывает массив байтов в вызывающий поток
`void write(byte буфер[], int смещение, int колБайтов)`	Записывает диапазон *колБайтов* байт из массива *буфер* начиная с *буфер[смещение]*
`void write(int b)`	Записывает одиночный байт в вызывающий поток. Из аргумента *b* записывается только младший байт
`void writeObject(Object объект)`	Записывает объект *объект* в вызывающий поток

Класс ObjectOutputStream

Класс ObjectOutputStream расширяет класс OutputStream и реализует интерфейс ObjectOutput**.** Этот класс отвечает за запись объекта в поток. Конструктор его выглядит так.

ObjectOutputStream(OutputStream *выходнойПоток)* throws IOException

Аргумент ***выходнойПоток*** представляет собой выходной поток, в который могут быть записаны сериализуемые объекты. Закрытие объекта класса ObjectOutputStream приводит к закрытию также внутреннего потока, определенного аргументом *выходнойПоток.*

Интерфейс ObjectInput

Интерфейс ObjectInput расширяет интерфейсы AutoCloseable и DataInput и определяет методы, перечисленные в табл. 4. Он поддерживает сериализацию объектов. Особо стоит отметить метод readObject **().** Он вызывается для десериализации объекта. В случае ошибок все эти методы передают исключение IOException**.** Метод readObject **()** также может передать исключение ClassNotFoundException**.**

Класс ObjectInputStream

Этот класс расширяет класс InputStream реализует интерфейс ObjectInput. Класс ObjectInputStream отвечает за чтение объектов из потока. Ниже показан конструктор этого класса.

ObjectlnputStream(lnputStream *входнойПоток)* throws IOException

Аргумент ***входнойПоток*** — это входной поток, из которого должен быть прочитан сериализованный объект. Закрытие объекта класса ObjectlnputStream приводит к закрытию также внутреннего потока, определенного аргументом ***входнойПоток****.*

Пример сериализации

В следующей программе показано, как использовать сериализацию и десериализацию объектов. Начинается она с создания экземпляра объекта

MyClass. Этот объект имеет три переменные экземпляра типа String, int и double. Именно эту информацию мы хотим сохранять и восстанавливать.

В программе создается объект класса FileOutputStream, который ссылается на файл по имени " **serial**", и для этого файлового потока создается объект класса ObjectOutputStream. Метод writeObject() этого объекта класса ObjectOutputStream используется затем для сериализации объекта. Объект выходного потока очищается и закрывается.

Далее создается объект класса FileInputStream, который ссылается на файл по имени " **serial**", и для этого файлового потока создается объект класса ObjectInputStream. Метод **readObject** () класса ObjectInputStream используется для последующей десериализации объекта. После этого входной поток закрывается.

Обратите внимание на то, что объект MyClass определен с реализацией интерфейса Serializable. Если бы этого не было, передалось бы исключение NotSerializableException. Поэкспериментируйте с этой программой, объявляя некоторые переменные экземпляра MyClass как transient. Эти данные не будут сохраняться при сериализации.

```
// Демонстрация сериализации
// Эта программа использует оператор try-с-ресурсами. Требует JDK 7
и выше.
//import java.io.*;
class SerializationDemo {
    public static void main(String args[]) {
        // Сериализации объекта
        try ( ObjectOutputStream objOStrm =
            new ObjectOutputStream(new FileOutputStream("serial"))
            )
        {
            MyClass object1 = new MyClass("Hello", -7, 2.7e10);
            System.out.println("object1:   " + object1);
            objOStrm.writeObject(object1);
        } catch (IOException e) {
            System.out.println("Исключение во время сериализации  :
" + e);
        }
// Десериализация объекта
        try (
          ObjectInputStream objIStrm = new ObjectInputStream(
                  new FileInputStream("serial"))
            )
        {
            MyClass object2 = (MyClass) objIStrm.readObject();
            System.out.println("object2:   " + object2);
        } catch (Exception e) {
```

```
            System.out.println("Исключение во время сериализации:  "
+ e);
            System.exit(0);
        }
    }
}


class MyClass implements Serializable {
    String s;
    int i;
    double d;

    public MyClass(String s, int i, double d) {
        this.s = s;
        this.i = i;
        this.d = d;
    }

    public String toString() {
        return "s=" + s + ";   i=" + i + " ;   d=" + d;
    }
}
```

Эта программа демонстрирует идентичность переменных экземпляра объектов **object**1 и **object2.** Вот ее вывод.

