10 ПАКЕТЫ

Под понятием “пакет” подразумевается объединение взаимосвязанных классов, интерфейсов и подпакетов.

Содержание

ПАКЕТЫ

Концепция пакета оказывается полезной по нескольким причинам:

- Пакеты позволяют группировать родственные интерфейсы и классы.
- В интерфейсах и классах, входящих в пакет, могут использоваться популярные имена (вроде get или put), которые имеют смысл в данном контексте, но конфликтуют с теми же именами в других пакетах.
- Пакет может включать типы и члены, с которыми можно работать только в пределах данного пакета. Соответствующие идентификаторы доступны для программ пакета, но закрыты для внешних методов.

Каждый исходный файл, классы и интерфейсы которого принадлежат пакету **attr**, должен указывать на свою принадлежность к пакету объявлением **package**:

package attr;

Если фрагменту программы вне пакета понадобится обратиться к типам, входящим в пакет, у него имеется две возможности.

Первая — указывать перед каждым именем типа префикс (имя пакета).

Другая возможность доступа к типам пакета заключается в частичном или полном *импортировании* пакета:

import attr.*;

Механизмы package и import помогают предотвращать потенциальные конфликты имен

10.1. Имена пакетов

Идентификаторы пакетов Java представляют собой обычные имена, так что способ обеспечить их уникальность — включить в них имя домена организации в Internet.

package COM.magic.attr;

Во многих средах разработки имена пакетов отражаются на уровне файловой системы — часто требуется, чтобы все программы из одного пакета находились в определенной папке или каталоге, а имя этого каталога соответствовало имени пакета

10.2. Пакетный доступ

Классы и интерфейсы, входящие в пакет, обладают одним из двух уровней доступа: пакетным (**package**) и открытым (**public**). Открытый класс или интерфейс доступен для программ, не входящих в пакет. Типы, не являющиеся открытыми, обладают областью видимости на уровне пакета: они скрыты за его пределами, в том числе даже от программ во вложенных пакетах.

Член класса, не объявленный с ключевым словом **public**, **protected** или **private**, может использоваться любой программой, входящей в пакет, но остается невидим за пределами пакета.

Поля или методы, не объявленные в пакете с ключевым словом **private**, доступны для всех программ этого пакета. Следовательно, классы, входящие в тот же пакет, считаются “дружественными”, или “заслуживающими доверия”. Однако подпакеты не пользуются доверием в своих внешних пакетах. Например, защищенные и пакетные идентификаторы в пакете dit недоступны для программ в пакете dit.dat и наоборот.

10.3. Содержимое пакета

Если в исходном файле отсутствует объявление **package**, то входящие в него типы считаются принадлежащими к “безымянному” пакету.

К проектированию пакета следует подходить внимательно и включать в него только функционально связанные классы и интерфейсы. Классы пакета могут свободно обращаться к незакрытым членам друг друга..

В пределах пакета не существует никаких ограничений доступа, кроме **private**.

Кроме того, пакет должен составляться исходя из логического разделения задач, чтобы помочь программистам, которые ищут полезные для себя интерфейсы и классы

Пакет может быть составной частью другого пакета. Например, пакет **java.lang** является вложенным, то есть пакет **lang** входит в более обширный пакет **java**. Пакет **java** не содержит ничего, кроме других пакетов. Вложение позволяет построить иерархическую систему имен для взаимосвязанных пакетов.

Вложение пакетов является средством организации взаимосвязанных пакетов и не имеет отношения к правам доступа

Вложение позволяет группировать взаимосвязанные пакеты и помогает программистам искать классы в иерархической структуре, но не дает никаких других преимуществ

11 ПАКЕТ ВВОДА/ВЫВОДА

Ввод/вывод в Java описывается в терминах потоков.

Потоком

/*(одним словом "поток" переводятся совершенно разнородные термины thread и stream, что создает определенную двусмысленность*/

называется упорядоченная последовательность данных, которая имеет источник (входной поток) или приемник (выходной поток).

Потоки ввода/вывода избавляют программиста от необходимости вникать в конкретные детали операционной системы и позволяют осуществлять доступ к файлам.

В основе работы всех потоков лежит ограниченный набор базовых интерфейсов и абстрактных классов; большинство типов потоков (например, потоки для работы с файлами) поддерживают базовые методы, иногда — с минимальными модификациями.

Пакет ввода/вывода в Java называется java.io.

11.1. Потоки

В пакете **java.io** определяется несколько абстрактных классов для базовых входных и выходных потоков на их основе создаются некоторые полезные типы потоков.

