Глава 9

ПОТОКИ ВВОДА/ВЫВОДА

Вопрос. Я скачал из интернета файл. Теперь мне он больше не нужен. Как закачать его обратно?

Ответ. Вот из-за таких, как ты, скоро в интернете все файлы кончатся.

Цитата из Интернет-форума

Потоки ввода/вывода используются для передачи данных в файловые потоки, на консоль или на сетевые соединения. Потоки представляют собой объекты соответствующих классов. Пакеты ввода/вывода: **java.io, java.nio** предоставляют пользователю большое число классов и методов и постоянно обновляются.

Байтовые и символьные потоки ввода/вывода

При разработке приложения регулярно возникает необходимость извлечения информации из какого-либо источника и хранения результатов. Действия по чтению/записи информации представляют собой стандартный и простой вид деятельности. Самые первые классы ввода/вывода связаны с передачей и извлечением последовательности байтов из потоков.

Все потоки ввода последовательности байтов являются подклассами абстрактного класса **InputStream**, потоки вывода — подклассами абстрактного класса **OutputStream**. При работе с файлами используются подклассы этих классов, соответственно, **FileInputStream** и **FileOutputStream**, конструкторы которых открывают поток и связывают его с соответствующим физическим файлом. Существуют классы-потоки для ввода массивов байтов, строк, объектов, а также для выбора из файлов и сетевых соединений.

Для чтения байта или массива байтов используются реализации абстрактных методов **int read()** и **int read(byte[] b)** класса **InputStream**. Метод **int read()** возвращает **–1**, если достигнут конец потока данных, поэтому возвращаемое значение имеет тип **int**, а не **byte**. При взаимодействии с информационными потоками возможны различные исключительные ситуации, поэтому обработка исключений вида **try-catch** при использовании методов чтения и записи является обязательной.

В классе **FileInputStream** метод **read()** читает один байт из файла, а поток **System.in** как объект подкласса **InputStream** позволяет вводить байт с консоли.

Реализация абстрактного метода **write(int b)** класса **OutputStream** записывает один байт в поток вывода. Оба эти метода блокируют поток до тех пор, пока байт не будет записан или прочитан. После окончания чтения или записи в поток его всегда следует закрывать с помощью метода **close()**, для того, чтобы освободить ресурсы приложения. Класс **FileOutputStream** используется для вывода одного или нескольких байт информации в файл.

Поток ввода связывается с одним из источников данных, в качестве которых могут быть использованы массив байтов, строка, файл, «pipe»-канал, сетевые соединения и др. Набор классов для взаимодействия с перечисленными источниками приведен на рисунках 9.1 и 9.2.

Класс **DataInputStream** предоставляет методы для чтения из потока данных значений базовых типов, но не рекомендуется к использованию. Класс **BufferedInputStream** присоединяет к потоку буфер для упрощения и ускорения следующего доступа.

Для вывода данных в поток используются следующие классы.

Абстрактный класс **FilterOutputStream** используется как шаблон для настройки производных классов. Класс **BufferedOutputStream** присоединяет буфер к потоку для ускорения вывода и ограничения доступа к внешним устройствам.

Начиная с версии 1.2, пакет **java.io** подвергся значительным изменениям. Появились новые классы, которые производят скоростную обработку потоков, хотя и не полностью перекрывают возможности классов предыдущей версии.

Для обработки символьных потоков в формате Unicode применяется отдельная иерархия подклассов абстрактных классов **Reader** и **Writer**, которые почти полностью повторяют функциональность байтовых потоков, но являются более

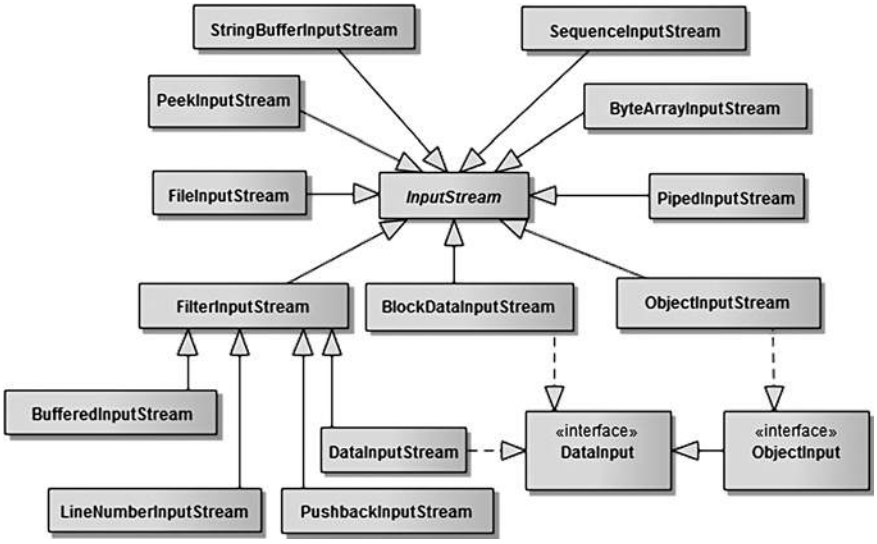

Рис. 9.1. *Иерархия классов байтовых потоков ввода*

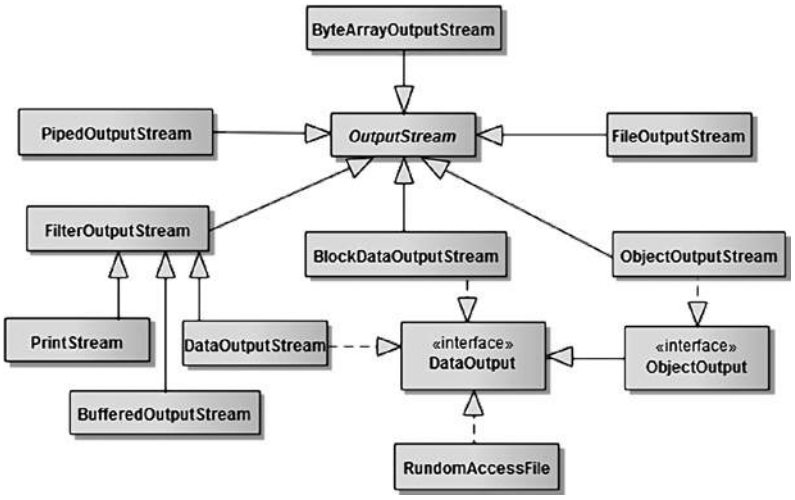

Рис. 9.2. *Иерархия классов байтовых потоков вывода*

актуальными при передаче текстовой информации. Например, аналогом класса **FileInputStream** является класс **FileReader**. Такой широкий выбор потоков позволяет выбрать наилучший способ записи в каждом конкретном случае.

