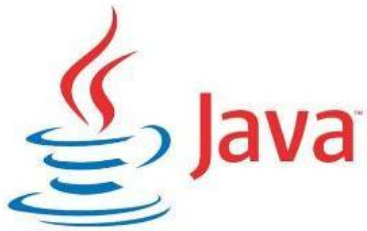


Консольный ввод-вывод

Байтовые и символьные потоки

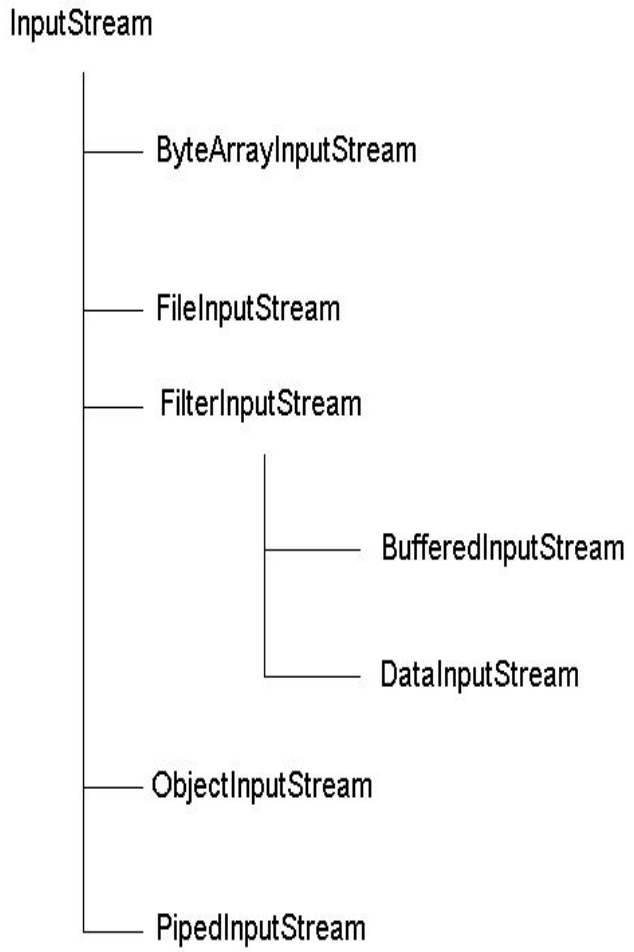
Аналогично языку C++ ввод/вывод в языке Java выполняется с использованием потоков. Поток ввода/вывода - это некоторый условный канал, по которому отсылаются и получаются данные.

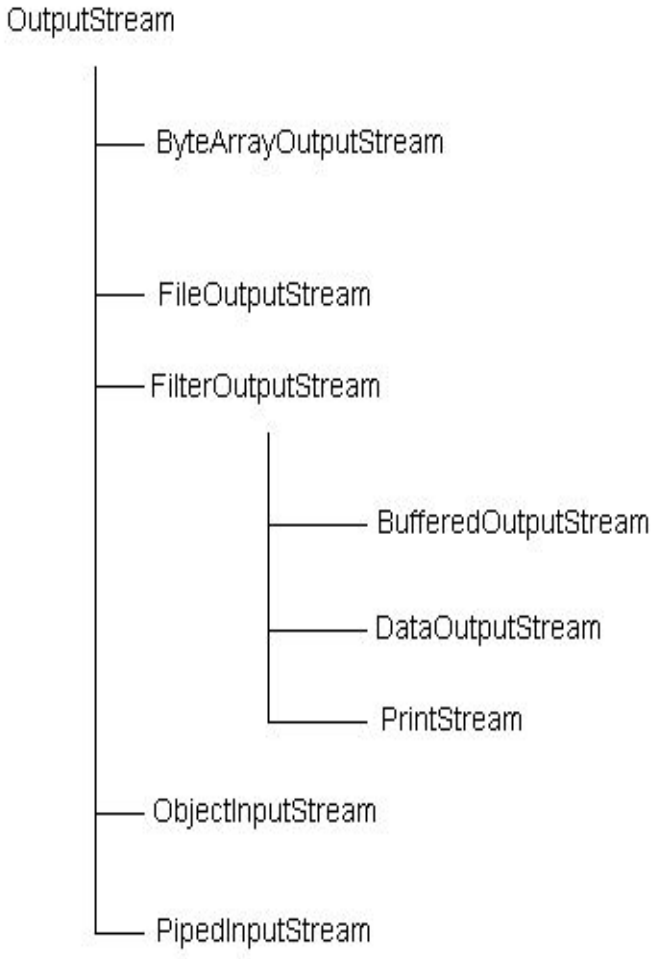
В Java 2 реализованы 2 типа потоков: байтовые и символьные. Байтовые потоки предоставляют удобные средства для обработки, ввода и вывода байтов или других двоичных объектов. Символьные потоки используются для обработки, ввода и вывода символов или строк в кодировке Unicode.

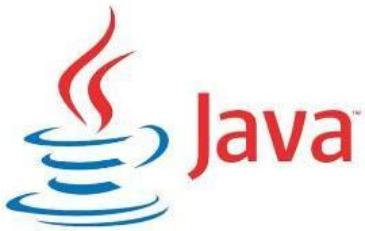
Все классы для консольного I/O – пакет java.io: import java.io.*;

Консольный ввод-вывод

Байтовые потоки

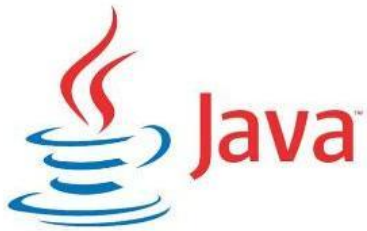






Консольный ввод-вывод
Байтовые потоки.
Абстрактный класс InputStream.

public abstract int read() throws IOException	Читает 1 байт из вх. потока. Результат в младшем байте.
public int read (byte[ ] b) throws IOException	Читает послед-ть байт в массив b[ ]. Возвращает кол-во прочитанных байт или -1.
public int read (byte[ ] b, int off, int len) throws IOException	Заполняет массив с указанного байта, читает не более len символов
public long skip (long n) throws IOException	Пропускает n байт в потоке
int available ()	Возвращает кол-во доступных байт в потоке.
void close ()	Закрывает поток ввода.