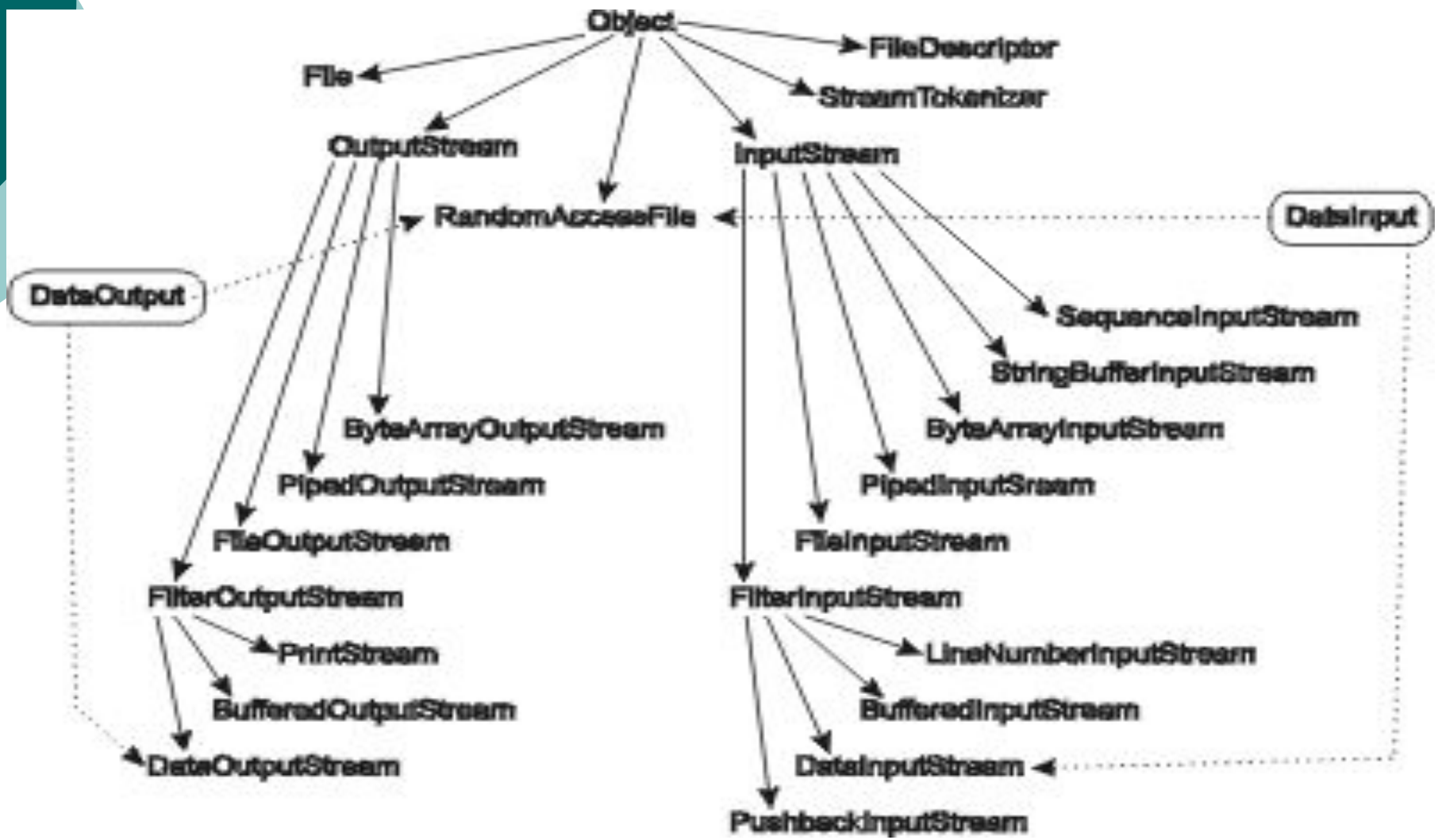
Потоки почти всегда являются парными: если существует **FileInputStream**, то есть и **FileOutputStream**.

Кроме того, существуют классы для работы с именами файлов, класс потока с возможностью чтения/записи с **именем RandomAccessFile** и анализатор для деления входного потока на отдельные лексемы.

Класс **IOException** используется многими методами **java.io** для сигнализации об исключительных состояниях. Некоторые классы, являющиеся расширениями **IOException**, сообщают о конкретных проблемах, однако в большинстве случаев все же применяются объекты **IOException** со строкой-описанием.

Иерархия типов пакета java.io изображена на следующем рисунке:

Иерархия типов пакета java.io



11.2. Класс InputStream

InputStream является базовым классом для большинства входных потоков в java.io:

public **InputStream()**

Класс **InputStream** содержит только безаргументный конструктор.

public abstract int **read()** throws **IOException**

Читает из потока один байт данных и возвращает прочитанное значение, лежащее в диапазоне от 0 до 255 При достижении конца потока возвращается флаг –1. Метод блокирует работу программы до появления значения на входе.

public int **read(byte[] buf)** throws **IOException**

Читает данные в массив байтов. Метод блокирует работу программы до появления вводимого значения, после чего заполняет buf всеми прочитанными байтами, в количестве не более buf.length. Метод возвращает фактическое количество прочитанных байтов или –1 при достижении конца потока.

public int **read(byte**[] **buf**, int **off**, int **len**) throws **IOException**

Читает данные в байтовый подмассив. Метод блокирует работу программы до начала ввода, после чего заполняет часть массива buf, начиная со смещения off, в количестве до len байтов, если не встретится конец массива buf.

public long **skip(long count)** throws **IOException**

Пропускает до count байтов во входном потоке. Количество пропущенных байтов может быть меньше count из-за достижения конца потока. Возвращает фактическое количество пропущенных байтов.

public int **available()** throws I**OException**

Возвращает количество байтов, которые могут быть прочитаны без блокировки работы программы.

public void **close()** throws **IOException**

Закрывает входной поток. Метод должен вызываться для освобождения любых ресурсов (например, файловых дескрипторов), связанных с потоком. Если не сделать это, то ресурсы будут считаться занятыми, пока сборщик мусора не вызовет метод finalize данного потока.

Класс InputStream

Приведенная ниже программа подсчитывает общее количество символов и разделителей (white-space characters) в файле:

```
import java.io.*;
class CountSpace {
    public static void main(String[] args)   throws IOException     {
        InputStream in;
         if (args.length == 0)  in = System.in;
         else   in = new FileInputStream(args[0]);
         int ch;
       int total;
       int spaces = 0;
         for (total = 0; (ch = in.read()) != -1; total++) {
              if (Character.isSpaceChar((char)ch)) spaces++;          }
        System.out.println(total + " chars, "              + spaces + " spaces");
   }}
```

Программа либо берет имя файла из командной строки, либо читает данные из стандартного входного потока, System.in. Входной поток представлен переменной in. Если имя файла не задано, используется стандартный входной поток; если же оно указано, то создается объект **FileInputStream**, являющийся расширением **InputStream**

Цикл for подсчитывает как общее количество символов в файле, так и количество символов-разделителей; для идентификации последних применяется метод **isSpace** класса **Character**.

