Глава 9

ФАЙЛЫ. ПОТОКИ ВВОДА/ВЫВОДА

Потоки ввода/вывода используются для передачи данных в файловые потоки, на консоль или на сетевые соединения. Потоки представляют собой объекты соответствующих классов. Библиотека ввода/вывода предоставляет пользователю большое число классов и методов и постоянно обновляется.

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 права 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 myFile = new File("\\com\\myfile.txt");
File myDir  = new File("c:\\jdk1.6.0\\src\\java\\io");
File myFile = new File(myDir, "File.java");
File myFile = new File("c:\\com", "myfile.txt");
File myFile = 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 myFile  = new File(File.separator + "com"
        + File.separator + "myfile.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=";"**, а для Windows – **pathSeparator=":"**.

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

```
/* пример # 1 : работа с файловой системой: FileTest.java */
package chapt09;
import java.io.*;
import java.util.*;

public class FileTest {
   public static void main(String[] args) {
   //с объектом типа File ассоциируется файл на диске FileTest2.java
       File fp = new File("chapt09" + File.separator
             + "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.toString());
          //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:     chapt09\FileTest2.java
Абсолютный путь:  D:\workspace\chapt09\FileTest2.java
Размер файла:     2091
Последняя модификация : Fri Mar 31 12:26:50 EEST 2006
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()**.

Байтовые и символьные потоки ввода/вывода

При создании приложений всегда возникает необходимость прочитать информацию из какого-либо источника и сохранить результат. Действия по чтению/записи информации представляют собой стандартный и простой вид деятельности. Самые первые классы ввода/вывода связаны с передачей и извлечением последовательности байтов.

