Программирование на Java

Лекция 15. Пакет java.io

20 апреля 2003 года

Авторы документа:

Николай Вязовик (Центр Sun технологий МФТИ) <vyazovick@itc.mipt.ru>
Евгений Жилин (Центр Sun технологий МФТИ) < gene@itc.mipt.ru>

Аннотация

Эта лекция описывает возможности Java для обмена или передачи информации, что является важной функциональностью для большинства программных систем. Сюда входит работа с файлами, с 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 потоками символов (Reader и Writer). Библиотека java.io содержит все эти классы и их многочисленных наследников, предоставляющих полезные возможности. Отдельно рассматривается механизм сериализации объектов и работа с файлами.

Оглавление

Лекция 15. Пакет java.io

Содержание лекции.

1. Система ввода/вывода. Потоки данных (stream)

Подавляющее большинство программ обменивается данными с внешним миром. Это, безусловно, делают любые сетевые приложения - они передают и получают информацию от других компьютеров и специальных устройств, подключенных к сети. Оказывается удобным точно таким же образом представлять обмен данными между устройствами внутри одной машины. Так, например, программа может считывать данные с клавиатуры и записывать их в файл, или же наоборот - считывать данные из файла и выводить их на

экран. Таким образом, устройства, откуда может производиться считывание информации, могут быть самыми разнообразными - файл, клавиатура, (входящее) сетевое соединение и т.д. То же самое касается и устройств вывода - это может быть файл, экран монитора, принтер, (исходящее) сетевое соединение и т.п. В конечном счете, все данные в компьютерной системе в процессе обработки передаются от устройств ввода к устройствам вывода.

Обычно часть вычислительной платформы, которая отвечает за обмен данным, так и называется - система ввода/вывода. В Java она представлена пакетом java.io (input/output). Реализация системы ввода/вывода осложняется не только широким спектром источников и получателей данных, но еще и различными форматами передачи информации. Ею можно обмениваться в двоичном представлении, символьном или текстовом с применением некоторой кодировки (только для русского языка их насчитывается от 4 штук), или передавать числа в различных представлениях. Доступ к данным может потребоваться как последовательный (например, считывание HTML страницы), так и произвольный (сложная работа с несколькими частями одного файла). Зачастую для повышения производительности применяется буферизация.

В Java для описания работы по вводу/выводу используется специальное понятие поток данных (stream). Поток данных связан с некоторым источником или приемником данных, способных получать или предоставлять информацию. Соответственно, потоки делятся на входные - читающие данные, и на выходные - передающие (записывающие) данные. Введение концепции stream позволяет отделить программу, обменивающуюся информацией одинаковым образом с любыми устройствами, от низкоуровневых операций с такими устройствами ввода/вывода.

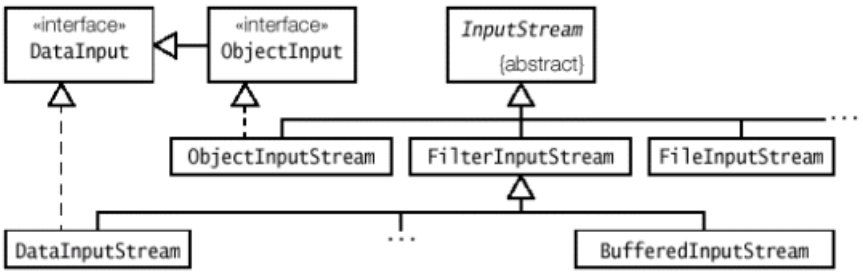
В Java потоки естественным образом представляются объектами. Описывающие их классы как раз и составляют основную часть пакета java.io. Они довольно разнообразны и отвечают за различную функциональность. Все классы разделены на две части - одни осуществляют ввод данных, другие вывод.

Существующие стандартные классы помогают решить большинство типичных задач. Минимальной "порцией" информации является, как известно, бит, принимающий значение 0 или 1 (это понятие также удобно применять на самом низком уровне, где данные передаются электрическим сигналом; условно говоря, 1 представляется прохождением импульса, 0 - его отсутствием). Традиционно используется более крупная единица измерения байт, объединяющая 8 бит. Таким образом, значение, представленное 1 байтом, находится в диапазоне от 0 до 28-1=255, или, если использовать знак, от -128 до +127. Примитивный тип byte в Java в точности соответствует последнему, знаковому диапазону.

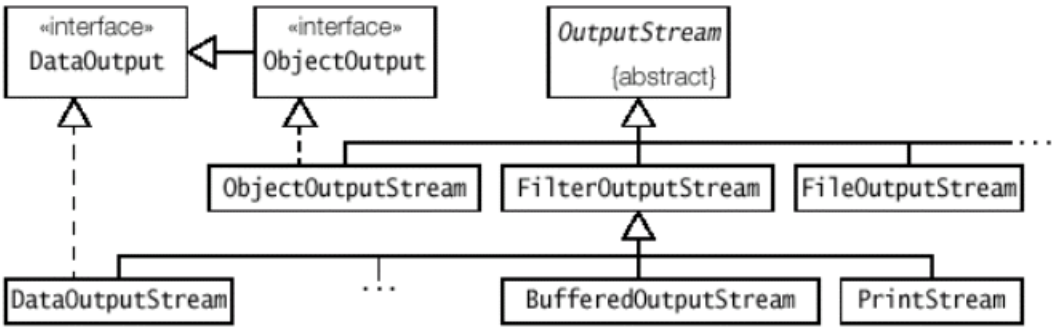
Базовые, наиболее универсальные классы позволяют считывать и записывать информацию именно в виде набора байт. Чтобы их было удобно применять в различных задачах, java.io содержит также классы, преобразующие любые данные в набор байт.

Например, если нужно сохранить результаты вычислений - набор значений типа double - в файл, то их можно сначала легко превратить в набор байт, а затем эти байты записать в файл. Аналогичные действия совершаются и в ситуации, когда требуется сохранить объект (т.е. его состояние) - преобразование в набор байт и последующая их запись в файл. Понятно, что при восстановлении данных в обоих рассмотренных случаях проделываются обратные действия - сначала считывается последовательность байт, а затем она преобразовывается в нужный формат.

На следующих диаграммах представлены иерархии классов ввода/вывода. Как и говорилось, все типы поделены на две группы. Представляющие входные потоки классы наследуются от InputStream:



а выходные - от OutputStream:



1.1. Классы InputStream и OutputStream

InputStream - это базовый класс для потоков ввода, т.е. чтения. Соответственно, он описывает базовые методы для работы с байтовыми потоками данных. Эти методы необходимы всем классам, наследующимся от InputStream.

Простейшая операция представлена методом read() (без аргументов). Он является абстрактным, и, соответственно, должен быть определен в классах-наследниках. Этот метод предназначен для считывания ровно одного байта из потока, однако возвращает при этом значение типа int. В случае если считывание произошло успешно, то возвращаемое значение лежит в диапазоне от 0 до 255 и представляет собой полученный байт (значение int содержит 4 байта и получается простым дополнением нулями в двоичном представлении). Обратите внимание, что полученный таким образом байт не обладает знаком и не находится в диапазоне от -128 до +127 как примитивный тип byte в Java.