11.3. Класс OutputStream

Абстрактный класс **OutputStream** во многих отношениях напоминает **InputStream;** он абстрагирует поток байтов, направляемых в приемник. Класс содержит следующие методы:

public **OutputStream()**

Класс OutputStream содержит только безаргументный конструктор.

public abstract void **write(int b)** throws IOException

Записывает в поток байт b. Байт передается в виде значения int, передаются только младшие 8 бит значения int — старшие 24 бита при этом теряются. Метод блокирует работу программы до завершения записи байта.

public void **write(byte[] buf)** throws IOException

Записывает в поток содержимое массива байтов. Метод блокирует работу программы до завершения записи.

public void **write(byte[] buf, int offset, int len)** throws IOException

Записывает в поток часть массива байтов, которая начинается с buf [offset] и насчитывает до count байтов, если ранее не будет встречен конец массива.

public void **flush()** throws IOException

Очищает поток, то есть направляет в него все байты, находящиеся в буфере.

public void **close()** throws IOException

Закрывает поток. Все эти методы возбуждают исключение **IOException**.

Пример

Приложение Translate получает два параметра: строку from и строку to. Если во входном потоке встречается символ, входящий в строку from, он заменяется символом строки to, находящимся в той же позиции:

```
import java.io.*;
class Translate {
    public static void main(String[] args) {
        InputStream in = System.in;
       OutputStream out = System.out;
        if (args.length != 2)       error ("must provide from/to arguments");
        String from = args[0], to = args[1];
        int ch, i;
        if (from.length() != to.length())  error ("from and to must be same length");
```

Пример

```
try {
        while ((ch = in.read()) != -1) {
            if ((i = from.indexOf(ch)) != -1)
  out.write(to.charAt(i));
            else      out.write(ch);          }       }
 catch (IOException e) {
          error ("I/O Exception: " + e);       }    }
   public static void error(String err) {
     System.err.print("Translate: " + err);
     System.exit(1);  // ненулевое значение означает
   // неблагополучное завершение    }}
```

Упражнение 11.1

Перепишите приведенную выше программу Translate в виде метода, который пересылает символы из **InputStream** в **OutputStream**, а метод трансляции (правило замены символов) и потоки являются параметрами. Для каждого типа I**nputStream** и **OutputStream**, о которых говорилось в этой главе, напишите новый метод main, в котором бы учитывалась возможность трансляции символов при вводе или выводе. Если потоки ввода и вывода оказываются симметричными, то для них может применяться общий метод main.

11.4. Стандартные типы потоков

В пакете **java.io** определяются несколько типов потоков. Обычно они составляют пары ввода/вывода:

Конвейерные потоки **Piped** спроектированы для парного использования, при котором байты, записываемые в **PipedOutputStream**, могут читаться из **PipedInputStream**.

Байтовые потоки **ByteArray** осуществляют ввод/вывод в массив байтов.

Фильтрующие потоки **Filtered** представляют собой абстрактные классы байтовых потоков, в которых с читаемыми байтами выполняются некоторые операции-фильтры.

Объект **FilterInputStream** получает ввод от другого объекта **InputStream**, некоторым образом обрабатывает (фильтрует) байты и возвращает результат.

Фильтрующие потоки могут объединяться в последовательности, при этом несколько фильтров превращаются в один сквозной фильтр.

Аналогичным образом осуществляется и фильтрация вывода — для этого применяются различные классы **FilterOutputStream**.

Буферизованные потоки **Buffered** расширяют понятие фильтрующих потоков, добавляя буферизацию, чтобы при каждом вызове **read и write** не приходилось обращаться к файловой системе.

Стандартные типы потоков

Потоки данных **Data** разделяются на две категории. Интерфейсы **DataInput** и **DataOutput** определяют методы для чтения и записи данных встроенных типов, причем вывод одного из них воспринимается в качестве ввода другого. Эти интерфейсы реализуются **классами DataInputStream** и **DataOutputStream**.

Файловые потоки **File** расширяют понятие фильтрующих потоков — байтовый поток в них связывается с определенным файлом. В них встроены некоторые методы, относящиеся к работе с файлами.

В пакет также входит ряд потоков ввода (вывода), для которых отсутствуют парные им потоки вывода (ввода):

Поток **SequenceInputStream** преобразует последовательность объектов **InputStream** в один общий **InputStream**, благодаря чему несколько объединенных входных потоков могут рассматриваться в виде единого входного потока.

StringBufferInputStream использует объект **StringBuffer** в качестве входного потока.

LineNumberInputStream расширяет **FilterInputStream** и следит за нумерацией строк входного потока.

PushbackInputStream расширяет **FilterInputStream**, добавляя возможность отката на один байт, что оказывается полезным при сканировании и синтаксическом анализе входного потока.

PrintStream расширяет **OutputStream** и включает методы **print и println** для форматирования данных на выводе. К этому типу относятся потоки **System.out и System.err.**

Стандартные типы потоков

Имеются еще несколько **полезных классов** ввода/вывода:

Класс **File** (не путать с потоковым классом File!) предназначен для работы с именами и путями файлов в локальной файловой системе. Он включает разделители для компонентов пути, локальный разделитель- суффикс и ряд полезных методов для работы с именами файлов.

RandomAccessFile позволяет работать с файлами на уровне потоков с произвольным доступом.

Он реализует ***интерфейсы*** **DataInput** и **DataOutput,** а также большинство методов ввода/вывода классов **InputStream и OutputStream**.

Класс **StreamTokenizer** разбивает **InputStream** на отдельные лексемы. Он представляет входной поток в виде понятных “слов”, что часто бывает необходимо при синтаксическом анализе введенных пользователем выражений.

Все эти классы могут расширяться и порождать новые разновидности потоковых классов, предназначенные для конкретных приложений.

11.5. Фильтрующие потоки

Фильтрующие потоки добавляют несколько новых конструкторов к базовым конструкторам классов **InputStream** и **OutputStream**.

