Лабораторная работа №5

Система ввода/вывода.

1. Введение

Описываются реализованные в Java возможности передачи информации: работа с файлами, сетью, долговременное сохранение объектов, обмен данными между потоками исполнения и т.п. Все эти действия базируются на потоках байт (представлены классами InputStream и OutputStream) и потоках символов (Reader и Writer). Все эти классы и их многочисленные наследники содержатся в библиотеке java.io. Отдельно рассматривается механизм сериализации объектов и работа с файлами.

2. Общие сведения

2.1 Система ввода/вывода. Потоки данных (stream)

Часть вычислительной платформы, которая отвечает за обмен данными, называется – система ввода/вывода. В Java она представлена пакетом java.io (input/output).

В Java для описания работы по вводу/выводу используется специальное понятие поток данных (stream). **Поток данных** связан с некоторым источником, или приемником, данных, способным получать или предоставлять информацию. Соответственно, потоки делятся на входящие – **читающие** данные и выходящие – **передающие** (записывающие) данные. Введение концепции stream позволяет отделить логику программы от низкоуровневых операций с устройствами ввода/вывода.

В Java потоки представляются объектами. Описывающие их классы составляют основную часть пакета java.io. Все классы разделены на две части – одни осуществляют ввод данных, другие – вывод.

Базовые, наиболее универсальные, классы позволяют считывать и записывать информацию именно в виде набора байт. Чтобы их было удобно применять в различных задачах, java.io содержит также классы, преобразующие любые данные в набор байт. Например, если нужно сохранить тип double – в файл, то можно сначала превратить его в набор байт, а затем эти байты записать в файл. Аналогичные действия совершаются когда требуется сохранить объект. При восстановлении данных проделываются обратные действия – сначала считывается последовательность байт, а затем она преобразуется в нужный формат.

На рис.1 представлены иерархии классов ввода/вывода. Входные потоки классы наследуются от InputStream, а выходные – от OutputStream.

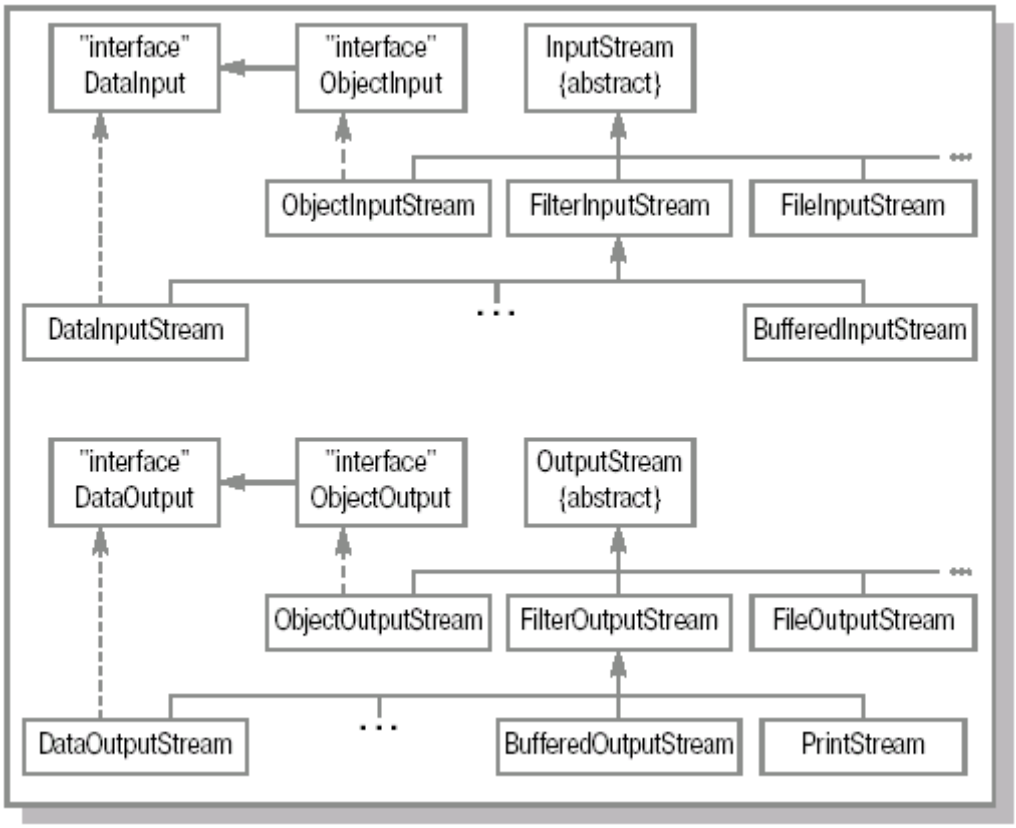

Рис. 15.1. Иерархия классов ввода/вывода.

2.1.1 Классы InputStream и OutputStream

InputStream – это базовый класс для потоков ввода, и содержит базовые методы для работы с байтовыми потоками данных. Эти методы необходимы всем классам, которые наследуются от InputStream.

Метод **read**() (без аргументов) - является абстрактным и должен быть определен в классах-наследниках. Метод предназначен для считывания одного байта из потока. Если считывание произошло успешно, возвращаемое значение лежит в диапазоне от 0 до 255 и представляет собой полученный байт. Если достигнут конец потока, то возвращаемое значение равно -1.

Если же считать из потока данные не удается из-за каких-то ошибок, или сбоев, будет брошено исключение java.io.IOException. Этот класс наследуется от Exception, т.е. его всегда необходимо обрабатывать явно.

На практике обычно приходится считывать не один, а сразу несколько байт – то есть массив байт. Для этого используется метод read(), где в качестве параметров передается массив byte[]. При выполнении этого метода в цикле производится вызов абстрактного метода read() (определенного без параметров) и результатами заполняется переданный массив. Количество байт, считываемое таким образом, равно длине переданного массива. Но при этом может так получиться, что данные в потоке закончатся еще до того, как будет заполнен весь массив. То есть возможна ситуация, когда в потоке данных (байт) содержится меньше, чем длина массива. Поэтому метод возвращает значение int, указывающее, сколько байт было реально считано. Понятно, что это значение может быть от 0 до величины длины переданного массива.

Если же мы изначально хотим заполнить не весь массив, а только его часть, то для этих целей используется метод read(), которому, кроме массива byte[], передаются еще два int значения. Первое – это позиция в массиве, с которой следует начать заполнение, второе – количество байт, которое нужно считать. Такой подход, когда для получения данных передается массив и два int числа – offset (смещение) и length (длина), является довольно

распространенным и часто встречается не только в пакете java.io.

При вызове методов read() возможно возникновение такой ситуации, когда запрашиваемые данные еще не готовы к считыванию. Например, если мы считываем данные, поступающие из сети, и они еще просто не пришли. В таком случае нельзя сказать, что данных больше нет, но и считать тоже нечего - выполнение останавливается на вызове метода read() и получается "зависание".

Чтобы узнать, сколько байт в потоке готово к считыванию, применяется метод available(). Этот метод возвращает значение типа int, которое показывает, сколько байт в потоке готово к считыванию. При этом не стоит путать количество байт, готовых к считыванию, с тем количеством байт, которые вообще можно будет считать из этого потока. Метод available() возвращает число – количество байт, именно на данный момент готовых к считыванию.

Когда работа с входным потоком данных окончена, его следует закрыть. Для этого вызывается метод **close**(). Этим вызовом будут освобождены все системные ресурсы, связанные с потоком.

Точно так же, как InputStream – это базовый класс для потоков ввода, класс OutputStream – это базовый класс для потоков вывода.

В классе **OutputStream** аналогичным образом определяются три метода **write**() – один принимающий в качестве параметра int, второй – byte[] и третий – byte[], плюс два int-числа. Все эти методы ничего не возвращают (void).

Метод **write**(int) является абстрактным и должен быть реализован в классах-наследниках. Этот метод принимает в качестве параметра int, но реально записывает в поток только byte – младшие 8 бит в двоичном представлении. Остальные 24 бита будут проигнорированы. В случае возникновения ошибки этот метод бросает java.io.IOException, как, впрочем, и большинство методов, связанных с вводом-выводом.

Для записи в поток сразу некоторого количества байт методу write() передается массив байт. Или, если мы хотим записать только часть массива, то передаем массив byte[] и два int-числа – отступ и количество байт для записи. Понятно, что если указать неверные параметры – например, отрицательный отступ, отрицательное количество байт для записи, либо если сумма отступ плюс длина будет больше длины массива, – во всех этих случаях кидается исключение IndexOutOfBoundsException.

Реализация потока может быть такой, что данные записываются не сразу, а хранятся некоторое время в памяти. Например, мы хотим записать в файл какие-то данные, которые получаем порциями по 10 байт, и так 200 раз подряд. В таком случае вместо 200 обращений к файлу удобней будет скопить все эти данные в памяти, а потом одним заходом записать все 2000 байт. То есть класс выходного потока может использовать некоторый внутренний механизм для буферизации (временного хранения перед отправкой) данных. Чтобы убедиться, что данные записаны в поток, а не хранятся в буфере, вызывается метод **flush**(), определенный в OutputStream. В этом классе его реализация пустая, но если какой-либо из наследников использует буферизацию данных, то этот метод должен быть в нем переопределен.