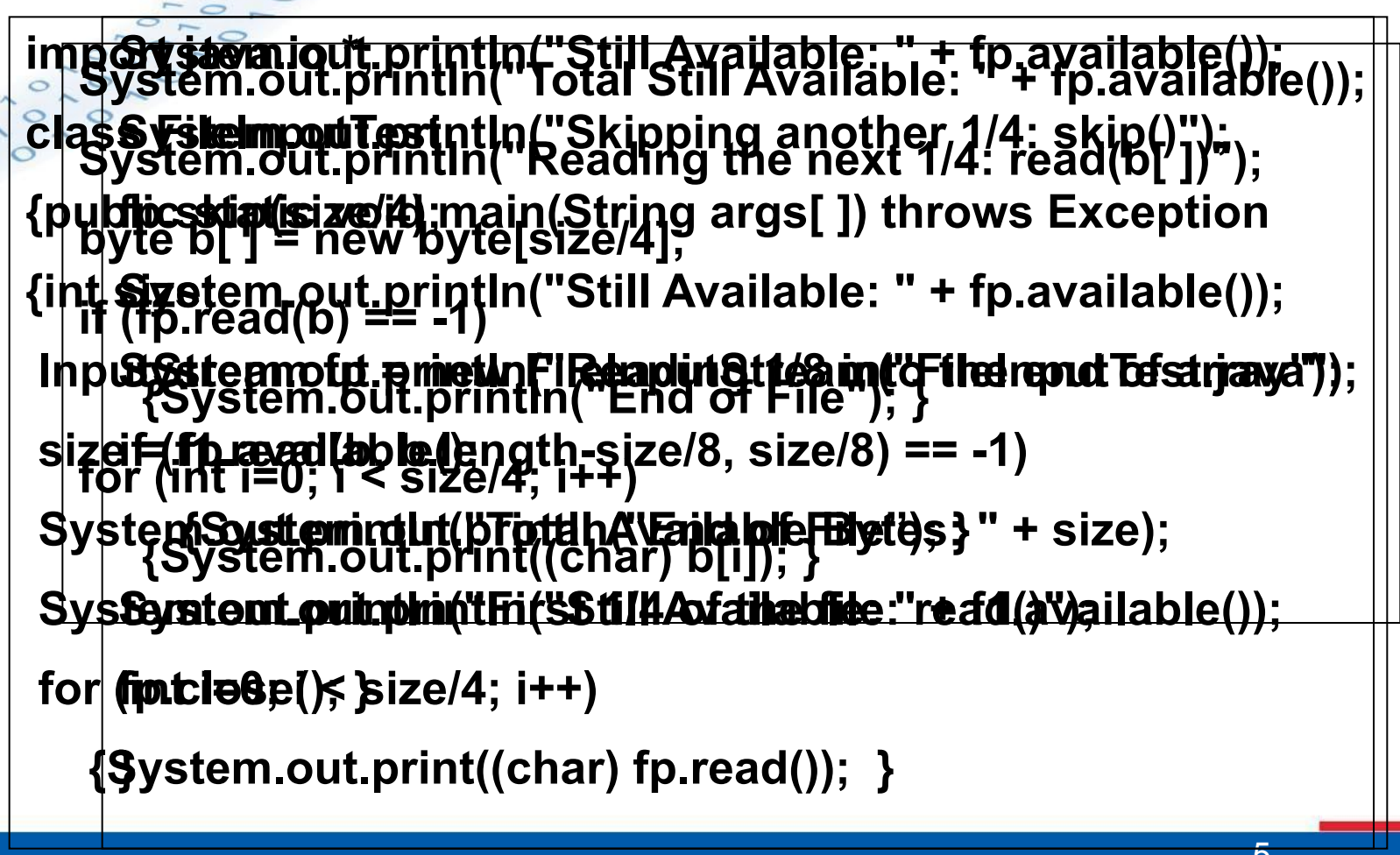


Консольный ввод-вывод
Байтовые потоки.
Класс FileInputStream.

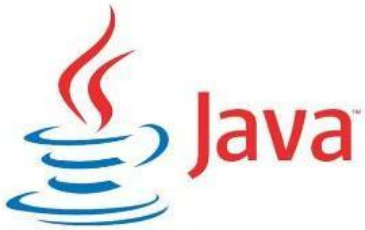
FileInputStream(String name) throws FileNotFoundException
FileInputStream (File file) throws FileNotFoundException

В классе FileInputStream переопределяется большая часть методов класса Input-Stream (в т.ч. абстрактный метод read()). Когда создается объект класса FileInputStream, он одновременно с этим открывается для чтения.

Консольный ввод-вывод
Байтовые потоки.
Класс FileInputStream.



```
import java.io.*;
class FileInputTest
{public static void main(String args[ ]) throws Exception
{int size;
InputStream fp = new FileInputStream("FileInputTest.java");
size = fp.available();
System.out.println("Total Available Bytes: " + size);
System.out.println("First Half Size: " + size/2);
for (int i=0; i < size/2; i++)
   {System.out.print((char) fp.read());  }
System.out.println("Still Available: " + fp.available());
System.out.println("Skipping another 1/4: skip()");
fp.skip(size/4);
System.out.println("Still Available: " + fp.available());
System.out.println("Reading the next 1/4: read(b[ ])");
byte b[ ] = new byte[size/4];
if (fp.read(b) == -1)
   {System.out.println("End of File"); }
for (int i=0; i < size/4; i++)
   {System.out.print((char) b[i]); }
System.out.println("Still Available: " + fp.available());
System.out.println("Reading 1/8 into the end of the array");
if (fp.read(b, size/8, size/8) == -1)
   {System.out.println("End of File"); }
for (int i=0; i < size/4; i++)
   {System.out.print((char) b[i]); }
System.out.println("Still Available: " + fp.available());
fp.close();
}
```

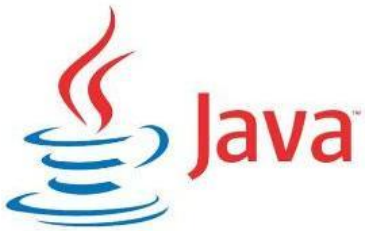


Консольный ввод-вывод
Байтовые потоки. Класс ByteArrayInputStream.