Им передается поток соответствующего типа (входной или выходной), с которым необходимо соединить объект. Фильтрующие потоки позволяют объединять потоки в “цепочки” и тем самым создавать составной поток с большими возможностями.

Приведенная программа печатает номер строки файла, в которой будет обнаружено первое вхождение заданного символа:

```
import java.io.*;
import java.lang.*;

class FindChar {
    public static void main (String[] args)
        throws Exception
    {
        if (args.length != 2)
            throw new Exception("need char and file");
        int match = args[0].charAt(0);
```

Фильтрующий поток

```
        FileInputStream
            fileIn = new FileInputStream(args[1]);

      LineNumberInputStream  lnum = new LineNumberInputStream(fileIn);

        int ch;
        while ((ch=lnum.read())!= -1) {
            if (ch == match) {
                System.out.println("'" + (char)ch +
                    " at line " + lnum.getLineNumber());
                System.exit(0);
            }
        }
        System.out.println(ch + " not found");
        System.exit(1);
    }
}
```

Комментарий

Программа создает поток класса **FileInputStream** с именем **fileIn** для чтения из указанного файла и затем вставляет перед ним объект класса **LineNumberInputStream** с именем **lnum**.

Объекты **LineNumberInputStream** получают ввод от входных потоков, за которыми они закреплены, и следят за нумерацией строк. При чтении байтов из **lnum** на самом деле происходит чтение из потока **fileIn**, который получает эти байты из входного файла. Если запустить программу с файлом ее собственного исходного текста и буквой 'I' в качестве аргументов, то результат работы будет выглядеть следующим образом:

'I' at line 15

Вы можете "сцепить" произвольное количество объектов **FilterInputStream**.

В качестве исходного источника байтов допускается произвольный объект **InputStream**, не обязательно относящийся к классу **FilterInputStream**. Возможность сцепления является одним из основных достоинств фильтрующих потоков, причем самый первый поток в цепочке не должен относиться к классу **FilterInputStream**.

Объекты **FilterOutputStream** могут сцепляться аналогичным образом. При этом байты, записанные в один выходной поток, будут подвергаться фильтрации и записываться в другой выходной поток. Все потоки, от первого до предпоследнего, должны относиться к классу **FilterOutputStream**, но последний поток может представлять собой любую из разновидностей **Output Stream**.

Фильтрующие потоки

Применение фильтрующих потоков позволяет усовершенствовать поведение стандартных потоков. Например, чтобы всегда знать номер текущей строки в **System.in,** можно вставить в начало программы следующий фрагмент:

```
LineNumberInputStream  lnum = new LinenumberInputStream(System.in);
```

Во всем остальном тексте программы производятся обычные операции с **System.in**, однако теперь появляется возможность следить за нумерацией строк. Для этого используется следующий вызов:

```
lnum.getLineNumber();
```

Поток **LineNumberInputStream**, закрепленный за другим потоком **Input Stream**, следует контракту последнего, если **InputStream** — единственный тип, к которому мог бы относиться данный поток.

System.in может быть отнесен только к **типу InputStream**, так что весь код программы, в котором он используется, вправе рассчитывать только на выполнение контракта этого типа.

Упражнение 11.2

Расширьте **FilterInputStream** для создания класса, который осуществляет построчное чтение и возврат данных, с использованием метода, блокирующего работу программы до появления полной строки ввода.

Упражнение 11.3

Расширьте **FilterOutputStream** для создания класса, который преобразует каждое слово входного потока в заглавный регистр (**title case**).

Упражнение 11.4

Создайте пару фильтрующих потоковых классов для работы со сжатыми в произвольном формате данными; при этом поток **CompressInputStream** должен уметь расшифровывать данные, созданные потоком **Compress OutputStream**.

11.6. Класс PrintStream

Класс **PrintStream** используется каждый раз, когда в вашей программе встречается вызов метода **print или println**.

PrintStream является расширением **FilterOutputStream**, так что передаваемые байты могут подвергаться фильтрации. Класс содержит методы **print и println** для следующих типов:

char int float Object booleanchar[] long double String

Кроме того, простой вызов **println** без параметров осуществляет переход на другую строку без вывода информации.

PrintStream содержит два конструктора.

Конструктор **FilterOutputStream**, получающий в качестве параметра объект-поток.

У другого конструктора имеется второй параметр логического типа, который управляет автоматической очисткой (**autoflushing**) потока.

Если значение этого аргумента равно true, то запись в поток символа перехода на новую строку '\n' приводит к вызову метода **flush**. В противном случае такой символ ничем не отличается от всех остальных, и **flush** не вызывается.

При включении автоматической очистки вызов какого-либо из методов **write**, записывающего массив байтов, приводит к обращению к **flush**. Символы '\n', которые встречаются внутри массивов, не вызывают **flush**, независимо от состояния флага автоматической очистки.

Методы **print(String) и print(char[])** являются синхронизированными. Все остальные методы **print и println** реализуются с помощью этих двух методов, так что печать в объект **PrintStream** является безопасной при многопоточной работе.

Кириллица и консоль

Вывод на консоль происходит в кодировке отличной о установленной в локали. Это характерно для MS Windows NT/2k/Server2k3.

Для них установлена локаль **CP1251** в то время как вывод на консоль происходит в кодировке **CP866**.

В этом случае надо заменить **PrintStream**, который не может работать с символьным выводом на **PrintWriter** и вставить «переходное кольцо» между потоком символов **Unicode** и потоком байтов **System.out,** выводимых на консоль в виде объекта **OutputStreamWriter.** В конструкторе этого объекта следует указать нужную кодировку **CP866.**

PrintWriter pw = **new** PrintWriter(new OutputStreamWriter(System.out,”CP866”), true);

true – принудительный сброс буфера в выходной поток.

