EPAM Systems, RD Dep.

Конспект и раздаточный материал

JAVA.SE.04 I/O Streams

REVISION HISTORY					
Ver.	**Description of Change**	**Author**	**Date**	**Approved**	
				Name	**Effective Date**
<1.0>	Первая версия	Игорь Блинов	<19.08.2011>		
<2.0>	Вторая версия. Конспект переделан под обновленное содержание материала модуля.	Ольга Смолякова	<05.03.2014>		

Содержание
JAVA.SE.04 I/O Streams

Потоки данных

В Java для описания работы по вводу/выводу используется специальное понятие – поток данных (**stream**).

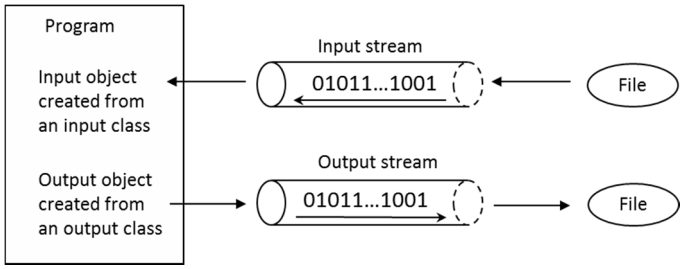
Поток данных связан с некоторым *источником* или *приемником* данных, способных получать или предоставлять информацию.

Соответственно, потоки делятся на **входные** - читающие данные, и на **выходные** - передающие (записывающие) данные.

Введение концепции *stream* позволяет отделить программу, обменивающуюся информацией одинаковым образом с любыми устройствами, от низкоуровневых операций с такими устройствами ввода/вывода.

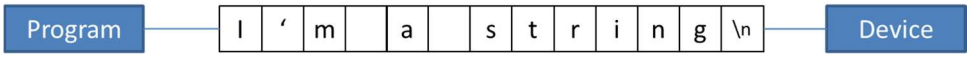
Потоки данных — это упорядоченные последовательности данных, которым соответствует определенный источник (*source*) (для потоков ввода) или получатель (*destination*) (для потоков вывода).

Классы ввода-вывода Java исключают необходимость вникать в особенности низкоуровневой организации операционных систем и предоставляют доступ к системным ресурсам посредством методов работы с файлами и иных инструментов.

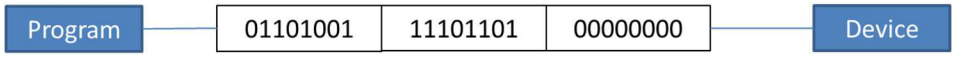


В JAVA существует **2 типа потоков данных**:

- **Символьные потоки** (text-streams, последовательности 16-битовых символов Unicode), содержащие символы.



- **Байтовые потоки** (binary-streams), содержащие 8-ми битную информацию.



Работа с потоками

Общая схема работы с потоками:

- Открыть поток на чтение/запись .
- Пока есть информация, читать/писать очередные данные в/из потока.
- Закрыть поток.

Общая схема работы с потоками в Java:

- Создать потоковый объект и ассоциировать его с файлом на диске.
  - Придать потоковому объекту требуемую функциональность.
- Пока есть информация, читать/писать очередные данные в/из потока
- Закрыть поток.

```
package _java._se._04._iostream;
import java.io.BufferedWriter;
import java.io.FileWriter;
import java.io.IOException;
import java.io.PrintWriter;
public class IOJavaScheme {
	public static void main(String[] args) {
		try {
			// создание потокового объекта (открытие потока)
			FileWriter out = new FileWriter("text.txt");
			// придание потоковому объекту требуемых свойств
			BufferedWriter br = new BufferedWriter(out);
			PrintWriter pw = new PrintWriter(br);
			// работа с потоком через потоковый объект
			pw.println("I'm a sentence in a text-file.");
			// закрытие потока
			pw.close();
		} catch (IOException e) {
			e.printStackTrace();
		}
	}
}
```

Исключения в потоках ввода-вывода

Методы классов потокового ввода-вывода могут сгенерировать исключительную ситуацию типа **IOException**. **IOException** – *проверяемое* исключение и должно быть обработано.

```
package _java._se._04._iostream;
import java.io.IOException;
import java.io.OutputStream;
public class IOExceptionGenerate {
	public static void main(String[] args) {
		OutputStream stdout = System.out;
		try {
			stdout.write(104); // 'h'
			stdout.write(105); // 'i'
			stdout.write(10); // '\n'
		} catch (IOException e) {
			e.printStackTrace();
		}
	}
}
```

Классы пакета java.io могут генерировать следующие типы исключений.

Exceptions	
CharConversionException	**EOFException**
FileNotFoundException	**InterruptedIOException**
InvalidClassException	**InvalidObjectException**
IOException	**NotActiveException**

NotSerializableException	**ObjectStreamException**
OptionalDataException	**StreamCorruptedException**
SyncFailedException	**UnsupportedEncodingException**
UTFDataFormatException	**WriteAbortedException**