Когда работа с потоком закончена, его следует закрыть. Для этого вызывается метод **close**(). Этот метод сначала освобождает буфер (вызовом метода **flush**), после чего поток закрывается и освобождаются все связанные с ним системные ресурсы. Закрытый поток не может выполнять операции вывода и не может быть открыт заново. В классе OutputStream реализация метода close() не производит никаких действий.

Итак, классы **InputStream** и **OutputStream** определяют необходимые методы для работы с байтовыми потоками данных. Эти классы являются абстрактными. Их задача – определить общий интерфейс для классов, которые получают данные из различных источников. Такими источниками могут быть, например, массив байт, файл, строка и т.д. Все они, или, по крайней мере, наиболее распространенные, будут рассмотрены далее.

Классы-реализации потоков данных

2.1.2 Классы ByteArrayInputStream и ByteArrayOutputStream

Самый естественный и простой источник, откуда можно считывать байты, – это, конечно, массив байт. Класс **ByteArrayInputStream** представляет поток, считывающий данные из массива байт. Этот класс имеет конструктор, которому в качестве параметра передается массив byte[]. Соответственно, при вызове методов read() возвращаемые данные будут браться именно из этого массива. Например:

```
byte[] bytes = {1,-1,0};
ByteArrayInputStream in =
   new ByteArrayInputStream(bytes);
int readedInt = in.read(); // readedInt=1
System.out.println("first element read is: "
                    + readedInt);
readedInt = in.read();
// readedInt=255. Однако
// (byte)readedInt даст значение -1
System.out.println("second element read is: "
                   + readedInt);
readedInt = in.read(); // readedInt=0
System.out.println("third element read is: "
                   + readedInt);
Если запустить такую программу, на экране отобразится следующее:
first element read is: 1
second element read is: 255
third element read is: 0
```

При вызове метода read() данные считывались из массива bytes, переданного в конструктор ByteArrayInputStream. Обратите внимание, в данном примере второе считанное значение равно 255, а не -1, как можно было бы ожидать. Чтобы понять, почему это произошло, нужно вспомнить, что метод read считывает byte, но возвращает значение int, полученное добавлением необходимого числа нулей (в двоичном представлении). Байт, равный -1, в двоичном представлении имеет вид 11111111 и, соответственно, число типа int, получаемое приставкой 24-х нулей, равно 255 (в десятичной системе). Однако если явно привести его к byte, получим исходное значение.

Аналогично, для записи байт в массив применяется класс ByteArrayOutputStream. Этот класс использует внутри себя объект byte[], куда записывает данные, передаваемые при вызове методов write(). Чтобы получить записанные в массив данные, вызывается метод toByteArray().

Пример:

```
ByteArrayOutputStream out =
   new ByteArrayOutputStream();
out.write(10);
out.write(11);
byte[] bytes = out.toByteArray();
```

В этом примере в результате массив bytes будет состоять из двух элементов: 10 и 11.

Использовать классы ByteArrayInputStream и ByteArrayOutputStream может быть очень удобно, когда нужно проверить, что именно записывается в выходной поток. Например, при отладке и тестировании сложных процессов записи и чтения из потоков. Эти классы хороши тем, что позволяют сразу просмотреть результат и не нужно создавать ни файл, ни сетевое соединение, ни что-либо еще.

2.1.3 Классы FileInputStream и FileOutputStream

Класс FileInputStream используется для чтения данных из файла. Конструктор такого класса в качестве параметра принимает название файла, из которого будет производиться

считывание. При указании строки имени файла нужно учитывать, что она будет напрямую передана операционной системе, поэтому формат имени файла и пути к нему может различаться на разных платформах. Если при вызове этого конструктора передать строку, указывающую на несуществующий файл или каталог, то будет брошено java.io.FileNotFoundException. Если же объект успешно создан, то при вызове его методов read() возвращаемые значения будут считываться из указанного файла.

Для записи байт в файл используется класс FileOutputStream. При создании объектов этого класса, то есть при вызовах его конструкторов, кроме имени файла, также можно указать, будут ли данные дописываться в конец файла, либо файл будет перезаписан. Если указанный файл не существует, то сразу после создания FileOutputStream он будет создан. При вызовах методов write() передаваемые значения будут записываться в этот файл. По окончании работы необходимо вызвать метод close(), чтобы сообщить системе, что работа по записи файла закончена. Пример:

```
      byte[] bytesToWrite = {1, 2, 3};
      byte[] bytesReaded = new byte[10];
      String fileName = "d:\\test.txt";
      try {
         // Создать выходной поток
         FileOutputStream outFile = new FileOutputStream(fileName);
         System.out.println("Файл открыт для записи");
         // Записать массив
         outFile.write(bytesToWrite);
         System.out.println("Записано: " + bytesToWrite.length + "
байт");
         // По окончании использования должен быть закрыт
         outFile.close();
         System.out.println("Выходной поток закрыт");
         // Создать входной поток
         FileInputStream inFile = new FileInputStream(fileName);
         System.out.println("Файл открыт для чтения");
         // Узнать, сколько байт готово к считыванию
         int bytesAvailable = inFile.available();
         System.out.println("Готово к считыванию: " + bytesAvailable +
" байт");
         // Считать в массив
         int count = inFile.read(bytesReaded,0,bytesAvailable);
         System.out.println("Считано: " + count + " байт");
         for (i=0;i<count;i++)
            System.out.print(bytesReaded[i]+",");
         System.out.println();
         inFile.close();
         System.out.println("Входной поток закрыт");
      } catch (FileNotFoundException e) {
         System.out.println("Невозможно произвести запись в файл: " +
fileName);
      } catch (IOException e) {
         System.out.println("Ошибка ввода/вывода: " + e.toString());
      }
```

Пример 15.1.

```
      Результатом работы программы будет:
      Файл открыт для записи
      Записано: 3 байт
      Выходной поток закрыт
      Файл открыт для чтения
      Готово к считыванию: 3 байт
```

```
Считано: 3 байт
1,2,3,
Входной поток закрыт
```

Пример 15.2.

При работе с FileInputStream метод available() практически наверняка вернет длину файла, то есть число байт, сколько вообще из него можно считать. Но не стоит закладываться на это при написании программ, которые должны устойчиво работать на различных платформах,– метод available() возвращает число байт, которое может быть на данный момент считано без блокирования. Тот факт, что, скорее всего, это число и будет длиной файла, является всего лишь частным случаем работы на некоторых платформах.

В приведенном примере для наглядности закрытие потоков производилось сразу же после окончания их использования в основном блоке. Однако лучше закрывать потоки в finally блоке.

```
...
} finally {
try{inFile.close();}catch(IOException e){};
}
```

Такой подход гарантирует, что поток будет закрыт и будут освобождены все связанные с ним системные ресурсы.

2.1.4 PipedInputStream и PipedOutputStream

Классы PipedInputStream и PipedOutputStream характеризуются тем, что их объекты всегда используются в паре – к одному объекту PipedInputStream привязывается (подключается) один объект PipedOutputStream. Они могут быть полезны, если в программе необходимо организовать обмен данными между модулями (например, между потоками выполнения).

Эти классы применяются следующим образом: создается по объекту PipedInputStream и PipedOutputStream, после чего они могут быть соединены между собой. Один объект PipedOutputStream может быть соединен с ровно одним объектом PipedInputStream, и наоборот. Затем в объект PipedOutputStream записываются данные, после чего они могут быть считаны именно в подключенном объекте PipedInputStream. Такое соединение можно обеспечить либо вызовом метода connect() с передачей соответствующего объекта PipedI/OStream (будем так кратно обозначать пару классов, в данном случае PipedInputStream и PipedOutputStream), либо передать этот объект еще при вызове конструктора.

Использование связки PipedInputStream и PipedOutputStream показано в следующем примере:

```
try {
   int countRead = 0;
   byte[] toRead = new byte[100];
   PipedInputStream pipeIn = new PipedInputStream();
   PipedOutputStream pipeOut = new PipedOutputStream(pipeIn);
   // Считывать в массив, пока он полностью не будет заполнен
   while(countRead<toRead.length) {
      // Записать в поток некоторое количество байт
      for(int i=0; i<(Math.random()*10); i++) {
         pipeOut.write((byte)(Math.random()*127));
      }
      // Считать из потока доступные данные,
      // добавить их к уже считанным.
      int willRead = pipeIn.available();
      if(willRead+countRead>toRead.length)
      //Нужно считать только до предела массива
      willRead = toRead.length-countRead;
```

```
            countRead += pipeIn.read(toRead, countRead, willRead);
        }
    } catch (IOException e) {
        System.out.println ("Impossible IOException occur: ");
        e.printStackTrace();
    }
```

Пример 15.3.

