Понятие потоков ввода/вывода

- Потоком ввода/вывода (I/O Stream) называется произвольный источник или приемник, который способен генерировать либо получать некоторые данные
- Все потоки ведут себя одинаковым образом, хотя физические устройства, с которыми они связаны, могут сильно различаться
- Реализация конкретным потоком низкоуровневого способа приема/передачи информации скрыта от программиста

Подсистема ввода/вывода Java

- Основная подсистема ввода/вывода Java представлена пакетом **java.io**
- В JDK 1.4 появился пакет **java.nio**, представляющий новую систему ввода/вывода.
- В основе java.io лежат 4 абстрактных класса:
  - **InputStream, OutputStream** – для байтовых потоков. Их называют потоками ввода и вывода
  - **Reader и Writer** – для символьных потоков. Их называют потоками чтения и записи.

Класс InputStream

- Абстрактный класс **InputStream** предоставляет минимальный набор методов для работы с входным потоком **байтов**:

  - **int available()** - Возвращает количество еще доступных байт потока
  - **int read()** - Возвращает очередной байт. Значения от 0 до 255. Если чтение невозможно, возвращает -1
  - **int read(byte[] buf, int offset, int count)** - Вводит байты в массив. Возвращает количество реально введенных байтов
  - **long skip(long n)** - Пропускает n байтов потока
  - **void close()** - Закрывает поток и освобождает занятые системные ресурсы

Потомки класса InputStream

- **ObjectInputStream** - поток объектов. Создается при сохранении объектов системными средствами
- **SequenceInputStream** - последовательное соединение нескольких входных потоков
- **ByteArrayInputStream** - использует массив байтов как источник данных
- **PipedInputStream** - совместно с PipedOutputStream обеспечивает обмен данными между двумя потоками выполнения
- **FileInputStream** - обеспечивает чтение из файла
- **StringBufferInputStream** - использует изменяемую строку StringBuffer как источник данных
- **FilterInputStream** - абстрактный класс надстройки над классом InputStream

Механизм надстраивания

- В Java при работе с потоками ввода-вывода возможен такой механизм, когда один поток использует в качестве источника данных другой поток

- В случае InputStream этот механизм реализован с помощью класса FilterInputStream.
- FilterInputStream агрегирует в себе InputStream, и все свои методы делегирует этому внутреннему объекту

- В отличие от наследования надстраивание не ведет к появлению большого числа библиотечных классов. Так если мы имеем классы A1, A2, …, An и хотим комбинировать их свойства путем наследования, мы вынуждены создать порядка n * n новых классов. Если делать то же путем надстраивания, понадобится всего n новых классов

- В java.io имеется несколько потомков FilterInputStream:
  - **PushBackInputStream**
  - **BufferedInputStream**
  - **DataInputStream**

Класс OutputStream

- Абстрактный класс OutputStream предоставляет минимальный набор методов для работы с выходным потоком **байтов**

  - **void write(int b)** - Абстрактный метод записи в поток одного байта
  - **void write(byte[] buf, int offset, int count)** - Запись в поток массива байтов или его части
  - **void flush()** - Форсированная выгрузка буфера для буферизированных потоков. Если получателем служит другой поток, его буфер тоже сбрасывается
  - **void close()** - Закрытие потока и высвобождение системных ресурсов

Потомки класса OutputStream

- **ObjectOutputStream** - поток двоичных представлений объектов. Создается при сериализации
- **ByteArrayOutputStream** - использует массив байтов как приемник данных
- **PipedOutputStream** - вместе с PipedInputStream составляет пару потоков для обмена данными между потоками выполнения (threads)
- **FileOutputStream** - поток для записи в файл
- **FilterOutputStream** - абстрактный класс надстройки

Надстройки над OutputStream

- Надстройки над OuptupStream являются наследниками **FilterOutputStream**
  - **PrintOutputStream** – добавляет возможность преобразования простых типов данных в последовательность байтов. Делает это при помощи перегруженного метода print(), который преобразует и помещает их в выходной поток. Метод print() никогда не возбуждает исключение IOException, а записывает ошибки во внутренние переменные, которые проверяет метод checkError().
  - **BufferedOutputStream** – буферизированный выходной поток. Ускоряет вывод.
  - **DataOutputStream** - поток для вывода значений простых типов. Имеет такие методы как writeBoolean(), writeInt(), writeLong(), writeFloat() и т.п.