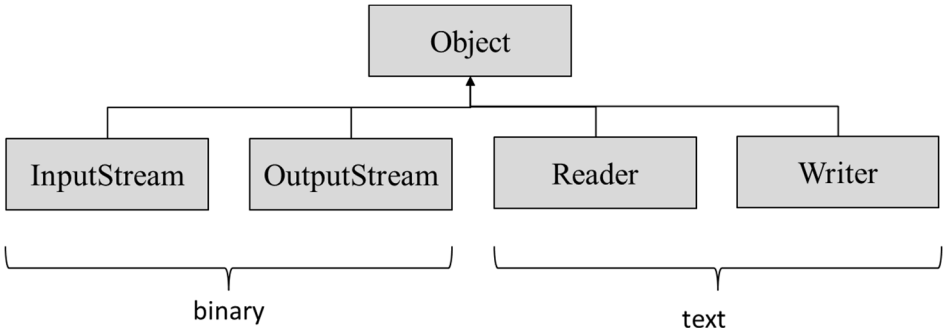
Байтовые и символьные потоки

В Java существует:

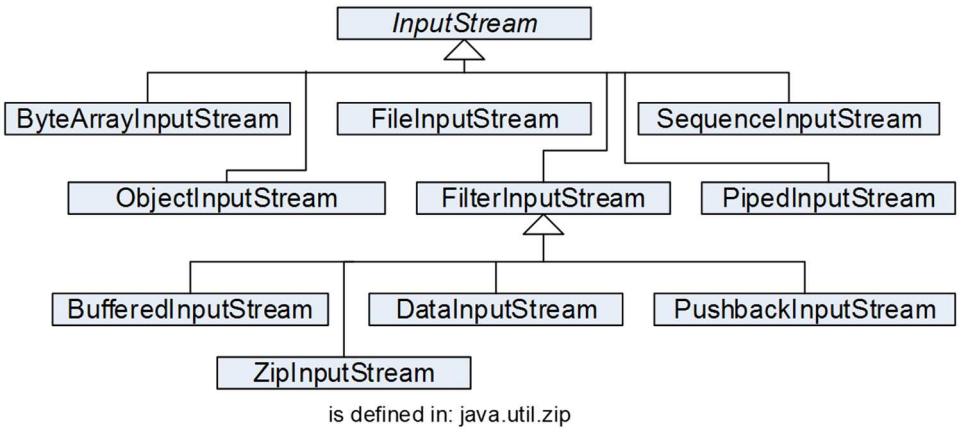
- *2 типа потоков* (символьные (**text**)/двоичные (**binary**));
- *2 направления потоков* (ввод(**input**)/вывод(**output**)).

В результате получаем 4 базовых класса, имеющих дело с вводом-выводом.

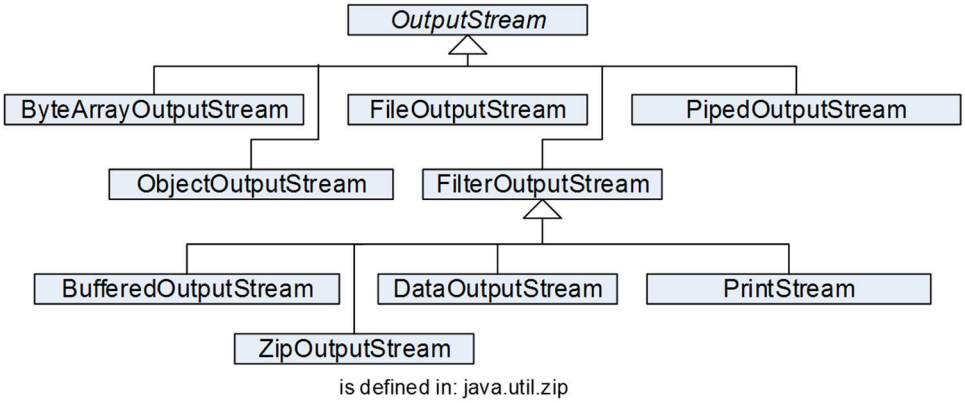
- **Reader**: text-input
- **Writer**: text-output
- **InputStream**: byte-input
- **OutputStream**: byte-output



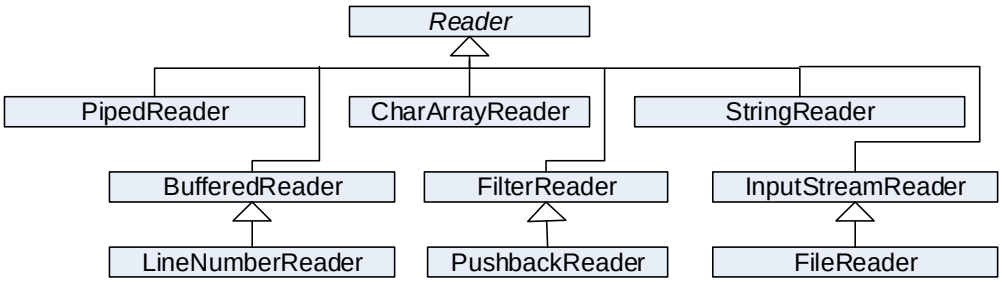
Иерархия классов байтового ввода.



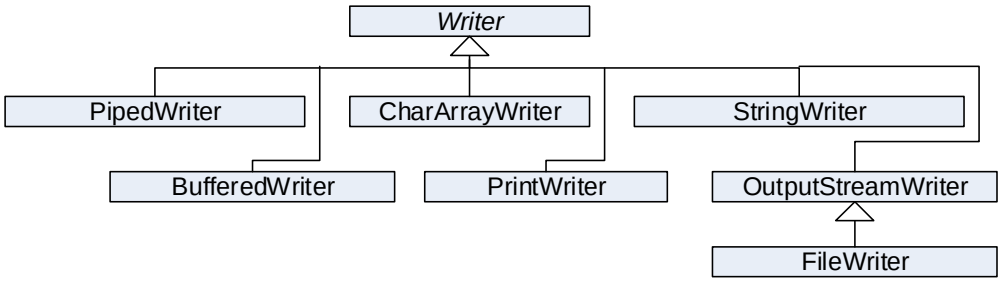
Иерархия классов байтового вывода.