В случае если достигнут конец потока, то есть в нем больше нет информации для чтения, то возвращаемое значение равно -1.

Если же считать из потока данные не удается из-за каких-то ошибок или сбоев, кидается исключение java.io.IOException. Этот класс наследуется от Exception, т.е. его всегда необходимо явно обрабатывать. Дело в том, что каналы передачи информации, будь то Интернет или, например, жесткий диск, могут давать сбои независимо от того, насколько

хорошо написана программа. А это означает, что нужно быть готовым к ним, чтобы пользователь не потерял нужные данные.

Метод read() - это абстрактный метод, но именно с соблюдением всех этих условий он должен быть реализован в классах-наследниках.

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 в потоке данные 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".

Что бы узнать, сколько байт в потоке готово к считыванию - используется метод available(). Этот метод возвращает значение типа int, которое показывает, сколько байт в потоке готово к считыванию. При этом не стоит путать это 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 возвращают void, то есть ничего не возвращают.

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 кидает 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+длина будет больше длины массива - во всех этих случаях кидается исключение IndexOutOfBoundsException.

Реализация потока может быть таковой, что данные записываются не сразу, а хранятся некоторое время в памяти. Например, мы хотим записать в файл какие-то данные, которые мы 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 свой внутренний механизм для буферизации (временного хранения перед отправкой) данных. Что бы 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, после чего поток закрывается, и освобождаются все системные ресурсы с ним связанные. 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 широко используемые, будут рассмотрены далее.

1.2. Классы-реализации потоков данных

1.2.1. 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. В данном примере второе считанное значение не равно -1, как можно было бы ожидать. Вместо этого оно равно 255. Что бы 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, получаемое приставкой 24-х нулей, равно 255(в десятичной системе). Однако если непосредственно привести его к byte, получим исходное значение.

Аналогично, для записи байт в массив, используется класс ByteArrayOutputStream. Этот класс использует внутри себя объект byte[], куда записывает данные, передаваемые при вызове методов write(). Что бы 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.

Использование классов 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. Использование этих классов выгодно тем, что можно сразу просмотреть результат, и не нужно создавать ни файл, ни сетевое соединение, ни что-либо еще.

1.2.2. Классы FileInputStream и FileOutputStream

Классы FileInputStream используется для чтения данных из файла. Конструктор этого класса в качестве параметра принимает название файла, из которого будет производиться считывание. При указании строки имени файла нужно учитывать, что именно эта строка и будет передана системе, поэтому формат имени файла и пути к нему может различаться на различных платформах. Если при вызове этого конструктора передать строку, указывающую на директорию либо на не существующий файл, то будет брошено java.io.FileNotFoundException.

Понятно, что если 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 указания файла, так же можно указать, будут ли данные дописываться в конец файла либо файл будет перезаписан. При этом, если указанный файл не существует, то сразу после создания FileOutputStream он будет создан. После, при вызовах методов write(), передаваемые значения будут записываться

в этот файл. По окончании работы необходимо вызвать метод close(), что бы 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");
 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());
}
```

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, сколько байт может быть на данный момент считано без блокирования. Тот, момент, что в подавляющем большинстве случаев это число и будет длиной файла, является всего лишь частным случаем работы на некоторых платформах.

В данном примере для наглядности закрытие потоков производится сразу же после окончания их использования в основном блоке. Однако считается хорошей практикой закрывать потоки в finally блоке.

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

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

1.2.3. PipedInputStream и PipedOutputStream

Классы PipedInputStream и PipedOutputStream характерны тем, что их объекты всегда используются в паре - к одному объекту PipedInputStream привязывается точно один объект PipedOutputStream. Эти классы могут быть полезны, если необходимо данные и записать и считать в пределах одного выполнения одной программы.

Используются следующим образом: создается по объекту PipedInputStream и PipedOutput-Stream, после чего они могут быть соединены между собой. Один объект PipedOutputStream может быть соединен с ровно одним объектом PipedInputStream и наоборот. Соединенный - означает, что если в объект PipedOutputStream записываются данные, то они могут быть считаны именно в соединенном объекте PipedInputStream. Такое соединение можно обеспечить либо вызовом метода connect() с передачей соответствующего объекта PipedStream, либо передать этот объект еще при вызове конструктора.

В следующем примере показано использование связки PipedInputStream и PipedOutput-Stream.

```
try {
int countRead = 0;
byte[] toRead = new byte[100];
PipedInputStream pipeIn = new PipedInputStream();
PipedOutputStream pipeOut = new PipedOutputStream(pipeIn);
// Считвать 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) {
pr("Impossible IOException occur: ");
e.printStackTrace();
}
```

Данный пример носит чисто демонстративный характер, так как выгода от использования этих классов в основном проявляется при разработке многопоточного приложения. Так, например, если программа выполняется в нескольких потоках выполнения (поток выполнения - Thread, но перевод на русский язык двух понятий Stream и Thread - совпадает), организовать передачу данных между потоками удобно с помощью объектов классов PipedStream. Для этого достаточно создать связанные объекты классов PipedInputStream

и PipedOutputStream, после чего передать их в соответствующие Thread. Тогда поток выполнения, где производится считывание из потока 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);
}
// do some work with readedBytes
// …
}catch(IOException e){
/* IOException будет брошено, когда поток inStream либо связанный с ним
PipedOutputStream будет закрыт, после чего будет произведена попытка считывания
 из inStream */
System.out.println("работа с потоком inStream завершена");
}
```

Так как с объектом inStream одновременно могут работать несколько потоков выполнения, блок synchronized(inStream){…} гарантирует, что во время между вызовами inStream.available() и inStream.read(…), ни в каком другом потоке выполнения не будет произведено считывание из inStream. Поэтому вызов inStream.read(readedBytes) не приведет к блокировке, и все данные, готовые к считыванию - будут считаны.

1.2.4. StringBufferInputStream

StringBufferInputStream - производит считывание данных, получаемых преобразованием символов строки в байты. По понятным причинам этот класс не имеет аналога в классах выходных потоков.

Если для каких-либо целей, мы хотим считать строку - объект String как набор byte, то можем воспользоваться классом StringBufferInputStream. Например, если мы уже используем потоки, а некоторые данные получили в виде строки, можно организовать считывание данных из нее через единый интерфейс. При создании объекта StringBufferInputStream необходимо конструктору передать объект String. Данные, возвращаемые методом read(), будут браться именно из этого объекта String. При этом символы будут преобразовываться в байты не совсем точно - будут учитываться только младшие 8 бит у символа(в то время как тип char состоит их 2-х байт).

1.2.5. SequenceInputStream

Класс SequenceInputStream считывает данные из других двух и более входных потоков. При этом можно указать два потока или их список. Данные будут вычитываться последовательно - сначала все данные из первого потока в списке, потом из второго, и так далее. Конец потока SequenceInputStream будет достигнут только тогда, когда будет достигнут конец потока, последнего в списке.