В примерах по возможности используются способы инициализации для различных семейств потоков ввода/вывода.

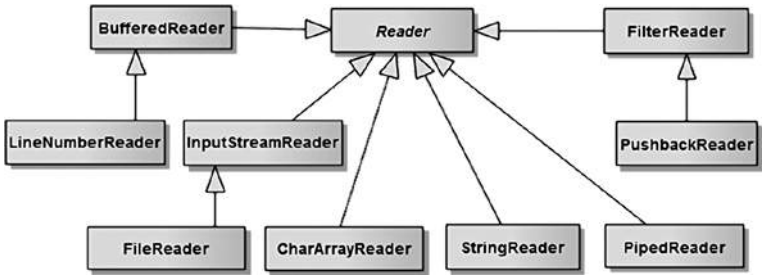

Рис. 9.3. *Иерархия символьных потоков ввода*

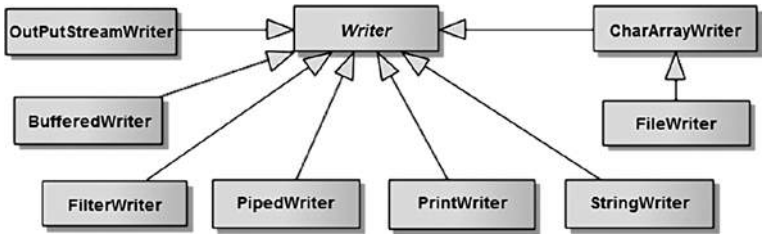

Рис. 9.4. *Иерархия символьных потоков вывода*

/ # 1 # чтение по одному символу (байту) из потока ввода # ReadDemo.java */*

```
package by.bsu.io;
import java.io.File;
import java.io.FileReader;
import java.io.IOException;
public class ReadDemo {
        public static void main(String[] args) {
                String file = "data\\file.txt";
                File f = new File(file); // объект для связи с файлом на диске
                int b, count = 0;
                FileReader is = null;
                // FileInputStream is = null; // альтернатива
                try {
                        is = new FileReader(f);
                        // is = new FileInputStream(f);
                        while ((b = is.read()) != -1) { // чтение
                                System.out.print((char) b);
                                count++;
                        }
                        System.out.print("\n число байт = " + count);
                } catch (IOException e) {
                        System.err.println("Ошибка файла: " + e );
                } finally {
                        try {
                                if (is != null) {
                                        is.close(); // закрытие потока ввода
                                }
                        } catch (IOException e) {
                                System.err.println("ошибка закрытия: " + e);
                        }
                }
        }
}
```

Один из конструкторов **FileReader(file)** или **FileInputStream(file)** открывает поток **is** и связывает его с файлом **file**, где директория **data** в корне проекта должна существовать.

Если файла по указанному пути не существует, то при попытке инициализации потока с несуществующим файлом будет сгенерировано исключение:

Ошибка файла: java.io.FileNotFoundException: data\file.txt (The system cannot find the file specified)

Если файл существует, то информация из него будет считана по одному символу; и результаты чтения, и количество прочитанных символов будут выведены на консоль.

Для закрытия потока используется метод **close()**. Закрытие потока должно произойти при любом исходе чтения: удачном или с генерацией исключения. Гарантировать закрытие потока может только помещение метода **close()** в блок

finally. При чтении из потока можно пропустить **n** байт с помощью метода **long skip(long n)**.

Закрытие потока ввода/вывода в блоке **finally** принято как негласное соглашение и является признаком хорошего кода. Однако выглядит достаточно громоздко. В Java 7 возможно автоматическое закрытие потоков ввода/вывода без явного вызова метода **close()** и блока **finally**:

```
try (FileReader is = new FileReader(new File("data\\file.txt"))) {
          int b = 0;
          while ((b = is.read()) != -1) { /* чтение */
                    System.out.print((char)b)
          }
} catch (IOException e) {
          System.err.println("ошибка файла: " + e);
}
```

С этой версии в список интерфейсов, реализуемых практически всеми классами потоков, добавлен интерфейс **AutoCloseable**. Метод **close()** вызывается неявно для всех потоков, открытых в инструкции

```
try(Поток1: Поток2:…: ПотокN)
```

Например:

```
String dest1 = "filename1";
String dest2 = "filename2";
   try (Writer writer = new FileWriter(dest1): OutputStream out = new FileOutputStream(dest2)) {
     // using writer & out
   } catch (IOException e) {
   e.printStackTrace();
   }
```

Для вывода символа (байта) или массива символов (байтов) в поток используются потоки вывода — объекты подкласса **FileWriter** суперкласса **Writer** или подкласса **FileOutputStream** суперкласса **OutputStream**. В следующем примере для вывода в связанный с файлом поток используется метод **write()**.

// # 2 # вывод массива в поток в виде символов и байт # WriteRunner.java

```
package by.bsu.io;
import java.io.*;
public class WriteRunner {
          public static void main(String[] args) {
                    String pArray[] = { "2013 ", "Java SE 8" };
                    File fbyte = new File("data\\byte.data");
                    File fsymb = new File("data\\symbol.txt");
                    FileOutputStream fos = null;
                    FileWriter fw = null;
                    try {
```

```
                    fos = new FileOutputStream(fbyte);
                    fw = new FileWriter(fsymb);
                    for (String a : pArray) {
                        fos.write(a.getBytes());
                        fw.write(a);
                    }
                } catch (IOException e) {
                    System.err.println("ошибка записи: " + e);
                } finally {
                    try {
                        if (fos != null) {
                            fos.close();
                        }
                        // ниже некорректно!
                        // каждому close() требуется свой try-catch
                        if (fw != null) {
                            fw.close();
                        }
                    } catch (IOException e) {
                        System.err.println("ошибка закрытия потока: " + e);
                    }
                }
        }
}
```

В результате будут получены два файла с идентичным набором данных, но созданные различными способами.