Иерархия классов символьного ввода.



Иерархия классов символьного вывода.



Назначение потоков

Type of I/O	Streams
Memory	CharArrayReader CharArrayWriter ByteArrayInputStream ByteArrayOutputStream
	StringReader StringWriter StringBufferInputStream
Pipe	PipedReader PipedWriter

	PipedInputStream PipedOutputStream
File	FileReader FileWriter FileInputStream FileOutputStream
Object Serialization	*N/A* ObjectInputStream ObjectOutputStream

Классы байтовых потоков

Байтовые потоки определяются в двух иерархиях классов.

Наверху этой иерархии — два **абстрактных** класса: **InputStream** и **OutputStream**.

Каждый из этих абстрактных классов имеет несколько конкретных подклассов, которые обрабатывают различия между разными устройствами, такими как дисковые файлы, сетевые соединения и даже буферы памяти.

Абстрактные классы **InputStream** и **OutputStream** определяют несколько ключевых методов, которые реализуются другими поточными классами. Два наиболее важных— **read()** и **write()**, которые, соответственно, *читают и записывают байты данных*.

Оба метода объявлены как абстрактные внутри классов **InputStream** и **OutputStream** и переопределяются производными поточными классами.

Поточный класс	Значение
BufferedInputStream	Класс, для буферизации ввода
BufferedOutputStream	Класс, для буферизации вывода
ByteArrayInputStream	Поток, читающий из массива байт
ByteArrayOutputStream	Поток, пишущий в массив байт
DataInputStream	Поток ввода, который содержит методы для чтения данных стандартных типов Java
DataOutputStream	Поток вывода, который содержит методы для записи данных стандартных типов Java
FileInputStream	Поток, читающий байты из файла
FileOutputStream	Поток, пишущий байты в файл
FilterInputStream	Реализует Inputstream (шаблон адаптер)
FilterOutputStream	Реализует OutputStream (шаблон адаптер)
InputStream	Абстрактный класс, который описывает поточный ввод

LineNumberInputStream	Расширяет функциональность InputStream тем, что дополнительно производит подсчет, сколько строк было считано из потока.
OutputStream	Абстрактный класс, который описывает поточный вывод
ObjectInputStream	Поток сериализации объектов
ObjectOutputStream	Поток десериализации объектов
PipedlnputStream	Поток, читающий данные из канала
PipedOutputStream	Поток, пишущий данные в канал
PrintStream	Используется для конвертации и записи строк в байтовый поток.
PushbacklnputStream	Фильтр позволяет вернуть во входной поток считанные из него данные
SequencelnputStream	Считывает данные из других двух и более входных потоков
StringBufferInputStream	Производит считывание данных, получаемых преобразованием символов строки в байты

Класс InputStream

Все байтовые потоки чтения наследуются от класса **InputStream.**

Чтение

- read()-методы будут блокированы, пока доступные данные не будут прочитаны.
  - Два из трех read()-методов возвращают число прочитанных байт.
- Возвращают -1 если данных в потоке нет.
- Выбрасывают исключение **IOException**, если происходит ошибка ввода-вывода.

Существует 3 основных read-метода:

- **int read()** - возвращает представление очередного доступного символа во входном потоке в виде целого
- **int read(byte[] buffer)** - пытается прочесть максимум *buffer.length* байт из входного потока в массив *buffer*. Возвращает количество байт, в действительности прочитанных из потока
- **int read(byte[] buffer, int offset, int length)** - пытается прочесть максимум *length* байт, расположив их в массиве *buffer*, начиная с элемента *offset*. Возвращает количество реально прочитанных байт
- **available()** - возвращает количество байт, доступных для чтения в настоящий момент
- **skip(long n)** - пытается пропустить во входном потоке **n** байт. Возвращает количество пропущенных байт
- **close()** - закрывает источник ввода. Последующие попытки чтения из этого потока приводят к возбуждению **IOException**

Некоторые потоки ввода поддерживают операцию перепозиционирования потока, могут "пометить" место в потоке указателем, а затем "перемотать" поток к помеченному месту.

Методы, поддерживающие перепозиционирование:

- **markSupported()** - возвращает **true**, если данный поток поддерживает операции **mark/reset**.
- **mark(int readlimit)** - ставит метку в текущей позиции входного потока, которую можно будет использовать до тех пор, пока из потока не будет прочитано readlimit байт.
- **reset()** - возвращает указатель потока на установленную ранее метку.

Класс OutputStream

Все байтовые потоки записи наследуются от класса **OutputStream.**

Запись:

- **write()**-методы пишут данные в поток, данные при этом буферизуются.
- Используйте **flush()**-метод для сбрасывания буферизованных данных в поток.
- write()-методы выбрасывают исключение **IOException**, если происходит ошибка ввода-вывода.

Существуют 3 основных write-метода:

- **void write(int data)** - записывает один байт в выходной поток. Аргумент этого метода имеет тип int, что позволяет вызывать write, передавая ему выражение, при этом не нужно выполнять приведение его типа к byte, 0 <= data <= 255.
- **void write(byte[] buffer)** - записывает в выходной поток весь указанный массив байт.
- **void write(byte[] buffer, int offset, int length)** - записывает в поток часть массива - length байт, начиная с элемента buffer[offset].