Данный пример носит чисто демонстративный характер (в результате его работы массив toRead будет заполнен случайными числами). Более явно выгода от использования PipedI/OStream в основном проявляется при разработке многопоточного приложения. Если в программе запускается несколько потоков исполнения, организовать передачу данных между ними удобно с помощью этих классов. Для этого нужно создать связанные объекты PipedI/OStream, после чего передать ссылки на них в соответствующие потоки. Поток выполнения, в котором производится чтение данных, может содержать подобный код:

```
// inStream - объект класса PipedInputStream
try {
   while(true) {
   byte[] readedBytes = null;
   synchronized(inStream) {
      int bytesAvailable = inStream.available();
      readedBytes = new byte[bytesAvailable];
      inStream.read(readedBytes);
   }
   // обработка полученных данных из readedBytes
   // …
} catch(IOException e) {
/* IOException будет брошено, когда поток inStream, либо
связанный с ним PipedOutputStream, уже закрыт, и при этом
производится попытка считывания из inStream */
System.out.println("работа с потоком inStream завершена");
}
```

Пример 15.4.

Если с объектом inStream одновременно могут работать несколько потоков выполнения, то необходимо использовать блок synchronized (как и сделано в примере), который гарантирует, что в период между вызовами inStream.available() и inStream.read(…) ни в каком другом потоке выполнения не будет производиться считывание из inStream. Поэтому вызов inStream.read(readedBytes) не приведет к блокировке и все данные, готовые к считыванию, будут считаны.

2.1.5 StringBufferInputStream

Иногда бывает удобно работать с текстовой строкой String как с потоком байт. Для этого можно воспользоваться классом StringBufferInputStream. При создании объекта этого класса необходимо передать конструктору объект String. Данные, возвращаемые методом read(), будут считываться именно из этой строки. При этом символы будут преобразовываться в байты с потерей точности – старший байт отбрасывается (напомним, что символ char состоит из двух байт).

2.1.6 SequenceInputStream

Класс SequenceInputStream объединяет поток данных из других двух и более входных потоков. Данные будут вычитываться последовательно – сначала все данные из первого потока в списке, затем из второго, и так далее. Конец потока SequenceInputStream будет достигнут только тогда, когда будет достигнут конец потока, последнего в списке.

В этом классе имеется два конструктора – принимающий два потока и принимающий Enumeration (в котором, конечно, должны быть только экземпляры

InputStream и его наследников). Когда вызывается метод read(), SequenceInputStream пытается считать байт из текущего входного потока. Если в нем больше данных нет (считанное из него значение равно -1), у него вызывается метод close() и следующий входной поток становится текущим. Так продолжается до тех пор, пока не будут получены все данные из последнего потока. Если при считывании обнаруживается, что больше входных потоков нет, SequenceInputStream возвращает -1. Вызов метода close() у SequenceInputStream закрывает все содержащиеся в нем входные потоки.

Пример:

```
FileInputStream inFile1 = null;
FileInputStream inFile2 = null;
SequenceInputStream sequenceStream = null;
FileOutputStream outFile = null;
try {
   inFile1 = new FileInputStream("file1.txt");
   inFile2 = new FileInputStream("file2.txt");
   sequenceStream = new SequenceInputStream(inFile1, inFile2);
   outFile = new FileOutputStream("file3.txt");
   int readedByte = sequenceStream.read();
   while(readedByte!=-1){
      outFile.write(readedByte);
      readedByte = sequenceStream.read();
   }
} catch (IOException e) {
   System.out.println("IOException: " + e.toString());
} finally {
   try{sequenceStream.close();}catch(IOException e){};
   try{outFile.close();}catch(IOException e){};
}
```

Пример 15.5.

В результате выполнения этого примера в файл file3.txt будет записано содержимое файлов file1.txt и file2.txt – сначала полностью file1.txt, потом file2.txt. Закрытие потоков производится в блоке finally. Поскольку при вызове метода close() может возникнуть IOException, необходим try-catch блок. Причем, каждый вызов метода close() взят в отдельный try-catch блок - для того, чтобы возникшее исключение при закрытии одного потока не помешало закрытию другого. При этом нет необходимости закрывать потоки inFile1 и inFile2 – они будут автоматически закрыты при использовании в sequnceStream - либо когда в них закончатся данные, либо при вызове у sequenceStream метода close().

Объект SequenceInputStream можно было создать и другим способом: сначала получить объект Enumeration, содержащий все потоки, и передать его в конструктор SequenceInputStream:

```
Vector vector = new Vector();
vector.add(new StringBufferInputStream("Begin file1\n"));
vector.add(new FileInputStream("file1.txt"));
vector.add(new StringBufferInputStream("\nEnd of file1, begin
file2\n"));
vector.add(new FileInputStream("file2.txt"));
vector.add(new StringBufferInputStream("\nEnd of file2"));
Enumeration enum = vector.elements();
sequenceStream = new SequenceInputStream(enum);
```

Пример 15.6.

Если заменить в предыдущем примере инициализацию sequenceStream на приведенную здесь, то в файл file3.txt, кроме содержимого файлов file1.txt и file2.txt, будут записаны еще три строки – одна в начале файла, одна между содержимым файлов file1.txt и file2.txt и еще одна в конце file3.txt.

2.1.7 Классы FilterInputStream и FilterOutputStream и их наследники

Задачи, возникающие при вводе/выводе весьма разнообразны - это может быть считывание байт из файлов, объектов из файлов, объектов из массивов, буферизованное считывание строк из массивов и т.д. В такой ситуации решение с использованием простого наследования приводит к возникновению слишком большого числа подклассов. Более эффективно применение надстроек (в ООП этот шаблон называется адаптер) Надстройки – наложение дополнительных объектов для получения новых свойств и функций. Таким образом, необходимо создать несколько дополнительных объектов – адаптеров к классам ввода/вывода. В java.io их еще называют фильтрами. При этом надстройка-фильтр включает в себя интерфейс объекта, на который надстраивается, поэтому может быть, в свою очередь, дополнительно надстроена.

В java.io интерфейс для таких надстроек ввода/вывода предоставляют классы FilterInputStream (для входных потоков) и FilterOutputStream (для выходных потоков). Эти классы унаследованы от основных базовых классов ввода/вывода – InputStream и OutputStream, соответственно. Конструктор FilterInputStream принимает в качестве параметра объект InputStream и имеет модификатор доступа protected.

Классы FilterI/OStream являются базовыми для надстроек и определяют общий интерфейс для надстраиваемых объектов. Потоки-надстройки не являются источниками данных. Они лишь модифицируют (расширяют) работу надстраиваемого потока.

2.1.8 BufferedInputStream и BufferedOutputStream

На практике при считывании с внешних устройств ввод данных почти всегда необходимо буферизировать. Для буферизации данных служат классы BufferedInputStream и BufferedOutputStream.

BufferedInputStream содержит массив байт, который служит буфером для считываемых данных. То есть когда байты из потока считываются либо пропускаются (метод skip()), сначала заполняется буферный массив, причем, из надстраиваемого потока загружается сразу много байт, чтобы не требовалось обращаться к нему при каждой операции read или skip. Также класс BufferedInputStream добавляет поддержку методов mark() и reset(). Эти методы определены еще в классе InputStream, но там их реализация по умолчанию бросает исключение IOException. Метод mark() запоминает точку во входном потоке, а вызов метода reset() приводит к тому, что все байты, полученные после последнего вызова mark(), будут считываться повторно, прежде, чем новые байты начнут поступать из надстроенного входного потока.

BufferedOutputStream предоставляет возможность производить многократную запись небольших блоков данных без обращения к устройству вывода при записи каждого из них. Сначала данные записываются во внутренний буфер. Непосредственное обращение к устройству вывода и, соответственно, запись в него, произойдет, когда буфер заполнится. Инициировать передачу содержимого буфера на устройство вывода можно и явным образом, вызвав метод flush(). Так же буфер освобождается перед закрытием потока. При этом будет закрыт и надстраиваемый поток (так же поступает BufferedInputStream).

Следующий пример наглядно демонстрирует повышение скорости считывания данных из файла с использованием буфера:

```
try {
   String fileName = "d:\\file1";
   InputStream inStream = null;
   OutputStream outStream = null;

   //Записать в файл некоторое количество байт
   long timeStart = System.currentTimeMillis();
```

```
         outStream = new FileOutputStream(fileName);
         outStream = new BufferedOutputStream(outStream);
         for(int i=1000000; --i>=0;) {
            outStream.write(i);
         }
         long time = System.currentTimeMillis() - timeStart;
         System.out.println("Writing time: " + time + " millisec");
         outStream.close();

         // Определить время считывания без буферизации
         timeStart = System.currentTimeMillis();
         inStream = new FileInputStream(fileName);
         while(inStream.read()!=-1){
         }
         time = System.currentTimeMillis() - timeStart;
         inStream.close();
         System.out.println("Direct read time: " + (time) + "
millisec");
         // Теперь применим буферизацию
         timeStart = System.currentTimeMillis();
         inStream = new FileInputStream(fileName);
         inStream = new BufferedInputStream(inStream);
         while(inStream.read()!=-1){
         }
         time = System.currentTimeMillis() - timeStart;
         inStream.close();
         System.out.println("Buffered read time: " + (time) + "
millisec");
      } catch (IOException e) {
         System.out.println("IOException: " + e.toString());
         e.printStackTrace();
      }
```

Пример 15.7.