ByteArrayInputStream – это реализация входного потока, в котором в качестве источника используется массив типа byte. У этого класса два конструктора, каждый из которых в качестве первого параметра требует байтовый массив.

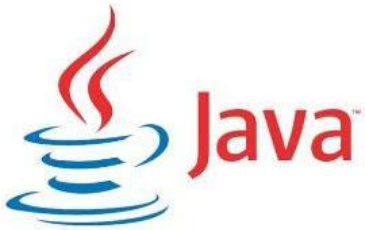
ByteArrayInputStream(byte array[ ])

ByteArrayInputStream(byte array[ ], int start, int numBytes)



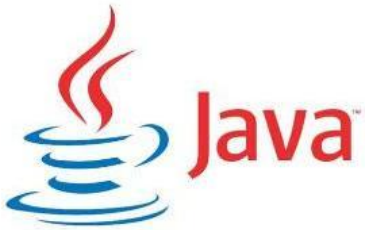
Консольный ввод-вывод
Байтовые потоки.
Класс ByteArrayInputStream.

```
import java.io.*;
class ByteArrayTest
{public static void main(String args [ ]) throws IOException
    {byte b[ ] = {0,1,2,3,4,5,6,7,8,9};
     ByteArrayInputStream input1 =
                new ByteArrayInputStream(b);
     ByteArrayInputStream input2 =
                new ByteArrayInputStream(b,0,3);
    }
}
```



Консольный ввод-вывод
Байтовые потоки.
Абстрактный класс OutputStream.

public abstract void write (int b) throws IOException	Записывает 1 байт в выходной поток.
public void write (byte[ ] b) throws IOException	Записывает в поток массив байт
public void write (byte[ ] b, int off, int len) throws IOException	Записывает часть массива в поток (len элементов начиная с элемента off)
public void flush() throws IOException	Немедленно выталкивает из буфера в поток все что накоплено в буфере.
public void close() throws IOException	Закрывает поток.



Консольный ввод-вывод
Байтовые потоки.
Класс FileOutputStream.

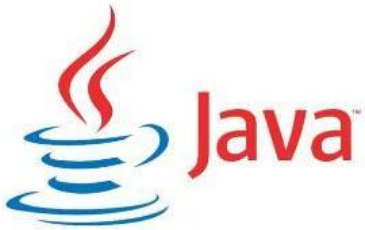
Класс FileOutputStream можно применять для записи байтов в файл. У класса FileOutputStream есть 3 конструктора:

FileOutputStream (String filePath) throws FileNotFoundException

FileOutputStream (File fileObj) throws FileNotFoundException

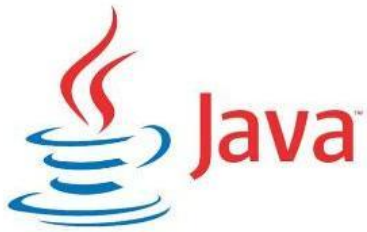
FileOutputStream (String filePath, boolean append)
throws FileNotFoundException

- filePath – полное имя файла,
- fileObj – объект типа File, который описывает файл
- append (=true – добавление информации в существующий файл)



Консольный ввод-вывод
Байтовые потоки.
Класс FileOutputStream.

```
try
{FileInputStream Source = new FileInputStream("infile.dat");
 FileOutputStream Dest = new  FileOutputStream("outfile.dat");
 int c;
 while ((c = Source.read()) != -1)
     {Dest.write(c); }
}
catch (FileNotFoundException ex)
{… }
finally
{Source.close(); Dest.close(); }
```



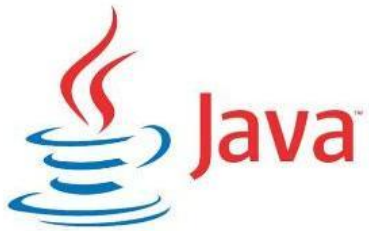
Консольный ввод-вывод
Байтовые потоки.
Класс ByteArrayOutputStream.

ByteArrayOutputStream( ); - 32 байта

ByteArrayOutputStream(int numBytes);

Метод, записывающий содержимое одного потока в другой:
public void writeTo(OutputStream out) throws IOException

```
byte buf [ ] = {‘a’,’b’,’c’,’d’};
ByteArrayOutputStream b = new ByteArrayOutputStream();
b.write(buf);
FileOutputStream f = new FileOutputStream(“result.txt”);
b.writeTo(f);
```



