# Основы ввода-вывода Java

Большинство java-приложений не являются текст-ориентированными консольными программами, а являются графически-ориентированными программами, которые основаны на Abstract Window Toolkit (AWT) или Swing для взаимодействия с пользователем. Поэтому поддержка консольного ввода-вывода в java ограничена и не слишком удобна в использовании.

## Потоки

Java-программы создают потоки ввода-вывода. **Поток (stream)** - это абстракция, которая либо порождает, либо принимает информацию. Поток связан с физическим устройством с помощью системы ввода-вывода java. Все потоки ведут себя одинаково, несмотря на то, что физические устройства, к которым они подключены, отличаются друг от друга. Т.е. одни и те же классы и методы ввода-вывода применимы к устройствам разного типа. Абстракция входного потока может охватить разные типы ввода: из дискового файла, клавиатуры или сетевого сокета. Аналогично выходной поток может ссылаться на консоль, дисковый файл или сетевое подключение.

Потоки – это способ обращения с вводом-выводом без необходимости учета различий между клавиатурой и сетью и т.п.

Java реализует потоки внутри иерархии классов, определенных в пакете **java.io.**

## Байтовые и символьные потоки

Java определяет два типа потоков: байтовые и символьные.

Байтовые потоки предоставляют удобные средства для управления вводом и выводом байтов. Байтовые потоки используются при чтении и записи бинарных данных.

Символьные потоки предлагают удобные возможности управления вводом и выводом символов. Они используют кодировку Unicode. В некоторых случаях символьные потоки более эффективны, чем байтовые. Исходная версия java (1.0) не включала символьных потоков, и потому весь ввод-вывод был байт-ориентированным. Символьные потоки появились в java 1.1, и при этом некоторые байт-ориентированные классы и методы устарели. На низком уровне весь ввод-вывод по-прежнему байт-ориентирован. Символьные потоки предлагают удобные и эффективные средства управления символами.

Java-символы представляют собой индексы в наборе символов Unicode. Это 16-битные значения, которые могут быть преобразованы в целые значения, с которыми можно выполнять целочисленные операции сложения и вычитания.

Символьные константы указываются внутри пары одинарных кавычек. Все отображаемые символы ASCII можно вводить, указывая их в кавычках.
**‘a’, ‘z’, ‘@’**

Можно использовать несколько управляющих последовательностей, которые позволяют вводить нужные символы, такие как ‘\’ для символа одинарной кавычки и ‘\n’ для символа новой строки.

Существует механизм для ввода значения символа в восьмеричном или шестнадцатеричном виде. Для ввода значения в восьмеричной форме используют символ \, за которым следует трехзначный номер ‘\141’ = ‘a’.

Для ввода шестнадцатеричного значения применяются символы \ и u (\u), за которыми следуют четыре шестнадцатеричные цифры ‘\u0061’ = ‘a’ из набора символов ISO-Latin-1.

**\123** - восьмеричный символ **\u0076** - шестнадцатеричный символ
**\’** - одинарная кавычка **\”** - двойная кавычка **** - обратная косая черта
**\r** - возврат каретки **\n** - новая строка **\t** – табуляция
**\b** - возврат на одну позицию.

**Классы байтовых потоков**

Байтовые потоки определены в двух иерархиях классов. На вершине находятся абстрактные классы **InputStream** и **OutputStream**. Каждый из этих абстрактных классов имеет несколько реальных подклассов, которые управляют различиями между различными устройствами, такими как дисковые файлы, сетевые подключения и даже буферы памяти. Абстрактные классы InputStream и OutputStream определяют несколько ключевых методов, которые реализуют другие потоковые классы. Два наиболее важных - это read() и write( ), которые, соответственно, читают и пишут байты данных. Оба метода объявлены как абстрактные внутри InputStream и OutputStream. В классах-наследниках они переопределяются.

**Классы символьных потоков**

Символьные потоки также определены в двух иерархиях классов. На их вершине находятся два абстрактных класса: **Reader** и **Writer**. Эти абстрактные классы управляют потоками символов Unicode. В Java предусмотрено несколько конкретных подклассов для каждого из них. Абстрактные классы Reader и Writer определяют несколько ключевых методов, которые реализуют другие потоковые классы. Два наиболее важных - это read() и write() , которые, соответственно читают и пишут символьные данные. Эти методы переопределяются в потоковых классах-наследниках.