Вывод информации на консоль: pw.println(«Это русский текст»);

Ввод с консоли производится методом read() класса InputStreams.

BufferedReader br = **new** BufferedReader (**new** InputStreamReader(System.in, “CP866”));

Далее ввод строки, например, осуществляется следующим образом:

s = br.readLine();

Консольный ввод/вывод

```
import java.io.*;
class PrWr{
Public static void main(String[] args){
Try{
PrintWriter pw = new PrintWriter(new OutputStreamWriter(System.out,”CP866”), true);
BufferedReader br = new BufferedReader (new InputStreamReader( System.in,
      “CP866”));
String s = «Это русская строка»;
System.out.println(“print russian string”+ s);
pw.println(«посимвольный ввод: »);
Int c =0;
 while((c=br.read())!=-1)
pw.println((char)c);
 pw.println(«построчный ввод: »);
do{ s = br.readLine();
pw.println(s);
}while(!s.equals(“q”));
}catch (Exeption e)(System,out.println(e);
}}}
```

11.7. Буферизованные потоки

Объекты классов **BufferedInputStream** и **BufferedOutputStream** обладают свойством буферизации, благодаря чему удается избежать вызова операций чтения/записи при каждом новом обращении к потоку.

Эти классы часто используются в сочетании с файловыми потоками — работа с файлом на диске происходит сравнительно медленно, и буферизация позволяет сократить количество обращений к физическому носителю.

При создании буферизованного потока можно явно задать размер буфера или положиться на значение, принятое по умолчанию. Буферизованный поток использует массив типа **byte** для промежуточного хранения байтов, проходящих через поток.

Если метод **read** вызывается для пустого потока **BufferedInputStream**, он выполняет следующие действия: обращается к методу **read** потока-источника, заполняет буфер максимально возможным количеством байтов и возвращает запрошенные данные из буфера.

Аналогично ведет себя и **BufferedOutputStream**.

Буферизованные потоки

Буферизованный выходной поток, используемый для записи данных в файл, создается следующим образом:

```
OutputStream bufferedFile(String path)     throws IOExceptioon{
 OutputStream out = new FileOutputStream(path);
  return new BufferedOutputStream(out);
}
```

Сначала для указанного пути создается **FileOutputStream**, затем порождается **BufferedOutputStream** и возвращается полученный буферизованный объект-поток. Подобная схема позволяет буферизовать вывод, предназначенный для занесения в файл.

Чтобы пользоваться методами объекта **FileOutputStream**, необходимо сохранить ссылку на него, поскольку для фильтрующих потоков не существует способа получить объект, следующий за данным объектом-потоком в цепочке. Перед тем как работать со следующим потоком, необходимо очистить буфер, иначе данные в буфере не достигнут следующего потока.

11.8. Байтовые потоки

Байтовые массивы, используемые в качестве источников входных или приемников выходных потоков, могут применяться для построение строк с данными для печати, декодирования данных и т. д.

Эти возможности предоставляются потоками **ByteArray**. Методы потоков **ByteArray** являются синхронизированными, а следовательно — безопасными в условиях многопоточной среды.

Класс **ByteArrayInput** использует в качестве источника данных массив типа **byte**. Он содержит два конструктора:

public **ByteArrayInputStream(byte[] buf)**

Создает объект **ByteArrayInputStream** по заданному байтовому массиву. Массив используется непосредственно, а не копируется. Достижение конца массива **buf** означает завершение ввода данных из потока.

public ByteArrayInputStream(byte[] buf, int offset, int length)

Создает объект **ByteArrayInputStream** по заданному байтовому массиву, однако используется лишь часть массива **buf** от **buf[offset]** до **buf [offset+length-1]** или до конца массива (в зависимости от того, какая величина окажется меньше).

Байтовые потоки

Класс **ByteArrayOutput** осуществляет вывод в динамически увеличиваемый байтовый массив. Он содержит следующие конструкторы и методы:

public **ByteArrayOutputStream()**

Создает объект **ByteArrayOutputStream**, размер которого выбирается по умолчанию.

public **ByteArrayOutputStream(int size)**

Создает объект **ByteArrayOutputStream** с заданным исходным размером.

public synchronized byte[] **toByteArray()**

Метод возвращает копию данных. Это позволяет программисту работать с массивом, не изменяя выходных данных.

public int **size()**

Возвращает текущий размер буфера.

public String **toString(int hiByte)**

Создает новый объект **String** на основе содержимого байтового массива. Старшие 8 бит каждого 16-разрядного символа в строке устанавливаются равными 8 младшим битам **hiByte**. Также имеется переопределенная безаргументная форма **toString**, эквивалентная **toString(0).**

11.9. Класс StringBufferInputStream

StringBufferInputStream читает данные из строки String, а не из байтового массива.

Класс содержит единственный конструктор, параметром которого является строка — источник ввода. Работа с символами строки осуществляется так, как если бы это были байты.

Программа читает символы из командной строки или из System.in:

```
class Factor {
    public static void main(String[] args) {
          if (args.length == 0) {
    factorNumbers(System.in);          }
    else { InputStream in;
              for (int i = 0; i < args.length; i++) {
            in = new StringBufferInputStream(args[i]);
    factorNumbers(in);                 }          }     }     // ...}
```

Класс StringBufferInputStream

Если команда вызывается без параметров, то **factorNumbers** берет числа из стандартного входного потока. Если же в командной строке присутствуют параметры, то для каждого из них создается объект **StringBufferInputStream** и вызывается метод **factorNumbers**.

Входные данные этого метода рассматриваются как единая последовательность байтов, независимо от того, взяты ли они из командной строки или из стандартного входного потока .

Конструктор **StringBufferInputStream** передается объект класса **String**, а не **StringBuffer**.