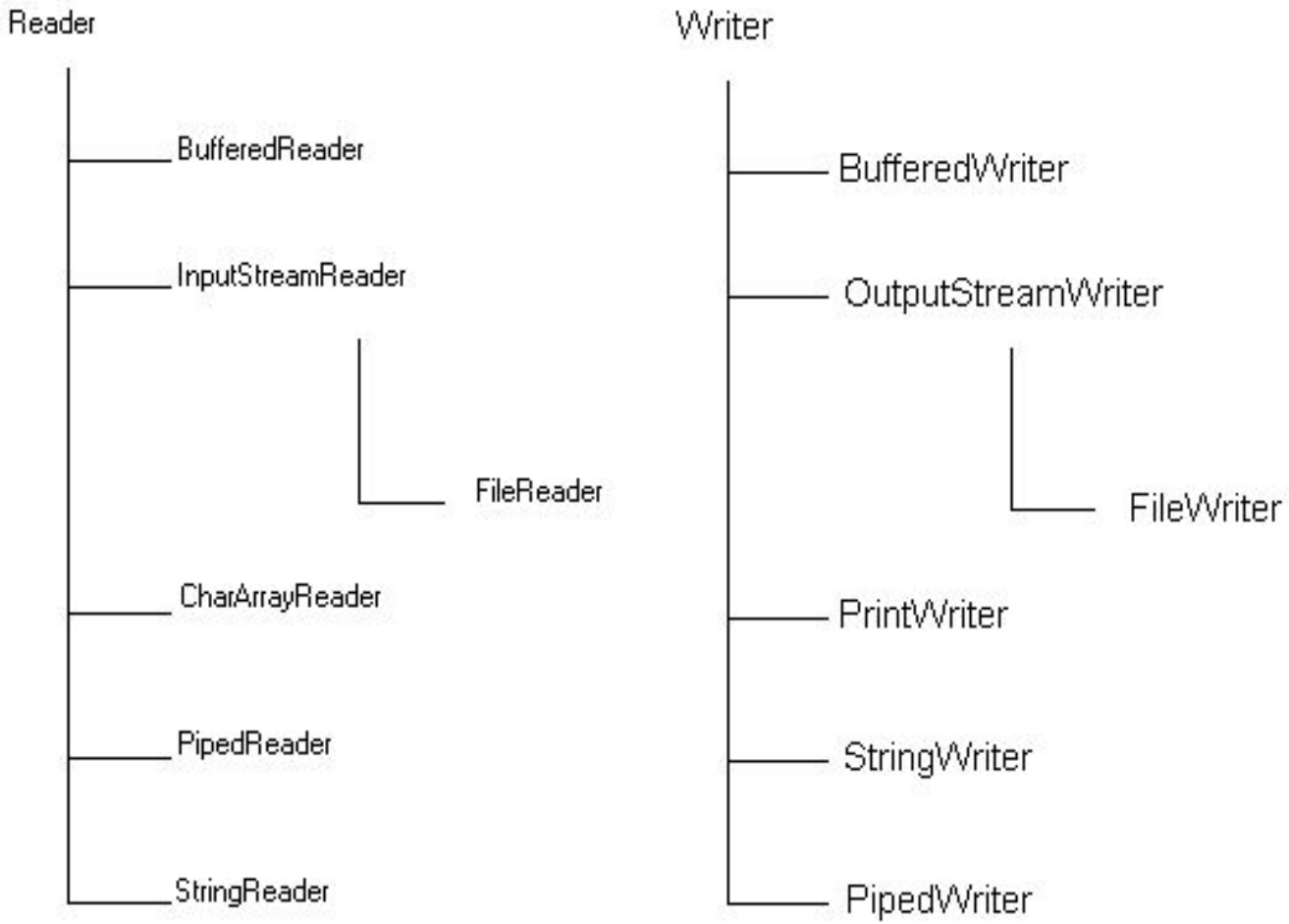
Консольный ввод-вывод

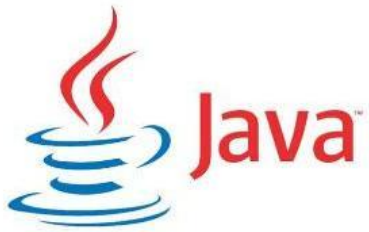
Символьные потоки.

В Java символы хранятся в кодировке Unicode. Символьный поток I/O автоматически транслирует Символы между форматом Unicode и локальной кодировкой, пользовательского ПК.

Консольный ввод-вывод

Символьные потоки.





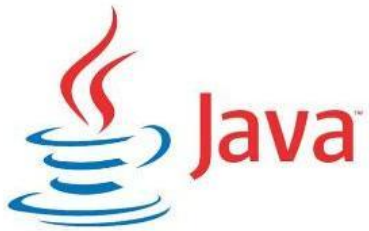
Консольный ввод-вывод

Символьные потоки.

Методы классов Reader и Writer аналогичны методам классов InputStream и OutputStream с той разницей, что все аргументы типа byte заменены на аргументы типа char.

Классы CharArrayReader и CharArrayWriter соответствуют классам ByteArrayInputStream и ByteArrayOutputStream с той же разницей.

Классы FileReader и FileWriter являются символьными версиями потоковых классов FileInputStream и FileOutputStream.



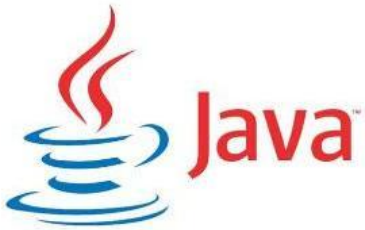
Консольный ввод-вывод

Символьные потоки.

Классы InputStreamReader и OutputStreamWriter – переходники между байтовыми и символьными потоками. Байтовый поток используется для физического ввода-вывода, а символьный поток преобразует байты в символы с учетом кодировки.

InputStreamReader (InputStream obj)

OutputStreamWriter (OutputStream out)



Консольный ввод-вывод

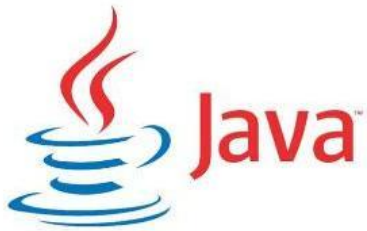
Буферизованный ввод-вывод.

В библиотеке Java имеются также буферизованные потоки I/O.

Для буферизованных потоков операции чтения и записи происходят с буфером, находящимся в памяти.

Когда буфер пуст выполняется реальная операция чтения из потока, когда буфер полон – реальная операция записи в поток.

Буферизация может быть добавлена к любому байтовому либо символьному потоку с помощью специальных классов «оболочек» (“wrappers”). При этом конструктору буферизованного потока передается не буферизованный поток.



Консольный ввод-вывод

Буферизованный ввод-вывод.