Результатом могут быть, например, такие значения:

Writing time: 359 millisec

Direct read time: 6546 millisec

Buffered read time: 250 millisec

Пример 15.8.

В данном случае не производилось никаких дополнительных вычислений, занимающих процессорное время, только запись и считывание из файла. При этом считывание с использованием буфера заняло в 10 (!) раз меньше времени, чем аналогичное без буферизации. Для более быстрого выполнения программы запись в файл производилась с буферизацией, однако ее влияние на скорость записи нетрудно проверить, убрав из программы строку, создающую BufferedOutputStream.

Классы BufferedI/OStream добавляют только внутреннюю логику обработки запросов, но не добавляют никаких новых методов. Следующие два фильтра предоставляют некоторые дополнительные возможности для работы с потоками.

2.1.9 LineNumberInputStream

Класс LineNumberInputStream во время чтения данных производит подсчет, сколько строк было считано из потока. Номер строки, на которой в данный момент происходит чтение, можно узнать путем вызова метода getLineNumber(). Также можно и перейти к определенной строке вызовом метода setLineNumber(int lineNumber).

Под строкой при этом понимается набор байт, оканчивающийся либо '\n', либо '\r', либо их комбинацией '\r\n', именно в этой последовательности.

Аналогичный класс для исходящего потока отсутствует. LineNumberInputStream, начиная с версии 1.1, объявлен deprecated, то есть использовать его не рекомендуется. Его заменил класс LineNumberReader (рассматривается ниже), принцип работы которого точно такой же.

2.1.10 PushBackInputStream

Этот фильтр позволяет вернуть во входной поток считанные из него данные. Такое действие производится вызовом метода unread(). Понятно, что обеспечивается подобная функциональность за счет наличия в классе специального буфера – массива байт, который хранит считанную информацию. Если будет произведен откат (вызван метод unread), то во время следующего считывания эти данные будут выдаваться еще раз как только полученные. При создании объекта можно указать размер буфера.

2.1.11 PrintStream

Этот класс используется для конвертации и записи строк в байтовый поток. В нем определен метод print(…), принимающий в качестве аргумента различные примитивные типы Java, а также тип Object. При вызове передаваемые данные будут сначала преобразованы в строку вызовом метода String.valueOf(), после чего записаны в поток. Если возникает исключение, оно обрабатывается внутри метода print и дальше не бросается (узнать, произошла ли ошибка, можно с помощью метода checkError()). При записи символов в виде байт используется кодировка, принятая по умолчанию в операционной системе (есть возможность задать ее явно при запуске JVM).

Этот класс также является deprecated, поскольку работа с кодировками требует особого подхода (зачастую у двухбайтовых символов Java старший байт просто отбрасывается). Поэтому в версии Java 1.1 появился дополнительный набор классов, основывающийся на типах Reader и Writer. Они будут рассмотрены позже. В частности, вместо PrintStream теперь рекомендуется применять PrintWriter. Однако старый класс продолжает активно использоваться, поскольку статические поля out и err класса System имеют именно это тип.

2.1.12 DataInputStream и DataOutputStream

До сих пор речь шла только о считывании и записи в поток данных в виде byte. Для работы с другими примитивными типами данных Java определены интерфейсы DataInput и DataOutput и их реализации – классы-фильтры DataInputStream и DataOutputStream. Их место в иерархии классов ввода/вывода можно увидеть на рис.15.1.

Интерфейсы DataInput и DataOutput определяют, а классы DataInputStream и DataOutputStream, соответственно, реализуют методы считывания и записи значений всех примитивных типов. При этом происходит конвертация этих данных в набор byte и обратно. Чтение необходимо организовать так, чтобы данные запрашивались в виде тех же типов, в той же последовательности, как и производилась запись. Если записать, например, int и long, а потом считывать их как short, чтение будет выполнено корректно, без исключительных ситуаций, но числа будут получены совсем другие.

Это наглядно показано в следующем примере:

```
try {
   ByteArrayOutputStream out = new ByteArrayOutputStream();
   DataOutputStream outData = new DataOutputStream(out);
   outData.writeByte(128);
   // этот метод принимает аргумент int, но записывает
   // лишь младший байт
   outData.writeInt(128);
   outData.writeLong(128);
```

```
            outData.writeDouble(128);
            outData.close();
            byte[] bytes = out.toByteArray();
            InputStream in = new ByteArrayInputStream(bytes);
            DataInputStream inData = new DataInputStream(in);
            System.out.println("Чтение в правильной последовательности:
");
            System.out.println("readByte: " + inData.readByte());
            System.out.println("readInt: " + inData.readInt());
            System.out.println("readLong: " + inData.readLong());
            System.out.println("readDouble: " + inData.readDouble());
            inData.close();
            System.out.println("Чтение в измененной последовательности:");
            in = new ByteArrayInputStream(bytes);
            inData = new DataInputStream(in);
            System.out.println("readInt: " + inData.readInt());
            System.out.println("readDouble: " + inData.readDouble());
            System.out.println("readLong: " + inData.readLong());
            inData.close();
            } catch (Exception e) {
               System.out.println("Impossible IOException occurs: " +
                  e.toString());
            e.printStackTrace();
         }
```

Пример 15.9.

Результат выполнения программы:

Чтение в правильной последовательности:
readByte: -128
readInt: 128
readLong: 128
readDouble: 128.0
Чтение в измененной последовательности:
readInt: -2147483648
readDouble: -0.0
readLong: -9205252085229027328

Итак, значение любого примитивного типа может быть передано и считано из потока данных.

2.2 Сериализация объектов (serialization)

Для объектов процесс преобразования в последовательность байт и обратно организован несколько сложнее – объекты имеют различную структуру, хранят ссылки на другие объекты и т.д. Поэтому такая процедура получила специальное название - сериализация (serialization), обратное действие, – то есть воссоздание объекта из последовательности байт – десериализация.

Поскольку сериализованный объект – это последовательность байт, которую можно легко сохранить в файл, передать по сети и т.д., то и объект затем можно восстановить на любой машине, вне зависимости от того, где проводилась сериализация. Разумеется, Java позволяет не задумываться при этом о таких факторах, как, например, используемая операционная система на машине-отправителе и получателе. Такая гибкость обусловила широкое применение сериализации при создании распределенных приложений, в том числе и корпоративных (enterprise) систем.

2.2.1 Стандартная сериализация

Для представления объектов в виде последовательности байт определены

унаследованные от DataInput и DataOutput интерфейсы ObjectInput и ObjectOutput, соответственно. В java.io имеются реализации этих интерфейсов – классы ObjectInputStream и ObjectOutputStream.

Эти классы используют стандартный механизм сериализации, который предлагает JVM. Для того, чтобы объект мог быть сериализован, класс, от которого он порожден, должен реализовывать интерфейс java.io.Serializable. В этом интерфейсе не определен ни один метод. Он нужен лишь для указания, что объекты класса могут участвовать в сериализации. При попытке сериализовать объект, не имеющий такого интерфейса, будет брошен java.io.NotSerializableException.

Чтобы начать сериализацию объекта, нужен выходной поток OutputStream, в который и будет записываться сгенерированная последовательность байт. Этот поток передается в конструктор ObjectOutputStream. Затем вызовом метода writeObject() объект сериализуется и записывается в выходной поток. Например:

```
ByteArrayOutputStream os =
   new ByteArrayOutputStream();
Object objSave = new Integer(1);
ObjectOutputStream oos =
   new ObjectOutputStream(os);
oos.writeObject(objSave);
```

Чтобы увидеть, во что превратился объект objSave, можно просмотреть содержимое массива:

```
byte[] bArray = os.toByteArray();
```

А чтобы восстановить объект, его нужно десериализовать из этого массива:

```
ByteArrayInputStream is =
   new ByteArrayInputStream(bArray);
ObjectInputStream ois =
   new ObjectInputStream(is);
Object objRead = ois.readObject();
```

Теперь можно убедиться, что восстановленный объект идентичен исходному:

```
System.out.println("readed object is: " +
                    objRead.toString());
System.out.println("Object equality is: " +
   (objSave.equals(objRead)));
System.out.println("Reference equality is: " +
                   (objSave==objRead));
```

Результатом выполнения приведенного выше кода будет:

```
readed object is: 1
Object equality is: true
Reference equality is: false
```

Как мы видим, восстановленный объект не совпадает с исходным (что очевидно – ведь восстановление могло происходить и на другой машине), но равен сериализованному по значению.