- **flush()** - очищает любые выходные буферы, завершая операцию вывода.
- **close()** - закрывает выходной поток. Последующие попытки записи в этот поток будут возбуждать **IOException**.

```
package _java._se._04._iostream;
import java.io.ByteArrayInputStream;
import java.io.ByteArrayOutputStream;
public class ByteArrayStreamExample {
	public static void main(String[] args) {
		{
			byte[] bytes = { 1, -1, 0 };
			ByteArrayInputStream in = new ByteArrayInputStream(bytes);
			int readedInt = in.read(); // readedInt=1
			System.out.println("first element read is: " + readedInt);
			readedInt = in.read();
					// readedInt=255. Однако (byte)readedInt даст
			// значение -1
			System.out.println("second element read is: " + readedInt);
			readedInt = in.read(); // readedInt=0
			System.out.println("third element read is: " + readedInt);
		}
```

```
		{
			ByteArrayOutputStream out = new ByteArrayOutputStream();
			out.write(10);
			out.write(11);
			byte[] bytes = out.toByteArray();
		}
	}
}
```

```
package _java._se._04._iostream;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;
public class FileStreamExample {
	public static void main(String[] args) {
		byte[] bytesToWrite = { 1, 2, 3 };
		byte[] bytesReaded = new byte[10];
		String fileName = "d:\\test.txt";
		FileOutputStream outFile = null;
		FileInputStream inFile = null;
		try {
			// Создать выходной поток
			outFile = new FileOutputStream(fileName);
			System.out.println("Файл открыт для записи");
			// Записать массив
			outFile.write(bytesToWrite);
			System.out.println("Записано: " + bytesToWrite.length + " байт");
			// По окончании использования должен быть закрыт
			outFile.close();
			System.out.println("Выходной поток закрыт");

			// Создать входной поток
			inFile = new FileInputStream(fileName);
			System.out.println("Файл открыт для чтения");
			// Узнать, сколько байт готово к считыванию
			int bytesAvailable = inFile.available();
			System.out.println("Готово к считыванию: " + bytesAvailable
					+ " байт");
			// Считать в массив
			int count = inFile.read(bytesReaded, 0, bytesAvailable);
			System.out.println("Считано: " + count + " байт");
			inFile.close();
			System.out.println("Входной поток закрыт");
		} catch (FileNotFoundException e) {
			System.out.println("Невозможно произвести запись в файл: "
					+ fileName);
		} catch (IOException e) {
			System.out.println("Ошибка ввода/вывода: " + e.toString());
		} finally {
			try {
				inFile.close();
			} catch (IOException e) {
			}
		}
	}
}
```

Классы **PipedInputStream** и **PipedOutputStream** характерны тем, что их объекты всегда используются в паре - к одному объекту **PipedInputStream** привязывается точно

один объект **PipedOutputStream**. *Эти классы могут быть полезны, если необходимо данные и записать и считать в пределах одного выполнения одной программы.*

Используются следующим образом: создается по объекту **PipedInputStream** и **PipedOutput-Stream**, после чего они могут быть соединены между собой. Один объект **PipedOutputStream** может быть соединен с ровно одним объектом **PipedInputStream** и наоборот.

Соединенный - означает, что если в объект **PipedOutputStream** записываются данные, то они могут быть считаны именно в соединенном объекте **PipedInputStream**. Такое соединение можно обеспечить либо вызовом метода **connect()** с передачей соответствующего объекта **PipedStream**, либо передать этот объект еще при вызове конструктора.

```
package _java._se._04._iostream;
import java.io.IOException;
import java.io.PipedInputStream;
import java.io.PipedOutputStream;
public class PipedStreamExample {
	public static void main(String[] args) {
		PipedInputStream pipeIn = null;
		PipedOutputStream pipeOut = null;
		try {
			int countRead = 0;
			int[] toRead = null;
			pipeIn = new PipedInputStream();
			pipeOut = new PipedOutputStream(pipeIn);
			for (int i = 0; i < 20; i++) {
				pipeOut.write(i);
			}
			int willRead = pipeIn.available();
			toRead = new int[willRead];
			for (int i = 0; i < willRead; i++) {
				toRead[i] = pipeIn.read();
				System.out.print(toRead[i] + " ");
			}
		} catch (IOException e) {
			System.out.println("Impossible IOException occur: ");
			e.printStackTrace();
		}
	}
}
package _java._se._04._iostream;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.SequenceInputStream;
public class SequenceInputStreamExample {
	public static void main(String[] args) {
		FileInputStream inFile1 = null;
		FileInputStream inFile2 = null;
		SequenceInputStream sequenceStream = null;
		FileOutputStream outFile = null;
		try {
			inFile1 = new FileInputStream("file 1.txt");
			inFile2 = new FileInputStream("file 2.txt");
			sequenceStream = new SequenceInputStream(inFile1, inFile2);
			outFile = new FileOutputStream("file 3.txt");
			int readedByte = sequenceStream.read();
			while (readedByte != -1) {
				outFile.write(readedByte);
				readedByte = sequenceStream.read();
```

```
                }
            } catch (IOException e) {
                System.out.println("IOException: " + e.toString());
            } finally {
                try {
                    sequenceStream.close();
                } catch (IOException e) {
                }
                try {
                    outFile.close();
                } catch (IOException e) {
                }
            }
        }
}
```

Классы символьных потоков

Символьные потоки определены в двух иерархиях классов.