**Классы байтовых потоков**

BufferedInputStream - Буферизированный входной поток.

BufferedOutputStream - Буферизированный выходной поток.

ByteArrayInputStream - Входной поток, читающий из массива байт.

ByteArrayOutputStream - Выходной поток, записывающий в массив байт.

DataInputStream - Входной ПОТОК, включающий методы для чтения стандартных типов данных Java.

DataOutputStream - Выходной поток, включающий методы для записи стандартных типов данных Java.

FileInputStream - Входной поток, читающий из файла.

FileOutputStream - Выходной поток, записывающий в файл.

FilterInputStream - Реализация InputStream.

FilterOutputStream - Реализация OutputStream.

InputStream - Абстрактный класс, описывающий поток ввода.

OutputStream - Абстрактный класс, описывающий поток вывода.

PrintStream - Выходной поток, включающий print() и println().

**Классы символьных потоков**

BufferedReader - Буферизованный входной символьный поток.

BufferedWriter - Буферизованный выходной символьный поток.

CharArrayReader - Входной поток, который читает из символьного массива.

CharArrayWriter - Выходной поток, который пишет в символьный массив.

FileReader - Входной поток, читающий файл.

FileWriter - Выходной поток, пишущий в файл.

FilterReader - Фильтрующий поток для чтения.

FilterWriter - Фильтрующий поток для записи.

InputStreamReader -Входной поток, транслирующий байты в символы.

OutputStreamWriter -Выходной поток, транслирующий байты в символы.

PrintWriter - Выходной поток, включающий print() и println().

Reader - Абстрактный класс, описывающий символьный ввод.

StringReader - Входной поток, читающий из строки.

StringWriter - Выходной поток, пишущий в строку.

Writer - Абстрактный класс, описывающий символьный вывод.

## Предопределенные потоки

Все java-программы автоматически импортируют пакет **java.lang**. В этом пакете определен класс System, инкапсулирующий некоторые аспекты среды времени выполнения. Например, текущее время и настройки различных параметров, ассоциированных с системой.

System содержит три предопределенных потоковых переменных: **in, out** и **err**. Эти переменные объявлены как **public static final** (доступны без объекта **System**).

**System.out** ссылается на стандартный выходной поток, по умолчанию - консоль.

**System.in** ссылается на стандартный входной поток, по умолчанию -консоль.

**System.err** ссылается на стандартный поток ошибок, по умолчанию - консоль.

Эти потоки могут быть перенаправлены на любое совместимое устройство ввода-вывода.

**System.in** - это объект типа **InputStream**.
**System.out** и **System.err** - объекты типа **PrintStream**.
Это байтовые потоки, но обычно они используются для чтения и записи символов с консоли и на консоль. При необходимости их можно поместить в оболочки символьных потоков.

## Чтение консольного ввода

Для чтения консольного ввода используется символ-ориентированный поток.
В Java консольный ввод выполняется чтением System.in. Чтобы получить символьный поток, присоединенный к консоли, нужно поместить System.in в оболочку объекта BufferedReader.

**BufferedReader** поддерживает буферизованный входной поток. Конструктор выглядит так: **BufferedReader(Reader inputReader)**.

**inputReader** - это поток, который связывается с создаваемым экземпляром **BufferedReader**. **Reader** - абстрактный класс, его наследником является **InputStreamReader**, который преобразует байты в символы.

Для получения объекта InputStreamReader, который присоединен к System.in, служит конструктор: **InputStreamReader(InputStream inputStream)** Поскольку System.in ссылается на объект типа InputStream, он может быть использован как параметр inputStream.

BufferedReader, соединенный с клавиатурой:

**BufferedReader br=new BufferedReader(new InputStreamReader(System.in))**; После выполнения этого оператора br представляет собой основанный на символах поток, подключенный к консоли через System.in.

## Чтение символов

Для чтения символа из BufferedReader применяется read ().

**int read() throws IOException**

Метод **read( )** читает символ из входного потока и возвращает его как целое значение. При достижении конца потока возвращается -l.

```
import java.io.*;
class Read {
  public static void main(String args[]) throws IOException {
       char c;
       BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
       System.out.println("Вводите символы, ‘q‘ - для выхода.");
      do {
            c = (char)br.read();
           System.out.println(c);
    } while(c != 'q') ;
}}
```

System.in является строчно-буферизованным по умолчанию. Это значит, что никакого ввода программе не передается до тех пор, пока не будет нажата клавиша <ENTER>.