Как обычно, для упрощения в примере была опущена обработка ошибок. Однако, сериализация (десериализация) объектов довольно сложная процедура, поэтому возникающие сложности не всегда очевидны. Рассмотрим основные исключения, которые может генерировать метод readObject() класса ObjectInputStream.

Предположим, объект некоторого класса TestClass был сериализован и передан по сети на другую машину для восстановления. Может случиться так, что у считывающей JVM на локальном диске не окажется описания этого класса (файл TestClass.class). Поскольку стандартный механизм сериализации записывает в поток байт лишь состояние объекта, для успешной десериализации необходимо наличие описание класса. В результате будет брошено исключение ClassNotFoundException.

Причина появления java.io.StreamCorruptedException вполне очевидна из названия – неправильный формат входного потока. Предположим, происходит попытка считать

сериализованный объект из файла. Если этот файл испорчен (для эксперимента можно открыть его в текстовом редакторе и исправить несколько символов), то стандартная процедура десериализации даст сбой. Эта же ошибка возникнет, если считать некоторое количество байт (с помощью метода read) непосредственно из надстраиваемого потока InputStream. В таком случае ObjectInputStream снова обнаружит сбой в формате данных и будет брошено исключение java.io.StreamCorruptedException.

Поскольку ObjectOutput наследуется от DataOutput, ObjectOutputStream может быть использован для последовательной записи нескольких значений как объектных, так и примитивных типов в произвольной последовательности. Если при считывании будет вызван метод readObject, а в исходном потоке следующим на очереди записано значение примитивного типа, будет брошено исключение java.io.OptionalDataException. Очевидно, что для корректного восстановления данных из потока их нужно считывать именно в том порядке, в каком были записаны.

2.2.2 Восстановление состояния

Итак, сериализация объекта заключается в сохранении и восстановлении состояния объекта. В Java в большинстве случаев состояние описывается значениями полей объекта. Причем, что важно, не только тех полей, которые были явно объявлены в классе, от которого порожден объект, но и унаследованных полей.

Предположим, мы бы попытались своими силами реализовать стандартный механизм сериализации. Нам передается выходной поток, в который нужно записать состояние нашего объекта. С помощью DataOutput интерфейса можно легко сохранить значения всех доступных полей (будем для простоты считать, что они все примитивного типа). Однако в большинстве случаев в родительских классах могут быть объявлены недоступные нам поля (например, private). Тем не менее, такие поля, как правило, играют важную роль в определении состояния объекта, так как они могут влиять на результат работы унаследованных методов. Как же сохранить их значения?

С другой стороны, не меньшей проблемой является восстановление объекта. Как говорилось раньше, объект может быть создан только вызовом его конструктора. У класса, от которого порожден десериализуемый объект, может быть несколько конструкторов, причем, некоторые из них, или все, могут иметь аргументы. Какой из них вызвать? Какие значения передать в качестве аргументов?

После создания объекта необходимо установить считанные значения его полей. Однако многие классы имеют специальные set-методы для этой цели. В таких методах могут происходить проверки, могут меняться значения вспомогательных полей. Пользоваться ли этими методами? Если их несколько, то как выбрать правильный и какие параметры ему передать? Снова возникает проблема работы с недоступными полями, полученными по наследству. Как же в стандартном механизме сериализации решены все эти вопросы?

Во-первых, рассмотрим подробнее работу с интерфейсом Serializable. Заметим, что класс Object не реализует этот интерфейс. Таким образом, существует два варианта – либо сериализуемый класс наследуется от Serializable-класса, либо нет. Первый вариант довольно прост. Если родительский класс уже реализовал интерфейс Serializable, то наследникам это свойство передается автоматически, то есть все объекты, порожденные от такого класса, или любого его наследника, могут быть сериализованы.

Если же наш класс впервые реализует Serializable в своей ветке наследования, то его суперкласс должен отвечать специальному требованию – у него должен быть доступный конструктор без параметров. Именно с помощью этого конструктора будет создан десериализуемый объект и будут проинициализированы все поля, унаследованные от классов, не наследующих Serializable.

Рассмотрим пример:

```
// Родительский класс, не реализующий Serializable
```

```
public class Parent {
   public String firstName;
   private String lastName;
   public Parent(){
      System.out.println("Create Parent");
      firstName="old_first";
      lastName="old_last";
   }
   public void changeNames() {
      firstName="new_first";
      lastName="new_last";
   }
   public String toString() {
      return
super.toString()+",first="+firstName+",last="+lastName;
   }
}
// Класс Child, впервые реализовавший Serializable
public class Child extends Parent implements Serializable {
   private int age;
   public Child(int age) {
      System.out.println("Create Child");
      this.age=age;
   }
   public String toString() {
      return super.toString()+",age="+age;
   }
}
// Наследник Serializable-класса
public class Child2 extends Child {
   private int size;
   public Child2(int age, int size) {
      super(age);
      System.out.println("Create Child2");
      this.size=size;
   }
   public String toString() {
      return super.toString()+",size="+size;
   }
}
// Запускаемый класс для теста
public class Test {
   public static void main(String[] arg) {
      try {
         FileOutputStream fos=new FileOutputStream("output.bin");
         ObjectOutputStream oos=new ObjectOutputStream(fos);
         Child c=new Child(2);
         c.changeNames();
         System.out.println(c);
         oos.writeObject(c);
         oos.writeObject(new Child2(3, 4));
         oos.close();
         System.out.println("Read objects:");
         FileInputStream fis=new FileInputStream("output.bin");
         ObjectInputStream ois=new ObjectInputStream(fis);
         System.out.println(ois.readObject());
         System.out.println(ois.readObject());
         ois.close();
```

```
            } catch (Exception e) { // упрощенная обработка для
краткости
               e.printStackTrace();
            }
         }
      }
```

Пример 15.10.

В этом примере объявлено 3 класса. Класс Parent не реализует Serializable и, следовательно, не может быть сериализован. В нем объявлено 2 поля, которые при создании получают значения, содержащие слово "old" ("старый"). Кроме этого, объявлен метод, позволяющий модифицировать эти поля. Он выставляет им значения, содержащие слово "new" ("новый'). Также переопределен метод toString(), чтобы дать возможность узнать значения этих полей.

Поскольку класс Parent имеет доступный конструктор по умолчанию, его наследник может реализовать интерфейс Serializable. Обратите внимание, что у самого класса Child такого конструктора уже нет. Также объявлено поле и модифицирован метод toString().

Наконец, класс Child2 наследуется от Child, а потому автоматически является допустимым для сериализации. Аналогично, имеет новое поле, значение которого отображает toString().

Запускаемый класс Test сериализует в файл output.bin два объекта. Обратите внимание, что у первого из них предварительно вызывается метод changeNames(), который модифицирует значения полей, унаследованных от класса Parent.

Результат выполнения примера:

```
Create Parent
Create Child
Child@ad3ba4,first=new_first,last=new_last,age=2
Create Parent
Create Child
Create Child2
Read objects:
Create Parent
Child@723d7c,first=old_first,last=old_last,age=2
Create Parent
Child2@22c95b,first=old_first,last=old_last,age=3,size=4
```

Пример 15.11.

Во всех конструкторах вставлена строка, выводящая сообщение на консоль. Так можно отследить, какие конструкторы вызываются во время десериализации. Видно, что для объектов, порожденных от Serializable-классов, конструкторы не вызываются вовсе. Идет обращение лишь к конструктору без параметров не-Serializable-суперкласса.

Сравним значения полей первого объекта и его копии, полученной десериализацией. Поля, унаследованные от не-Serializable-класса (firstName, lastName), не восстановились. Они имеют значения, полученные в конструкторе Parent без параметров. Поля, объявленные в Serializable-классе, свои значения сохранили. Это верно и для второго объекта – собственные поля Child2 и унаследованные от Child имеют точно такие же значения, что и до сериализации. Их значения были записаны, а потом считаны и напрямую установлены из потока данных.

Иногда в классе есть поля, которые не должны участвовать в сериализации. Тому может быть несколько причин. Например, это поле малосущественно (временная переменная) и сохранять его нет необходимости. Если сериализованный объект передается по сети, то исключение такого поля из сериализации позволяет уменьшить нагрузку на сеть и ускорить работу приложения.

Некоторые поля хранят значения, которые не будут иметь смысла при пересылке объекта на другую машину, или при воссоздании его спустя какое-то время. Например,

сетевое соединение, или подключение к базе данных, в таких случаях нужно устанавливать заново.

Затем, в объекте может храниться конфиденциальная информация, например, пароль. Если такое поле будет сериализовано и передано по сети, его значение может быть перехвачено и прочитано, или даже подменено.