Парного потока для **StringBufferOutputStream** не существует. При необходимости его можно имитировать, применяя метод **toString** к потоку **ByteArrayOutputStream.**

11.10. Файловые потоки и FileDescriptor

Ввод и вывод в приложениях часто связан с чтением/записью файлов. Файловый ввод/вывод в Java представлен двумя потоками — **FileInputStream** и **FileOutputStream**.

Объекты каждого из этих типов создаются одним из трех конструкторов:

Конструктор с параметром типа **String**, содержащим имя файла.

Конструктор с параметром **File** (см. раздел “Класс File”).

Конструктор с параметром класса **FileDescriptor**.

Файловый дескриптор **FileDescriptor** представляет собой системно-зависимый объект, служащий для описания открытого файла.

Он может быть получен вызовом метода **getFD** для любого объекта класса **File** или **RandomAccessFile**.

Объекты **FileDescriptor** позволяют создавать новые потоки **File** или **RandomAccessFile** для тех же файлов, что и другие потоки, но при этом не требуется знание имени файлов.

Необходимо соблюдать осторожность и следить за тем, чтобы различные потоки не пытались одновременно совершать с файлом различные операции.

Например, невозможно предсказать, что случится, когда два потока попытаются одновременно записать информацию в один и тот же файл с использованием двух разных объектов **File Descriptor**.

Метод **flush** класса **FileOutputStream** гарантирует лишь сброс содержимого буфера в файл. Он *не* гарантирует, что данные будут записаны на диск — файловая система может осуществлять свою собственную буферизацию.

11.11. Конвейерные потоки

Конвейерные (piped) потоки используются парами, предназначенными для ввода/вывода; байты, записанные во входной поток пары, считываются на выходе.

Конвейерные потоки безопасны в многопоточной среде; на самом деле, один из вполне надежных способов работы с конвейерными потоками заключается в использовании двух программных потоков — одного для чтения, а другого для записи. В случае заполнения конвейера происходит блокировка программного потока, осуществляющего запись. Если же чтение и запись производятся в одном программном потоке, то он блокируется навсегда.

В примере создается новый программный поток, получающий входные данные от некоторого объекта-генератора, а его вывод направляется в объект

```
 package Pipe;
import java.io.*;
class Target extends Thread {
    private PipedReader pr;
    Target(PipedWriter pw){
         try {            pr =new PipedReader(pw);
       } catch (IOException e) {            System.out.println("Exception from Target(): " + e);
        }}
       PipedReader getStream(){return pr;}
       public void run(){
          while(true){              try{
          System.out.println("Reading: "+ pr.read());
           } catch (IOException e) {            System.out.println("Exception the job is finished");
           System.exit(0);}       }}}
```

КОД

```
class Source extends Thread {
   private PipedWriter pw;
   Source(){
pw =new PipedWriter();}
       PipedWriter getStream(){return pw;}
       public void run(){
           for(int i=0;i<10;i++){
    try{
       pw.write(i);
        System.out.println("Writing: "+i);}
   catch(Exception e){System.err.println("From
   Source.run(): "+e);
   }   }}}
```

Конвейерные потоки

```
class  PapedPrWr  {
 public static void main(String[] args) {
Source s=new Source();
Target t=new Target(s.getStream());
s.start();
t.start();
 }
}
```

Создаются конвейерные потоки, заданием **PipedWriter** в качестве параметра конструктора **PipedReader**.

Порядок значения не имеет: с тем же успехом можно было передавать выходной поток конструктору входного. Важно, чтобы парные потоки ввода/вывода были соединены друг с другом.

Далее конструируется объект **Source** и выходным потоком сгенерированных данных назначается **PipedWriter**.

Затем в цикле происходит чтение данных от генератора и запись их в системный выходной поток.

В конце необходимо убедиться, что последняя выводимая строка будет должным образом завершена.

11.12. Класс SequenceInputStream

Класс **SequenceInputStream** создает единый входной поток, читая данные из одного или нескольких входных потоков: сначала первый поток читается до самого конца, затем — следующий за ним, и так далее, до последнего потока.

Этот класс содержит два конструктора: один — для простейшего случая двух входных потоков, которые передаются в качестве параметров конструктора; другой конструктор предназначен для произвольного количества **входных потоков, в нем используется абстрактное представление Enumeration**

Реализация интерфейса **Enumeration** позволяет получить упорядоченный список объектов любого типа.

Для потока **Sequence InputStream** перечисление может содержать только объекты типа **Input Stream**. Если в нем окажется что-либо еще, то при попытке получения объекта из списка возбуждается исключение **SequenceInputStream**.

Например, приложение **Factor** вызывает метод **factor Numbers** для каждого аргумента, входящего в командную строку. Все числа обрабатываются отдельно, так что подобное разобщение параметров не имеет особого значения. Тем не менее, если бы приложение суммировало числа из входного потока, то было бы необходимо собрать все значения воедино

Класс SequenceInputStream

В приложении SequenceInputStream используется для создания единого потока из объектов StringBufferInputStream для каждого из параметров:

```
import java.io.*;
import java.util.Vector;
class Sum {
    public static void main(String[] args) {
          InputStream in;  // поток, из которого читаются числа
          if (args.length == 0) {  in = System.in;          }
 else {  InputStream stringIn;
              Vector inputs = new Vector(args.length);
              for (int i = 0; i <args.length; i++) {
                 String arg = args[i] + " ";
                stringIn = new StringBufferInputStream(arg);
                    inputs.addElement(stringIn);              }
               in = new SequenceInputStream(inputs.elements());          }
       try {
             double total = sumStream(in);
            System.out.println("The sum is " + total);          }
catch (IOException e) {
           System.out.println(e);
            System.exit(-1); //          }     }     // ...}
```

SequenceInputStream

Если параметры отсутствуют, то для ввода данных используется **System.in**. В противном случае создается объект **Vector**, размер которого позволяет хранить столько объектов **StringBufferInputStream**, сколько аргументов в командной строке. Затем мы создаем поток для каждого из аргументов и добавляем в концы строк пробелы, чтобы разделить их. Затем потоки заносятся в вектор **streams**.