Буферизированный ввод/вывод

```
public class FileCopy {
   public static void main(String[] args) {
      try {
        BufferedInputStream bis = new BufferedInputStream(new FileInputStream("erste.jpg"));
        BufferedOutputStream bos = new BufferedOutputStream(new FileOutputStream("zweite.jpg"));

        int c = 0;
        while (true) {
       c = bis.read();
       if (c != -1)
              bos.write(c);
       else
              break;
        }
        bis.close();
        bos.flush(); //освобождаем буфер (принудительно записываем содержимое буфера в файл)
        bos.close(); //закрываем поток записи (обязательно!)
      }
      catch (java.io.IOException e) {
        System.out.println(e.toString());
      }
    }
}
```

Символьные потоки

- Для работы с символьными потоками в Java существуют два базовых класса – Reader и Writer

- **Reader** содержит абстрактные методы read(…) и close(). Дополнительные методы объявлены в потомках этого класса

- **Writer** содержит абстрактные методы write(…), flush() и close()

Некоторые потомки класса Writer

- BufferedWriter - буферизированный выводной поток. Размер буфера можно менять, хотя размер, принятый по умолчанию, пригоден для большинства задач.
- CharArrayWriter - позволяет выводить символы в массив как в поток.
- StringWriter - позволяет выводить символы в изменяемую строку как в поток.
- PrintWriter - поток, снабженный операторами print() и println().
- PipedWriter - средство межпоточного общения.
- OutputStreamWriter – мост между классом OutputStream и классом Writer. Символы, записанные в этот поток, превращаются в байты. При этом можно выбирать способ кодирования символов.
- FileWriter - поток для записи символов в файл.
- FilterWriter – служит для быстрого создания пользовательских надстроек

Потомки класса Reader

- BufferedReader - буферизированный вводной поток символов
- CharArrayReader - позволяет читать символы из массива как из потока.
- StringReader - то же из строки
- PipedReader - парный поток к PipedWriter
- InputStreamReader – при помощи методов класса Reader читает байты из потока InputStream и превращает их в символы. В процессе превращения использует разные системы кодирования
- FileReader - поток для чтения символов из файла
- FilterReader – служит для создания надстроек

Пример программы

- Вводить строки с клавиатуры и записывать их в файл на диске.

```
try {
    // Создаем буферизованный  символьный входной поток
    BufferedReader in = new BufferedReader(
      new InputStreamReader(System.in));
    // Используем класс PrintWriter для вывода
    PrintWriter out = new PrintWriter (new FileWriter("data.txt"));
    // Записываем строки, пока не введем строку "stop"
    while (true) {
      String s = in.readLine();
      if (s.equals("stop"))
            break;
      out.println(s);
    }
    out.close();
} catch (IOException ex) {
   // Обработать исключение
}
```

Класс RandomAccessFile

- RandomAccessFile применяется для работы с файлами произвольного доступа.
- Для перемещения по файлу в RandomAccessFile применяется метод seek().
- RandomAccessFile не участвует в рассмотренной выше иерархии, но реализует интерфейсы DataInput и DataOutput (те же, что реализованы классами InputStream и OutputStream).