Для исключения поля объекта из сериализации его необходимо объявить с модификатором transient. Например, следующий класс:

```
class Account implements
      java.io.Serializable {
   private String name;
   private String login;
   private transient String password;
   /* объявление других элементов класса
   ...
   */
}
```

У такого класса поле password в сериализации участвовать не будет и при восстановлении оно получит значение по умолчанию (в данном случае null).

Особого внимания требуют статические поля. Поскольку они принадлежат классу, а не объекту, они не участвуют в сериализации. При восстановлении объект будет работать с таким значением static-поля, которое уже установлено для его класса в этой JVM.

2.2.3 Граф сериализации

До этого мы рассматривали объекты, которые имеют поля лишь примитивных типов. Если же сериализуемый объект ссылается на другие объекты, их также необходимо сохранить (записать в поток байт), а при десериализации – восстановить. Эти объекты, в свою очередь, также могут ссылаться на следующие объекты. При этом важно, что если несколько ссылок указывают на один и тот же объект, то этот объект должен быть сериализован лишь однажды, а при восстановлении все ссылки должны вновь указывать на него одного. Например, сериализуемый объект A ссылается на объекты B и C, каждый из которых, в свою очередь, ссылается на один и тот же объект D. После деесериализации не должно возникать ситуации, когда B ссылается на D1, а C – на D2, где D1 и D2 – равные, но все же различные объекты.

Для организации такого процесса стандартный механизм сериализации строит граф, включающий в себя все участвующие объекты и ссылки между ними. Если очередная ссылка указывает на некоторый объект, сначала проверяется – нет ли такого объекта в графе. Если есть – объект второй раз не сериализуется. Если нет – новый объект добавляется в граф.

При построении графа может встретиться объект, порожденный от класса, не реализующего интерфейс Serializable. В этом случае сериализация прерывается, генерируется исключение java.io.NotSerializableException.

Рассмотрим пример:

```
import java.io.*;
class Point implements Serializable {
   double x;
   double y;
   public Point(double x, double y) {
      this.x = x;
      this.y = y;
   }
   public String toString() {
      return "("+x+","+y+") reference="+super.toString();
   }
}
```

```
      class Line implements Serializable {
         Point point1;
         Point point2;
         int index;
         public Line() {
            System.out.println("Constructing empty line");
         }
         Line(Point p1, Point p2, int index) {
            System.out.println("Constructing line: " + index);
            this.point1 = p1;
            this.point2 = p2;
            this.index = index;
         }
         public int getIndex() { return index; }
         public void setIndex(int newIndex) { index = newIndex; }
         public void printInfo() {
            System.out.println("Line: " + index);
            System.out.println(" Object reference: " +
super.toString());
            System.out.println(" from point "+point1);
            System.out.println(" to point "+point2);
         }
      }
      public class Main {
         public static void main(java.lang.String[] args) {
            Point p1 = new Point(1.0,1.0);
            Point p2 = new Point(2.0,2.0);
            Point p3 = new Point(3.0,3.0);
            Line line1 = new Line(p1,p2,1);
            Line line2 = new Line(p2,p3,2);
            System.out.println("line 1 = " + line1);
            System.out.println("line 2 = " + line2);
            String fileName = "d:\\file";
            try{
               // записываем объекты в файл
               FileOutputStream os = new FileOutputStream(fileName);
               ObjectOutputStream oos = new ObjectOutputStream(os);
               oos.writeObject(line1);
               oos.writeObject(line2);
               // меняем состояние line1 и записываем его еще раз
               line1.setIndex(3);
               //oos.reset();
               oos.writeObject(line1);
               // закрываем потоки
               // достаточно закрыть только поток-надстройку
               oos.close();
               // считываем объекты
               System.out.println("Read objects:");
               FileInputStream is = new FileInputStream(fileName);
               ObjectInputStream ois = new ObjectInputStream(is);
               for (int i=0; i<3; i++) { // Считываем 3 объекта
                  Line line = (Line)ois.readObject();
                  line.printInfo();
               } ois.close();
            } catch(ClassNotFoundException e) {
               e.printStackTrace();
            } catch(IOException e) {
               e.printStackTrace();
```

```
        }
    }
}
```

Пример 15.12.

В этой программе работа идет с классом Line (линия), который имеет 2 поля типа Point (линия описывается двумя точками). Запускаемый класс Main создает два объекта класса Line, причем, одна из точек у них общая. Кроме этого, линия имеет номер (поле index). Созданные линии (номера 1 и 2) записываются в поток, после чего одна из них получает новый номер (3) и вновь сериализуется.

Выполнение этой программы приведет к выводу на экран примерно следующего:

```
Constructing line: 1
Constructing line: 2
line 1 = Line@7d39
line 2 = Line@4ec
Read objects:
Line: 1
   Object reference: Line@331e
   from point (1.0,1.0) reference=Point@36bb
   to point (2.0,2.0) reference=Point@386e
Line: 2
   Object reference: Line@6706
   from point (2.0,2.0) reference=Point@386e
   to point (3.0,3.0) reference=Point@68ae
Line: 1
   Object reference: Line@331e
   from point (1.0,1.0) reference=Point@36bb
   to point (2.0,2.0) reference=Point@386e
```

Пример 15.13.

Из примера видно, что после восстановления у линий сохраняется общая точка, описываемая одним и тем же объектом (хеш-код 386e).

Третий записанный объект идентичен первому, причем, совпадают даже объектные ссылки. Несмотря на то, что при записи третьего объекта значение index было изменено на 3, в десериализованном объекте оно осталось равным 1. Так произошло потому, что объект, описывающий первую линию, уже был задействован в сериализации и, встретившись во второй раз, повторно записан не был.

Чтобы указать, что сеанс сериализации завершен, и получить возможность передавать измененные объекты, у ObjectOutputStream нужно вызвать метод reset(). В рассматриваемом примере для этого достаточно убрать комментарий в строке

//oos.reset();

Если теперь запустить программу, то можно увидеть, что третий объект получит номер 3.

```
Constructing line: 1
Constructing line: 2
line 1 = Line@ea2dfe
line 2 = Line@7182c1
Read objects:
Line: 1
   Object reference: Line@a981ca
   from point (1.0,1.0) reference=Point@1503a3
   to point (2.0,2.0) reference=Point@a1c887
Line: 2
   Object reference: Line@743399
   from point (2.0,2.0) reference=Point@a1c887
   to point (3.0,3.0) reference=Point@e7b241
Line: 3
   Object reference: Line@67d940
```

```
from point (1.0,1.0) reference=Point@e83912
to point (2.0,2.0) reference=Point@fae3c6
```

Пример 15.14.

Однако это будет уже новый объект, ссылка на который отличается от первой считанной линии. Более того, обе точки будут также описываться новыми объектами. То есть в новом сеансе все объекты были записаны, а затем восстановлены заново.

Расширение стандартной сериализации

Некоторым сложно организованным классам требуется особый подход для сериализации. Для расширения стандартного механизма можно объявить в классе два метода с точно такой сигнатурой:

```
private void writeObject(
                java.io.ObjectOutputStream out)
   throws IOException;
private void readObject(
                java.io.ObjectInputStream in)
   throws IOException, ClassNotFoundException;
```

Если в классе объявлены такие методы, то при сериализации объекта для записи его состояния будет вызван writeObject, который должен сгенерировать последовательность байт и записать ее в поток out, полученный в качестве аргумента. При этом можно вызвать стандартный механизм записи объекта путем вызова метода

```
out.defaultWriteObject();
```

Этот метод запишет все не-transient и не-static поля в поток данных.

В свою очередь, при десериализации метод readObject должен считать данные из потока in (также полученного в качестве аргумента) и восстановить значения полей класса. При реализации этого метода можно обратиться к стандартному механизму с помощью метода:

```
in.defaultReadObject();
```

Этот метод считывает описание объекта из потока и присваивает значения соответствующих полей в текущем объекте.

Если же процедура сериализации в корне отличается от стандартной, то для таких классов предназначен альтернативный интерфейс java.io.Externalizable.

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 объявлены 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.

Метод writeExternal имеет сигнатуру:

```
void writeExternal(ObjectOutput out)
   throws IOException;
```

Для сохранения состояния вызываются методы ObjectOutput, с помощью которых можно записать как примитивные, так и объектные значения. Для корректной работы в соответствующем методе

```
void readExternal(ObjectInput in)
   throws IOException,ClassNotFoundException;
```

эти значения должны быть считаны в том же самом порядке.

2.2.4 Классы Reader и Writer и их наследники

Рассмотренные классы – наследники InputStream и OutputStream – работают с байтовыми данными. Если с их помощью записывать или считывать текст, то сначала необходимо сопоставить каждому символу его числовой код. Такое соответствие называется кодировкой.

Известно, что Java использует кодировку Unicode, в которой символы представляются двухбайтовым кодом. Байтовые потоки зачастую работают с текстом упрощенно – они просто отбрасывают старший байт каждого символа. В реальных же приложениях могут использовать различные кодировки (даже для русского языка их существует несколько). Поэтому в версии Java 1.1 появился дополнительный набор классов, основывающийся на типах Reader и Writer. Их иерархия представлена на рис. 15.2.