При создании объекта этого класса в конструктор в качестве параметров передаются объекты InputStream - либо два экземпляра либо Enumeration из них. Когда вызывается метод read(), SequenceInputStream пытается считать байт из текущего входного потока. Если в нем больше нет данных(считанное из него значение равно -1), у этого входного потока вызывается метод close(), и следующий входной поток становится текущим. Так до тех пор, пока последний входной поток не станет текущим, и из него не будут считаны все данные. Тогда, если при считывании обнаруживается, что в текущем входном потоке нет больше данных, и больше нет входных потоков SequenceInputStream возвращет -1, то есть объявляет, что данных больше нет. SequenceInputStream автоматически вызывает вызов метода close() у входных потоков, у которых достигнут конец. Вызов метода close() у SequenceeInputStream закрывает этот поток, предварительно закрыв 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){};
}
```

В результате выполнения этого примера в файл file3.txt будет записано содержимое файлов file1.txt и file2.txt - сначала полностью file1.txt и потом file2.txt. Закрытие потоков производится в блоке finally. При этом, так как при вызове метода close() может возникнуть IOException, то каждый такой вызов должен быть взят в try-catch блок. Причем в отдельный try-catch блок взят каждый вызов метода close() - для того, что бы все потоки были закрыты независимо от возникновения исключений при закрытии других потоков. 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 sequencecStream метода close().

Объект SequenceInputStream можно было создать и другим способом: сначала получить объект Enumeration, содержащий все потоки, и передать этот объект Enumeration 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);
```

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

1.3. Классы FilterInputStreeam и FilterOutputStream. Их наследники.

Задачи, возникающие при вводе/выводе крайне разнообразны - этот может быть считывание байтов из файлов, объектов из файлов, объектов из массивов, буферизованное считывание строк из массивов. В такой ситуации решение путем простого наследования приводит к возникновению слишком большого числа подклассов. Решение же, когда требуется совмещение нескольких свойств, высокоэффективно в виде надстроек. (В ООП этот паттерн 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, и поэтому может быть в свою очередь дополнительно быть надстроена.

В java.io интерфейс для таких надстроек ввода/вывода предоставляют классы FilterInput-Stream (для входных потоков) и FilterOutputStream (для выходных потоков). Эти классы унаследованы от основных базовых классов ввода/вывода - InputStream и OutputStream соответственно. Конструкторы этих классов принимают в качестве параметра объект InputStream и имеют модификатор доступа protected. Сами же эти классы являются базовыми для надстроек. Поэтому только наследники могут вызывать его(при их создании), передавая переданный им поток. Таким образом обеспечивается некоторый общий интерфейс для надстраиваемых объектов.

Надстройки можно попробовать разделить на два основных типа - одни меняют данные, с которыми может работать поток, другие меняют внутренние механизмы потока: производят буферизацию, позволяют вернуть в поток считанные байты, или же посчитать количество считанных строк.

1.3.1. BufferedInputStream и BufferedOutputStream

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

BufferedInputStream - содержит в себе массив байт, который служит буфером для считываемых данных. То есть, когда байты из потока считываются (вызов метода read()) либо пропускаются(метод skip()), сначала перезаписывается этот буферный массив, при

этом считываются сразу много байт за раз. Так же класс BufferedInputStream добавляет поддержку методов mark() и reset(). Эти методы определены еще в классе InputStream, но их реализация по умолчанию бросает исключение IOException. Метод mark() запоминает точку во входном потоке и метод reset() приводит к тому, что все байты, считанные после наиболее позднего вызова метода mark(), будут считаны заново, прежде чем новые байты будут считываться из содержащегося входного потока.

BufferedOutputStream - при использовании объекта этого класса, запись производится без необходимости обращения к устройству ввода/вывода при записи каждого байта. Сначала данные записываются во внутренний буфер. Непосредственное обращение к устройству вывода и, соответственно, запись в него произойдет, когда буфер будет полностью заполнен. Освобождение буфера с записью байт на устройство вывода можно обеспечить и непосредственно - вызовом метода flush(). Так же буфер будет освобожден непосредственно перед закрытием потока (вызов метода close()). При вызове этого метода также будет закрыт и поток, над которым буфер надстроен.

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 durates: " + 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 durates " + (time) + " millisec");
 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 durates " + (time) + " millisec");
```

```
 } catch (IOException e) {
  pr("IOException: " + e.toString());
  e.printStackTrace();
 }
```

Результатом могут быть, например, такие значения:

```
Writing durates: 359 millisec
Direct read durates 6546 millisec
Buffered read durates 250 millisec
```

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 файл производилась с буферизацией, однако ее влияние на скорость записи нетрудно проверить, убрав из программы строку, создающую BufferedOutputStream. Оставляем это для самостоятельной проверки.

Классы BufferedInputStream и BufferedOutputStream добавляют только внутреннюю логику по обработке запросов, они не добавляют никаких дополнительных методов. Следующие два фильтра как раз добавляют некоторые дополнительные возможности при работе с потоками. Однако, первый из них - LineNumberInputStream объявлен deprecated начиная с версии jdk 1.1, и использовать его не рекомендуется.

1.3.2. LineNumberInputStream

Расширяет функциональность InputStream тем, что дополнительно производит подсчет, сколько строк было считано из потока. Номер строки, на которой в данный момент происходит чтение можно узнать вызовом метода getLineNumber(). Так же 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 OutputFilter - отсутствует.

1.3.3. PushBackInputStream

Этот фильтр позволяет вернуть во входной поток считанные из него данные. Это действие производится вызовом метода unread(). Понятно, что обеспечивается такая функциональность за счет наличия в классе некоторого буфера - массива байт, который заполняется при считывании из потока. Если будет произведен откат, то потом просто эти данные будут выдаваться еще раз, как будто только были считаны. При конструировании можно указать максимальное количество байт, которое может быть возвращено в поток. Аналогичный класс для OutputFilter - отсутствует.

Следующий класс также является deprecated, и его не рекомендуется применять при написании программ, работающих с различными кодировками. Однако он продолжает активно использоваться в силу исторических причин, так как статические поля out и err класса System ссылаются на объекты именно этого класса, и, поэтому его необходимо знать.

1.3.4. PrintStream

Используется для конвертации и записи строк в байтовый поток. В классе PrintStream определен метод print(), принимающий различные примитивные типы java а так же Object. При вызове этих методов передаваемые данные будут сначала превращены в строку вызовом метода String.valueOf(), после чего записаны в поток. То есть, в поток данные будут записаны в представлении, удобном для прочтения человеком. При этом, даже если возникает исключение, то оно обрабатывается внутри метода print и дальше НЕ кидается. При записи строки используется кодировка, определяемая из настроек системы, где выполняется программа. При этом PrintStream не поддерживает Unicode. Вообще же для работы с текстовыми данными следует использовать Writer`ы. Аналогичный класс для InputFilter - отсутствует.

Итак, ряд классов-потоков уже устарел и не рекомендуется к применению. Тем не менее они были описаны в основном для справочного материала и полноты картины. Действительный интерес и практические выгоды от использования предоставляет следующая пара классов.

1.3.5. 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 существующие их реализации - классы-фильтры DataInputStream и DataOutputStream. Их место в иерархии классов ввода/вывода можно увидеть на диаграммах классов, приведенных на рис.1 и рис.2.

Интерфейсы DataInput и DataOutput определяют, а классы DataInputStream и DataOutput-Stream, соответственно реализуют, методы считывания и записи всех примитивных типов данных. При этом происходит конвертация этих данных в byte и обратно. При этом в поток будут записаны байты, а ответственность за восстановление данных лежит только на разработчике - нужно считывать данные в виде тех же типов, в той же последовательности, как и производилась запись. То есть, можно, конечно, записать несколько раз int или long, а потом считывать их как short или что-нибудь еще - считывание произойдет корректно и никаких предупреждений о возникшей ошибке не возникнет, но результат будет соответствующий - значения, которые никогда не записывались. Это наглядно показано в следующем примере:

```
try {
 ByteArrayOutputStream out = new ByteArrayOutputStream();
 DataOutputStream outData = new DataOutputStream(out);
 outData.writeByte(128); // it would write -128, as casted to byte
 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("Read data in order it was writen: ");
 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("read the same data in other order (first byte is
skipped):");
  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();
 }
```

Интерфейсы DataInput и DataOutput представляют возможность записи/считывания данных примитивных типов Java. Для аналогичной работы с объектами определены унаследованные от них интерфейсы ObjectInput и ObjectOutput соответственно. В java.io имеются реализации этих интерфейсов - классы ObjectInputStream и ObkectOutputStream. Процесс превращения в байты и обратно для объектов несколько сложнее чем для примитивных типов - записываться могут объекты разных классов, они могут иметь ссылки на другие объекты и т.д. Процесс превращения объекта в набор байт носит название сериализация (Serialization) и, соответственно, для обратного процесса - то есть из набора байт в объект - десериализация.

2. Serialization

В java имеется стандартный механизм превращения объекта в набор байт - сериализации. Для того, что бы объект мог быть сериализован, он должен реализовать интерфейс java.io.Serializable(соответствующее объявление должно явно присутствовать в классе объекта или, по правилам наследования, неявно в родительском классе вверх по иерархии). Интерфейс java.io.Serializable не определяет никаких методов. Его присутствие только определяет, что объекты этого класса разрешено сериализовывать. При попытке сериализовать объект, не реализующий этот интерфейс, будет брошено

```
  java.io.NotSerializableException
```

После того, как объект был сериализован, то есть превращен в последовательность байт, его по этой последовательности можно восстановить, при этом восстановление можно проводить на любой машине (вне зависимости от того, где проводилась сериализация), то есть последовательность байт можно передать на другую машину по сети или любым другим образом, и там провести десериализацию. При этом не имеет значения опереционная система под которой запущена Java - например, можно создать объект на машине с ОС Windows, превратить его в последовательность байт, после чего передать их по сети на машину с ОС Unix, где восстановить тот же объект.

Для работы по сериализации в java.io определены интерфейсы ObjectInput, ObjectOutput и реализующие их классы ObjectInputStream и ObjectOutputStream соответственно.

Для сериализации объекта нужен выходной поток OutputStream, который следует передать при конструировании ObjectOutputStream. После чего вызовом метода writeObject() сериализовать объект и записать его в выходной поток. Например:

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

Что бы просмотреть, во что превратился объект objSave, можно просмотреть содержимое массива

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

А чтобы получить этот объект, можно десериализовать его из этого массива:

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

Теперь можно убедиться, что считанный объект идентичен записанному:

```
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 видим, восстановленный объект не является тем же самым(что очевидно - ведь восстановление могло происходить и на другой машине), но равным сериализованому.

Сериализуемый объект может хранить ссылки на другие объекты, которые в свою очередь так же могут хранить ссылки на другие объекты. И все они тоже должны быть восстановлены при десериализации. При этом, важно, что если несколько ссылок указывают на один и тот же объект, то в восстановленных объектах эти ссылки так же указывали на один и тот же объект. Следующий пример демонстрирует это свойство:

```
package demo.io;
import java.io;
class Point implements java.io.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 java.io.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{
  // write objects to file
  FileOutputStream os = new FileOutputStream(fileName);
  ObjectOutputStream oos = new ObjectOutputStream(os);
  oos.writeObject(line1);
  oos.writeObject(line2);
  //change state of line 1 and write it again
  line1.setIndex(3);
  //oos.reset();
  oos.writeObject(line1);
```

```
  // close stream
  // it is enought to close only filter stream
  oos.close();
  os.close();
  //read objects
  System.out.println("Read objects:");
  FileInputStream is = new FileInputStream(fileName);
  ObjectInputStream ois = new ObjectInputStream(is);
  while(is.available()>0){
   Line line = (Line)ois.readObject();
   line.printInfo();
  }
  }catch(ClassNotFoundException e){
   e.printStackTrace();
  }catch(IOException e){
   e.printStackTrace();
  }
}
}
```

Первым делом стоит обратить внимание, что метод available() вызывался не у ObjectInput-Stream, а именно у InputStream, над которым объектный поток надстроен. Сделано так потому, что вызов метода available() у ObjectOutputStream возвращает количество байт, определяемое следующим образом: если достигнут конец потока - значение (-1), если поток находится в заблокированном режиме, то есть производит непосредственное считывание байт объекта из содержащегося в нем потока, то возвращается количество - сколько еще должно быть считано для завершения десериализации объекта(например, такое возможно, если идет долгое считывание объекта по сети), если же поток не находится в режиме блокировки, возвращается 0.

Выполнение этой программы приведет к выводу на экран примерно следующего:

```
Constructing line: 1
Constructing line: 2
Read objects:
Line: 1
 Object reference: study.java.Line@2dc5
 from point (1.0,1.0) reference=study.java.Point@322b
 to point (2.0,2.0) reference=study.java.Point@3481
Line: 2
 Object reference: study.java.Line@5214
 from point (2.0,2.0) reference=study.java.Point@3481
 to point (3.0,3.0) reference=study.java.Point@5435
Line: 1
 Object reference: study.java.Line@2dc5
 from point (1.0,1.0) reference=study.java.Point@322b
 to point (2.0,2.0) reference=study.java.Point@3481
```

Можно заметить, что объектные ссылки на Point - point2 у первой линии и point1 у второй линии совпадают - это по-прежнему один и тот же объект. Далее, третий записанный объект

- тот же, что и первый, при этом совпадают объектные ссылки. Но, когда мы записывали его, значение index было присвоено 3, а в десериализованном объекте это значение равно 1. Так произошло потом, что объект уже был сериализован, а в таком случае, что не быть записанным дважды, механизм сериализации некоторым образом для себя помечает, что объект уже записан в граф, и когда в очередной раз попадется ссылка на него, она будет указывать на уже сериализованный объект. Такой механизм необходим, что бы иметь возможность записывать связанные объекты, которые могут иметь перекрестные ссылки. В таких случаях необходимо отслеживать был ли объект уже сериализован, то есть нужно ли его записывать или достаточно указать ссылку на него.

Если класс содержит в качестве полей другие объекты, то эти объекты так же будут сериализовываться и поэтому тоже должны быть сериализуемы. В свою очередь, сериализуемы должны быть и все объекты, содержащиеся в этих сериализуемых объектах и т.д. Полный путь ссылок объекта по всем объектным ссылкам, имеющимся у него и у всех объектов на которые у него имеются ссылки, и т.д. - называется графом исходного объекта.