Наверху этой иерархии два **абстрактных** класса: **Reader** и **Writer**. Они обрабатывают потоки символов Unicode. Абстрактные классы **Reader** и **Writer** определяют несколько ключевых методов, которые реализуются другими поточными классами. Два самых важных метода — **read()** и **write()**, которые читают и записывают символы данных, соответственно. Они переопределяются производными поточными классами.

Поточный класс	Значение
BufferedReader	Буферизированный символьный поток ввода
BufferedWriter	Буферизированный символьный поток вывода
CharArrayReader	Поток ввода, которой читает из символьного массива
CharArrayWrite	Выходной поток, который записывает в символьный массив
FileReader	Поток ввода, который читает из файла
FileWriter	Выходной поток, который записывает в файл
FilterReader	Отфильтрованный поток ввода
FilterWriter	Отфильтрованный поток вывода
InputStreamReader	Поток ввода, который переводит байты в символы
BufferedReader	Буферизированный символьный поток ввода
BufferedWriter	Буферизированный символьный поток вывода
CharArrayReader	Поток ввода, которой читает из символьного массива
CharArrayWrite	Выходной поток, который записывает в символьный массив
FileReader	Поток ввода, который читает из файла
FileWriter	Выходной поток, который записывает в файл

FilterReader	Отфильтрованный поток ввода
FilterWriter	Отфильтрованный поток вывода
InputStreamReader	Поток ввода, который переводит байты в символы

Класс Reader

Все символьные потоки вывода наследуются от класса **Reader.**

Чтение

- read()-методы будут блокированы, пока доступные данные не будут прочитаны.
- Два из трех read()-методов возвращают число прочитанных символов.
  - Возвращают -1 если данных в потоке нет.
- Выбрасывают исключение **IOException**, если происходит ошибка ввода-вывода.

Существует 3 основных read-метода:

- **int read()** - возвращает представление очередного доступного символа во входном потоке в виде целого
- **int read(char[] buffer)** - пытается прочесть максимум *buffer.length* символов из входного потока в массив *buffer*. Возвращает количество симоволов, в действительности прочитанных из потока
- **int read(char[] buffer, int offset, int length)** - пытается прочесть максимум *length* символов, расположив их в массиве *buffer*, начиная с элемента *offset*. Возвращает количество реально прочитанных символов

- **close()** – метод закрывает поток
- **mark(int readAheadLimit)** - помечает текущее положение потока, параметр указывает количество символов, которые могут быть прочитаны до тех пор, пока метка не станет недействительной
- **ready()** – возвращает true, если в потоке есть данные, доступные для чтения
- **reset()** – возвращает поток в положение, указанное меткой
- **skip(long n)** –пропускает n-байт в потоке

Класс Writer

Все символьные потоки ввода наследуются от класса **Writer**.

Существуют 5 основных write-метода:

- **void write(int c) –** записывает один символ в поток
- **void write(char[] buffer) –** записывает массив символов в поток
- **void write(char[] buffer, int offset, int length) –** записывает в поток подмассив символов длиной length, начиная с позиции offset
- **void write(String aString) –** записывает строку в поток
- **void write(String aString, int offset, int length)** – записывает в поток подстроку символов длиной length, начиная с позиции offset

```
package _java._se._04._iostream;
import java.io.CharArrayReader;
import java.io.IOException;
public class CharArrayReaderExample {
	public static void main(String args[]) throws IOException {
		String tmp = "abcdefghijklmnopqrstuvwxyz";
		int length = tmp.length();
		char c[] = new char[length];
		tmp.getChars(0, length, c, 0);
		CharArrayReader input1 = new CharArrayReader(c);
		CharArrayReader input2 = new CharArrayReader(c, 0, 5);
		int i;
		System.out.println("input1 is:");
		while ((i = input1.read()) != -1) {
			System.out.print((char) i);
		}
		System.out.println();
		System.out.println("input2 is:");
		while ((i = input2.read()) != -1) {
			System.out.print((char) i);
		}System.out.println();
	}
}
package _java._se._04._iostream;
import java.io.BufferedWriter;
import java.io.FileOutputStream;
import java.io.OutputStreamWriter;
import java.io.Writer;
public class InputStreamWriterExample {
	public static void main(String[] argv) throws Exception {
		Writer out = new BufferedWriter(new OutputStreamWriter(
				new FileOutputStream("outfilename"), "UTF8"));
		out.write("asdf");
		out.close();
	}
}
package _java._se._04._iostream;
import java.io.CharArrayReader;
import java.io.IOException;
import java.io.PushbackReader;
public class PushbackReaderExample {
	public static void main(String args[]) throws IOException {
		String s = "if (a == 4) a = 0;\n";
		char buf[] = new char[s.length()];
		s.getChars(0, s.length(), buf, 0);
		CharArrayReader in = new CharArrayReader(buf);
		PushbackReader f = new PushbackReader(in);
		int c;
		while ((c = f.read()) != -1) {
			switch (c) {
			case '=':
				if ((c = f.read()) == '=')
					System.out.print(".eq.");
				else {
					System.out.print("<-");
					f.unread(c);
				}break;
			default:
				System.out.print((char) c);break;
			}
		}
	}
```

```
}
package _java._se._04._iostream;
import java.io.BufferedReader;
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.IOException;
public class FileReaderExample {
	public static void main(String args[]) {
		try {
			FileReader fr = new FileReader("FileReaderExample.java");
			BufferedReader br = new BufferedReader(fr);
			String s;
			while ((s = br.readLine()) != null) {
				System.out.println(s);
			}
			fr.close();
		} catch (FileNotFoundException e) {
			e.printStackTrace();
		} catch (IOException e) {
			e.printStackTrace();
		}
	}
}
```

Сравнение байтовых и символьных потоков

Reader и **InputStream** определяют похожее APIs, которое *отличается* только *типами данных*.

int read()	**Reader**
int read(**char** cbuf[])	
int read(**char** cbuf[], int offset, int length)	
int read()	**InputStream**
int read(**byte** cbuf[])	
int read(**byte** cbuf[], int offset, int length)	

Writer и **OutputStream** определяют похожее APIs, которое *отличается* только *типами данных*.

int write()	**Writer**
int write(**char** cbuf[])	
int write(**char** cbuf[], int offset, int length)	
int write()	**OutputStream**
int write(**byte** cbuf[])	
int write(**byte** cbuf[], int offset, int length)	

В таблице приведены соответствия классов для байтовых и символьных потоков.