Эта иерархия очень схожа с аналогичной для байтовых потоков InputStream и OutputStream. Главное отличие между ними – Reader и Writer работают с потоком символов (char). Только чтение массива символов в Reader описывается методом read(char[]), а запись в Writer – write(char[]).

В таблице 15.1 приведены соответствия классов для байтовых и символьных потоков.

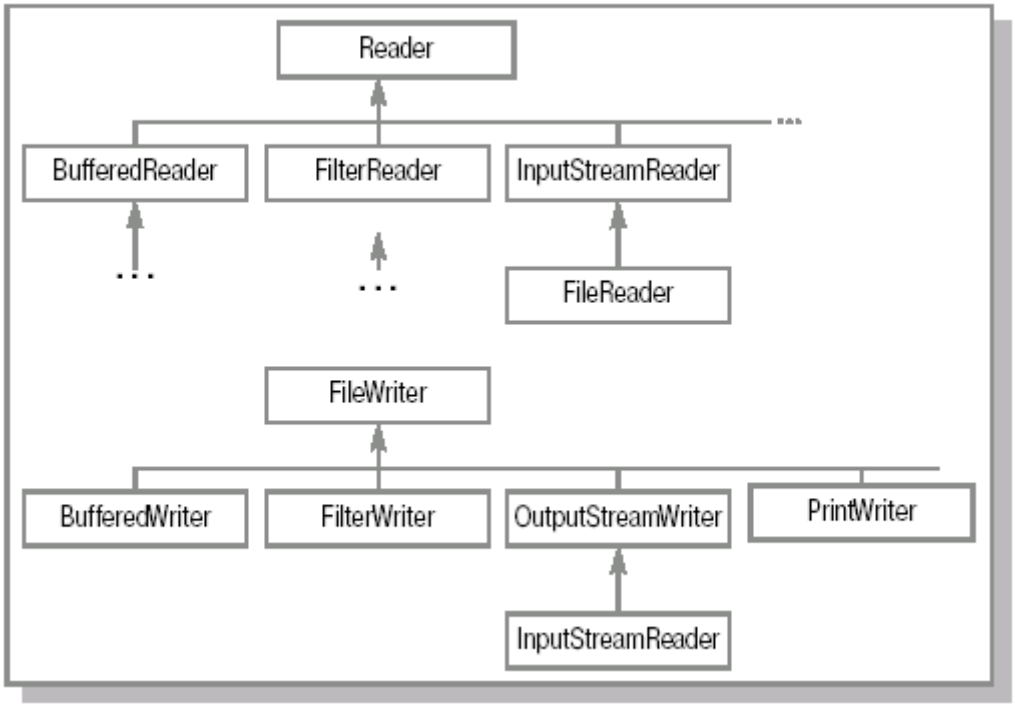

Рис. 15.2. Иерархия классов Reader и Writer.

Таблица 15.1. Соответствие классов для байтовых и символьных потоков.

Байтовый поток	Символьный поток
InputStream	Reader
OutputStream	Writer
ByteArrayInputStream	CharArrayReader
ByteArrayOutputStream	CharArrayWriter
Нет аналога	InputStreamReader
Нет аналога	OutputStreamWriter
FileInputStream	FileReader
FileOutputStream	FileWriter
FilterInputStream	FilterReader
FilterOutputStream	FilterWriter
BufferedInputStream	BufferedReader
BufferedOutputStream	BufferedWriter
PrintStream	PrintWriter
DataInputStream	Нет аналога
DataOutputStream	Нет аналога
ObjectInputStream	Нет аналога
ObjectOutputStream	Нет аналога

PipedInputStream	PipedReader
PipedOutputStream	PipedWriter
StringBufferInputStream	StringReader
Нет аналога	StringWriter
LineNumberInputStream	LineNumberReader
PushBackInputStream	PushBackReader
SequenceInputStream	Нет аналога

Как видно из таблицы, различия крайне незначительны и предсказуемы.

Например, конечно же, отсутствует преобразование в символьное представление примитивных типов Java и объектов (DataInput/Output, ObjectInput/Output). Добавлены классы-мосты, преобразующие символьные потоки в байтовые: InputStreamReader и OutputStreamWriter . Именно на их основе реализованы FileReader и FileWriter. Метод available() класса InputStream в классе Reader отсутствует, он заменен методом ready(), возвращающим булевое значение, – готов ли поток к считыванию (то есть будет ли считывание произведено без блокирования).

В остальном же использование символьных потоков идентично работе с байтовыми потоками. Так, программный код для записи символьных данных в файл будет выглядеть примерно следующим образом:

```
String fileName = "d:\\file.txt";
//Строка, которая будет записана в файл
String data = "Some data to be written and read.\n";
try{
   FileWriter fw = new FileWriter(fileName);
   BufferedWriter bw = new BufferedWriter(fw);
   System.out.println("Write some data to file: " + fileName);
   // Несколько раз записать строку
   for(int i=(int)(Math.random()*10);--i>=0;)
      bw.write(data);
   bw.close();
   // Считываем результат
   FileReader fr = new FileReader(fileName);
   BufferedReader br = new BufferedReader(fr);
   String s = null;
   int count = 0;
   System.out.println("Read data from file: " + fileName);
   // Считывать данные, отображая на экран
   while((s=br.readLine())!=null)
      System.out.println("row " + ++count + " read:" + s);
   br.close();
} catch(Exception e) {
   e.printStackTrace();
}
```

Пример 15.15.

Классы-мосты InputStreamReader и OutputStreamWriter при преобразовании символов также используют некоторую кодировку. Ее можно задать, передав в конструктор в качестве аргумента ее название. Если оно не будет соответствовать никакой из известных кодировок, будет брошено исключение UnsupportedEncodingException. Вот некоторые из корректных значений этого аргумента (чувствительного к регистру!) для распространенных кодировок: "Cp1251", "UTF-8", "8859_1" и т.д.

2.2.5 Класс StreamTokenizer

Экземпляр StreamTokenizer создается поверх существующего объекта, либо InputStream, либо Reader. Как и java.util.StringTokenizer, этот класс позволяет разбивать

данные на лексемы (token), выделяемые из потока по определенным свойствам. Поскольку работа ведется со словами, конструктор, принимающий InputStream, объявлен как deprecated (предлагается оборачивать байтовый поток классом InputStreamReader и вызывать второй конструктор). Общий принцип работы такой же, как и у StringTokenizer, – задаются параметры разбиения, после чего вызывается метод nextToken(), пока не будет достигнут конец потока. Способы задания разбиения у StreamTokenizer довольно разнообразны, но просты, и поэтому здесь не рассматриваются.

2.3 Работа с файловой системой

2.3.1 Класс File

Если классы потоков осуществляют реальную запись и чтение данных, то класс File – это вспомогательный инструмент, призванный обеспечить работу с файлами и каталогами.

Объект класса File является абстрактным представлением файла и пути к нему. Он устанавливает только соответствие с ним, при этом для создания объекта неважно, существует ли такой файл на диске. После создания можно выполнить проверку, вызвав метод exists, который возвращает значение true, если файл существует. Создание или удаление объекта класса File никоим образом не отображается на реальных файлах. Для работы с содержимым файла можно получить экземпляры FileI/OStream.

Объект File может указывать на каталог (узнать это можно путем вызова метода isDirectory). Метод list возвращает список имен (массив String) содержащихся в нем файлов (если объект File не указывает на каталог – будет возвращен null).

Следующий пример демонстрирует использование объектов класса File:

```
import java.io.*;
public class FileDemo {
   public static void findFiles(File file, FileFilter filter,
         PrintStream output) throws IOException{
      if (file.isDirectory()) {
         File[] list = file.listFiles();
         for (int i=list.length; --i>=0;) {
            findFiles(list[i], filter, output);
         }
      } else {
         if (filter.accept(file))
            output.println("\t" + file.getCanonicalPath());
      }
   }
   public static void main(String[] args) {
      class NameFilter implements FileFilter {
         private String mask;
         NameFilter(String mask) {
            this.mask = mask;
         }
         public boolean accept(File file){
            return (file.getName().indexOf(mask)!=-1)?true:false;
         }
     ë }
      File pathFile = new File(".");
      String filterString = ".java";
      try {
         FileFilter filter = new NameFilter(filterString);
         findFiles(pathFile, filter, System.out);
      } catch(Exception e) {
         e.printStackTrace();
```

```
            }
            System.out.println("work finished");
        }
    }
```

Пример 15.16.

При выполнении этой программы на экран будут выведены названия (в каноническом виде) всех файлов, с расширением .java, содержащихся в текущем каталоге и всех его подкаталогах.