Поэтому, когда мы повторно записывали line1, состояние этого объекта не было записано, поскольку этот объект уже был помечен, как записанный в граф. Что бы указать, что сеанс сериализации завершен, и мы хотим заново записывать объекты, у ObjectOutputStream нужно вызвать метод reset(). В рассматриваемом примере для этого достаточно убрать комментарий вв строке

```
  //reset();
```

Если теперь запустить программу, то можно заметить, что line1 во второй раз записано полностью. Но при этом объектная ссылка на ее point2 не совпадает с объектной ссылкой point1 у line1. Этого, впрочем, и следовало ожидать - ведь во второй раз line1 была записана уже в новом сеансе сериализации, а значит, и все объекты Point были записаны заново.

Далее, в теле конструкторов Line специально были вставлены строки, выводящие на экран некоторые сообщения, сообщающие о вызове этих конструкторов. Как можно заметить, при десериализации никакие из этих конструкторов не вызывались - даже конструктор по умолчанию, а Point вообще такого не имеет. То есть при десериализации объект просто восстанавливается в том виде в каком он был, а не конструируется заново.

Однако, вопрос, на который следует обратить внимание - что происходит с состоянием объекта, унаследованным от суперкласса. Ведь состояние объекта определяется не только значениями полей, определенными в нем самом, но так же и таковыми, унаследованными от супепркласса. Сериализуемый подтип берет на себя такую ответственность, но только в том случае, если у суперкласса определен конструктор по умолчанию, объявленный с модификатором доступа таким, что будет доступен для вызова из рассматриваемого наследника. Этот конструктор будет вызван при десериализации. В противном случае, во время выполнения будет брошено исключение java.io.InvalidClassException . Что бы проверить, когда вызывается этот конструктор, можно в предыдущем примере добавить объявление класса

```
class AbstractEntity{
  public AbstractEntity(){
   System.out.println("Create Abstract Entity");
}
```

```
}
```

А какому-нибудь классу, скажем Point, указать наследование от этого класса AbstractEntity.

```
class Point extends AbstractEntity implements java.io.Serializable
```

В процессе десериализации, поля НЕ сериализуемых классов (родительских классов, НЕ реализующих интерфейс Serializable) инициируются вызовом конструктора без параметров. Такой конструктор должен быть доступен из сериализуемого их подкласса. Поля сериализуемого класса будут восстановлены из потока.

При обходе графа, может попасться объект, не реализующий интерфейс Serializable. В этом случае будет брошено java.io.NotSerializableException .

Весьма познавательно будет так же попробовать провести следующий эксперимент: выполнить программу из предыдущего примера. После ее выполнения на диске останется сохраненным файл, куда были сериализованы три объекта. Теперь удалим класс Point, то есть нужно сделать так, что бы файл Point.class не был доступен по путям, указанным в переменной CLASSPATH. Если теперь попробовать считать объект из файла "d:\file", при вызове метода readObject() произойдет исключение ClassNotFoundException. Если исключение обработать и попытаться продолжить считывание из ObjectInputStream, будет брошено java.io.StreamCorruptedException. Но из InputStream над которой надстроен ObjectInputStream в таком случае будет считано лишь некоторое количество байт, но это количество, спецификацией не оговаривается и может изменяться в зависимости от реализации Java-машины, поэтому лучше избегать попыток продолжать считывание из такого потока в подобных ситуациях. Получить java.io.StreamCorruptedException можно и более простым путем, для этого достаточно считать произвольное количество байт непосредственно из потока InputStream. В таком случае ObjectInputStream замечает, что окольными путями были считаны данные из потока, и будет брошено исключение java.io.StreamCorruptedException . Однако, если произвести считывание не из InputStream а вызовом того же метода read() у ObjectInputStream, будет возвращено значение (-1), то есть как будто достигнут конец потока. В ObjectOutputStream так же могли быть записаны примитивные типы Java(ведь ObjectOutput наследуется от DataOutput). При попытке считывания объекта, когда следующим на очереди считывания идет некоторый примитивный тип Java, будет брошено исключение java.io.OptionalDataException. Можно рассмотреть еще несколько возможных вариантов некорректного использования ObjectInputStream, но совершенно ясно, что если мы хотим считать именно те данные, которые были записаны в поток, нужно считывать их именно в том порядке, в каком были записаны.

Если классу для сериализации и десериализации требуется своеобразный подход, его можно осуществить путем реализации следующих методов, с точным сохранением сигнатуры:

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

Методом writeObject записываются данные состояния объекта, и методом readObject они, соответственно, считываются. При этом можно вызвать стандартный механизм записи объекта, вызовом метода

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

И стандартный механизм считывания вызовом метода

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

Этот метод использует информацию из потока что бы присвоить значения полей сохраненного объекта, полям с соответствующими именами в текущем объекте. Использование такой комбинации методов удобно, если ожидается, что со временем классу будут добавлены новые поля.

Возможны ситуации, когда по некоторым причинам, необходимо самостоятельно управлять ходом сериализации и восстановления объекта. Например, опираясь на некоторую замысловатую логику, записывать не все поля, а только некоторые из них. Для такого гибкого управления процессом сериализации предназначен интерфейс java.io.Externalizable .

При использовании этого интерфейса в поток автоматически записывается только идентификация класса. Сохранить и восстановить всю информацию по состоянию экземпляра, должен обеспечить сам класс. Для получения классом полного контроля за форматом и содержанием потока, для объекта и его суперклассов, должны быть реализованы методы writeExternal() и readExternal() интерфейса Externalizable. Эти методы должны полностью координировать сохранение состояния, унаследованное от суперкласса.

При сериализации, сериализуемый объект первым делом проверяется на поддержку интерфейса Externalizable. Если объект реализует Externalizable, вызывается его метод writeExternal . Если объект не поддерживает Externalizable, но реализует Serializable, используется стандартная сериализация ObjectOutputStream. При восстановлении Externalizable объекта, экземпляр создается путем вызова public конструктора без аргументов, после чего вызывается метод readExternal . В противовес этому, объекты Serializable восстанавливаются путем считывания из потока ObjectInputStream.

Метод writeExternal имеет сигнатуру:

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

Для сохранения состояния, вызываются методы ObjectInput для сохранения как примитивных значений так и объектов, связанных с сериализуемым. Для корректной работы, в соответствующем методе

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

В том же самом порядке эти значения должны быть считаны.

При управлении процессом сериализации не всегда имеет смысл обращаться к реализации интерфейса Externalizable или методов readObject, writeObject. Если вопрос состоит только в том, что нежелательно сохранять и восстанавливать некоторый объекты-члены экземпляра. Обычно подобное требование возникает, когда в объекте хранится некоторая конфиденциальная информация, например пароль. Даже если поле такого объекта описано как private, оно все равно будет сериализовано, то есть записано в поток, а значит, это

значение можно будет прочитать, а при умелом обращении и изменить. Так же может потребоваться пропустить сохранение объекта, десериализация которого все равно не будет иметь смысла в последующем - например сетевое соединение все равно нужно будет устанавливать заново. Один способ пропустить сохранение объекта - реализовать интерфейс Externalizable, или определить методы readObject и writeObject - тогда вообще вся запись и чтение проходят как будет указано. Однако в данном случае можно поступить намного проще - достаточно такое поле объявить с модификатором transient. Например, при сериализации класса Account, приведенном в следующем примере, сериализовываться будут только поля login и name

```
class Account implements java.io.Serializable {
 private String name;
 private String login;
 private transient String password;
 /* Some accessors and mutators for fields
 ...
 */
}
```

Когда объект восстанавливается, таким полям выставляется значение по умолчанию, для объектов это null.