Буферизованные потоки I/O:

BufferedInputStream, BufferedOutputStream - байтовые

BufferedReader, BufferedWriter – символьные

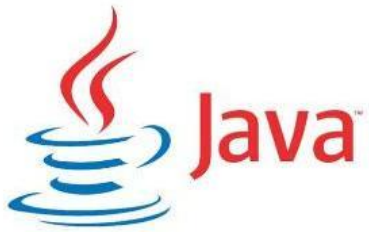
Конструкторы:

BufferedInputStream(InputStream in)

BufferedOutputStream(OutputStream out)

BufferedReader(Reader in)

BufferedWriter(Writer out)



Консольный ввод-вывод

Пример организации консольного ввода-вывода

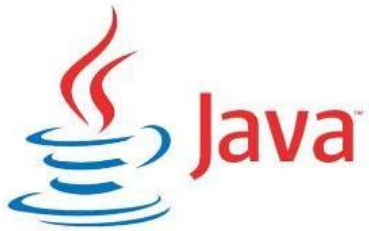
В пакете java.lang есть класс с именем System.

В классе System определены три переменные:

```
public static InputStream in;
public static PrintStream out;
public static PrintStream err;
```

System.out и System.err – байтовые потоки, эмулирующие поддержку локального character set !

System.in – байтовый поток !



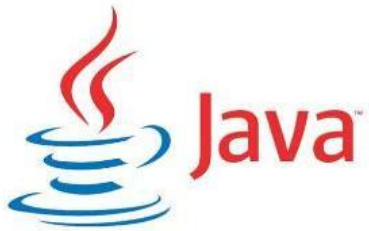
Консольный ввод-вывод

Пример организации консольного вывода

Класс PrintStream содержит методы print() и println() для всех базовых типов данных.

Т.о. для консольного вывода надо вызвать System.out.print() или System.out.println()

```
System.out.println (“Значение a = ” +a);
System.out.print (str);
```



Консольный ввод-вывод

Класс Scanner начиная с JSDK 1.5

```
import java.util.Scanner;
…
Scanner sc = new Scanner(System.in);
…
System.out.println(“Input a:”);
int a = sc.nextInt();
System.out.println(“Input str:”);
String str = sc.next();
```

Подробнее см.

http://docs.oracle.com/javase/1.5.0/docs/api/index.html?java/util/Scanner.html

Консольный ввод-вывод

Класс File

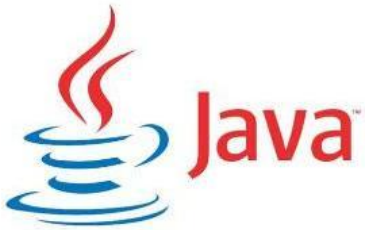
Класс File представляет имя файла, но не сам файл (если файл не существует, он не создается, если существует с ним можно проводить производить операции через объект File) !

Конструкторы:

public File(String pathname)

public File(String pathname, String filename)

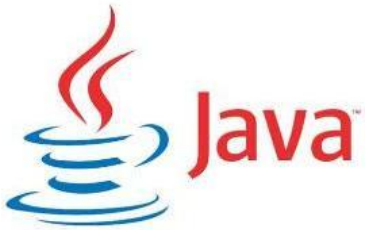
public File (File parent, String child)



Консольный ввод-вывод

Класс File

public boolean exists()	Существует ли файл?
public boolean canRead()	Возможен ли доступ на чтение?
public boolean canWrite()	Возможен ли доступ на запись
public boolean isHidden()	Является ли файл скрытым?
public boolean isFile()	Файл?
public boolean isDirectory()	Каталог?
public boolean delete()	Удаляет файл, если он существует. Каталог удаляется только если он пуст.
public String[ ] list()	Возвращает список файлов в каталоге



Консольный ввод-вывод

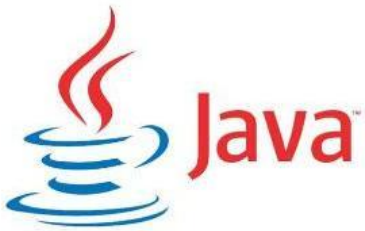
Класс File

public boolean mkdir()	Создает пустой каталог
public boolean renameTo (File newName)	Переименование файла
public boolean setReadOnly()	Устанавливает атрибут ReadOnly

Сериализация в Java

Что такое сериализация?

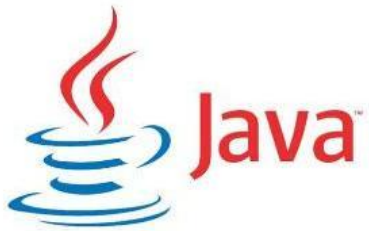
Сериализация это процесс сохранения состояния объекта в последовательность байт; десериализация это процесс восстановления объекта из этих байт. Java Serialization API предоставляет стандартный механизм для создания сериализуемых объектов.



Сериализация в Java

Для чего нужна сериализация?

В Java всё представлено в виде объектов. Если двум компонентам Java необходимо общаться друг с другом, то им необходим механизм для обмена данными. Следовательно, должен быть универсальный и эффективный протокол передачи объектов между компонентами. Сериализация создана для этого, и компоненты Java используют этот протокол для передачи объектов.



Сериализация в Java

3 механизма сериализации

Сериализация

1) используя протокол по умолчанию

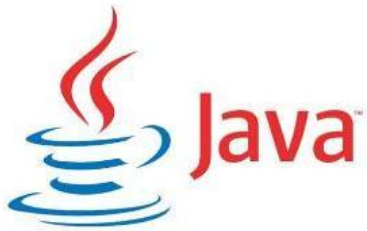
2) модифицируя протокол по умолчанию

3) создавая свой собственный протокол

Сериализация в Java

1) Протокол по умолчанию

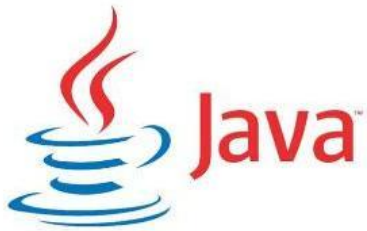
Чтобы объект стал сериализуемым, необходимо, чтобы он реализовывал интерфейс java.io.Serializable. Интерфейс Serializable это интерфейс-маркер; в нём не задекларировано ни одного метода. Но он говорит сериализующему механизму, что класс может быть сериализован.