Для определения того, что файл имеет расширение .java, использовался интерфейс FileFilter с реализацией в виде внутреннего класса NameFilter. Интерфейс FileFilter определяет только один метод accept, возвращающий значение, определяющее, попадает ли переданный файл в условия фильтрации. Помимо этого интерфейса, существует еще одна разновидность интерфейса фильтра – FilenameFilter, где метод accept определен несколько иначе: он принимает не объект файла к проверке, а объект File, указывающий на каталог, где находится файл для проверки, и строку его названия. Для проверки совпадения, с учетом регулярных выражений, нужно соответствующим образом реализовать метод accept. В конкретном приведенном примере можно было обойтись и без использования интерфейсов FileFilter или FilenameFilter. На практике их можно использовать для вызова методов list объектов File – в этих случаях будут возвращены файлы с учетом фильтра.

Также класс File предоставляет возможность получения некоторой информации о файле.

Методы canRead и canWrite – возвращается boolean значение, можно ли будет приложению производить чтение и изменение содержимого из файла, соответственно.

getName – возвращает строку – имя файла (или каталога).

getParent, getParentName – возвращают каталог, где файл находится в виде строки названия и объекта File, соответственно.

getPath – возвращает путь к файлу (при этом в строку преобразуется абстрактный путь, на который указывает объект File).

isAbsolutely – возвращает boolean значение, является ли абсолютным путь, которым указан файл. Определение, является ли путь абсолютным, зависит от системы, где запущена Java-машина. Так, для Windows абсолютный путь начинается с указания диска, либо символом '\'. Для Unix абсолютный путь начинается символом '/' .

isDirectory, isFile – возвращает boolean значение, указывает ли объект на каталог либо файл, соответственно.

isHidden – возвращает boolean значение, указывает ли объект на скрытый файл.

lastModified – дата последнего изменения.

length – длина файла в байтах.

Также можно изменить некоторые свойства файла – методы setReadOnly, setLastModified, назначение которых очевидно из названия. Если нужно создать файл на диске, это позволяют сделать методы createNewFile, mkDir, mkDirs. Соответственно, createNewFile создает пустой файл (если таковой еще не существует), mkDir создает каталог, если для него все родительские уже существуют, а mkDirs создаст каталог вместе со всеми необходимыми родительскими.

Файл можно и удалить – для этого предназначены методы delete и deleteOnExit. При вызове метода delete файл будет удален сразу же, а при вызове deleteOnExit по окончании работы Java-машины (только при корректном завершении работы) отменить запрос уже невозможно.

Таким образом, класс File дает возможность достаточно полного управления файловой системой.

2.3.2 Класс RandomAccessFile

Этот класс реализует сразу два интерфейса – DataInput и DataOutput – следовательно, может производить запись и чтение всех примитивных типов Java. Эти операции, как следует из названия, производятся с файлом. При этом их можно производить поочередно, произвольным образом перемещаясь по файлу с помощью вызова метода seek(long) (переводит на указанную позицию в файле). Узнать текущее положение указателя в файле можно вызовом метода getFilePointer.

При создании объекта этого класса конструктору в качестве параметров нужно передать два параметра: файл и режим работы. Файл, с которым будет проводиться работа, указывается либо с помощью String – название файла, либо объектом File, ему соответствующим. Режим работы (mode) – представляет собой строку либо "r"(только чтение), либо "rw"(чтение и запись). Попытка открыть несуществующий файл только на чтение приведет к исключению FileNotFoundException. При открытии на чтение и запись он будет незамедлительно создан (или же будет брошено исключение FileNotFoundException, если это невозможно осуществить).

После создания объекта RandomAccessFile можно воспользоваться методами интерфейсов DataInput и DataOutput для проведения с файлом операций считывания и записи. По окончании работы с файлом его следует закрыть, вызвав метод close.

Заключение

Рассмотрено понятие потоков данных (stream). Потоки являются очень эффективным способом решения задач, связанных с передачей и получением данных, независимо от особенностей используемых устройств ввода/вывода. Как вы теперь знаете, именно в пакете java.io содержатся стандартные классы, решающие задачи обмена данными в самых различных форматах.

Были описаны базовые классы байтовых потоков InputStream и OutputStream, а также символьных потоков Reader и Writer. Все классы потоков явным или неявным образом наследуются от них. Краткий обзор показал, для чего предназначен каждый класс, как с ним работать, какие классы не рекомендованы к использованию. Изучено, как передавать в потоки значения примитивных типов Java. Особое внимание было уделено операциям с объектами, для которых существует специальный механизм сериализации.

Наконец, были описаны классы для работы с файловой системой – File и RandomAccessFile.

3. Дополнительная литература

1. Вязовик Н.А. **Программирование на JAVA. Курс лекций на intuit.ru**
2. Хорстманн К., Корнелл Г. **Java 2. Том 2. Тонкости программирования**
3. Герберт Шилдт, Джеймс Холмс **Искусство программирования на JAVA.**
4. Патрик Нотон, Герберт Шилдт **Полный справочник по Java.**

4. Порядок выполнения работы

В соответствии с вариантом выполните следующее основное задание:

1. Создайте новый проект в среде Eclipse.
2. Создать классы на основе лабораторной работы №3 и в соответствии с заданием.
3. Поместить созданные классы и интерфейсы в пакет.
4. Создайте вне пакета управляющий класс, предусмотрев в нем точку входа (main), который будет подключать пакет и осуществлять тестирование разработанных классов.
5. Подготовьте отчет о выполнении лабораторной работы:

Для успешной сдачи лабораторной работы необходимо:

- представить преподавателю отлаженный код программы для указанного варианта

задания;

- подготовить отчет по работе.

5. Порядок оформления отчета

Отчет о выполнении лабораторной работы должен содержать:

1) титульный лист;
2) задание;
3) текст программы;
4) результаты работы программы.

6. Варианты заданий

Общее требование: Приложение должно состоять из набора классов и интерфейса реализующих функциональность задачи и тестового класса.

1. Разработайте приложение осуществляющее транслитерацию по принципу Punto Switcher. Приложение получает из входного потока InputStream строку символов, производит транслитерацию и помещает обработанную строку в выходной поток OutputStream. Предусмотреть возможность получения данных с клавиатуры и из файла и, соответственно, вывод данных в файл и на экран. Использовать классы InputStream, OutputStream, File и другие.
2. Разработайте приложение реализующее функциональность простого архиватора. Создайте класс-наследник от FilterInputStream, например CompressInputStream, который сжимает данные используя алгоритм Хаффмана. Создайте класс-наследник от FilterOutputStream, например CompressOutputStream, который умеет распаковывать данные, созданные потоком CompressInputStream. Предусмотреть возможность получения данных с клавиатуры и из файла и, соответственно, вывод данных в файл и на экран. Алгоритм сжатия Хаффмана реализовать в отдельном классе.
3. Разработайте программу, которая умеет работать с данными в формате *.ini – файлов. По типу <имя параметра> = <значение параметра>. Организуйте в программе возможность создания нового файла, добавление, удаление, редактирование записей, чтение ini - файла. Для хранения данных используйте HashMap, HashTable или TreeMap. Используйте класс StreamTokenizer.
4. Разработайте программу, которая создает новый программный поток, получающий случайные входные данные от некоторого объекта-генератора, а его вывод направляется в выходной поток (экран либо файл). Использовать классы `PipedOutputStream` и `PipedInputStream`.
5. Создать программу выводящую: a) дерево каталогов b) содержимое каталога. Предусмотреть возможность задания корневого каталога, вывод результатов на экран и в файл.
6. Создать программу, получающую имя файла и выводящую всю информацию о соответствующем файле (название, размер, время создания, атрибуты и тд), если он существует. Предусмотреть возможность задания имя файла в различном виде (полный путь, сокращенный путь, в текущем каталоге) Предусмотреть вывод результатов на экран и в файл.
7. Напишите программу, которая получает в качестве параметров имя каталога и расширение файла и выводит список всех файлов каталога с заданным расширением. Использовать FilenameFilter. Предусмотреть возможность задания имени каталога в различном виде (полный путь, сокращенный путь, в текущем каталоге), вывод результатов на экран и в файл.
8. Создать программу считывающую данные (int) из файла(Reader), помещающую их в список, сортирующую их по убыванию (использовать любую быструю

сортировку) и записывающую в другой файл(Writer). Имена файлов определяет пользователь. Предусмотреть возможность задания имя файла в различном виде (полный путь, сокращенный путь, в текущем каталоге)

9. Разработать программу, реализующую простейший грамматический анализатор. Программа имеет базу слов, хранящихся в файле. При запуске они загружаются в HashSet. На вход программы подается файл с текстом. Задача программы выделить слова, в которых есть ошибки и сформировать из них список и выдать на экран или в файл по выбору пользователя. Предусмотреть возможность автоматического исправления ошибок. Использовать StreamTokenizer, PushbackInputStream
10. Разработать программу, реализующую внешнюю сортировку файла алгоритмом простого слияния (см. http://www.citforum.ru/programming/theory/sorting/sorting1.shtml#3_1) Вывести входной и выходной файлы на экран. Использовать классы InputStream, OutputStream, File и другие.