Класс File

Для работы с физическим файлами и каталогами (директориями), расположенными на внешних носителях, в приложениях Java используются классы из пакета **java.io**.

Класс **File** служит для хранения и обработки в качестве объектов каталогов и имен файлов. Этот класс не содержит методы для работы с содержимым файла, но позволяет манипулировать такими свойствами файла, как право доступа, дата и время создания, путь в иерархии каталогов, создание, удаление файла, изменение его имени и каталога и т. д.

Объект класса **File** создается одним из нижеприведенных способов:

```
File obFile = new File("\\com\\file.txt");
File obDir = new File("c:/jdk/src/java/io");
File obFile1 = new File(obDir, "File.java");
File obFile2 = new File("c:\\com", "file.txt");
File obFile3 = new File(new URI("Интернет-адрес"));
```

В первом случае создается объект, соответствующий файлу, во втором — подкаталогу. Третий и четвертый случаи идентичны. Для создания объекта

указывается каталог и имя файла. В пятом — создается объект, соответствующий адресу в Интернете.

При создании объекта класса **File** любым из конструкторов компилятор не выполняет проверку на существование физического файла с заданным путем.

Существует разница между разделителями, употребляющимися при записи пути к файлу: для системы Unix — «/», а для Windows — «\\». Для случаев, когда неизвестно, в какой системе будет выполняться код, предусмотрены специальные поля в классе **File**:

```
public static final String separator;
public static final char separatorChar;
```

С помощью этих полей можно задать путь, универсальный в любой системе:

```
File file = new File(File.separator + "com" + File.separator + "data.txt" );
```

Также предусмотрен еще один тип разделителей — для директорий:

```
public static final String pathSeparator;
public static final char pathSeparatorChar;
```

К примеру, для ОС Unix значение **pathSeparator** принимает значение «**:**», а для Windows — «**;**».

В классе **File** объявлено более тридцати методов, наиболее используемые из них рассмотрены в следующем примере:

/ # 3 # работа с файловой системой: FileTest.java */*

```
package by.bsu.io;
import java.io.File;
import java.util.Date;
import java.io.IOException;
public class FileTest {
   public static void main(String[] args) {
     // с объектом типа File ассоциируется файл на диске FileTest2.java
     File fp = new File("FileTest2.java");
     if(fp.exists()) {
        System.out.println(fp.getName() + " существует");
          if(fp.isFile()) { // если объект - дисковый файл
             System.out.println("Путь к файлу:\t" + fp.getPath());
             System.out.println("Абсолютный путь:\t" + fp.getAbsolutePath());
             System.out.println("Размер файла:\t" + fp.length());
             System.out.println("Последняя модификация :\t"+ new Date(fp.lastModified()));
             System.out.println("Файл доступен для чтения:\t" + fp.canRead());
             System.out.println("Файл доступен для записи:\t" + fp.canWrite());
             System.out.println("Файл удален:\t"  + fp.delete());
             }
          } else
             System.out.println("файл " + fp.getName() + " не существует");
      try {
```

```
        if(fp.createNewFile()) {
               System.out.println("Файл " + fp.getName() + " создан");
        }
      } catch(IOException e) {
            System.err.println(e);
      }
// в объект типа File помещается каталог\директория
// в корне проекта должен быть создан каталог com.learn с несколькими файлами
   File dir = new File("com" + File.separator + "learn");
   if (dir.exists() && dir.isDirectory()) { /*если объект является каталогом и если этот
                                                        каталог существует */
               System.out.println("каталог " + dir.getName() + " существует");
   }
            File[] files = dir.listFiles();
            for(int i = 0; i < files.length; i++) {
               Date date = new Date(files[i].lastModified());
               System.out.print("\n"+files[i].getPath()+" \t| "+files[i].length()+"\t|"+date);
               // использовать toLocaleString() или toGMTString()
            }
         // метод listRoots() возвращает доступные корневые каталоги
         File root = File.listRoots()[1];
         System.out.printf("\n%s %,d из %,d свободно.", root.getPath(), root.getUsableSpace(),
                                                                  root.getTotalSpace());
   }
}
```

В результате файл **FileTest2.java** будет очищен, а на консоль выведено:

```
FileTest2.java существует
Путь к файлу:                FileTest2.java
Абсолютный путь:             C:\workspace\FileTest2.java
Размер файла:                2091
Последняя модификация:       Fri Nov 21 12:26:50 EEST 2008
Файл доступен для чтения:    true
Файл доступен для записи:    true
Файл удален:                 true
Файл FileTest2.java создан
каталог learn существует
com\learn\bb.txt       | 9      | Fri Mar 24 15:30:33 EET 2006
com\learn\byte.txt| 8    | Thu Jan 26 12:56:46 EET 2006
com\learn\cat.gif | 670  | Tue Feb 03 00:44:44 EET 2004
C:\ 3 665 334 272 из 15 751 376 896 свободно.
```

У каталога как объекта класса **File** есть дополнительное свойство — просмотр списка имен файлов с помощью методов **list()**, **listFiles()**, **listRoots()**.

Предопределенные потоки

Система ввода/вывода языка Java содержит стандартные потоки ввода, вывода и вывода ошибок. Класс **System** пакета **java.lang** содержит поле **in**, которое является ссылкой на объект класса **InputStream**, и поля **out, err** — ссылки на объекты класса **PrintStream**, объявленные со спецификаторами **public static** и являющиеся стандартными потоками ввода, вывода и вывода ошибок соответственно. Эти потоки связаны с консолью, но могут быть переназначены на другое устройство.

Для назначения вывода текстовой информации в произвольный поток следует использовать класс **PrintWriter**, являющийся подклассом абстрактного класса **Writer**.

Для наиболее удобного вывода информации в файл (или в любой другой поток) следует организовать следующую последовательность инициализации потоков с помощью класса **PrintWriter**:

```
new PrintWriter(new BufferedWriter(new FileWriter(new File("text\\data.txt"))));
```

В итоге класс **PrintWriter** выступает классом-оберткой для класса **BufferedWriter**, и далее так же, как и класс **BufferedReader** для **FileReader**. По окончании работы с потоками закрывать следует только самый последний класс. В данном случае — **PrintWriter**. Все остальные, в него обернутые, будут закрыты автоматически.