**String str = br.readLine();** - для ввода строки

## Консольный вывод

Консольный вывод в основном осуществляется с помощью методов print() и println(). Эти методы определены в классе **PrintStream** (который является типом объекта System.out).

**System.out** - байтовый поток. Поскольку PrintStream - выходной поток, унаследованный от **OutputStream,** он реализует низкоуровневый метод **write().** Метод write () может применяться для записи на консоль.

**void write(int byteval)**

Этот метод пишет в поток байт, переданный в **byteval**. Хотя параметр byteval объявлен как целочисленный, записываются только 8 его младших бит.

```
class WriteDemo {
 public static void main(String args[]) {
   int b;
        b = 'A';
        System.out.write(b);
         System.out.write('\n');        }   }
```

## Класс **PrintWriter**

**PrintWriter** - это один из классов, основанных на символах, определяет несколько конструкторов.

Один из них:
**PrintWriter(OutputStream outputStream, boolean flushOnNewline)**
**outputStream** - объект типа OutputStream,
**flushOnNewline** - управляет тем, будет ли java сбрасывать буфер в выходной поток каждый раз при вызове метода println (). Если flushOnNewline = true, то происходит автоматический сброс буфера, если же false, то сброс не делается.

PrintWriter поддерживает методы print () и println() для всех типов, включая Object. Если аргумент не примитивного типа, то PrintWriter вызывает метод toString () и затем печатает результат.

Чтобы выводить на консоль с помощью объекта PrintWriter, нужно определить System.out в качестве выходного потока и сбрасывать буфер при вызове метода println().

PrintWriter, который подключен к консольному выводу:
**PrintWriter pw = new PrintWriter(System.out, true);**

```
import java.io.*;
  public class PrintWriterDemo {
      public static void main(String args[]) {
                PrintWriter pw = new PrintWriter(System.out,  true);
                pw.println ("Это строка") ;
                int i = -7;
                pw.println(i);
                double d = 4.5e-7;
                pw.println(d);  } }
```

## Чтение и запись файлов

В Java все файлы байт-ориентированы, и Java предоставляет методы для чтения и записи байтов в файл. Java позволяет также поместить байт-ориентированные файловые потоки в оболочки символ-ориентированных объектов.

Два из наиболее часто используемых потоковых класса - это FileInputStream и FileOutputStream, которые создают байтовые потоки, связанные с файлами. Чтобы открыть файл нужно создать объект одного из этих классов, указав имя файла в качестве аргумента конструктора.

Некоторые конструкторы:
**FileInputStream(String fileName) throws FileNotFoundException**
**FileOutputStream(String fileName) throws FileNotFoundException**

fileName - имя открываемого файла. При создании входного потока, если файл не существовал, выбрасывается исключение FileNotFoundException. Для выходных потоков, если файл не может быть создан, выбрасывается исключение FileNotFoundException. При открытии выходного файла любой ранее существовавший файл с тем же именем уничтожается.

При завершении работы с файлом нужно закрыть его вызовом метода close(). Этот метод определен и в FileInputStream и в FileOutputStream.

**void close() throws IOException**

Чтобы читать файл можно использовать версию метода read(), который определен в FileInputStream.

**int read() throws IOException**

Метод read() читает единственный байт из файла и возвращает его как целое число. read() возвращает -1, когда достигнут конец файла. Метод read() может возбуждать исключение IOException.

```
import java.io.*;
class ReadFile {
  public static void main(String args[ ]) throws IOException {
    int i;
    FileInputStream fin;

      try {
            fin = new FileInputStream(args[0]);  }

     catch(FileNotFoundException e) {
            System.out.println("Файл не найден"); }
            return; }

     catch(ArrayIndexOutOfBoundsException e) {
           System.out.println (“Файл не задан”);
           return;  }

  do {
    i = fin.read();
    if(i !=-l) System.out.print((char) i);
      } while(i !=-l);

  fin.close();
} }
```

```
import java.io.*;
class ReadFile {
  public static void main(String args[ ]) throws IOException {
   int i;
   FileInputStream fin;
     try {
           fin = new FileInputStream("input.txt");  }
    catch(FileNotFoundException e) {
           System.out.println("Файл не найден");
           return; }
  do {
   i = fin.read();
   if(i!=-1) System.out.print((char)i);
     } while(i!=-1);

  fin.close();
} }
```