Потоки ввода последовательности байтов являются подклассами абстрактного класса **InputStream**, потоки вывода – подклассами абстрактного класса **OutputStream**. Эти классы являются суперклассами для ввода массивов байтов, строк, объектов, а также для выбора из файлов и сетевых соединений. При работе с файлами используются подклассы этих классов соответственно

FileInputStream и **FileOutputStream**, конструкторы которых открывают поток и связывают его с соответствующим физическим файлом.

Для чтения байта или массива байтов используются абстрактные методы **read()** и **read(byte[] b)** класса **InputStream**. Метод возвращает **–1**, если достигнут конец потока данных, поэтому возвращаемое значение имеет тип **int**, 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 конкретных классах потоков ввода указанные выше методы реализованы в соответствии с предназначением класса. В классе **FileInputStream** данный метод читает один байт из файла, а поток **System.in** как встроенный объект подкласса **InputStream** позволяет вводить информацию с консоли. Абстрактный метод **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.

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 рис. 9.1.

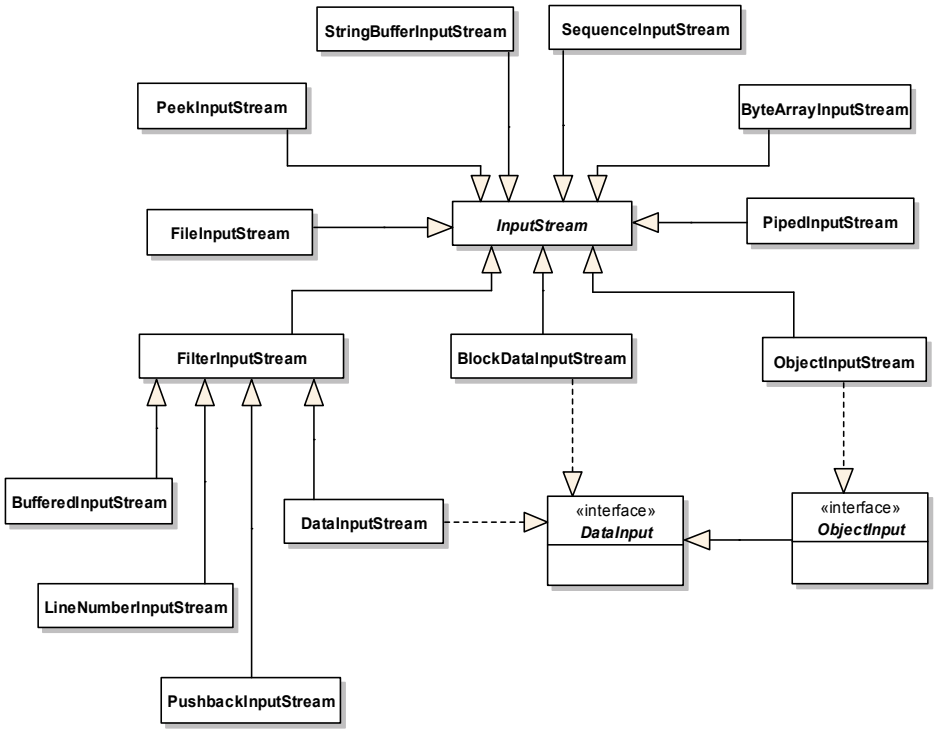

Рис. 9.1. Иерархия классов байтовых потоков ввода

Абстрактный класс **`FilterInputStream`** используется как шаблон для настройки классов ввода, наследуемых от класса **`InputStream`**. Класс **`DataInputStream`** предоставляет методы для чтения из потока данных значений базовых типов, но начиная с версии 1.2 класс был помечен как deprecated и не рекомендуется к использованию. Класс **`BufferedInputStream`** присоединяет к потоку буфер для ускорения последующего доступа.

Для вывода данных используются потоки следующих классов.

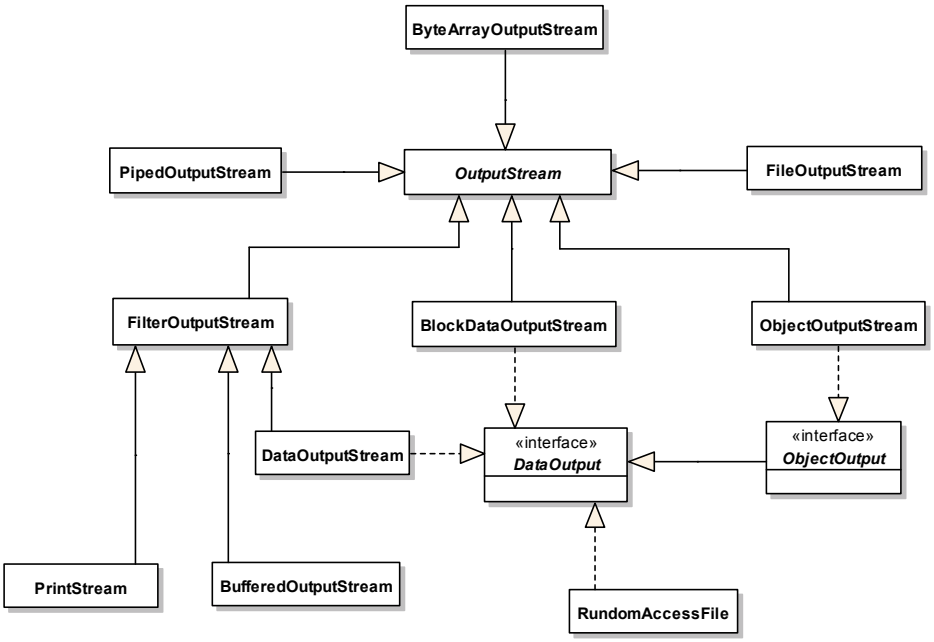

Рис. 9.2. Иерархия классов байтовых потоков вывода

Абстрактный класс **`FilterOutputStream`** используется как шаблон для настройки производных классов. Класс **`BufferedOutputStream`** присоединяет буфер к потоку для ускорения вывода и уменьшения 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 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.

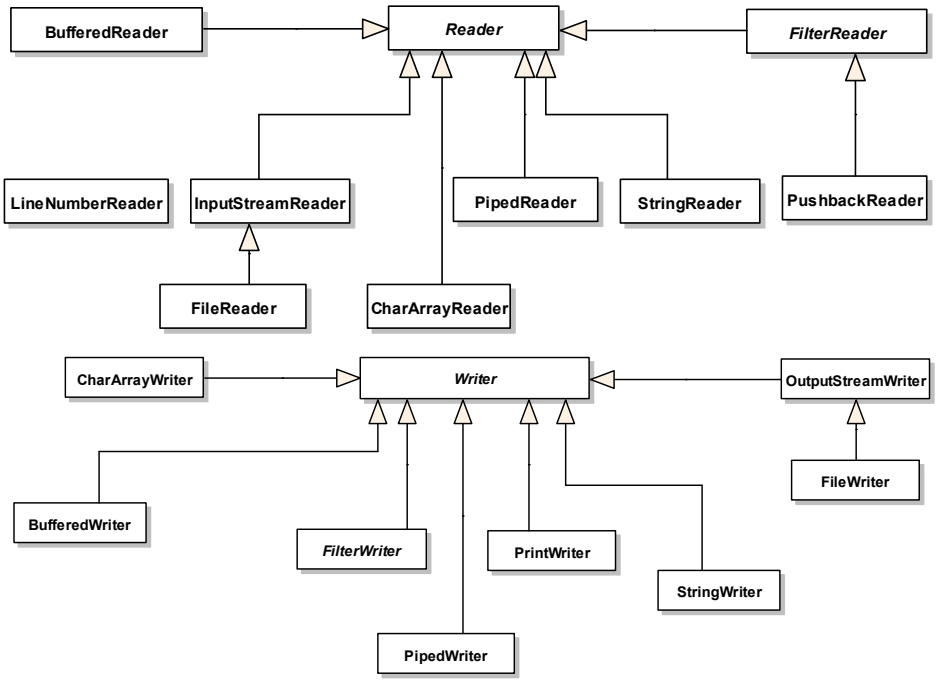

Рис. 9.3. Иерархия символьных потоков ввода/вывода