Приведенный ниже пример демонстрирует вывод в файл строк и чисел с плавающей точкой.

// # 4 # буферизованный вывод в файл # DemoWriter.java

```
package by.bsu.io;
import java.io.*;
public class DemoWriter {
          public static void main(String[] args) {
                    File f = new File("data\\res.txt");
                    double[ ] v = { 1.10, 1.2, 1.401, 5.01, 6.017, 7, 8 };
                    FileWriter fw = null;
                    BufferedWriter bw = null;
                    PrintWriter pw = null;
                    try {
                              fw = new FileWriter(f, true);
                              bw = new BufferedWriter(fw);
                              pw = new PrintWriter(bw);
                              for (double version : v) {
                                        pw.printf("Java %.2g%n", version);// запись прямо в файл
                              }
                    } catch (IOException e) {
                              System.err.println("ошибка открытия потока " + e);
                    } finally {
                              if (pw != null) {
```

```
                                        try { // закрывать нужно только внешний поток
                                                pw.close();
                                        } catch (IOException e) {
                                                e.printStackTrace();
                                        }
                                }
                        }
                }
}
```

В итоге в файл **res.txt** будет помещена следующая информация:

Java 1.1
Java 1.2
Java 1.4
Java 5.0
Java 6.0
Java 7.0
Java 8.0

Для вывода данных в файл в текстовом формате использовался фильтрованный поток вывода **PrintWriter** и метод **printf()**. После соединения этого потока с дисковым файлом посредством символьного потока **BufferedWriter** и удобного средства записи в файл **FileWriter** становится возможной запись текстовой информации с помощью обычных методов **println()**, **print()**, **printf()**, **format()**, **write()**, **append()**.

В отличие от Java 1.1 в языке Java 1.2 для консольного ввода используется не байтовый, а символьный поток. В этой ситуации для ввода используется подкласс **BufferedReader** абстрактного класса **Reader** и методы **read()** и **readLine()** для чтения символа и строки соответственно. Этот поток для организации чтения из файла лучше всего инициализировать объектом класса **FileReader** в виде:

```
new BufferedReader(new FileReader(new File("res.txt")));
```

Чтение из созданного в предыдущем примере файла с использованием удобной технологии можно произвести следующим образом:

// # 5 # чтение из файла # DemoReader.java

```
package by.bsu.io;
import java.io.*;
public class DemoReader {
        public static void main(String[ ] args) {
                BufferedReader br = null;
                try {
                        br = new BufferedReader(new FileReader("data\\res.txt"));
                        String tmp = "";
                        while ((tmp = br.readLine()) != null) {
```

```
                                        // пробел использовать как разделитель
                                        String[] s = tmp.split("\\s");
                                        // вывод полученных строк
                                        for (String res : s) {
                                                System.out.println(res);
                                        }
                                }
                        } catch (IOException e) {
                                e.printStackTrace();
                        } finally {
                                if (br != null) {
                                        try {
                                                br.close();
                                        } catch (IOException e) {
                                                e.printStackTrace();
                                        }
                                }
                        }
                }
}
```

В консоль будет выведено:

Java
1.1
Java
1.2
Java
1.4
Java
5.0
Java
6.0
Java
7.0
Java
8.0

Сериализация объектов

Кроме данных базовых типов, в поток можно отправлять объекты классов целиком для передачи клиентскому приложению или для хранения.

Процесс преобразования объектов в потоки байтов для хранения называется сериализацией. Процесс извлечения объекта из потока байтов называется десериализацией. Существует два способа сделать объект сериализуемым.

Для того, чтобы объекты класса могли быть подвергнуты процессу сериализации, этот класс должен расширять интерфейс **java.io.Serializable**. Все подклассы такого класса также будут сериализованы. Многие стандартные классы реализуют этот интерфейс. Этот процесс заключается в сериализации каждого поля объекта, но только в том случае, если это поле не имеет спецификатора **static** или **transient**. Спецификаторы **transient** и **static** означают, что поля, помеченные ими, не могут быть предметом сериализации, но существует различие в десериализации. Так, поле со спецификатором **transient** после десериализации получает значение по умолчанию, соответствующее его типу (объектный тип всегда инициализируется по умолчанию значением **null**), а поле со спецификатором **static** получает значение по умолчанию в случае отсутствия в области видимости объектов своего типа, а при их наличии получает значение, которое определено для существующего объекта.

Интерфейс **Serializable** не имеет методов, которые необходимо реализовать, поэтому его использование ограничивается упоминанием при объявлении класса. Все действия в дальнейшем производятся по умолчанию. Для записи объектов в поток необходимо использовать класс **ObjectOutputStream**. После этого достаточно вызвать метод **writeObject(Object ob)** этого класса для сериализации объекта **ob** и пересылки его в выходной поток данных. Для чтения используются, соответственно, класс **ObjectInputStream** и его метод **readObject()**, возвращающий ссылку на класс **Object**. После чего следует преобразовать полученный объект к нужному типу.

Необходимо знать, что при использовании **Serializable** десериализация происходит следующим образом: под объект выделяется память, после чего его поля заполняются значениями из потока. Конструктор объекта при этом не вызывается.

/ # 6 # сериализуемый класс # Student.java */*

```
package by.bsu.serial;
import java.io.Serializable;
public class Student implements Serializable {
   protected static String faculty;
   private String name;
   private int id;
   private transient String password;
   private static final long serialVersionUID = 1L;
   /* смысл поля serialVersionUID для класса будет объяснен ниже */
        public Student(String nameOfFaculty, String name, int id, String password) {
          faculty = nameOfFaculty;
          this.name = name;
          this.id = id;
          this.password = password;
   }
   public String toString() {
     return "\nfaculty " + faculty + "\nname " + name + "\nID " + id + "\npassword " + password;
   }
}
```