Byte-streams	Char-streams
InputStream	Reader
OutputStream	Writer
ByteArrayInputStream	CharArrayReader
ByteArrayOutputStream	CharArrayWriter
Нет аналога	InputStreamReader
Нет аналога	OutputStreamWriter
FileInputStream	FileReader
FileOutputStream	FileWriter
FilterInputStream	FilterReader
FilterOutputStream	FilterWriter
BufferedInputStream	BufferedReader
BufferedOutputStream	BufferedWriter
PrintStream	PrintWriter
DataInputStream	Нет аналога
DataOutputStream	Нет аналога
ObjectInputStream	Нет аналога
ObjectOutputStream	Нет аналога
PipedInputStream	PipedReader
PipedOutputStream	PipedWriter
StringBufferInputStream	StringReader
Нет аналога	StringWriter
LineNumberInputStream	LineNumberReader
PushBackInputStream	PushBackReader
SequenceInputStream	Нет аналога

Предопределенные потоки

Все программы Java автоматически импортируют пакет **java.lang**. Этот пакет определяет класс с именем **System**, инкапсулирующий некоторые аспекты исполнительной среды Java.

Класс System содержит также три предопределенные поточные переменные **in**, **out** и **err**. Эти поля объявлены в System со спецификаторами **public** и **static**.

Объект **System.out** называют *потоком стандартного вывода*. По умолчанию с ним связана консоль.

На объект **System.in** ссылаются как на *стандартный ввод*, который по умолчанию связан с клавиатурой.

К объекту **System.err** обращаются как к *стандартному потоку ошибок*, который по умолчанию также связан с консолью.

Однако эти потоки могут быть переназначены на любое совместимое устройство ввода/вывода.

```
package _java._se._04._iostream;
import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;
public class StandartOutInExample {
	public static void main(String[] args) {
		try {
			OutputStream stdout = System.out;
			stdout.write(104); // ASCII 'h'
			stdout.flush();
			stdout.write('\n');
			byte[] b1 = new byte[5];
			InputStream stdin1 = System.in;
			stdin1.read(b1);
			System.out.write(b1);
			System.out.write('\n');
			System.out.flush();
			InputStream stdin2 = System.in;
			byte[] b2 = new byte[stdin2.available()];
			int len = b2.length;
			for (int i = 0; i < len; i++)
				b2[i] = (byte) stdin2.read();
			System.out.println(b2[0] + " " + b2[1]);
			System.out.flush();
		} catch (IOException e) {
			e.printStackTrace();
		}
	}
}
```

Упаковка (wrapping) потоков

Потокам можно придать новые свойства, заключив один поток в оболочку другого потока.

Класс **BufferedReader** может быть применен для более *эффективного чтения* символов, массивов и строк.

```
BufferedReader in = new BufferedReader(
                    new FileReader("foo.in"));
```

Классы **BufferedWriter** и **PrintWriter** могут быть использованы для более *эффективной записи* символов, массивов, строк и других типов данных.

```
BufferedWriter out = new BufferedWriter(
                     new FileWriter("foo.out"));
PrintWriter out = new PrintWriter(
                    new BufferedWriter(new FileWriter(
                          "foo.out")));
```

Класс Scanner

Объект класса **java.util.Scanner** принимает *форматированный объект* (ввод) и преобразует его в двоичное представление. При вводе могут использоваться данные из консоли, файла, строки или любого другого источника, реализующего интерфейсы **Readable** или **ReadableByteChannel**.

Класс определяет следующие конструкторы:

```
Scanner (File source)      throws FileNotFoundException
Scanner (File source, String charset)   throws FileNotFoundException
Scanner (InputStream source)
Scanner (InputStream source, String charset)
Scanner (Readable source)
Scanner (ReadableByteChannel source)
Scanner (ReadableByteChannel source, String charset)
Scanner(String source)
```

где **source** – источник входных данных, а **charset** – кодировка.

Объект класса **Scanner** читает лексемы из источника, указанного в конструкторе, например из строки или файла. Лексема – это набор данных, выделенный набором разделителей (по умолчанию пробелами). В случае ввода из консоли следует определить объект:

```
Scanner con = new Scanner (System.in);
```

После создания объекта его используют для ввода, например целых чисел, следующим образом:

```
while (con.hasNextInt ()) {
      int n = con.nextInt ();
}
```

В классе **Scanner** определены группы методов, проверяющих данные заданного типа на доступ для ввода. Для проверки наличия произвольной лексемы используется метод **hasNext()**. Проверка конкретного типа производится с помощью одного из методов **boolean hasNextТип()** или **boolean hasNextТип(int radix)**, где **radix** – основание системы счисления. Например, вызов метода **hasNextInt()** возвращает **true**, только если следующая входящая лексема – целое число. Если данные указанного типа доступны, они считываются