2.1. Версии классов

Сериализованный объект может храниться сколь угодно долго, например, если записать его на диск. Тогда, на момент его десериализации может возникнуть такая ситуация, что в его класс уже внесены изменения - добавлены или изменены методы, поля и т.д. Некоторые такие изменения могут изменить класс таким образом, что десериализация станет невозможной. В этом случае попытка десериализации приведет к возникновению InvalidClassException. Например, если сериализовать объект класса User, определенного следующим образом

```
class User implements java.io.Serializable{
 String name;
}
```

После чего модифицировать класс, заменив поле name на два:

```
 class User implements java.io.Serializable{
  protected String firstName;
  protected String lastName;
 }
```

То при попытке десериализовать записанный ранее объект, будет брошено исключение InvalidClassException . Однако этого не произошло бы, если класс изменить следующим образом:

```
class User implements java.io.Serializable{
```

```
 private String name;
 String lastName;
}
```

Для отслеживания таких ситуаций, каждому классу присваивается его идентификатор (ID) версии. Он представляет собой число long (длина 64 бита), полученное при помощи хэш-функции. Для его вычисления используются имена классов, всех реализуемых интерфейсов, всех методов и полей класса. При десериализации объекта, идентификаторы класса и идентификатор, взятый из потока сравниваются.

Изменения, проводимые с классом можно разбить на две группы - совместимые, то есть изменения, которые можно производить в классе, и при этом поддерживать совместимость с ранними версиями, и несовместимые - изменения, нарушающие поддержку с ранней версией класса. Определить к какому типу относится изменение, можно, руководствуясь следующей логикой: так как предполагается, что поздние версии будут поддерживать более ранние, то поздние должны иметь такой набор полей, что в них возможно восстановить поля из старых записей.

Итак, к совместимым относятся следующие изменения:

- добавление поля к классу
- добавление или удаление суперкласса
- изменение модификаторов доступа полей
- удаление у полей модификаторов static или transient
- изменение кода методов, инициализаторов, конструкторов

К несовместимым относятся:

- Удаление поля
- изменение название пакета класса
- изменение типа поля
- добавление к полю экземпляра ключевого слова static или transient
- реализация Serializable вместо Externalizable или наоборот.

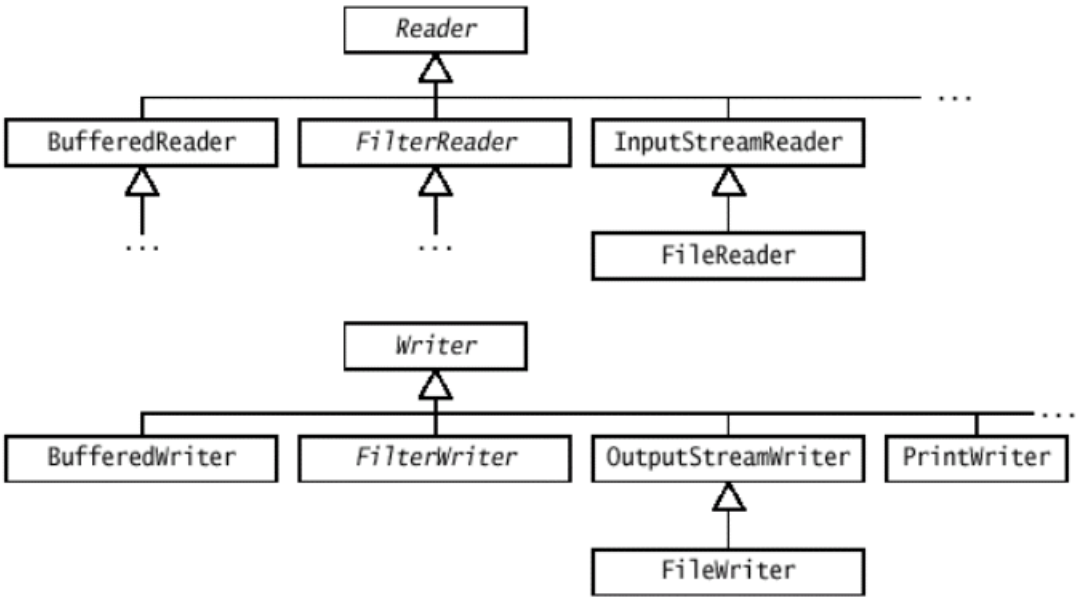
Другими словами, должна остаться возможность восстановить именно те поля, которые были записаны в поток при сериализации.

Обеспечение такой совместимости может стать неприятной задачей, если не позаботиться об этом заранее. В таких случаях сразу следует объявить, что класс реализует интерфейс Externalizable, и потом просто дорабатывать методы readExternalizable и writeExternalizable.

3. Классы Reader и Writer. Их наследники.

Рассмотренные классы - наследники InputStream и OutputStream работают с байтовыми данными. При этом, в обращении со строковыми данными поддерживаются 8-ми битные символы и зачастую неверно происходит при интернационализации обращение с 16-ти битными символами Unicode - именно на которых и основан примитивный тип char(символ) в Java. За правильное использование символов в операциях ввода/вывода предназначены

наследники классов Reader (чтение) и Writer (запись). Их иерархия представлена диаграммой на рис.3 .



Эта иерархия очень схожа с аналогичной для байтовых потоков InputStream и OutputStream.

В таблице 1 приведены соответствия классов для байтовых и символьных потоков.

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:

Для символьных потоков нет потока, аналогичного потоку SequenceInputStream последовательного считывания, и, конечно же, отсутствует преобразование в символьное представление примитивных типов Java и объектов (DateInput/Output, ObjectInput/Output).

В свою очередь, в символьных потоках присутствует поток записи символьных данных в строку, чего нет для байтовых потоков (скорее в силу излишества такой надобности при использовании байтовых потоков). Кроме того, в символьных потоках имеются классы-мосты, преобразующие символьные потоки в байтовые: InputStreamReader и OutputStreamWriter .