/ # 7 # запись сериализованного обьекта в файл и его десериализация # Serializator.java */*

```
package by.bsu.serial;
import java.io.File;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.InvalidObjectException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.io.IOException;
import java.io.InvalidClassException;
import java.io.FileNotFoundException;
public class Serializator {
          public boolean serialization(Student s, String fileName) {
                    boolean flag = false;
                    File f = new File(fileName);
                    ObjectOutputStream ostream = null;
                    try {
                              FileOutputStream fos = new FileOutputStream(f);
                              if (fos != null) {
                                        ostream = new ObjectOutputStream(fos);
                                        ostream.writeObject(s); // сериализация
                                        flag = true;
                              }
                    } catch (FileNotFoundException e) {
                              System.err.println("Файл не может быть создан: "+ e);
                    } catch (NotSerializableException e) {
                              System.err.println("Класс не поддерживает сериализацию: " + e);
                    } catch (IOException e) {
                              System.err.println(e);
                    } finally {
                              try {
                                        if (ostream != null) {
                                                  ostream.close();
                                        }
                              } catch (IOException e) {
                                        System.err.println("ошибка закрытия потока");
                              }
                    }
                    return flag;
          }
          public Student deserialization(String fileName) throws InvalidObjectException {
                    File fr = new File(fileName);
                    ObjectInputStream istream = null;
                    try {
                              FileInputStream fis = new FileInputStream(fr);
                              istream = new ObjectInputStream(fis);
                              // десериализация
                              Student st = (Student) istream.readObject();
                              return st;
                    } catch (ClassNotFoundException ce) {
```

```
                    System.err.println("Класс не существует: " + ce);
                } catch (FileNotFoundException e) {
                    System.err.println("Файл для десериализации не существует: "+ e);
                } catch (InvalidClassException ioe) {
                    System.err.println("Несовпадение версий классов: " + ioe);
                } catch (IOException ioe) {
                    System.err.println("Общая I/O ошибка: " + ioe);
                } finally {
                    try {
                        if (istream != null) {
                            istream.close();
                        }
                    } catch (IOException e) {
                        System.err.println("ошибка закрытия потока ");
                    }
                }
                throw new InvalidObjectException("объект не восстановлен");
        }
}
```

/ # 8 # запуск процессов сериализации и десериализации # RunnerSerialization.java */*

```
package by.bsu.serial;
import java.io.InvalidObjectException;
public class RunnerSerialization {
        public static void main(String[] args) {
                // создание и запись объекта
                Student ob = new Student("MMF", "Goncharenko", 1, "G017s9");
                System.out.println(ob);
                String file = "data\\demo.data";
                Serializator sz = new Serializator();
                boolean b = sz.serialization(ob, file);
                Student.faculty = "GEO"; // изменение значения static-поля
                // чтение и вывод объекта
                Student res = null;
                try {
                    res = sz.deserialization(file);
                } catch (InvalidObjectException e) {
                    // обработка
                    e.printStackTrace();
                }
                System.out.println(res);
        }
}
```

В результате выполнения данного кода в консоль будет выведено:

faculty MMF
name Goncharenko
ID 1
password G017s9

faculty GEO
name Goncharenko
ID 1
password null

В итоге поля **name** и **id** нового объекта **res** сохранили значения, которые им были присвоены до записи в файл. Поле **password** со спецификатором **transient** получило значение по умолчанию, соответствующее типу (объектный тип всегда инициализируется по умолчанию значением **null**). Поле **faculty**, помеченное как статическое, получает то значение, которое имеет это поле на текущий момент, то есть при создании объекта **goncharenko** поле получило значение **MMF**, а затем значение статического поля было изменено на **GEO**. Если же объекта данного типа нет в области видимости, то статическое поле также получает значение по умолчанию.

Если полем класса является ссылка на другой тип, то необходимо, чтобы агрегированный тип также реализовывал интерфейс **Serializable**, иначе при попытке сериализации объекта такого класса будет сгенерировано исключение **NotSerializableException**.

/ # 9 # класс, сериализация которого невозможна # Student.java */*

```
public class Student implements Serializable {
          protected static String faculty;
          private int id;
          private String name;
          private transient String password;
          private Address addr = new Address(); // не поддерживающее сериализацию поле
          private static final long serialVersionUID = 2L;
          // more code
}
```

Если класс **Address** не имплементирует интерфейс **Serializable**, а именно:

```
public class Address {
          // поля, методы
}
```

то при таком объявлении класса **Address** сериализация объекта класса **Student** невозможна.

При сериализации объекта класса, реализующего интерфейс **Serializable**, учитывается порядок объявления полей в классе. Поэтому при изменении порядка, имен и типов полей или добавлении новых полей в класс структура информации, содержащейся в сериализованном объекте, будет серьезно отличаться от новой структуры класса. Поэтому десериализация может пройти некорректно. Этим обусловлена необходимость добавления программистом в каждый класс, реализующий интерфейс **Serializable**, поля **private static final long serialVersionUID** на стадии разработки класса. Это поле содержит уникальный

идентификатор версии сериализованного класса. Оно вычисляется по содержимому класса — полям, их порядку объявления, методам, их порядку объявления. Для этого применяются специальные программы-генераторы UID.

Это поле записывается в поток при сериализации класса. Это единственный случай, когда **static**-поле сериализуется.

При десериализации значение этого поля сравнивается с имеющимся у класса в виртуальной машине. Если значения не совпадают, инициируется исключение **java.io.InvalidClassException**. Соответственно, при любом изменении в первую очередь полей класса значение поля **serialVersionUID** должно быть изменено программистом или генератором.

Если набор полей класса и их порядок жестко определены, методы класса могут меняться. В этом случае сериализации и десериализации ничто не угрожает**.**

Вместо реализации интерфейса **Serializable** можно реализовать **Externalizable**, который содержит два метода:

```
void writeExternal(ObjectOutput out)
void readExternal(ObjectInput in)
```

При использовании этого интерфейса в поток автоматически записывается только идентификация класса. Сохранить и восстановить всю информацию о состоянии экземпляра должен сам класс. Для этого в нем должны быть переопределены методы **writeExternal()** и **readExternal()** интерфейса **Externalizable**. Эти методы должны обеспечить сохранение состояния, описываемого полями самого класса и его суперкласса.

При восстановлении **Externalizable**-объекта экземпляр создается путем вызова конструктора без аргументов, после чего вызывается метод **readExternal()**, поэтому необходимо проследить, чтобы в классе был конструктор по умолчанию. Для сохранения состояния вызываются методы **ObjectOutput**, с помощью которых можно записать как примитивные, так и объектные значения. Для корректной работы в соответствующем методе **readExternal()** эти значения должны быть считаны в том же порядке.