Пример работы с RandomAccessFile

- Создать файл прямого доступа, выполнить запись в файл и чтение из файла

```
RandomAccessFile rf = new RandomAccessFile("rtest.dat", "rw");
// Записать в файл 10 чисел и закрыть файл
for(int i = 0; i < 10; i++)
    rf.writeDouble(i * 1.414);
rf.close();
// Открыть файл, записать в него еще одно число и снова закрыть
rf = new RandomAccessFile("rtest.dat", "rw");
rf.seek(5 * 8);
rf.writeDouble(47.0001);
rf.close();
// Открыть файл с возможностью только чтения "r"
rf = new RandomAccessFile("rtest.dat", "r");
// Прочитать 10 чисел и показать их на экране
for(int i = 0; i < 10; i++)
  System.out.println("Value " + i + ": " + rf.readDouble());
rf.close();
```

Класс File

- Класс File предназначен для работы с элементами файловой системы – каталогами и файлами
- Каждый объект File представляет абстрактный файл или каталог, возможно и не существующий
- Абстрактный путь, который заключает в себе объект File, состоит из не обязательного системно-зависимого префикса и последовательности имен
  - Префикс выглядит по-разному в различных операционных системах: символ устройства "C:", "D:" в системе Windows, символ корневого каталога "/" в системе UNIX, символы "\\" в UNC и т.д.
  - Каждое имя последовательности является именем каталога, а последнее имя может быть именем каталога или файла
- Путь может быть абсолютным или относительным

Конструкторы класса File

- **File(String filePath),** где filePath – имя файла на диске
- **File(String dirPath, String filePath)**, здесь параметры dirPath и filePath вместе задают то же, что один параметр в предыдущем конструкторе
- **File(File dirObj, String fileName)**, вместо имени каталога выступает другой объект File
- Объект File является неизменяемым объектом !

Каталоги

- Каталог – это особый файл, который содержит в себе список других файлов и каталогов
- Для каталога метод isDirectory() возвращает true
- Метод File[] listFiles() возвращает список подкаталогов и файлов данного каталога
- Пример: получить массив файлов и каталогов, которые находятся в рабочем (или текущем) каталоге

```
File path = new File(".");
File[] list = path.listFiles();
for(int i = 0; i < list.length; i++)
    System.out.println(list[i].getName());
```

Фильтры (интерфейс FileFilter)

- Интерфейс FileFilter применяется для проверки, подпадает ли объект File под некоторое условие
- Метод boolean accept(File file) возвращает истину, если аргумент удовлетворяет условию
- Метода listFiles(FileFilter filter) класса File принимает в качестве аргумента объект FileFilter и возвращает уже профильтрованный массив из объектов

Пример работы с фильтрами

- Выбрать из текущего каталога лишь те файлы, которые содержат в своем последнем имени буквосочетание, заданное в командной строке

```
public static void main(final String[] args) {
    File path = new File(".");

    // Получить массив объектов
    File[] list = path.listFiles(new FileFilter() {
      public boolean accept(File file) {
            String f = file.getName();
            return f.indexOf(args[0]) != -1;
      }
    });
    // Напечатать имена файлов
    for(int i = 0; i < list.length; i++) {
      System.out.println(list[i].getName());
    }
}
```

Новый ввод/вывод

- Библиотека нового ввода-вывода появилась в версии JDK 1.4
- Ее цель – увеличение производительности и обеспечения безопасности при одновременном конкурентном доступе к данным из нескольких потоков.
- Основными понятиями нового ввода/вывода являются
  - Канал (Channel)
  - Буфер (Buffer)
- При работе с каналом прямого взаимодействия с ним нет. Приложение "посылает" буфер в канал, который затем либо извлекает данные из буфера, либо помещает их в него

Буфер

- Буфер представляет собой контейнер для данных простых типов, таких как byte, int, float и др. кроме boolean

- Кроме собственно данных, буфер имеет
  - текущую позицию
  - лимит
  - емкость

- Операции над буфером можно поделить на
  - абсолютные - считывают или записывают один или несколько элементов начиная с текущей позиции и увеличивают или уменьшают текущую позицию на количество прочитанных элементов
  - относительные - производятся начиная с указанного индекса и не изменяют текущей позиции

Методы класса Buffer

- clear() – подготавливает буфер для операции записи в него данных. Он устанавливает лимит равным емкости и позицию равной нулю. Таким образом, при чтении данных из канала и записи их в буфер, они будут туда помещаться с начальной позиции до тех пор, пока буфер не будет полностью заполнен.