Можно обратить внимание и на остальные различия, которые впрочем, являются незначительными:

- Если классы - основы фильтров для символьных потоков: FilterReader и FilterWriter являются абстрактными, то аналогичные для байтовых: InputStream и OutputStream - не абстрактные.
- Методы: close класса Reader и flush и close класса Writer являются абстрактными, в то время как в соответствующих классах InputStream и OutputStream байтовых потоков они имеют просто пустую реализацию
- BufferedReader наследуется не от фильтра FilterReader, а напрямую от Reader
- LineNumberReader не наследуется от FilterReader, вместо этого он унаследован от BufferedReader, который в свою очередь напрямую унаследован от Reader
- FileReader и FileWriter унаследованы от классов-мостов InputStreamReader и OutputStreamWriter
- Метод available() класса InputStream, отсутствует в классе Reader, он заменен методом ready() - возвращающим вместо количества данных готовых к считыванию, 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";
FileWriter fw = null;
BufferedWriter bw = null;
FileReader fr = null;
BufferedReader br = null;
//Строка, которая будет записана в файл
String data = "Some data to be written and readed\n";
```

```
try{
 fw = new FileWriter(fileName);
 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();
 fr = new FileReader(fileName);
 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();
}
```

Классы-мосты InputStreamReader и OutputStreamWriter имеют возможность производить преобразование символов, используя различные кодировки. Кодировка задается при конструировании потока, путем передачи в конструктор в качестве параметра строки - названия кодировки. При указании названия, которое не соответствует никакой из известных кодировок, будет брошено исключение UnsupportedEncodingException. Вот некоторые из допустимых значений этой строки: "Cp1521", "UTF-8", "8859_1", "8859_2" и т.д..

4. Класс StreamTokenizer

Этот класс создается поверх существующего объекта либо InputStream либо Reader. Позволяет работать с данными посредством токенов - кусков данных, выделяемых из потока по определенным свойствам. Процесс разбора данных контролируется через таблицу со множеством флагов котрые могут быть выставлены в различные состояния. StreamTokenizer может распознавать идентификаторы, числа, строки в кавычках и различные стили комментариев. Обычно приложение сначала создает экземпляр этого класса, выставляет синтаксические таблицы, после чего многократно повторяет в цикле вызов метода nextToken, пока не будет возвращено значение StreamTokenizer.TT_EOF.

После вызова метода nextToken поле ttype содержит значение определяющее тип последнего считанного токена. Если последний считанный токен распознан как число, его значение содержится в поле nval. Если же токен распознан как слово, его значение заносится в поле sval.

Способ определения, как разбивать данные на эти куски - настраивается вызовом методов commentChar, ordinaryChar, parseNumbers, quoteChar, resetSyntax, slashSlashComments, slashStarComments, whitespaceChars, wordChars. По умолчанию используются такие настройки, что будут выделяться числа, и тексты(символы с соответствующими кодами) разделяемые переводом строки.

5. Работа с файловой системой.

5.1. Класс File

Если классы потоков осуществляют реальную запись и чтение данных, то класс File - это вспомогательный инструмент, призванный облегчить обращение с файлами и директориями.

Объект класса File является абстрактным представлением файла и пути к нему. Он устанавливает только соответствие с ним, при этом для создания объекта не важно, существует ли такой файл на диске. После создания можно можно сделать проверку, вызвав метод exists, который возвращает значение true, если файл существует. Создание или удаление объекта класса File никоим образом не отображается на реальных файлах. Этот класс не имеет методов для работы с содержимым файла. Объект File может указывать на директорию (узнать это можно вызовом метода isDirectory), тогда вызовом метода list можно получить список имен (массив String) файлов в ней (если объект File не указывает на директорию - 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;
   }
  }
  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");
  }
}
```

При выполнении этой программы на экран будут выведены названия (в каноническом виде) всех файлов, с расширением ".java", содержащихся в текущей директории и всех ее под-директориях. Для определения, что файл имеет расширение ".java" использовался интерфейс FileFilter с реализацией в виде внутреннего класса NameFilter. Интерфейс FileFilter определяет только один метод accept, возвращающий значение, 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 объекта файла к проверке, а объект файл директории 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.

Так же класс File предоставляет возможность получения некоторой информации про файл:

- Методы canRead и canWrite - возвращается boolean значение, возможно ли будет приложению производить чтение и изменение содержимого из файла соответственно
- exists - возвращается boolean значение, существует ли такой файл на диске
- getName - возвращает строку - имя файла (или директории)
- getParent, getParentName - возвращают директорию, 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 директорию либо файл соответственно
- isHidden - возвращает boolean значение, указывает ли объект на скрытый файл
- lastModified
- length

Так же возможно изменить некоторые свойства файла - методы setReadOnly, setLastModified назначение которых очевидно из названия. Если нужно создать файл на диске - это можно сделать методами createNewFile, mkDir, mkDirs. Соответственно createNewFile создает файл (если таковой еще не существует), mkDir создает директорию если для нее все родительские уже существуют, а mkDir создаст директорию, вместе со всеми необходимыми родительскими.

Можно и удалить файл - 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), при этом отменить запрос не является возможным.

Таким образом класс File дает возможность достаточно полного управления файловой системы.

5.2. Класс RandomAccessFile

Этот класс реализует сразу два интерфейса: DataInput и DataOutput и, соответственно, может производить и запись, и чтение всех примитивных типов Java. Эти операции записи и чтения, как следует из названия, производятся с файлами. При этом эти операции можно производить поочередно, вдобавок произвольным образом перемещаясь по файлу - вызовом метода seek(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 объект String, который принимает одно из значений: "r"(чтение) либо "rw"(чтение и запись). При этом, если передать, что mode равно "r" и указать несуществующий файл(либо директорию), будет брошено исключение FileNotFoundException. Если указать, что mode равно "rw" и указать несуществующий файл, он будет незамедлительно создан(или же 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.

6. Заключение

В данной главе Вы ознакомились с таким важным понятием, как потоки данных. Потоки являются очень эффективным способом решения задач по передаче и получению данных независимо от используемых устройств ввода/вывода. Как Вы теперь знаете, именно в пакете java.io содержатся стандартные классы, решающие задачи обмена данными через наиболее распространенные источники (и приемники) данных. Классы этого пакета эффективно решают задачу как широкого спектра источников и получателей данных, так и различных форматов передачи информации.

Были рассмотрены базовые классы байтовых потоков InputStream и OutputStream, а также символьных потоков Reader и Writer. Все классы потоков явным или неявным образом наследуются от этих классов. Были детально рассмотрены классы потоков, реализующих работу с различными источниками данных – массив байт, символьный массив, файл, pipe (“канал”), строки. Показано назначение и способы эффективного использования фильтров, особое внимание уделено использованию BufferedInputStream и BufferedOutputStream, указано, какие классы являются не рекомендованными к использованию. Показано как можно записать (считать) в поток (из потока) различные примитивные значения Java – классы DataInputStream, DataOutputStream (а так же ObjectInputStream, ObjectOutputStream

и отдельно PrintWriter, BufferedReader). Особое внимание уделено описанию механизма сериализации. Показана возможность чтения данных из потоков с разбиением на токены (слова) – класс StreamTokenizer.

Так же приведены и описаны классы для работы с файловой системой – классы File и RandomAccessFile.

7. Контрольные вопросы

15-1. Какие источники могут быть использованы классами стандартных входных потоков java в качестве источника данных? Какие могут быть использованы выходными потоками?

a.) Следующие ресурсы могут быть использованы стандартными потоками в качестве источников данных:

- Файл – представляется объектом класса File
- Массив – представляется массивом byte[] или char[]
- Строка – представляется объектом String
- Сетевое соединение – входной поток получается вызовом getInputStream() у объекта класса java.net.Socket
- “Pipe” – получаемые данные передаются из соединенного объекта PipedOutputStream(PipedWriter)

Выходными потоками могут использоваться эти же самые ресурсы, кроме строк (String). Также, понятно, для сетевого соединения существует различие в получении выходного потока: производится вызовом getOutputStream() у объекта класса Socket.