Для чтения и записи в поток значений отдельных полей объекта можно использовать, соответственно, методы внутренних классов:

```
ObjectInputStream.GetField
ObjectOutputStream.PutField.
```

Класс Scanner

Объект класса **java.util.Scanner** принимает форматированный объект или ввод из потока и преобразует его в двоичное представление. При вводе могут использоваться данные из консоли, файла, строки или любого другого источника, реализующего интерфейсы **Readable**, **InputStream** или **ReadableByteChannel**.

На настоящий момент класс **Scanner** предлагает наиболее удобный и полный интерфейс для извлечения информации практически из любых источников.

Некоторые конструкторы класса:

```
Scanner(String source)
Scanner(File source) throws FileNotFoundException
Scanner(File source, String charset) throws FileNotFoundException
Scanner(InputStream source)
Scanner(InputStream source, String charset)
Scanner(Path source)
Scanner(Path source, String charset),
```

где **source** — источник входных данных, а **charset** — кодировка.

Объект класса **Scanner** читает лексемы из источника, указанного в конструкторе, например из строки или файла. Лексема — это набор символов, выделенный набором разделителей (по умолчанию пробельными символами). В случае ввода из консоли следует определить объект:

```
Scanner con = new Scanner(System.in);
```

После создания объекта его используют для ввода информации в приложение, например лексему или строку,

```
String str1 = con.next();
String str2 = con.nextLine();
```

или типизированных лексем, например, целых чисел,

```
if(con.hasNextInt()) {
         int n = con.nextInt();
}
```

В классе **Scanner** определены группы методов, проверяющих данные заданного типа на доступ для ввода. Для проверки наличия произвольной лексемы используется метод **hasNext()**. Проверка конкретного типа производится с помощью одного из методов **boolean hasNext***Tun***()** или **boolean hasNext***Tun***(int radix)**, где **radix** — основание системы счисления. Например, вызов метода **hasNextInt()** возвращает **true**, только если следующая входящая лексема — целое число. Если данные указанного типа доступны, они считываются с помощью одного из методов *Tun* **next***Tun***()**. Произвольная лексема считывается методом **String next()**. После извлечения любой лексемы текущий указатель устанавливается перед следующей лексемой.

В качестве примера можно рассмотреть форматированное чтение из файла **scan.txt**, содержащего информацию следующего вида:

2 Java 1,6
true 1.7

// # 10 # разбор текстового файла # ScannerDemo.java

```
package by.bsu.reading;
import java.io.*;
```

```
import java.util.Scanner;
public class ScannerDemo {
          public static void main(String[ ] args) {
                    String filename = "text\\scan.txt";
                    Scanner scan = null;
                    try {
                              FileReader fr = new FileReader(filename);
                              scan = new Scanner(fr);
                              // чтение из файла
                              while (scan.hasNext()) {
                                        if (scan.hasNextInt()) {
                                          System.out.println(scan.nextInt() + " :int");
                                        } else if (scan.hasNextBoolean()) {
                                          System.out.println(scan.nextBoolean() + " :boolean");
                                        } else if (scan.hasNextDouble()) {
                                          System.out.println(scan.nextDouble() + " :double");
                                        } else {
                                          System.out.println(scan.next() + " :String");
                                        }
                              }
                    } catch (FileNotFoundException e) {
                              System.err.println(e);
                    } finally {
                              if (scan != null) {
                                        scan.close();
                              }
                    }
          }
}
```

В результате выполнения кода (при русских региональных установках) будет выведено:

2 :int
Java :String
1.6 :double
true :boolean
1.7 :String

Процедура проверки типа реализована с помощью методов **hasNext***Тип***()**. Такой подход предпочтителен из-за отсутствия возможности возникновения исключительной ситуации, так как ее обработка требует на порядок больше ресурсов, чем нормальное течение программы.

Объект класса **Scanner** определяет границы лексемы, основываясь на наборе разделителей. Можно задавать разделители с помощью метода **useDelimiter(Pattern pattern)** или **useDelimiter(String regex)**, где **pattern** и **regex** содержит набор разделителей в виде регулярного выражения. Применение метода **useLocale(Locale loc)** позволяет задавать правила чтения информации, принятые в заданной стране или регионе.

/ # 11 # применение разделителей и локалей # ScannerDelimiterDemo.java*/*

```
package by.bsu.scan;
import java.util.Locale;
import java.util.Scanner;
public class ScannerDelimiterDemo {
          public static void main(String args[]) {
                    double sum = 0.0;
                    Scanner scan = new Scanner("1,3;2,0; 8,5; 4,8;9,0; 1; 10;");
                    scan.useLocale(Locale.FRANCE);
                    // scan.useLocale(Locale.US);
                    scan.useDelimiter(";\\s*");
                    while (scan.hasNext()) {
                              if (scan.hasNextDouble()) {
                                        sum += scan.nextDouble();
                              } else {
                                        System.out.println(scan.next());
                              }
                    }
                    scan.close();
                    System.out.printf("Сумма чисел = " + sum);
          }
}
```

В результате выполнения программы будет выведено:

Сумма чисел = 36.6

Если заменить **Locale** на американскую, то результат будет иным, так как представление чисел с плавающей точкой отличается.

Использование шаблона **";\\s*"** указывает объекту класса **Scanner**, что символ «**;**» и ноль или более пробелов следует рассматривать как разделитель.

Метод **String findInLine(Pattern pattern)** или **String findInLine(String pattern)** ищет заданный шаблон в текущей строке текста. Если шаблон найден, соответствующая ему подстрока извлекается из строки ввода. Если совпадений не найдено, то возвращается **null**.

Методы **String findWithinHorizon(Pattern pattern, int count)** и **String findWithinHorizon(String pattern, int count)** производят поиск заданного шаблона в ближайших **count** символах. Можно пропустить образец с помощью метода **skip(Pattern pattern)**.

Если в строке ввода найдена подстрока, соответствующая образцу **pattern**, метод **skip()** просто перемещается за нее в строке ввода и возвращает ссылку на вызывающий объект. Если подстрока не найдена, метод **skip()** генерирует исключение **NoSuchElementException**.

Архивация

Для хранения классов языка Java и связанных с ними ресурсов в языке Java используются сжатые архивные **jar**-файлы.