# NIO

Одним из наиболее примечательных пакетов в java, появившихся в последнее время, является NIO (New I/O). Этот пакет интересен тем, что для выполнения операций ввода-вывода он поддерживает каналы.

java.nio - пакет верхнего уровня в системе NIO. Инкапсулирует различные типы буферов, содержащих данные, с которыми работает система NIO.
java.nio.channels - поддерживает каналы, открывающие соединения ввода-вывода.
java.nio.channels.spi - поддерживает поставщиков услуг для каналов.
java.nio.charset - инкапсулирует наборы символов, поддерживает работу кодеров и декодеров, преобразующих символы в байты, и байты в символы.
java.nio.charset.spi - поддерживает поставщиков услуг для наборов символов.

# ОСНОВЫ NIO

Система NIO построена на двух элементах: буферах и каналах. Буфер хранит данные, а канал представляет открытое соединение с устройством ввода-вывода с файлом или сокетом. Для использования системы NIO необходимо получить канал для устройства ввода-вывода и буфер для хранения данных.

## Буферы

Буферы определены в пакете java.nio. Все буферы являются подклассами класса Buffer, который определяет основные функциональные особенности, характерные для каждого буфера: текущая позиция, лимит и вместимость. Текущая позиция представляет собой индекс в буфере, с которого в следующий раз начнется операция чтения или записи данных. Текущая позиция перемещается после выполнения большинства операций чтения или записи. Лимит представляет собой индекс, обозначающий конец буфера. Вместимость определяет количество элементов, которые может хранить буфер.

На основе класса Buffer можно получить специфические классы буферов (тип хранимых данных определяется по именам):
ByteBuffer CharBuffer DoubleBuffer FloatBuffer IntBuffer LongBuffer ShortBuffer.

Каждый буфер поддерживает различные методы get() и put(), которые позволяют получать данные из буфера или вносить их.

## Каналы

Каналы определены в пакете java.io.channels. Канал представляет открытое соединение с источником или приемником ввода-вывода. Получение канала осуществляется с помощью метода getChannel() объекта, поддерживающего каналы. Метод getChannel() поддерживается следующими классами ввода-вывода:

DatagramSocket FileInputStream FileOutputStream RandomAccessFile ServerSocket Socket.

Для получения канала потребуется сначала получить объект одного из этих классов, а затем вызвать метод getChannel() для данного объекта. Тип возвращаемоrо канала зависит от типа объекта, для которого вызывается метод getChannel(). Например, при вызове FileInputStream, FileOutputStream или RandomAccessFile метод getChannel() возвращает канал типа FileChannel. При вызове Socket метод getChannel() возвр.ащает SocketChannel. Каналы FileChannel и SocketChannel поддерживают различные методы read() и write(), которые позволяют выполнять операции ввода-вывода через канал.

Чтение из файла

```
import java.io.*;
import java.nio.*;
import java.nio.channels.*;
public class ChannelRead {
   public static void main(String args[]) {
       FileInputStream   fin;
       FileChannel   fchan;
       long fsize;
       ByteBuffer mbuf;
         try {  fin = new FileInputStream (“input.txt“); //файл для ввода данных
                fchan = fin.getChannel();                    //канал для этого файла
                fsize = fchan.size();                         //узнаем размер файла
                mbuf = ByteBuffer.allocate((int)fsize);    // выделяем буфер
                fchan.read(mbuf);                           //считываем файл в буфер
                mbuf.rewind();                               // переходим в начало буфера
                  for(int i=0; i < fsize; i++)
                      System.out.print((char)mBuf.get()); //Считываем байты из буфера
              System.out.println();
              fchan.close();     // закрываем канал
              fin.close();         // закрываем файл
        catch (IOException exc) { System.out.println(exc);
                                  System.exit(1);  }
}}
```