```
/* пример # 2 : чтение по одному байту (символу) из потока ввода : ReadDemo.java */
package chapt09;
import java.io.File;
import java.io.FileReader;
import java.io.IOException;

public class ReadDemo {
   public static void main(String[] args) {
      File f = new File("file.txt");//должен существовать!

      int b, count = 0;
      try {
         FileReader is = new FileReader(f);
      /* FileInputStream is = new FileInputStream(f);*/ //альтернатива
         while ((b = is.read()) != -1) {/*чтение*/
            System.out.print((char)b);
            count++;
         }
         is.close(); // закрытие потока ввода
       } catch (IOException e) {
            System.err.println("ошибка файла: " + e);
      }
      System.out.print("\n число байт = " + count);
   }
}
```

Один из конструкторов **FileReader(f)** или **FileInputStream(f)** открывает поток **is** и связывает его с файлом **f** . Для закрытия потока используется метод **close()**. При чтении из потока можно пропустить **n** байт с помощью метода **long skip(long n)**.

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()**.

```
// пример # 3 : вывод массива в поток в виде символов и байтов: WriteRunner.java
package chapt09;
import java.io.*;

public class WriteRunner {
      public static void main(String[] args) {
            String pArray[] = { "2007 ", "Java SE 6" };
            File fbyte = new File("byte.txt");
            File fsymb = new File("symbol.txt");
            try {
                  FileOutputStream fos =
                        new FileOutputStream(fbyte);
                  FileWriter fw = new FileWriter(fsymb);
                  for (String a : pArray) {
                        fos.write(a.getBytes());
                        fw.write(a);
                  }
                  fos.close();
                  fw.close();
            } catch (IOException e) {
                  System.err.println("ошибка файла: " + e);
            }
      }
}
```

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

В отличие от классов **FileInputStream** и **FileOutputStream** класс **RandomAccessFile** позволяет осуществлять произвольный доступ к потокам как ввода, так и вывода. Поток рассматривается при этом как массив байтов, доступ к элементам осуществляется с помощью метода **seek(long poz)**. Для создания потока можно использовать один из конструкторов:

```
RandomAccessFile(String name, String mode);
RandomAccessFile(File file, String mode);
```

Параметр **mode** равен **"r"** для чтения или **"rw"** для чтения и записи.