Для работы с архивами в спецификации Java существует два пакета — **java.util.zip** и **java.util.jar** соответственно для архивов **zip** и **jar**. Различие форматов **jar** и **zip** заключается только в расширении архива **zip**. Пакет **java.util.jar** аналогичен пакету **java.util.zip**, за исключением реализации конструкторов и метода **void putNextEntry(ZipEntry e)** класса **JarOutputStream**. Ниже будет рассмотрен только пакет **java.util.jar**. Чтобы переделать все примеры на использование **zip**-архива, достаточно всюду в коде заменить **Jar** на **Zip**.

Пакет **java.util.jar** позволяет считывать, создавать и изменять файлы форматов **jar**, а также вычислять контрольные суммы входящих потоков данных.

Класс **JarEntry** (подкласс **ZipEntry**) используется для предоставления доступа к записям **jar**-файла. Некоторые методы класса:

void setMethod(int method) — устанавливает метод сжатия записи;

void setSize(long size) — устанавливает размер несжатой записи;

long getSize() — возвращает размер несжатой записи;

long getCompressedSize() — возвращает размер сжатой записи.

У класса **JarOutputStream** существует возможность записи данных в поток вывода в **jar**-формате. Он переопределяет метод **write()** таким образом, чтобы любые данные, записываемые в поток, предварительно сжимались. Основными методами данного класса являются:

void setLevel(int level) — устанавливает уровень сжатия. Чем больше уровень сжатия, тем медленней происходит работа с таким файлом;

void putNextEntry(ZipEntry e) — записывает в поток новую **jar**-запись. Этот метод переписывает данные из экземпляра **JarEntry** в поток вывода;

void closeEntry() — завершает запись в поток **jar**-записи и заносит дополнительную информацию о ней в поток вывода;

void write(byte b[], int off, int len) — записывает данные из буфера **b**, начиная с позиции **off,** длиной **len** в поток вывода;

void finish() — завершает запись данных **jar**-файла в поток вывода без закрытия потока.

Пусть необходимо архивировать только файлы в указанной директории. Если директория содержит другие директории, то их файлы архивироваться не будут.

/ # 12 # создание jar-архива # PackJar.java */*

```
package by.bsu.packing;
import java.io.File;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;
```

```
import java.util.ArrayList;
import java.util.jar.JarEntry;
import java.util.jar.JarOutputStream;
import java.util.zip.Deflater;
public class PackJar {
          private String jarFileName;
          public final int BUFFER = 2_048;
          public PackJar(String jarFileName) {
                    this.jarFileName = jarFileName;
          }
          public void pack(String dirName) throws FileNotFoundException {
                    // получение списка имен файлов в директории
                    File dir = new File(dirName);
                    if (!dir.exists() || !dir.isDirectory()) {
                              throw new FileNotFoundException(dir + " not found");
                    }
                    File[] files = dir.listFiles();
                    ArrayList<String> listFilesToJar = new ArrayList<>();
                    for (int i = 0; i < files.length; i++) {
                              if (!files[i].isDirectory()) {
                                        listFilesToJar.add(files[i].getPath());
                              }
                    }
                    String[] temp = {};
                    String[] filesToJar = listFilesToJar.toArray(temp);
                    // собственно архивирование
                    try (FileOutputStream fos = new FileOutputStream(jarFileName);
                                        JarOutputStream jos = new JarOutputStream(fos)) {
                              byte[] buffer = new byte[BUFFER];
                              // метод сжатия
                              jos.setLevel(Deflater.DEFAULT_COMPRESSION);
                              for (int i = 0; i < filesToJar.length; i++) {
                                    jos.putNextEntry(new JarEntry(filesToJar[i]));
                                    try (FileInputStream in = new FileInputStream(filesToJar[i])) {
                                              int len;
                                              while ((len = in.read(buffer)) > 0) {
                                                        jos.write(buffer, 0, len);
                                              }
                                              jos.closeEntry();
                                        } catch (FileNotFoundException e) {
                                              System.err.println("Файл не найден" + e);
                                        }
                              }
                    } catch (IllegalArgumentException e) {
                              System.err.println(e + "Некорректный аргумент" + e);
                    } catch (IOException e) {
                              System.err.println("Ошибка доступа" + e);
                    }
          }
}
```

Расширить класс до архивации файлов из вложенных директорий относительно просто.

Класс **JarFile** обеспечивает гибкий доступ к записям, хранящимся в **jar**-файле. Это достаточно эффективный способ, поскольку доступ к данным осуществляется гораздо быстрее, чем при считывании каждой отдельной записи. Единственным недостатком является то, что доступ может осуществляться только для чтения. Метод **entries**() извлекает все записи из **jar**-файла. Этот метод возвращает список экземпляров **JarEntry** — по одной для каждой записи в **jar**-файле. Метод **getEntry(String name)** извлекает запись по имени. Метод **getInputStream()** создает поток ввода для записи. Этот метод возвращает поток ввода, который может использоваться приложением для чтения данных записи.

/ # 13 # запуск процесса архивации # PackDemo.java */*

```
package by.bsu.packing;
import java.io.FileNotFoundException;
public class PackDemo {
          public static void main(String[] args) {
                    String dirName = "путь_к_директории";
                    PackJar pj = new PackJar("example.jar");
                    try {
                              pj.pack(dirName);
                    } catch (FileNotFoundException e) {
                              e.printStackTrace();
                    }
          }
}
```

В результате выполнения кода будет создан архивный файл **example.jar**, размещенный в корне проекта.

Класс **JarInputStream** читает данные в **jar**-формате из потока ввода. Он переопределяет метод **read()** таким образом, чтобы любые данные, считываемые из потока, предварительно распаковывались.

Теперь следует распаковать файл из архива и разместить по заданному пути, к которому добавится исходный путь к файлам.