Запись в файл

```
import java.io.*;
import java.nio.*;
import java.nio.channels.*;
public class ChannelWrite {
   public static void main(String args[]) {
      FileOutputStream fout;
      FileChannel fchan;
      ByteBuffer mbuf;
         try { fout = new FileOutputStream(“out.txt”);
               fchan = fout.getChannel();     //канал для выходного файла
               mbuf = ByteBuffer.allocateDirect(26);  // создаем буфер
                for(int i=0; i<26; i++)
                   mbuf.put( (byte) ('A' + i));  //записываем несколько байтов
                mbuf.rewind();                    //переходим в начало буфера
                fchan.write(mbuf);           //записываем буфер в выходной файл
                fchan.close();
                fout.close();   }
         catch (IOException exc) { System.out.println(exc);
                                   System.exit(l); }
}}
```

## Класс Formatter (java.util)

Осуществляет поддержку форматированного вывода, предлагает преобразования формата, позволяющие отображать числа, строки, время и даты в любом виде. Formatter работает подобно функции C/C++ printf(), преобразуя бинарную форму данных в форматированный текст. Formatter сохраняет форматированный текст в буфере, содержимое которого может быть доступно программе в любой момент. Formatter может создавать буфер автоматически, или можно указать этот буфер явно при создании объекта Formatter. Formatter может направлять свой буфер в файл.

Конструкторы:

**Formatter()**

**Formatter(String filename) throws FileNotFoundException**

**Formatter(String filename, String charset) throws FileNotFoundException, UnsupportedEncodingException**

**Formatter(File outF) throws FileNotFoundException**

**Formatter(OutputStream outStrm)**

Параметр **filename** специфицирует имя файла, который будет принимать форматированный вывод. Параметр **charset** указывает набор символов. Если не указано никакого набора символов, то используется набор символов по умолчанию. Параметр **outF** представляет собой ссылку на открытый файл, который должен принимать вывод. Параметр **outStrm** задает ссылку на выходной поток, куда будет направлен вывод. Когда используется файл, вывод также пишется в файл.

```
public static void main(String args[]){
    Formatter fmt = new Formatter();
    fmt.format("Форматирование %s просто %d %f", "с Java", 10, 98.6);
    System.out.println(fmt);  }
```

## Класс Scanner (java.util)

-дополнение к Formatter. Scanner читает форматированный ввод и преобразует его в бинарную форму. Scanner может при меняться для чтения ввода с консоли, из файла, из строки или из другого источника.
Scanner может быть создан для String, InputStream, File или любого другого объекта, реализующего интерфейсы Readable, либо ReadableByteChannel.

```
FileReader fin = new FileReader("Test.txt");
Scanner src = new Scanner(fin);
```

FileReader реализует интерфейс Readable. Это класс для чтения символов из файла.

## Параллельные утилиты

Java имеет встроенную поддержку многопоточности и синхронизации. Например, новые потоки можно создавать путем реализации интерфейса Runnable или расширения класса Thread, синхронизация доступна посредством использования ключевого слова synchronized, а межпотоковые коммуникации поддерживаются методами wait() и notify() из класса Object.

Первоначальная поддержка многопоточности была лишена некоторых высокоуровневых функциональных возможностей (например, семафоров, пулов потоков и диспетчеров), которые способствуют созданию программ, работающих в параллельном режиме. Для поддержки параллельных программ в jdk5 были добавлены параллельные утилиты. Параллельные утилиты предоставляют множество возможностей, предлагают семафоры, циклические барьеры, пулы потоков, диспетчеры выполнения, блокировки, множество параллельных коллекций.

## Пакеты параллельного API

Параллельные утилиты содержатся в пакете **java.util.concurrent** и в двух подпакетах **java.util.concurrent.atomic** и **java.util.concurrent.locks**.

**java.util.concurrent** определяет основные функциональные возможности, которые поддерживают альтернативные варианты встроенных методов синхронизации и межпотоковых коммуникаций. Он определяет следующие ключевые функциональные особенности: синхронизаторы, исполнители, параллельные коллекции.

Синхронизаторы предлагают высокоуровневые способы синхронизации взаимодействий между несколькими потоками.

Классы синхронизаторов, определенные в пакете java. util. concurrent

**Semaphore** - классический семафор.

**CountDownLatch** – класс-синхронизатор, который заставляет один или несколько потоков ждать, пока набор операций, выполняемых в других потоках завершится.