с помощью одного из методов **Тип nextТип()**. Произвольная лексема считывается методом **String next()**. После извлечения любой лексемы текущий указатель устанавливается перед следующей лексемой.

```
package _java._se._04._iostream;
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;
import java.util.Scanner;
public class ScannerExample {
	static String filename = "scan.txt";
	public static void scanFile() {
		try {
			FileReader fr = new FileReader(filename);
			Scanner scan = new Scanner(fr);
			while (scan.hasNext()) {
				if (scan.hasNextInt()) {
					System.out.println(scan.nextInt() + ":int");
				} else {
				   if (scan.hasNextDouble())
				      System.out.println(scan.nextDouble() + ":double");
				   else {
				     if (scan.hasNextBoolean())
				        System.out.println(scan.nextBoolean() + ":boolean");
				     else {
				        System.out.println(scan.next() + ":String");
				     }
				   }
				}
			}
		} catch (FileNotFoundException e) {
			System.err.println(e);
		}
	}
	public static void makeFile() {
		try {
			FileWriter fw = new FileWriter(filename);
			fw.write("2 Java 1,5 true 1.6 ");
			fw.close();
		} catch (IOException e) {
			System.err.println(e);
		}
	}
	public static void main(String[] args) {
		ScannerExample.makeFile();
		ScannerExample.scanFile();
	}
}
```

Процедура проверки типа реализована при с помощью методов **hasNextТип()**. Такой подход предпочтителен из-за отсутствия возможности возникновения исключительной ситуации, так как ее обработка требует на порядок больше ресурсов, чем нормальное течение программы. Для чтения строки из потока ввода применяются методы **next()** или **nextLine()**.

Объект класса **Scanner** определяет границы лексемы, основываясь на наборе разделителей. Можно задавать разделители с помощью метода **useDelimiter(Pattern pattern)** или **useDelimiter(String pattern)**, где **pattern** содержит набор разделителей.

```
package _java._se._04._iostream;
import java.util.Scanner;
public class ScannerDelimiterExample {
	public static void main(String args[]) {
		double sum = 0.0;
		Scanner scan = new Scanner("1,3;2,0; 8,5; 4,8; 9,0; 1; 10");
		scan.useDelimiter(";\\s*");
		while (scan.hasNext()) {
			if (scan.hasNextDouble()) {
				sum += scan.nextDouble();
			} else {
				System.out.println(scan.next());
			}
		}
		System.out.printf("Сумма чисел = " + sum);
	}
}
```

Метод **String findInLine(Pattern pattern)** или **String findInLine(String pattern)** ищет заданный шаблон в следующей строке текста. Если шаблон найден, соответствующая ему подстрока извлекается из строки ввода. Если совпадений не найдено, то возвращается **null**. Методы String **findWithinHorizon(Pattern pattern, int count)** и **String findWithinHorizon(String pattern, int count)** производят поиск заданного шаблона в ближайших **count** символах. Можно пропустить образец с помощью метода **skip (Pattern pattern)**. Если в строке ввода найдена подстрока, соответствующая образцу **pattern**, метод **skip()** просто перемещается за нее в строке ввода и возвращает ссылку на вызывающий объект. Если подстрока не найдена, метод **skip()** генерирует исключение **NoSuchElementException**.

```
package _java._se._04._iostream;
import java.util.Scanner;
public class FindInLineExample {
	public static void main(String args[]) {
		String instr = "Name: Joe Age: 28 ID: 77";
		Scanner conin = new Scanner(instr);
		conin.findInLine("Age:"); // find Age
		if (conin.hasNext())
			System.out.println(conin.next());
		else
			System.out.println("Error!");
	}
}
```

Хотя **Scanner** и не является потоком, у него тоже обязательно вызывать метод **close()**, который закроет используемый основной источник.

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. Процесс **сериализации** заключается в сериализации каждого поля объекта, но только в том случае, если это поле не имеет спецификатора **static** или **transient**. Поля ,

помеченные ими не могут быть предметом сериализации. Для того, что бы объект мог быть сериализован, он должен реализовать интерфейс **Serializable**. Интерфейс **java.io.Serializable** *не определяет никаких методов*. Его присутствие только определяет, что объекты этого класса разрешено сериализовывать. При попытке сериализовать объект, не реализующий этот интерфейс, будет брошено **java.io.NotSerializableException.**