После завершения цикла мы вызываем метод **elements** вектора, чтобы получить объект **Enumeration** с элементами. **Enumeration** используется в конструкторе **SequenceInputStream**, который сцепляет все потоки параметров в единый поток **InputStream**. Затем все числа в этом потоке суммируются методом **sumStream** и выводится результат. Реализация **sumStream** приведена в примере из раздела “Класс StreamTokenizer”.

11.13. Класс LineNumberInputStream

Объекты класса **LineNumberInputStream** позволяют следить за нумерацией строк во время чтения данных из входного потока. Метод **getLine Number**, возвращает текущий номер строки. Нумерация строк начинается с единицы.

Текущий номер строки может быть задан методом **setLineNumber**. Это может оказаться полезным, когда вы работаете с несколькими входными потоками как с одним целым, однако нумерация строк должна осуществляться относительно начала каждого из потоков. Например, если **SequenceInput Stream** используется для чтения из нескольких файлов как из одного потока, то может возникнуть надобность в отдельной нумерации строк для каждого из файлов, из которого поступили данные.

Упражнение 11.5

Напишите программу, которая читает заданный файл и ищет в нем некоторое слово. Программа должна выводить каждую строку, в которой встретилось это слово, и ее номер.

11.14. Класс PushbackInputStream

Класс **PushbackInputStream** обеспечивает возможность отката на один символ потока назад. Это особенно полезно при разделении входного потока на отдельные лексемы. Например, чтобы определить, где кончается лексема, часто приходится читать символ, следующий за ее концом. После просмотра символа, завершающего текущую лексему, необходимо вернуть его во входной поток, чтобы он послужил началом следующей лексемы.

В примере класс PushbackInputStream используется для поиска самой длинной последовательности повторений любого байта в потоке:

```
import java.io.*;
class SequenceCount {
    public static void main(String[] args) {
       try {
            PushbackInputStream  in = new PushbackInputStream(System.in);
              int max = 0;     // длина найденной последовательности
             int maxB = -1;   // байт, из которого она состоит
            int b;           // текущий байт входного потока
             do {int cnt;   int b1 = in.read();  // первый байт в последовательности
                 for (cnt = 1; (b = in.read()) == b1; cnt++)  continue;
                  if (cnt > max) {   max = cnt; // запомнить длину
                      maxB = b1; // запомнить байт                    }
```

Класс PushbackInputStream

```
in.unread(b);   // откат к началу
                                // следующей последовательности                }
while (b != -1);   // продолжать до конца потока
            System.out.println(max + " bytes of " + maxB);          }
 catch (IOException e) {
            System.out.println(e);
          System.exit(1);          }     }}
```

При достижении конца одной последовательности происходит чтение байта, с которого начинается следующая последовательность. Метод **unread** позволяет вернуться на одну позицию назад, чтобы снова прочитать байт при выполнении цикла do для следующей последовательности.

Буфер отката представляет собой защищенное поле типа int с именем **pushBack**. Подклассы могут модифицировать это поле. Значение –1 показывает, что буфер отката пуст. Любое другое значение возвращается в качестве первого байта входного потока методом **Pushback.read**.

11.15. Класс StreamTokenizer

Разделение входного потока на отдельные лексемы встречается довольно часто, поэтому пакет java.io содержит специальный класс **StreamTokenizer** для выполнения простейшего лексического анализа.

Этот класс в полной мере работает лишь с младшими 8 битами Unicode, составляющими подмножество символов Latin-1, поскольку внутренний массив класса, хранящий информацию о категориях символов, состоит только из 256 элементов. Символы, превышающие \u00ff, считаются алфавитными.

Чтобы выделить лексемы в потоке, следует создать объект **StreamTokenizer** на основе объекта **InputStream** и затем установить параметры анализа. Цикл сканирования вызывает метод **nextToken**, который возвращает тип следующей лексемы в потоке. С некоторыми типами лексем **связываются значения, содержащиеся в полях объекта StreamTokenizer**.

Класс StreamTokenizer

Данный класс спроектирован в первую очередь для анализа потоков, содержащих текст в стиле Java; он не универсален.

Когда метод **nextToken** распознает следующую лексему, он возвращает ее тип и присваивает это же значение полю **ttype**. Имеются четыре типа лексем:

TT_WORD: обнаружено слово. Найденное слово помещается в поле sval типа String.

TT_NUMBER: обнаружено число. Найденное число помещается в поле nval типа double. Распознаются только десятичные числа с плавающей точкой (с десятичной точкой или без нее). Анализатор не распознает 3.4e79 как число с плавающей точкой, или 0xffff как шестнадцатеричное число.

TT_EOL: обнаружен конец строки.

TT_EOF: обнаружен конец файла.

Символы входного потока делятся на специальные и ординарные.

Специальными считаются символы, которые особым образом обрабатываются в процессе анализа, — пробелы, символы, образующие числа и слова, и так далее. Все остальные символы относятся к ординарным. Если следующий символ потока является ординарным, то тип лексемы совпадает с символом. Например, если в потоке встречается символ ‘B‘ и он не является специальным, то тип лексемы (и поле **ttype**) равен эквиваленту символа ‘B‘ в типе int.

Класс StreamTokenizer

В качестве примера рассмотрим реализацию метода Sum.sum Stream из класса Sum:

```
static double sumStream(InputStream in) throws IOException
    {
     StreamTokenizer nums = new StreamTokenizer(in);
    double result = 0.0;
    while (nums.nextToken() ==
    StreamTokenizer.TT_NUMBER)
         result +=nums.nval;
     return result;}
```

Объект **StreamTokenizer** создается для исходного потока, после чего в цикле происходит чтение лексем из потока. Если обнаруженная лексема является числом, то оно прибавляется к накапливаемому результату. Когда числа во входном потоке кончаются, возвращается окончательное значение суммы.