Сериализация в Java

1) Протокол по умолчанию. Сохранение объекта

```
import java.io.Serializable;

import java.util.Date;

import java.util.Calendar;

public class PersistentTime implements Serializable  {
     private Date time;


     public PersistentTime() {
         time = Calendar.getInstance().getTime();
     }
     public Date getTime() {
         return time;
      }

}
```

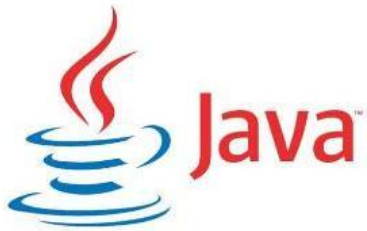


Сериализация в Java

1) Протокол по умолчанию. Сохранение объекта

Для сохранения объекта как последовательности байт используется класс java.io.ObjectOutputStream

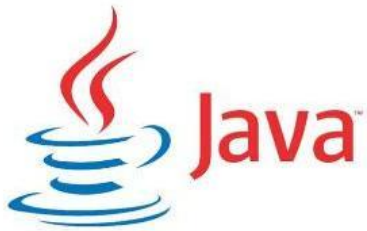
Этот класс является фильтрующим потоком (filter stream) - он окружает низкоуровневый поток байтов (называемый узловым потоком (node stream)) и предоставляет нам поток сериализации. Узловые потоки могут быть использованы для записи в файловую систему, сокеты и т.д. Это означает, что можно, например, передавать разложенные на байты объекты по сети и затем восстанавливать их на других компьютерах!



Сериализация в Java

1) Протокол по умолчанию. Сохранение объекта

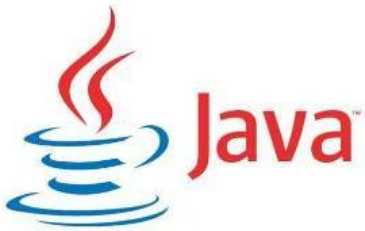
```
import java.io.ObjectOutputStream;
import java.io.FileOutputStream;
import java.io.IOException;
public class FlattenTime  {
    public static void main(String [] args)   {
        String filename = "time.ser";
        PersistentTime time = new PersistentTime();
        FileOutputStream fos = null;
       ObjectOutputStream out = null;
```



Сериализация в Java

1) Протокол по умолчанию. Сохранение объекта

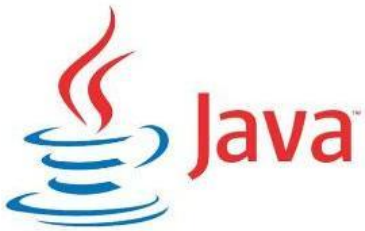
```
        try  {
            fos = new FileOutputStream(filename);
            out = new ObjectOutputStream(fos);
            out.writeObject(time);    //сериализация
            out.close();
          }
          catch(IOException ex)   {
             ex.printStackTrace();
          }
    }
}
```



Сериализация в Java

1) Протокол по умолчанию. Восстановление объекта

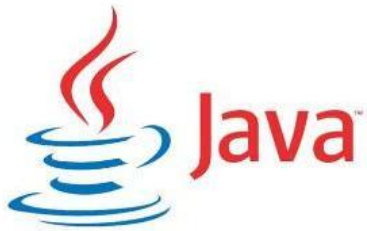
Для восстановления объекта используется метод readObject() класса java.io.ObjectInputStream. Метод считывает последовательность байтов и создает объект, полностью повторяющий оригинал. Поскольку readObject() может считывать любой сериализуемый объект, необходимо его присвоение соответствующему типу. Т.о., из системы, в которой происходит восстановление объекта, должен быть доступен файл класса. Т.е. при сериализации не сохраняется ни файл класса объекта, ни его методы, сохраняется лишь состояние объекта.



Сериализация в Java

1) Протокол по умолчанию. Восстановление объекта

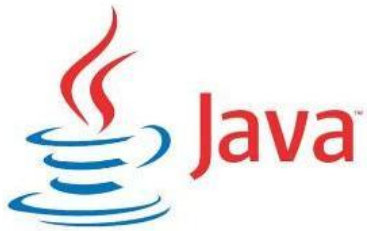
```
import java.io.ObjectInputStream;
import java.io.FileInputStream;
import java.io.IOException;
import java.util.Calendar;
public class InflateTime {
    public static void main(String [] args)  {
        String filename = "time.ser";
        PersistentTime time = null;
        FileInputStream fis = null;
        ObjectInputStream in = null;
```



Сериализация в Java

1) Протокол по умолчанию. Восстановление объекта

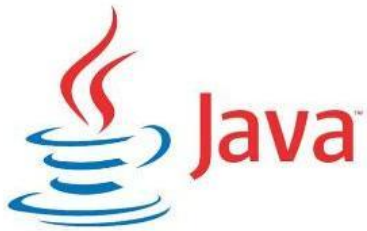
```
try {
    fis = new FileInputStream(filename);
    in = new ObjectInputStream(fis);
    time = (PersistentTime)in.readObject();  //десериализация
    in.close();
}
catch(IOException ex)   {
    ex.printStackTrace();
}
catch(ClassNotFoundException ex)  {
    ex.printStackTrace();
 }
```



Сериализация в Java

1) Протокол по умолчанию. Восстановление объекта

```
    // распечатать восстановленное время
    System.out.println("Время сохранения: " + time.getTime());
    System.out.println();
    // распечатать текущее время
    System.out.println("Текущее время: " +
     Calendar.getInstance().getTime());
  }
}
```



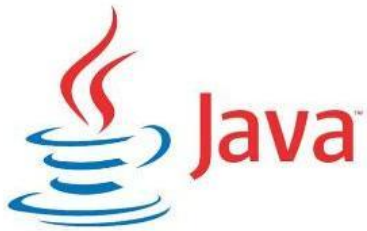
Сериализация в Java

1) Протокол по умолчанию. Ограничения сериализации

1) Если в состав сериализуемого класса A входит поле класса B, то класс B тоже должен быть сериализуемым. Иначе в процессе сериализации возникнет исключение NotSerializableException .