```
/* пример # 4 : запись и чтение из потока: RandomFiles.java */
package chapt09;
import java.io.*;

public class RandomFiles {
```

```
    public static void main(String[] args) {
        double data[] = { 1, 10, 50, 200, 5000 };
        try {
        RandomAccessFile rf =
            new RandomAccessFile("temp.txt", "rw");
            for (double d : data)
                rf.writeDouble(d); // запись в файл
        /* чтение в обратном порядке */
        for (int i = data.length - 1; i >= 0; i--) {
                rf.seek(i * 8);
        // длина каждой переменной типа double равна 8-и байтам
            System.out.println(rf.readDouble());
            }
            rf.close();
        } catch (IOException e) {
            System.err.println(e);
        }
    }
}
```

В результате будет выведено:

```
5000.0
200.0
50.0
10.0
1.0
```

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**.

При 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("file.txt"))));
```

В итоге класс **BufferedWriter** выступает классом-оберткой для класса **FileWriter**, так же как и класс **BufferedReader** для **FileReader**.

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

```
// пример # 5 : вывод в файл: DemoWriter.java
package chapt09;
import java.io.*;

public class DemoWriter {
      public static void main(String[] args) {
            File f = new File("res.txt");
            FileWriter fw = null;
            try {
                  fw = new FileWriter(f, true);
            } catch (IOException e) {
      System.err.println("ошибка открытия потока " + e);
      System.exit(1);
            }
            BufferedWriter bw = new BufferedWriter(fw);
            PrintWriter pw = new PrintWriter(bw);

            double[] v = { 1.10, 1.2, 1.401, 5.01 };
            for (double version : v)
                  pw.printf("Java %.2g%n", version);
            pw.close();
      }
}
```

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

```
Java 1.1
Java 1.2
Java 1.4
Java 5.0
```

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("f.txt")));
```

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

```
// пример # 6 : чтение из файла: DemoReader.java
package chapt09;
import java.io.*;
```

```
public class DemoReader {
      public static void main(String[] args) {
            try {
      BufferedReader br =
            new BufferedReader(new FileReader("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);
                  }
                  br.close();
            } catch (IOException e) {
                  e.printStackTrace();
            }
      }
}
```

В консоль будет выведено:

```
Java
1.1
Java
1.2
Java
1.4
Java
5.0
```

Сериализация объектов

Кроме данных базовых типов, в поток можно отправлять объекты классов.

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 **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 используется 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.

```
/* пример # 7 : запись сериализованного объекта в файл и его десериализация :
Student.java : DemoSerialization.java */
package chapt09;
import java.io.*;

class Student implements Serializable {
   protected static String faculty;
   private String name;
   private int id;
   private transient String password;
   private static final long serialVersionUID = 1L;
   /*значение этого поля для класса будет дано далее*/

   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;
   }
}

public class DemoSerialization {
   public static void main(String[] args) {
      // создание и запись объекта
      Student goncharenko =
            new Student("MMF", "Goncharenko", 1, "G017s9");
            System.out.println(goncharenko);
```

```
      File fw = new File("demo.dat");
      try {
         ObjectOutputStream ostream =
            new ObjectOutputStream(
                  new FileOutputStream(fw));

         ostream.writeObject(goncharenko);
         ostream.close();
      } catch (IOException e) {
          System.err.println(e);
      }
      Student.faculty = "GEO";//изменение значения static-поля
   // чтение и вывод объекта
      File fr = new File("demo.dat");
      try {
         ObjectInputStream istream =
                  new ObjectInputStream(
                        new FileInputStream(fr));