- flip() – подготавливает буфер для чтения из него данных. Он устанавливает лимит равным текущей позиции и после этого устанавливает позицию равной нулю. Таким образом, при записи данных в канал они будут считываться из буфера начиная с начала до того места, до которого он был заполнен

- rewind() – подготоваливает буфер для повторного прочтения данных. Он не изменяет лимит и устанавливает позицию равной нулю

Файловый канал

- Канал представляет собой открытое соединение к некоторой сущности, такой как, например, аппаратное устройство, файл, сетевой сокет или программный компонент, которая может производить операции ввода/вывода
- Класс FileChannel позволяет организовать канал доступа к файлу
- Для получения файлового канала служат метод getChannel() классов FileInputStream, FileOutputStream и RandomAccessFile

Работа с FileChannel

- Файловый канал имеет свою позицию, которая устанавливается методом position(long)
- Методы read(ByteBuffer) и read(ByteBuffer, int) служат для чтения данных из канала в переданный буфер с текущей позиции (относительно) или с указанной позиции (абсолютно) соответственно
- Аналогично используются методы write(...)
- Для блокировки файла или его части используются методы lock(...). Их использование гарантирует то, что файл, к которому осуществляется доступ, будет блокирован для других процессов

Пример работы с FileChannel

```
public class GetChannel {
  private static final int BSIZE = 1024;

  public static void main(String[] args) throws Exception {
    // Запись в файл:
    FileChannel fc = new FileOutputStream("data.txt").getChannel();
    fc.write(ByteBuffer.wrap("Немного текста ".getBytes()));
    fc.close();
    // Добавление в конец файла:
    fc = new RandomAccessFile("data.txt", "rw").getChannel();
    fc.position(fc.size()); // Переходим в конец
    fc.write(ByteBuffer.wrap("Еще немного".getBytes()));
    fc.close();
    // Чтение файла:
    fc = new FileInputStream("data.txt").getChannel();
    ByteBuffer buff = ByteBuffer.allocate(BSIZE);
    fc.read(buff);
    buff.flip();
    while(buff.hasRemaining())
      System.out.print((char)buff.get());
  }
}
```

Копирование файлов с использованием FileChannel

```
public class ChannelCopy {
   private static final int BSIZE = 1024;
   public static void main(String[] args) throws Exception {
   if(args.length != 2) {
     System.out.println("параметры: ФайлИсточник
     ФайлПолучатель");
     System.exit(1);
   }
   FileChannel in = new FileInputStream(args[0]).getChannel(), out =
     new FileOutputStream(args[1]).getChannel();
   ByteBuffer buffer = ByteBuffer.allocate(BSIZE);
   while(in.read(buffer) != -1) {
      buffer.flip(); // Подготовим для записи
      out.write(buffer);
      buffer.clear(); // Подготовим для чтения
   }
 }
}
```

Сериализация

- Сериализация позволяет превратить объект в поток байтов, чтобы, когда понадобится, полностью восстановить объект из потока

- Сериализация необходима для
  - сохранения объектов в постоянной памяти,
  - транспортировки параметров при удаленном вызове методов (RMI - Remote Methods Invocation),
  - сохранения на диске компонентов JavaBeans

Интерфейс Serializable

- Чтобы обладать способностью к сериализации, класс должен реализовать интерфейс-метку Serializable

- Интерфейс Serializable не содержит никаких методов. Он просто служит индикатором того, что класс может быть сериализован