Класс StreamTokenizer

Программа читает содержимое файла, ищет в нем атрибуты в виде пар *имя=значение* и сохраняет их в объектах AttributedImpl:

```
public static Attributed readAttrs(String file)     throws IOException{
 FileInputStream fileIn = new FileInputStream(file);
   StreamTokenizer in = new StreamTokenizer(fileIn);
     AttributedImpl attrs = new AttributedImpl();
    Attr attr = null;
in.commentChar('#');
   // '#' - комментарий до конца строки
    in.ordinaryChar('/');   // ранее являлся символом комментария
     while (in.nextToken() != StreamTokenizer.TT_EOF) {
          if (in.ttype() == StreamTokenizer.TT_WORD) {
               if (attr != null) {    attr.valueOf(in.sval);
                    attr = null;        // использован
}
 else {attr = new Attr(in.sval);
                    attrs.add(attr);
             }          } else if (in.ttype == '=') {
           if (attr == null) throw new IOException("misplaced '='");          }
 else {    if (attr == null)        // ожидалось слово
  throw new IOException("bad Attr name");
           attr.valueOf(new Double(in.nval));
             attr = null;          }     }     return attrs;}
```

Класс StreamTokenizer

Программа ищет в потоке строковую лексему, за которой может (хотя и не обязан) следовать знак =, сопровождаемый строкой или числом. Каждый такой атрибут заносится в объект Attr, добавляемый к набору атрибутов объекта AttributedImpl. После завершения анализа файла возвращается набор атрибутов.

Задавая символ # в качестве символа комментария, мы тем самым устанавливаем его категорию. Анализатор распознает несколько категорий символов, которые определяются следующими методами:

public void **wordChars(int low, int hi)**

Символы в этом диапазоне образуют слова; они могут входить в лексему типа TT_WORD. Допускается многократный вызов этого метода с разными диапазонами. Слово состоит из одного или нескольких символов, входящих в любой их допустимых диапазонов.

public void **whitespaceChars(int low, int hi)**

Символы в этом диапазоне являются разделителями. При анализе они игнорируются; их единственное назначение заключается в разделении лексем — например, двух последовательных слов. Как и в случае wordChars, можно вызывать этот метод несколько раз, при этом объединение всех диапазонов определяет набор символов-разделителей.

Класс StreamTokenizer

public void ordinaryChar (int ch)

Символ ch является ординарным. Ординарный символ при анализе потока возвращается сам по себе, а не в виде лексемы. В качестве иллюстрации см. приведенный выше пример.

public void ordinaryChars (int low, int hi)

Символы в диапазоне являются ординарными.

public void commentChar (int ch)

Символ ch начинает однострочный комментарий — символы от ch до ближайшего конца строки считаются одним длинным разделителем.

public void quoteChar (int ch)

Пары символов ch являются ограничителями для строковых констант. Когда в потоке распознается строковая константа, символ ch возвращается в качестве лексемы, а поле sval содержит тело строки (без символов-ограничителей).

В потоке могут присутствовать несколько разных символов-ограничителей, но строки должны начинаться и заканчиваться одним и тем же ограничителем.

Класс StreamTokenizer

public void **parseNumbers()**

Указывает на необходимость выделения чисел из потока. StreamTokenizer выдает числа с плавающей точкой двойной точности и возвращает тип лексемы TT_NUMBER, а значение лексемы помещается в поле nval. Просто отказаться от поиска чисел невозможно — для этого придется либо вызвать ordinaryChars для всех символов, входящих в состав числа (не забудьте о десятичной точке и знаке "минус"), либо вызвать **resetSyntax**.

public void **resetSyntax()**

Сбрасывает синтаксическую таблицу, в результате чего все символы становятся ординарными. Если вы вызовете **resetSyntax** и затем начнете читать поток, то nextToken всегда будет выдавать следующий символ потока, как будто вы используете метод **InputStream.read**.

Остальные методы управляют поведением анализатора:

public void **eolIsSignificant(boolean flag)**

Если значение flag равно true, то конец строки является существенным, и nextToken может возвращать TT_EOL. В противном случае концы строк считаются символами-разделителями и TT_EOL никогда не возвращается. Значение по умолчанию равно false.

public void **slashStarComments(boolean flag)**

Если значение flag равно true, анализатор распознает комментарии вида /*...*/. Значение по умолчанию равно false.

Класс StreamTokenizer

public void **slashSlashComments(boolean flag)**

Если значение flag равно true, анализатор распознает комментарии от // до конца строки. Значение по умолчанию равно false.

public void **lowerCaseMode(boolean flag)**

Если значение flag равно true, все символы в лексемах типа TT_WORD преобразуются в нижний регистр, если имеется соответствующий эквивалент (то есть к слову применяется метод String.toLowerCase). Значение по умолчанию равно false.

Имеется также несколько методов общего назначения:

public void **pushBack()**

Заносит предыдущую лексему обратно в поток. Следующий вызов nextToken снова вернет ту же самую лексему. Глубина отката ограничивается одной лексемой; несколько последовательных вызовов pushBack эквивалентны одному вызову.

public int **lineno()**

Возвращает текущий номер строки. Обычно это бывает полезно для вывода сообщений о найденных ошибках.

public String **toString()**

Возвращает строковое представление последней возвращенной лексемы, включающее номер строки.

Упражнение 11.6

Напишите программу, которая получает входные данные в форме "*имя оператор значение*", где *имя* — одно из трех имен по вашему выбору, *оператор* равен +, – или =, а *значение* является числом. Примените все операторы к именованным величинам, а в конце работы программы выведите все три значения. Усложним задание — воспользуйтесь классом Hashtable, который применялся при разработке AttributedImpl, чтобы можно было работать с произвольным количеством именованных величин, не обязательно тремя.