object1: s=Hello; i=-7; d=2.7E10

object2: s=Hello; i=-7; d=2.7E10

1.4. Контрольный пример по сериализации объектов

Имеется 3 класса:

1) **Pupil** – содержит информацию об учениках и методы для работы с ними. Помимо базовых методов (конструкторов, геттеров, сеттеров и toString()), присутствуют следующие методы:
   - save() и load() – ничего не делают, но пробрасывают исключения выше.
   - compareTo() – метод для сравнения учеников (по фамилиям)
2) **Pupils** – класс содержит список учеников (item) и методы для работы с этим списком
   - getItem() – метод возвращает список
   - setItem() – метод позволяет записывать список
   - add() – добавляет нового ученика в список
   - show() – метод для распечатки содержимого списка
3) **TestPupils** – класс содержит меню для работы с программой и методы для сериализации и десериализации объектов

- Saving() – метод для сохранения списка учеников в файл (сериализация)
- Loading() – метод для загрузки списка учеников из файла (десериализация)

Класс Pupil

```
import java.io.*;
import java.util.Random;
import java.util.Scanner;

public class Pupil implements Serializable, Comparable<Pupil> {
    private String firstName;
    private String lastName;
    private int age;
    private Scanner sc = new Scanner(System.in);

    public void save(DataOutputStream out) throws IOException {  }
    public Pupil load(DataInputStream in) throws IOException{ return
null;  }

    @Override
    public int compareTo(Pupil pupil) {
        return this.firstName.compareTo(pupil.firstName);
    }

    public Pupil(String firstName, String lastName, int age) {
        this.firstName = firstName; this.lastName = lastName;
this.age = age;
    }

    public Pupil(){
        this.firstName = sc.nextLine();
        this.lastName = sc.nextLine();
        this.age = sc.nextInt();
    }

    public String getFirstName() { return firstName;   }

    public void setFirstName(String firstName) {  this.firstName =
firstName; }

    public String getLastName() {  return lastName;   }

    public void setLastName(String lastName) {  this.lastName =
lastName;  }

    @Override
    public String toString() {
        return "Pupil{" + "firstName='" + firstName + '\'' +
                ", lastName='" + lastName + '\'' + ", age=" + age +
'}';
```

```
    }

    public int getAge() {  return age; }

    public void setAge(int age) { this.age = age; }
}
```

Класс Pupils

```
import java.io.*;
import java.util.*;

public class Pupils implements Serializable {
    private ArrayList<Pupil> item = new ArrayList<>();
    private Comparator<Pupil> comparator = null;

    public ArrayList<Pupil> getItem() {  return item;  }

    public void setItem(ArrayList<Pupil> item) {this.item = item; }

    public void add(Pupil pupil){
        item.add(pupil);
    }

    public void show(){
        if (comparator!=null)   Collections.sort(item, comparator);
        for(Pupil pupil : item){
            System.out.println(pupil);
        }
    }
}
```

Класс TestPupils

```
import java.io.*;
import java.util.ArrayList;
import java.util.Scanner;

public class TestPupil {
    public static void Saving(Pupils pupils){

        try(ObjectOutputStream out1 = new ObjectOutputStream(new
BufferedOutputStream( new FileOutputStream("test1.txt", false)));)
        {
            try {
               out1.writeObject(pupils.getItem());
            }catch (IOException e){
                System.out.println("Maybe disk is full, or you
haven't permissions to write");
            }
        } catch (FileNotFoundException e) {
```

```
            System.out.println("Отсутствует файл.");
        }catch (IOException e1) {
            e1.printStackTrace();
        }
    }

    public static Pupils Loading(Pupils pupils){
        try (ObjectInputStream in1 = new ObjectInputStream(new
BufferedInputStream(new FileInputStream(new File("test2.txt"))));
){
          try{
              pupils.setItem((ArrayList<Pupil>) in1.readObject());
          } catch (IOException e) {
              System.out.println(e.getMessage());
              e.printStackTrace();
              System.out.println("Maybe you haven't permissions to
read");
          } catch (ClassNotFoundException e){
              e.printStackTrace();
          }
        } catch (FileNotFoundException e){
            System.out.println("Отсутствует файл");
        }catch (IOException e1){
            e1.printStackTrace();
        }
        return pupils;
    }

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        Pupils pupils = new Pupils();
        while (true){
        System.out.println("" +
                "1. Save pupils to file\n" +
                "2. Load pupils from file\n" +
                "3. Add new pupil\n" +
                "4. Print list \n " +
                "5. Exit");

           int choice = sc.nextInt();
           switch (choice){
               case 1: Saving(pupils);break;
               case 2: {
                   pupils.getItem().clear();
                   pupils = Loading(pupils);
                   pupils.show();
                   break;
               }
               case 3:
                   System.out.println("How many pupils to add: ");
                   int n = sc.nextInt();
                   for (int i = 0; i < n; i++) {
                       pupils.add(new Pupil());
```

```
                    }
                    pupils.show();
                     break;
                case 4: pupils.show();break;
                default: System.exit(0);
            }
        }
    }
}
```