- Для того, чтобы значения полей объекта могли быть восстановлены в процессе десерилизации, к ним должен быть доступ посредством стандартного конструктора без параметров, который, в принципе, может не содержать никакого кода

```
public class MyClass implements Serializable{
…
}
```

Запись-чтение объектов

- Сериализованные объекты можно записывать и считывать при помощи классов ObjectOutputStream и ObjectInputStream.
- Они таже реализуют интерфейсы DataInput / DataOutput, что дает возможность записывать в поток не только объекты, но и простые типы данных.
- wirteObject(Object obj) – запись объекта (класс ObjectOutputStream)
- Object readObject() – чтение объекта (класс ObjectInputStream). Метод readObject может также генерировать java.lang.ClassNotFoundException
- При десериализации объекта, он возвращается в виде объекта класса Object - верхнего класса всей иерархии классов Java. Для того, чтобы использовать десериализованный класс, необходимо произвести явное преобразование его к необходимому типу

Пример сериализации объектов

```
public class Point implements java.io.Serializable {
   private int x=0, y = 0;
   public Point() {}
   public Point(int x, int y) {
     this.x = x; this.y = y;
   }
   public String toString() { return "("+x+","+y+")"; }
}
// Сериализация
java.io.ObjectOutputStream ois = new java.io.ObjectOutputStream(new
      java.io.FileOutputStream("state.bin"));
ois.writeDouble(3.14159265D);
ois.writeObject("The value of PI");
ois.writeObject(new Point(10,253)); //запись объекта класса Point
ois.flush();
ois.close();

// Десериализация
java.io.ObjectInputStream ois = new java.io.ObjectInputStream(new
      java.io.FileInputStream("state.bin"));
System.out.println("Double: " + ois.readDouble());
System.out.println("String: " + ois.readObject().toString());
System.out.println("Point: " + (Point) ois.readObject());
ois.close();
```

Архивирование

- Библиотека ввода/вывода Java содержит классы, поддерживающие чтение и запись потоков в компрессированном формате
- Эти классы являются оберткой для существующих классов ввода/вывода для обеспечения возможности компрессирования
- Они являются частью иерархии InputStream и OutputStream

Классы для работы с архивами

- DeflaterOutputStream – базовый класс для классов компрессии
- InflaterInputStream – базовый класс для классов декомпрессии.
- ZipOutputStream - DeflaterOutputStream, который компрессирует данные в файл формата Zip.
- ZipInputStream - InflaterInputStream, который декомпрессирует данные, хранящиеся в файле формата Zip.
- GZIPOutputStream – DeflaterOutputStream, который компрессирует данные в файл формата GZIP.
- GZIPInputStream – InflaterInputStream, который декомпрессирует данные, хранящиеся в файле формата GZIP

Работа с ZipOutputSream

```
ZipOutputStream out = new ZipOutputStream(new
FileOutputStream(“archive.zip”));
pack("111.txt", out);
pack(“222.txt", out);
out.close();

// Упаковывает файл по имени fin
static void pack(String fin, ZipOutputStream out) throws IOException {
    // Открыть вводной файл
    FileInputStream in = new FileInputStream(fin);
    // Создать вход
    out.putNextEntry(new ZipEntry(fin));
    // Выполнить сжатие
    int c;
    while((c = in.read()) != -1)
      out.write(c);
    in.close();
}
```

Работа с ZipInputStream

```
ZipInputStream in = new ZipInputStream(new BufferedInputStream(
                    new FileInputStream("111.zip")));

    ZipEntry entry;
    while ((entry = in.getNextEntry()) != null) {
      unpack(in, entry.getName());
    }

static void unpack(ZipInputStream in, String fout) throws IOException {
    // Создать выходной поток
    BufferedOutputStream out = new BufferedOutputStream(
                        new FileOutputStream(fout));
    int c;
    while((c = in.read()) != -1) {
      out.write(c);
    }
    out.close();
}
```