2) В процессе сериализации/ десериализации не участвуют статические поля класса, поскольку они фактически являются не полями объектов (экземпляров класса), а полями класса в целом.

3) Влияние наследования на сериализацию. Пусть класс A объявлен, как сериализуемый. От него унаследован класс B, для которого не указано implements Serializable . От B порожден класс C - сериализуемый. Тогда при сериализации будут сохранены все поля классов A и C , но не B .



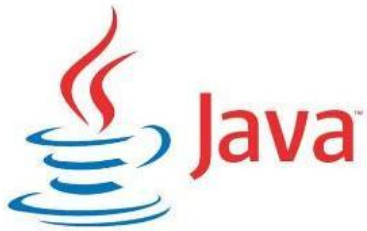
Сериализация в Java

1) Протокол по умолчанию. Несериализуемые поля

Класс java.lang.Object не реализует Serializable, поэтому не все объекты Java могут быть автоматически сохранены.

Например, некоторые системные классы, такие как Thread, OutputStream и его подклассы, и Socket - не сериализуемые.

Причина: сериализация таких классов бессмысленна.

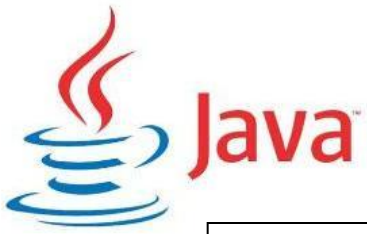


Сериализация в Java

1) Протокол по умолчанию. Несериализуемые поля

Проблема: есть класс, который содержит экземпляр Thread? Можем ли мы в этом случае сохранить объект такого типа?

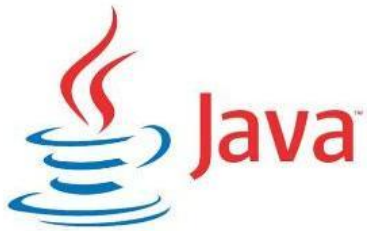
Решение: мы имеем возможность сообщить механизму сериализации о своих намерениях, пометив объект Thread нашего класса как несохраняемый - transient.



Сериализация в Java

1) Протокол по умолчанию. Несериализуемые поля

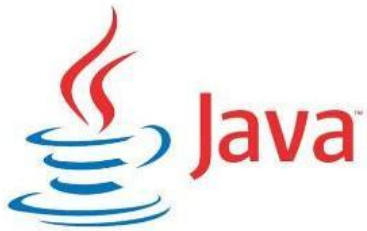
```
import java.io.Serializable;
public class PersistentAnimation implements Serializable, Runnable {
    transient private Thread animator;
    private int animationSpeed;
    public PersistentAnimation(int animationSpeed)   {
        this.animationSpeed = animationSpeed;
        animator = new Thread(this);
        animator.start();
    }
   public void run()  {   …  }
}
```



Сериализация в Java
2) Модификация протокола по умолчанию.

Проблема: как перезапустить анимацию?

Когда мы создаем объект при помощи new, конструктор объекта вызывается только при создании нового экземпляра. Читая объект методом readObject() мы не создаем нового экземпляра, мы просто восстанавливаем сохраненный объект. В результате анимационный поток запустится лишь однажды, при первом создании экземпляра этого объекта.



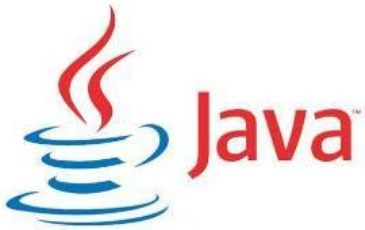
Сериализация в Java
2) Модификация протокола по умолчанию.

Решение:

private void writeObject(ObjectOutputStream out)
throws IOException;

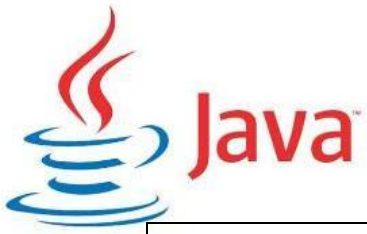
private void readObject(ObjectInputStream in)
throws IOException, ClassNotFoundException;

Виртуальная машина при вызове соответствующего метода автоматически проверяет, не были ли они объявлены в классе объекта.



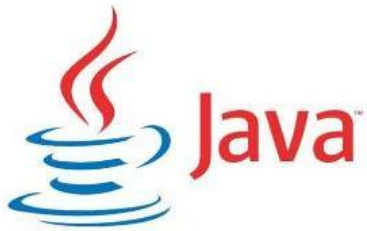
Сериализация в Java
2) Модификация протокола по умолчанию.

```
import java.io.Serializable;
public class PersistentAnimation implements Serializable, Runnable {
    transient private Thread animator;
    private int animationSpeed;
    public PersistentAnimation(int animationSpeed)   {
        this.animationSpeed = animationSpeed;
        startAnimation();
     }
   public void run()  {   …  }
```



Сериализация в Java
2) Модификация протокола по умолчанию.

```
private void startAnimation()  {
     animator = new Thread(this);
     animator.start();
}
private void writeObject(ObjectOutputStream out) throws IOException  {
   out.defaultWriteObject();  //мы не меняем нормальный процесс
}
private void readObject(ObjectInputStream in) throws IOException,
   ClassNotFoundException {
    in.defaultReadObject();    //мы лишь дополняем его
    startAnimation();
}
```



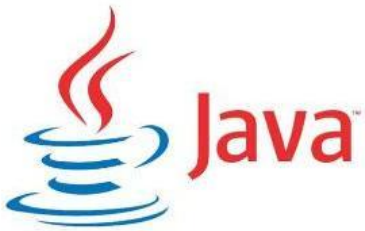
Сериализация в Java

2) Модификация протокола по умолчанию. Запрет сериализации для класса.

Проблема:

```
class A implements Serializable  {
   …
}
class B extends A  {
   …
}
```

Как запретить сериализацию для класса B?