Содержание отчета

Отчет о выполнении лабораторной работы должен включать:

1. Задание на работу
2. Листинг выполнения программы, отражающий все этапы ее выполнения. (с комментариями)
3. Скриншоты работы программ и содержимого файлов.
4. Ответы на контрольные вопросы

Варианты заданий для самостоятельной работы

Общее требование: во всех вариантах по заданию 1 и 2 требуется предусмотреть методы для создания исходных файлов (и наполнение их информацией).

Задание 1: (работа с байтовыми потоками)

(Номер задачи совпадает с номером варианта)

1. Дан файл *f*, компоненты которого являются целыми числами. Записать в файл *g* все четные числа файла *f*, а в файл *h* - все нечетные. Порядок следования чисел сохранить
2. Дан файл, содержащий вещественные числа. Описать функцию, подсчитывающую сумму отрицательных элементов в этом файле.
3. Даны два файла, содержащих целые числа. Переписать содержимое первого файла во второй без отрицательных чисел.
4. Дан файл f, содержащий вещественные числа. Описать функцию less(f) от непустого файла f, меньших среднего арифметического всех элементов этого файла.
5. Дан файл вещественных чисел f. Определить количество элементов в наиболее длинной возрастающей последовательности файла f.
6. Дан файл f, компоненты которого являются целыми числами. Записать в файл g наибольшее значение первых пяти компонент файла f, затем - следующих пяти компонент и т.д. Если в последней группе окажется менее пяти компонент, то последняя компонента файла g должна быть равна наибольшей из компонент файла f, образующих последнюю (неполную) группу.
7. Дан файл f, компоненты которого являются целыми числами. Получить файл g, образованный из файла f исключением повторных вхождений одного и того же числа.
8. Дан файл f, содержащий целые числа. Создать новый файл g и записать в него все простые числа файла f.

9. Дан файл целых чисел. Требуется переписать его содержимое в новый файл без автоморфных чисел
10. Дан файл целых чисел. Требуется определить есть ли в нем 3 подряд идущих простых числа, после которых следует совершенное число.
11. Дан файл целых чисел. Найти количество элементов файла, расположенных между максимальным и минимальным элементами файла
12. Дан файл целых чисел. Переписать в новый файл четные элементы, расположенные между первым нечетным элементом и предпоследним четным элементов файлв.
13. Дан файл целых чисел. Переписать в новый файл все элементы, кратные минимальному нечетному элементу файла.
14. Дан файл целых чисел. Переписать в новый файл все элементы исходного файла, расположенные до первого простого элемента и после максимального элемента.
15. Даны 2 файла целых чисел. Требуется записать в третий файл попарные суммы элементов. Конечный файл должен содержать число элементов в коротком файле (если исходные файлы имеют различное число элементов)

Задание 2 *(Работа с символьными потоками)*

(Номер задачи совпадает с номером варианта)