         Student unknown =
                  (Student)istream.readObject();
         istream.close();
         System.out.println(unknown);
      } catch (ClassNotFoundException ce) {
            System.err.println(ce);
            System.err.println("Класс не существует");
      } catch (FileNotFoundException fe) {
            System.err.println(fe);
            System.err.println("Файл не найден");
      } catch (IOException ioe) {
            System.err.println(ioe);
            System.err.println("Ошибка доступа");
      }
   }
}
```

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 **unknown** сохранили значения, которые им были присвоены до записи в файл. Поле **password** со спецификатором **transient** получило значение по умолчанию, соответствующее типу (объект-

ный 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 поля класса являются объектами другого класса, то необходимо, чтобы тот класс тоже реализовал интерфейс **Serializable.**

При сериализации объекта класса, реализующего интерфейс **Serializable**, учитывается порядок объявления полей в классе. Поэтому при изменении порядка десериализация пройдет некорректно. Это обусловлено тем, что в каждый класс, реализующий интерфейс **Serializable**, на стадии компиляции добавляется поле **private static final long serialVersionUID**. Это поле содержит уникальный идентификатор версии сериализованного класса. Оно вычисляется по содержимому класса – полям, их порядку объявления, методам, их порядку объявления.

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 классе это поле поменяет свое значение.

Если набор полей класса и их порядок жестко определены, методы класса могут меняться. В этом случае сериализации ничего не угрожает, однако стандартный механизм не даст десериализовать данные. В таких случаях можно вручную в классе определить поле **private static final long serialVersionUID.**

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 пустой конструктор. 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.

Консоль

Одним из классов, предоставляющих дополнительные возможности чтения и последующей типизации информации консоли (или любого другого потока), является **java.util.Scanner**, введенный в пятой версии языка. Также для взаимодействия с консолью применяется класс **java.io.Console**, введенный в шестой версии языка.

```
// пример # 8 : ввод информации : UserHelper.java
package chapt01;
//подключение классов ввода
import java.io.Console;
// обработчик ошибок ввода
import java.util.InputMismatchException;

public class Helper {
  //чтение информации из консоли с помощью класса Console
  public void readFromConsole() {
     Console con = System.console();
     if (con != null) {
        con.printf("Введите числовой код:");
        int code = 0;
        try {
           code = Integer.valueOf(con.readLine());
           System.out.println("Код доступа:" + code);
     } catch (InputMismatchException e) {
           con.printf("неправильный формат кода" + e);
        }
        if (code != 0) {
           con.printf("Введите пароль:");
           String password;
           char passTemp[] =
                  con.readPassword("Введите пароль: ");
           password = new String(passTemp);
           System.out.println("Пароль:" + password);
                     }
     } else {
        System.out.println("Консоль недоступна");
     }
   }
}
// пример # 9 : инициализация объектов и вызов методов: Runner.java
package chapt01;

public class Runner {
      public static void main(String[] args) {
            Helper helper = new Helper();
```

```
            helper.readFromConsole();
        }
}
```

В ответ на запрос можно ввести некоторые данные и получить следующий результат:

```
Введите числовой код:
1001
Введите пароль:
****
Код доступа: 1001
Пароль: pass
```

При вводе значения **code**, не являющегося цифрой, на экран будет выдано сообщение об ошибке при попытке его преобразования в целое число, так как метод **valueOf()** пытается преобразовать строку в целое число, не проверив предварительно, может ли быть выполнено это преобразование.

Класс Scanner

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

Класс определяет следующие конструкторы:

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

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 данных, выделенный набором разделителей (по умолчанию пробелами). В случае ввода из консоли следует определить объект:

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

После создания объекта его используют для ввода, например целых чисел, следующим образом:

```
write(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**Тип**()** или **boolean**

hasNextТип**(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 Тип **next**Тип**()**. Произвольная лексема считывается методом **String next()**. После извлечения любой лексемы текущий указатель устанавливается перед следующей лексемой.

```
// пример # 10 : разбор файла: ScannerLogic.java : ScannerDemo.java
package chap09;
import java.io.*;
import java.util.Scanner;

class ScannerLogic {
   static String filename = "scan.txt";
   public static void scanFile() {
      try {
         FileReader fr =
            new FileReader(filename);
         Scanner scan = new Scanner(fr);
         while (scan.hasNext()) {//чтение из файла

            if (scan.hasNextInt())
                  System.out.println(
                        scan.nextInt() + ":int");
            else if (scan.hasNextDouble())
                  System.out.println(
                        scan.nextDouble() + ":double");
            else if (scan.hasNextBoolean())
                  System.out.println(
                        scan.nextBoolean() + ":boolean");
            else
                  System.out.println(
                        scan.next() + ":String");
         }
      } catch (FileNotFoundException e) {
         System.err.println(e);
      }
   }
   public static void makeFile() {
      try {
         FileWriter fw =
            new FileWriter(filename);//создание потока для записи
         fw.write("2 Java 1,5 true 1.6 ");//запись данных
         fw.close();
      } catch (IOException e) {
         System.err.println(e);
      }
   }
```

```
}
public class ScannerDemo {
      public static void main(String[] args) {
            ScannerLogic.makeFile();
            ScannerLogic.scanFile();
      }
}
```

В результате выполнения программы будет выведено:

```
2:int
Java:String
1.5:double
true:boolean
1.6:String
```

Процедура проверки типа реализована при 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. Для чтения строки из потока ввода применяются методы **next()** или **nextLine()**.

Объект класса **Scanner** определяет границы лексемы, основываясь на наборе разделителей. Можно задавать разделители с помощью метода **useDelimiter(Pattern pattern)** или **useDelimiter(String pattern)**, где **pattern** содержит набор разделителей.

```
/* пример # 11 : применение разделителей: ScannerDelimiterDemo.java*/
package chapt09;
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.useDelimiter(";\\s*");
         while (scan.hasNext()) {
            if (scan.hasNextDouble())
                sum += scan.nextDouble();
            else System.out.println(scan.next());
         }
         System.out.printf("Сумма чисел = " + sum);
      }
   }
}
```

В результате выполнения программы будет выведено:

```
Сумма чисел = 36.6
```

Использование шаблона **"; *"** указывает объекту класса **Scanner**, что ‘;’ и ноль или более пробелов следует рассматривать как разделитель.

Метод **String findInLine(Pattern pattern)** или **String findInLine(String pattern)** ищет заданный шаблон в следующей 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 существуют 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. Наиболее важными методами класса являются:

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

int getMethod() – возвращает метод сжатия записи;

void setComment(String comment) – устанавливает комментарий записи;

String getComment() – возвращает комментарий 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;

`void close()` – закрывает поток записи.

```
/* пример # 12 : создание jar-архива: PackJar.java */
package chapt09;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;
import java.util.jar.JarEntry;
import java.util.jar.JarOutputStream;
import java.util.zip.Deflater;

public class PackJar {
   public static void pack(String[] filesToJar,
                  String jarFileName, byte[] buffer) {
      try {
            JarOutputStream jos =
                  new JarOutputStream(
                        new FileOutputStream(jarFileName));
            // метод сжатия
            jos.setLevel(Deflater.DEFAULT_COMPRESSION);
            for (int i = 0; i < filesToJar.length; i++) {
                  System.out.println(i);
            jos.putNextEntry(new JarEntry(filesToJar[i]));

            FileInputStream in =
                        new FileInputStream(filesToJar[i]);
             int len;
              while ((len = in.read(buffer)) > 0)
                  jos.write(buffer, 0, len);
              jos.closeEntry();
              in.close();
            }
            jos.close();
      } catch (IllegalArgumentException e) {
            e.printStackTrace();
            System.err.println("Некорректный аргумент");
      } catch (FileNotFoundException e) {
            e.printStackTrace();
```

```
            System.err.println("Файл не найден");
        } catch (IOException e) {
            e.printStackTrace();
            System.err.println("Ошибка доступа");
        }
    }
    public static void main(String[] args) {
        System.out.println("Создание jar-архива");
        // массив файлов для сжатия
        String[] filesToJar = new String[2];
        filesToJar[0] = "chapt09//UseJar.java";
        filesToJar[1] = "chapt09//UseJar.class";
        byte[] buffer = new byte[1024];
        // имя полученного архива
        String jarFileName = "example.jar";
        pack(filesToJar, jarFileName, buffer);
    }
}
```

Класс **JarFile** обеспечивает гибкий доступ к записям, хранящимся в **jar**-файле. Это очень 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.