15-2. Имея два объекта класса File, каким образом будет наиболее корректно узнать, указывают ли они на одну и ту же директорию (и на директорию ли)? Возможно ли только с помощью этих двух объектов удалить директорию? Если да, то как изменится содержимое другого объекта (если они действительно указывают на одну и туже директорию)?

a.) Вызов метода equals() объекта File приведет к лексикографическому сравнению пути файлов. То есть будут сравниваться строки, по которым был создан объект File. Правильным является привести оба объекта к каноническому виду (методы getCanonicalPath() или getCanonicalFile()) и сравнить эти представления. Проверить, указывает ли объект File на директорию, можно вызовом метода isDirectory(). Удалить файл (равно и директорию) можно вызовом метода delete() у объекта File. При этом директория будет удалена только в том случае, если в ней не содержится ни одног³о файл»а или директории. Если объект File указывает на несуществующий файл, то вызов его метода exist() вернет значение false – указывающее, что файл не существует на диске. Именно это и произойдет, если директория, на которую указывает некоторый объект File, будет удалена вызовом delete() через другую ссылку File.

15-3. От какого класса наследуются InputStream и OutputStream? Остальные классы потоков ввода/вывода?

a.) Все входные байтовые потоки наследуются(явно или неявно) от InputStream, а все выходные от OutputStream. Сами же абстрактные классы InputStream и OutputStream унаследованы от Object.

15-4. Если вызвать write(0x01234567) у экземпляра OutputStream, в каком порядке и какие байты будут записаны в выходной поток?

a.) Притом, что метод write() класса OutputStream принимает в качестве параметра значение int, на самом деле в поток будет записан только младший байт – в данном случае 0x67, то есть 16*6+7=103.

15-5. При каких условиях следующий метод вернет значение false?

```
public static boolean test(InputStream is) throws IOException {
 int value = is.read();
 return value == (value &amp;amp; 0xff);
}
```

a.) Выражение value == (value & 0xff) принимает значение true только в случае, если в двоичном представлении значение value содержит единицы только в младшем байте (значение 0xff имеет тип int и равно 255, но не –1). Такой результат будет получен в том случае, если считывание из потока произведено корректно – не был достигнут конец потока, и не было брошено исключение IOException.

15-6. Какие классы предоставляют методы для записи в поток двоичного представления значений примитивных типов Java?

a.) Для записи значений примитивных типов Java, предназначен интерфейс DataOutput. В пакете java.io этот интерфейс реализует класс DataOutputStream. Так же, неявно этот интерфейс реализован классом ObjectOutputStream – этот класс реализует интерфейс ObjectOutput, который расширяет интерфейс DataOutput.

15-7. Если необходимо записать (и после считать) несколько строк в файл (из файла), в каком порядке и какие следует настроить фильтры (и для чтения, и для записи)? Какие из них можно пропустить?

a.) Наиболее распространенным способом является следующая последовательность:

```
new PrintWriter(new BufferedWriter(new
FileWriter("file.txt")))
```

для записи, и, соответственно, для чтения:

```
new BufferedReader(new FileReader("file.txt"))
```

При этом BufferedWriter может быть опущен – он нужен только для обеспечения лучшей производительности путем буферизации (если требуется записать лишь несколько строк, этот класс может быть

действительно лишним). Так же, записать строки можно при желании и используя только лишь FileWriter, а считать - используя FileReader, хотя для удобства желательно воспользоваться классами PrintWriter и BufferedReader (или LineNumberReader – в зависимости от задач).

15-8. Что произойдет при попытке к одному объекту PipedWriter присоединить несколько различных объектов PipedReader? Что произойдет, если несколько раз подряд присоединять один и тот же PipedReader?

a.) К одному объекту PipedWriter может быть присоединен только один объект PipedReader. При попытке присоединить более одного PipedReader, будет брошено исключение new IOException("Already connected") . Это же исключение будет брошено, если подключать более одного раза один и тот же объект PipedReader – метод connect() реализован таким образом, что исключение new IOException("Already connected") будет брошено при попытке вызвать этот метод, когда либо сам объект PipedWriter либо переданный объект PipedReader уже подключен к некоторому объекту PipedWriter.

15-9. Значения каких примитивных типов Java могут быть переданы в качестве параметров методу write() класса Writer? Методу print() класса PrintWriter?

a.) В классе Writer метод write() является перегруженным и определен со следующими параметрами:

- write(char cbuf[])
- write(char cbuf[], int off, int len)
- write(int c)
- write(String str)
- write(String str, int off, int len)

То есть, из примитивных типов может быть передан только int (в случае вызова с передачей byte, char или short – будет произведено приведение к int).

Метод print() класса PrintWriter определен со всеми примитивными типами Java (а так же Object и String).

15-10. Какая кодировка используется классом OutputStreamWriter по умолчанию?

a.) Используемая кодировка зависит от системы, где запущена Java-машина.

15-11. Что будет записано в поток, если вызвать метод print() класса PrintWriter, передав в качестве параметра new File(“d:\\word.txt”) ?

a.) При таком вызове будет вызван метод print(Object obj) и, соответственно, в поток будет записано значение String.valueOf(obj). В свою очередь метод valueOf(Object obj) класса String определен таким образом, что в случае, если переданный объект не равен null, то будет возвращено значение obj.toString() . В данном случае, для объекта File, это будет значение, возвращаемое его методом getPath(), то есть строка, переданная в конструктор – “d:\\word.txt”.

15-12. Какие значения могут быть переданы в конструктор RandomAccessFile для указания режима доступа (чтение/запись)?

a.) Оба конструктора класса RandomAccessFile принимают объект String, указывающий режим доступа к файлу. Значение этой строки может принимать только одно из двух значений – либо “r”(чтение) либо “rw”(чтение и запись). В любом другом случае будет брошено исключение new IllegalArgumentException("mode must be r or rw") (в случае, если переданная строка является null, будет брошено исключение NullPointerException).

15-13. Какое значение следует передать методу seek() объекта RandomAccessFile, чтобы последний байт файла был считан одиночным вызовом read()?

a.) Узнать длину файла, на который указывает объект RandomAccessFile, можно вызовом метода length(). Что бы считать первый байт файла, следует вызвать seek() передав значение 0. соответственно, что бы считать последний байт, нужно методу seek() передать значение randomAccessFile.length()-1 .

15-14. Какие методы объявлены в интерфейсе Serializable?

a.) В интерфейсе java.io.Serializable не объявлено ни одного метода. Классы реализуют этот интерфейс, что бы указать, что объекты этого класса разрешены к сериализации.

15-15. Что произойдет, если записать в файл, используя ObjectOutputStream, значения типов long, int, byte именно в таком порядке, а считать в обратном, используя DataInputStream?

a.) DataInputStream запишет в поток байтовые представления long, int и byte, то есть сначала будут записаны 8 байт, образующие long, потом 4 байта, образующие int, и, наконец, еще один байт. При считывании, будут произведены обратные действия, то есть: для считывания byte будет считан один байт, для считывания int будут считаны следующие 4 байт и для считывания long будут считаны последние 8 байт. Понятно, что в случае, когда считывание производится не в том порядке, в каком была произведена запись, как в данном случае – чтение будет произведено успешно, но полученные значения byte, int и long могут (точнее - будут) отличаться от тех, которые были записаны.