1. Дан текстовый файл f. В каждой строке найти позицию самого длинного слово среди слов, вторая буква которых «е»; если слов с наибольшей длиной несколько, то найти последнее. Результат сохранить в отдельный файл. Если таких слов нет вообще, то сообщить об этом.
2. Дан текстовый файл f. Записать "в перевернутом виде" строки файла f в файл g, оставляя только строки, в которых записано не менее двух предложений. Порядок строк в файле g должен совпадать с порядком исходных строк в файле f;
3. Дан текстовый файл f, содержащий сведения о сотрудниках учреждения, записанные по следующему образцу: фамилия имя отчество фамилия имя отчество ... Записать эти сведения в файле g, используя образец: фамилия и. о. фамилия и. о. ...;
4. Дан текстовый файл, в каждой строке котором записана следующая информация о студентах: имя студента, оценка по текущему контролю и оценка по рубежному контролю. Найти лучшего и худшего по успеваемости студентов группы. Определить средний балл по рейтингу в группе. Всех студентов, рейтинг которых выше среднего записать в новый файл.
5. Дан текстовый файл, в каждой строке котором записана следующая информация о студентах: имя студента, количество пропущенных часов, средний балл по итогам рейтингов. Получить новый файл, в котором записаны студенты, имеющие более 20 часов пропущенных занятий. Найти лучшего и худшего по успеваемости студентов группы.
6. Дан текстовый файл *f*, содержащий сведения о веществах: указывается название вещества, его удельный вес и проводимость (проводник, полупроводник, изолятор). Найти удельные веса и названия всех полупроводников. Переписать в отдельный файл вещества, имеющие максимальную проводимость.
7. Дан текстовый файл *f*, содержащий сведения о книгах. Сведения о каждой из книг - это фамилия автора, название и год издания. 1) Найти названия книг заданного автора, изданных с 1960 г. 2) Определить год последнего издания книги с названием "Информатика". Записать в отдельный файл список книг, изданных за последние 5 лет.
8. Дан текстовый файл *f*, содержащий сведения об экспортируемых товарах: указывается наименование товара, страна, импортирующая товар, и объем поставляемой партии в штуках. Найти страны, в которые экспортируется данный товар, и общий объем его экспорта. Найти самый востребованный товар (максимальный суммарный объем партии).

9. Дан текстовый файл. Описать процедуру, которая записывает в новый текстовый файл все цифры из исходного файла (разбиение на строки сохранить). В новом файле определить максимальное число, образованное выписанными цифрами.

10. Пусть текстовый файл t разбит на непустые строки. Описать функции: 1) для подсчета числа строк, которые состоят из одинаковых литер; 2) для подсчета строк, в которых нет гласных; 3) для определения, какая из гласных букв встречается в нем чаще всего.
11. Пусть текстовый файл t разбит на непустые строки. Описать функции: 1) для подсчета числа строк, являющимися предложениями (начинаются с заглавной буквы, и заканчиваются «.», «!» или «?»); 2) для определения строки, в которой наименьшее количество цифр и знаков пунктуации.
12. Описать функцию, которая находит максимальную длину строк текстового файла t. Переписать содержимое файла в другой файл без строк, длина которых совпадает с максимальной. Оставшиеся строки записать в третий файл без гласных букв и знаков препинания.
13. Дан текстовый файл f. Определить, есть ли в файле строки, начинающиеся и заканчивающиеся цифрами. Если да, то определить максимальное из чисел, которые можно составить из этих цифр (отдельно для каждой строки).
14. Дан текстовый файл g. Создать новый файл f и записать в него все слова файла g с количеством символов >4 и < 10, в которых нет повторяющихся символов. Если таких слов нет, то полностью переписать файл g в файл f.
15. Дан текстовый файл f. Создать новый файл g и записать в него построчно все слова файла f, предварительно перенеся последнюю букву слова в начало.

Задание 3 *(Сериализация и десериализация объектов)*

Для всех вариантов. Организовать сохранение объектов из коллекции в текстовый файл и загрузку объектов из текстового файла в коллекцию. Изменить пользовательское меню для добавления работы с внешними файлами.

Контрольные вопросы

1. Что такое потоки ввода-вывода и для чего они нужны?
2. Байтовые потоки. Базовые абстрактные классы байтовых потоков.
3. Символьные потоки. Базовые абстрактные классы символьных потоков.
4. InputStreamReader и OutputStreamWriter.
5. Как получить свойства файла? Какие свойства файла можно узнать?
6. Какие стандартные потоки ввода-вывода существуют в Java, каково их назначение? На базе каких классов создаются стандартные потоки?
7. Как создать файловый поток для чтения и записи данных?
8. За счет чего буферизация ускоряет работу приложений с потоками ввода/вывода? Как определить сделать поток буферизированным?
9. Когда применяется принудительный сброс буферов?
10. Для выполнения каких операций применяется класс File?
11. Для чего предназначен класс RandomAccessFile? Чем он отличается от потоков ввода и вывода?
12. Какие средства позиционирования могут использоваться при прямом доступе к содержимому файла
13. Для чего используются потоки DataOutputStream и DataInputStream?
14. Что такое сериализация объектов? Что такое десериализация объектов?
15. Как объявить класс сериализуемым?

16. Какие поля класса не сериализуются?
17. Сериализация объектов. Подготовка классов к сериализации.