**CyclicBarrier** - класс-синхронизатор, который позволяет набору потоков позволяет набору потоков войти в режим ожидания в заданной программы.

**Exchanger** – осуществляет обмен данными между двумя потоками.

Исполнители управляют выполнением потоков. Первым в иерархии исполнителей является интерфейс **Executor,** который служит для запуска потока.

**ExecutorService** расширяет Executor и предлагает методы, управляющие выполнением. Существуют две реализации **ExecutorService**: **ThreadPoolExecutor** и **ScheduledThreadPoolExecutor**.

Пакет **java.util.concurrent** определяет служебный класс **Executors**, который содержит несколько статических методов, упрощающих создание разнообразных исполнителей. С исполнителями связаны интерфейсы **Future** и **Callable**.

**Future** содержит значение, возвращаемое потоком после его выполнения. Таким образом, его значение определяется "в будущем", когда поток завершит свое выполнение.

**Callable** определяет поток, возвращающий значение.

java.util.concurrent определяет несколько параллельных классов коллекций, включая **ConcurrentHashMap**, **ConcurrentLinkedQueue** и **CopyOnWriteArraylist**. Эти классы предлагают параллельные альтернативные варианты для связанных с ними классов, определенных в каркасе коллекций.

**java.util.concurrent.atomic** упрощает использование переменных в параллельной среде. Он предлагает средства эффективного обновления значений переменной без применения блокировок.

**java.util.concurrent.locks** предлагает альтернативный вариант работы с методами synchronized. В основе лежит интерфейс Lock, определяющий основной механизм, который используется для получения доступа к объекту и отказа в доступе. Ключевыми методами являются lock(), tryLock() и unlock(). Эти методы расширяют возможности управления синхронизацией.

Класс **Semaphore** реализует классический семафор. Семафор управляет доступом к общему ресурсу с помощью счетчика. Если счетчик больше нуля, доступ разрешается. Если он равен нулю, в доступе будет отказано. Этот счетчик подсчитывает разрешения, открывающие доступ к общему ресурсу. Чтобы получить доступ к ресурсу, поток должен получить разрешение на доступ у семафора.

**Сервлеты**

Сервлетами (servlet) называются небольшие программы, которые выполняются на стороне сервера Web-соединения.

Для создания сервлетов вам будет необходим доступ к среде разработки сервлетов. Это может быть
**JSDK – Java Servlet Development Kit** с **http:// java.sun.com**
или **Tomcat**. Tomcat представляет собой открытый продукт, который поддерживает проект **jakarta** от фонда Apache Software Foundation. Этот продукт содержит библиотеки классов, документацию и среду времени выполнения, что будет необходимо для создания и тестирования сервлетов. **http://jakarta.apache.org**

Классы и интерфейсы, необходимые для построения сервлетов, содержатся в двух пакетах: **javax.servlet** и **javax.servlet.http**
Они являются стандартными расширениями, предлагаемыми Tomcat. (не входят в состав Java SE6)

Жизненный цикл сервлета определяют три основных метода:
**init(), service()** и **destroy().** Они реализуются каждым сервлетом и вызываются сервером в определенное время.

1. Пусть пользователь ввел в окне браузера URL-адрес. На основании этого URL-адреса браузер генерирует HTTP-запрос, посылаемый соответствующему серверу.

2. HTTP-запрос принимает Web-сервер. Сервер находит соответствие между запросом и конкретным сервлетом. Сервлет динамически принимается и загружается в адресное пространство сервера.

3. Сервер вызывает метод init() сервлета. Этот метод вызывается только тогда, когда сервлет впервые загружается в память компьютера. Сервлету можно передавать параметры инициализации, поэтому он может конфиурировать себя самостоятельно.

4. Сервер вызывает метод service() сервлета. Этот метод вызывается для обработки HTTP-запроса. Сервлет может считывать данные, содержащиеся в HTTP-запросе. Он может также сформулировать HTTP-отклик клиенту. Сервлет остается в адресном пространстве и является доступным для обработки любых других HTTP-запросов, полученных от клиентов. Метод service () вызывается для каждого HTTP-запроса.

5. Сервер может принять решение выгрузить сервлет из памяти. Для освобождения ресурсов, выделенных для сервлета, сервер вызывает метод destroy ().