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

```
/* пример # 13 : чтение jar-архива: UnPackJar.java */
package chapt09;
import java.io.*;
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 = 2048;

  public void unpack(String destinationDirectory,
                                          String nameJar) {
            File sourceJarFile = new File(nameJar);
            try {
            File unzipDestinationDirectory =
```

```
                    new File(destinationDirectory);
              // открытие zip-архива для чтения
        JarFile jFile = new JarFile(sourceJarFile);
        Enumeration jarFileEntries = jFile.entries();
        while (jarFileEntries.hasMoreElements()) {
              // извлечение текущей записи из архива
        JarEntry entry =
              (JarEntry) jarFileEntries.nextElement();

        String entryname = entry.getName();
              //entryname = entryname.substring(2);
        System.out.println("Extracting: " + entry);
              destFile =
        new File(unzipDestinationDirectory, entryname);
              // определение каталога
              File destinationParent =
                          destFile.getParentFile();
              // создание структуры каталогов
                    destinationParent.mkdirs();
              // распаковывание записи, если она не каталог
                    if (!entry.isDirectory()) {
                          writeFile(jFile, entry);
                    }
              }
              jFile.close();
        } catch (IOException ioe) {
              ioe.printStackTrace();
        }
    }
private void writeFile(JarFile jFile, JarEntry entry)
                          throws IOException {
        BufferedInputStream is =
              new BufferedInputStream(
                    jFile .getInputStream(entry));
        int currentByte;
        byte data[] = new byte[BUFFER];
        // запись файла на диск
        BufferedOutputStream dest =
              new BufferedOutputStream(
              new FileOutputStream(destFile), BUFFER);

    while ((currentByte = is.read(data, 0, BUFFER)) > 0){
              dest.write(data, 0, currentByte);
        }
        dest.flush();
        dest.close();
        is.close();
    }
```

```
    public static void main(String[] args) {
        System.out.println(
            "Извлечение данных из jar-архива");
        // расположение и имя архива
        String nameJar = "c:\\work\\example.jar";
        // куда файлы будут распакованы
        String destination = "c:\\temp\\";
        new UnPackJar().unpack(destination, nameJar);
    }
}
```

Задания к главе 9

Вариант A

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 Александра Блока 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 главы 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 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 градусов 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.

Тестовые задания к главе 9

Вопрос 9.1.

Можно ли изменить корневой каталог, в который вкладываются все пользовательские каталоги, используя объект **`myfile`** класса **`File`**? Если это возможно, то с помощью какой инструкции?

1) `myfile.chdir("NAME");`
2) `myfile.cd("NAME");`
3) `myfile.changeDir("NAME");`
4) методы класса **`File`** не могут изменять корневой каталог.

Вопрос 9.2.

Экземпляром какого класса является поле **`System.in`**?

1) `java.lang.System;`
2) `java.io.InputStream;`
3) `java.io.BufferedInputStream;`
4) `java.io.PrintStream;`
5) `java.io.Reader.`

Вопрос 9.3.

Какие из следующих операций можно выполнить применительно к файлу на диске с помощью методов объекта класса **`File`**?

1) добавить запись в файл;
2) вернуть имя родительской директории;
3) удалить файл;
4) определить, текстовую или двоичную информацию содержит файл.

Вопрос 9.4.

Какой абстрактный класс является суперклассом для всех классов, используемых для чтения байтов?

1) `Reader;`
2) `FileReader;`
3) `ByteReader;`
4) `InputStream;`
5) `FileInputStream.`

Вопрос 9.5.

При объявлении какого из приведенных понятий может быть использован модификатор **`transient`**?

1) класса;
2) метода;
3) поля класса;
4) локальной переменной;
5) интерфейса.