/ # 14 # чтение jar-архива # UnPackJar.java */*

```
package by.bsu.packing;
import java.io.BufferedInputStream;
import java.io.BufferedOutputStream;
import java.io.File;
import java.io.FileOutputStream;
import java.io.IOException;
import java.util.Enumeration;
import java.util.jar.JarEntry;
import java.util.jar.JarFile;
public class UnPackJar {
```

```
        private File destFile;
        // размер буфера для распаковки
        public final int BUFFER = 2_048;
        public void unpack(String destinationDirectory, String nameJar) {
            File sourceJarFile = new File(nameJar);
            try (JarFile jFile = new JarFile(sourceJarFile)) {
                  File unzipDir = new File(destinationDirectory);
                  // открытие jar-архива для распаковки
                  Enumeration<JarEntry> jarFileEntries = jFile.entries();
                  while (jarFileEntries.hasMoreElements()) {
                            // извлечение текущей записи из архива
                            JarEntry entry = jarFileEntries.nextElement();
                            String entryname = entry.getName();
                            System.out.println("Extracting: " + entry);
                            destFile = new File(unzipDir, entryname);
                            // определение каталога
                            File destinationParent = destFile.getParentFile();
                            // создание структуры каталогов
                            destinationParent.mkdirs();
                            // распаковывание записи, если она не каталог
                            if (!entry.isDirectory()) {
                            writeFile(jFile, entry);
                            }
                  }
             } catch (IOException e) {
             e.printStackTrace();
             }
        }
        private void writeFile(JarFile jFile, JarEntry entry) {
                  int currentByte;
                  byte data[] = new byte[BUFFER];
                  try (BufferedInputStream is = new BufferedInputStream(jFile.getInputStream(entry));
                  FileOutputStream fos = new FileOutputStream(destFile);
                  BufferedOutputStream dest = new BufferedOutputStream(fos,BUFFER)) {
                            // запись файла на диск
                            while ((currentByte = is.read(data, 0, BUFFER)) > 0) {
                                      dest.write(data, 0, currentByte);
                            }
                            dest.flush();
                  } catch (IOException e) {
                            e.printStackTrace();
                  }
        }
}
```

При указании пути к архивному файлу и пути, по которому требуется разместить разархивированные файлы, следует указывать либо абсолютный путь, либо путь относительно корневой директории проекта.

```
/* # 15 # запуск процесса архивации # UnpackDemo.java */
package by.bsu.packing;
public class UnpackDemo {
          public static void main(String[] args) {
                    // расположение и имя архива
                    String nameJar = "example.jar";
                    // куда файлы будут распакованы
                    String destinationPath = "c:\\temp\\";
                    new UnPackJar().unpack(destinationPath, nameJar);
          }
}
```

На консоль будет выведен список разархивированных файлов, например:

Extracting: c:\temp\by\bsu\market\Broker.java
Extracting: c:\temp\by\bsu\market\Main.java
Extracting: c:\temp\by\bsu\market\Market.java

Задания к главе 9

Вариант А

В следующих заданиях требуется ввести последовательность строк из текстового потока и выполнить указанные действия. При этом могут рассматриваться два варианта:

- каждая строка состоит из одного слова;
- каждая строка состоит из нескольких слов.

Имена входного и выходного файлов, а также абсолютный путь к ним могут быть введены как параметры командной строки или храниться в файле.

1. В каждой строке найти и удалить заданную подстроку.
2. В каждой строке стихотворения найти и заменить заданную подстроку на подстроку иной длины.
3. В каждой строке найти слова, начинающиеся с гласной буквы.
4. Найти и вывести слова текста, для которых последняя буква одного слова совпадает с первой буквой следующего слова.
5. Найти в строке наибольшее число цифр, идущих подряд.
6. В каждой строке стихотворения Сергея Есенина подсчитать частоту повторяемости каждого слова из заданного списка и вывести эти слова в порядке возрастания частоты повторяемости.
7. В каждом слове сонета Вильяма Шекспира заменить первую букву слова на прописную.
8. Определить частоту повторяемости букв и слов в стихотворении Александра Пушкина.

Вариант B

Выполнить задания из варианта B гл. 4, сохраняя объекты приложения в одном или нескольких файлах с применением механизма сериализации. Объекты могут содержать поля, помеченные как **static**, а также **transient**. Для изменения информации и извлечения информации в файле создать специальный класс-коннектор с необходимыми для выполнения этих задач методами.

Вариант C

При выполнении следующих заданий для вывода результатов создавать новую директорию и файл средствами класса **File**.

1. Создать и заполнить файл случайными целыми числами. Отсортировать содержимое файла по возрастанию.
2. Прочитать текст Java-программы и все слова **public** в объявлении атрибутов и методов класса заменить на слово **private**.
3. Прочитать текст Java-программы и записать в другой файл в обратном порядке символы каждой строки.
4. Прочитать текст Java-программы и в каждом слове длиннее двух символов все строчные символы заменить прописными.
5. В файле, содержащем фамилии студентов и их оценки, записать прописными буквами фамилии тех студентов, которые имеют средний балл более 7.
6. Файл содержит символы, слова, целые числа и числа с плавающей запятой. Определить все данные, тип которых вводится из командной строки.
7. Из файла удалить все слова, содержащие от трех до пяти символов, но при этом из каждой строки должно быть удалено только максимальное четное количество таких слов.
8. Прочитать текст Java-программы и удалить из него все «лишние» пробелы и табуляции, оставив только необходимые для разделения операторов.
9. Из текста Java-программы удалить все виды комментариев.
10. Прочитать строки из файла и поменять местами первое и последнее слова в каждой строке.
11. Ввести из текстового файла, связанного с входным потоком, последовательность строк. Выбрать и сохранить *m* последних слов в каждой из последних n строк.
12. Из текстового файла ввести последовательность строк. Выделить отдельные слова, разделяемые пробелами. Написать метод поиска слова по образцу-шаблону. Вывести найденное слово в другой файл.
13. Сохранить в файл, связанный с выходным потоком, записи о телефонах и их владельцах. Вывести в файл записи, телефоны в которых начинаются на *k* и на *j*.
14. Входной файл содержит совокупность строк. Строка файла содержит строку квадратной матрицы. Ввести матрицу в двумерный массив (размер матрицы найти). Вывести исходную матрицу и результат ее транспонирования.

15. Входной файл хранит квадратную матрицу по принципу: строка представляет собой число. Определить размерность. Построить 2-мерный массив, содержащий матрицу. Вывести исходную матрицу и результат ее поворота на 90˚ по часовой стрелке.
16. В файле содержится совокупность строк. Найти номера строк, совпадающих с заданной строкой. Имя файла и строка для поиска — аргументы командной строки. Вывести строки файла и номера строк, совпадающих с заданной.