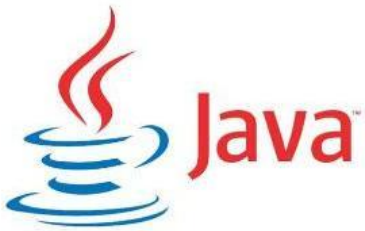


Сериализация в Java

2) Модификация протокола по умолчанию. Запрет сериализации для класса.

Решение:

```
private void writeObject(ObjectOutputStream out) throws IOException
{
   throw new NotSerializableException(“Non serializable class!");
}
private void readObject(ObjectInputStream in) throws IOException
{
   throw new NotSerializableException (“Non serializable class!");
}
```



Сериализация в Java

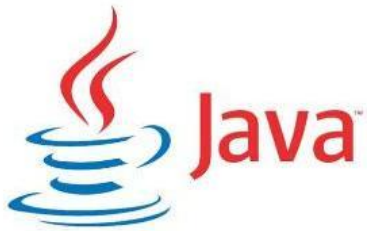
3) Создание собственного протокола

Вместо реализации интерфейса Serializable, можно реализовать интерфейс Externalizable, который содержит два метода:

public void writeExternal(ObjectOutput out) throws IOException;

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

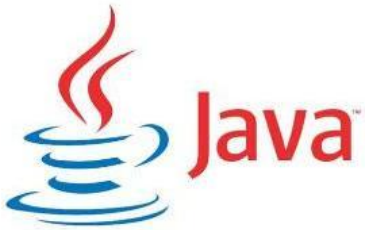
Для создания собственного протокола надо переопределить эти методы. Здесь ничего не делается автоматически. Это наиболее сложный, но и наиболее контролируемый способ.



Сериализация в Java

Кэширование объектов в потоке

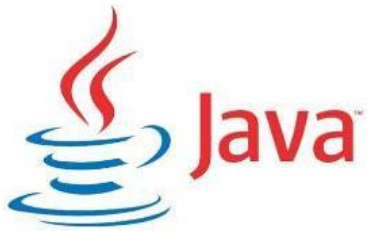
Проблема: Рассмотрим ситуацию, когда объект однажды уже записанный в поток, спустя какое-то время записывается в него снова. По умолчанию, ObjectOutputStream сохраняет ссылки на объекты, которые в него записываются. Это означает, что если состояние записываемого объекта, который уже был записан, будет записано снова, новое состояние не сохраняется!



Сериализация в Java

Кэширование объектов в потоке

```
ObjectOutputStream out = new ObjectOutputStream(...);
MyObject obj = new MyObject(); // должен быть Serializable
obj.setState(100);
out.writeObject(obj); // сохраняет объект с состоянием = 100
obj.setState(200);
out.writeObject(obj); // не сохраняет новое состояние объекта
```

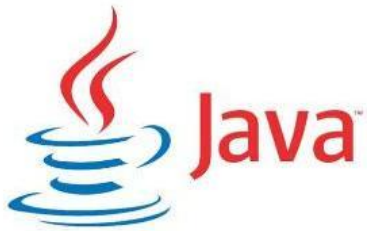


Сериализация в Java

Кэширование объектов в потоке

Решение:

1) Можно каждый раз после вызова метода записи убеждаться в том, что поток закрыт;
2) Можно вызвать метод objectOutputStream.reset(), который сигнализирует потоку о том, что необходимо освободить кэш от ссылок, которые он хранит, чтобы новые вызовы методов записи действительно записывали данные. Будьте осторожны, reset очищает весь кэш объекта, поэтому все ранее записанные объекты могут быть перезаписаны заново.

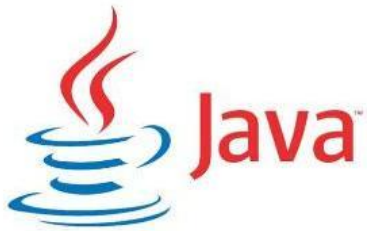


Сериализация в Java

Производительность

Сериализация « по умолчанию» является «медленной» операцией. Она в среднем в 2 – 2,5 раза медленнее записи в поток стандартными средствами ввода-вывода.

Кроме того, так как ссылки на объекты кэшируются в поток вывода, система не может выполнять сбор мусора для записанных в поток объектов если поток не был закрыт. Лучшее решение (как всегда при помощи операций ввода/вывода) - это как можно скорее закрывать потоки после выполнения записи.



Сериализация в Java

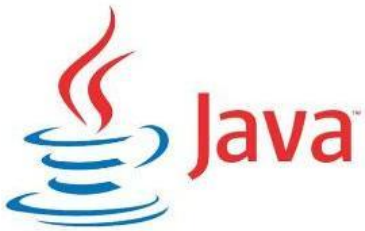
Контроль версий

Что произойдет если класс поменялся и мы пытаемся прочитать объект, сериализованный в старой структуре класса.

Возникнет исключительная ситуация, а именно java.io.InvalidClassException, потому что всем классам, которые могут быть сохранены, присваивается уникальный идентификатор. Если идентификатор класса не совпадает с идентификатором разложенного объекта, возникает исключительная ситуация.

Идентификатор, который является частью всех классов, хранится в поле, которое называется serialVersionUID.

static final long *serialVersionUID = 766141998445722174 3L;*



Сериализация в Java

Контроль версий

Если вы хотите контролировать версии, вы должны вручную задать поле serialVersionUID и убедиться в том, что оно такое же, и не зависит от изменений, внесенных вами в объект.

Можно использовать утилиту, входящую в состав JDK, которая называется serialver, чтобы посмотреть какой код будет присвоен по умолчанию (hash код объекта по умолчанию).

```
> serialver Baz

> Baz: static final long serialVersionUID = 1027553947283749 5L;
```

Просто скопируйте возвращенную строку с идентификатором версии и поместите ее в ваш код. Теперь, если вы внесли какие-либо изменения в файл класса Baz, просто убедитесь что указан тот же идентификатор версии и все будет в порядке.