После того, как объект был сериализован (превращен в последовательность байт), его можно восстановить, при этом восстановление можно проводить на любой машине (вне зависимости от того, где проводилась сериализация). При **десериализации** поле со спецификатором transient получает значение по умолчанию, соответствующее его типу, а поле со спецификатором **static** получает значение по умолчанию в случае отсутствия в области видимости объектов своего типа, а при их наличии получает значение, которое определено для существующего объекта. При использовании **Serializable** десериализация происходит следующим образом: под объект выделяется память, после чего его поля заполняются значениями из потока. КОНСТРУКТОР объекта при этом НЕ ВЫЗЫВАЕТСЯ. Для работы по сериализации в java.io определены интерфейсы **ObjectInput**, **ObjectOutput** и реализующие их классы **ObjectInputStream** и **ObjectOutputStream** соответственно. Для сериализации объекта нужен выходной поток **OutputStream**, который следует передать при конструировании **ObjectOutputStream**. После чего вызовом метода **writeObject()** сериализовать объект и записать его в выходной поток.

```
package _java._se._04._iostream;
import java.io.ByteArrayInputStream;
import java.io.ByteArrayOutputStream;
import java.io.IOException;
import java.io.NotSerializableException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
public class SerializationExample {
	public static void main(String[] args) {
		try {
			// сериализация
			ByteArrayOutputStream os = new ByteArrayOutputStream();
			Object objSave = new Integer(1);
			ObjectOutputStream oos = new ObjectOutputStream(os);
			oos.writeObject(objSave);
			// смотрим, во что превратился сериализованный объект
			byte[] bArray = os.toByteArray();
			for (byte b : bArray) {
				System.out.print((char) b);
			}
			System.out.println();
			// десериализация
			ByteArrayInputStream is = new ByteArrayInputStream(bArray);
			ObjectInputStream ois = new ObjectInputStream(is);
			Object objRead = ois.readObject();
			// проверяем идентичность объектов
			System.out.println("readed object is: " + objRead.toString());
			System.out.println("Object equality is: "
					+ (objSave.equals(objRead)));
			System.out
					.println("Reference equality is: " + (objSave == objRead));
		} catch (NotSerializableException e) {
			e.printStackTrace();
		} catch (IOException e) {
			e.printStackTrace();
```

```
			} catch (ClassNotFoundException e) {
				e.printStackTrace();
			}
		}
}
```

Сериализуемый объект может хранить **ссылки на другие объекты**, которые в свою очередь так же могут хранить ссылки на другие объекты.

И все **ссылки** тоже **должны быть восстановлены** при десериализации. Важно, что если несколько ссылок указывают *на один и тот же* объект, то в **восстановленных объектах** эти ссылки так же указывали *на один и тот же объект*.

Чтобы сериализованный объект не был записан дважды, механизм сериализации некоторым образом для себя помечает, что *объект уже записан* в граф, и когда в очередной раз попадется ссылка на него, она будет указывать на уже сериализованный объект. Такой механизм необходим, что бы иметь возможность записывать связанные объекты, которые могут иметь перекрестные ссылки. В таких случаях необходимо отслеживать был ли объект уже сериализован, то есть нужно ли его записывать или достаточно указать ссылку на него.

Если класс содержит **в качестве полей другие объекты**, то эти объекты так же будут сериализовываться и поэтому тоже должны быть **сериализуемы**. В свою очередь, сериализуемы должны быть и все объекты, содержащиеся в этих сериализуемых объектах и т.д.

Полный путь ссылок объекта по всем объектным ссылкам, имеющимся у него и у всех объектов на которые у него имеются ссылки, и т.д. - называется графом исходного объекта.

```
package _java._se._04._iostream;
public class Point implements java.io.Serializable  {
	private double x;
	private double y;

	public Point(double x, double y) {
		this.x = x;
		this.y = y;
	}

	public String toString() {
		return "(" + x + "," + y + ") reference=" + super.toString();
	}
}
```

```
package _java._se._04._iostream;
public class Line implements java.io.Serializable {
	private Point point1;
	private Point point2;
	private int index;

	public Line() {
		System.out.println("Constructing empty line");
	}

	Line(Point p1, Point p2, int index) {
		System.out.println("Constructing line: " + index);
		this.point1 = p1;
		this.point2 = p2;
		this.index = index;
	}

	public int getIndex() {
		return index;
```

```
	}
	public void setIndex(int newIndex) {
		index = newIndex;
	}
	public void printInfo() {
		System.out.println("Line: " + index);
		System.out.println(" Object reference: " + super.toString());
		System.out.println(" from point " + point1);
		System.out.println(" to point " + point2);
	}
}

package _java._se._04._iostream;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
public class SomeReferencesSerialization {
	public static void main(String[] args) {
		Point p1 = new Point(1.0, 1.0);
		Point p2 = new Point(2.0, 2.0);
		Point p3 = new Point(3.0, 3.0);
		Line line1 = new Line(p1, p2, 1);
		Line line2 = new Line(p2, p3, 2);
		System.out.println("line 1 = " + line1);
		System.out.println("line 2 = " + line2);
		String fileName = "d:\\file";
		try {
			FileOutputStream os = new FileOutputStream(fileName);
			ObjectOutputStream oos = new ObjectOutputStream(os);
			oos.writeObject(line1);

			line1.setIndex(3);

			oos.writeObject(line1);
			oos.close();
			os.close();

			System.out.println("Read objects:");
			FileInputStream is = new FileInputStream(fileName);
			ObjectInputStream ois = new ObjectInputStream(is);
			while (is.available() > 0) {
			     Line line = (Line) ois.readObject();
			     line.printInfo();
			}
		      } catch (ClassNotFoundException e) {
			e.printStackTrace();
		      } catch (IOException e) {
			e.printStackTrace();
		      }
	}
}
```

Что бы указать, что сеанс сериализации завершен, и мы хотим заново записывать объекты, у ObjectOutputStream нужно вызвать метод reset().

```
...
line1.setIndex(3);
oos.reset();
oos.writeObject(line1);}
...
```

При **десериализации** производного **класса, наследуемого от несериализуемого класса**, вызывается **конструктор без параметров** родительского НЕ сериализуемого класса. И если такого конструктора не будет – при десериализации возникнет ошибка **java.io.InvalidClassException**. Конструктор же дочернего объекта, того, который мы десериализуем, не вызывается.

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

Попытка **десериализации** объекта, **класс** которого к этому времени **был изменен**, приведет к возникновению **InvalidClassException**.

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

Применение потоков ввода-вывода

Поток – дорогой ресурс.

Количество потоков, которые вы можете держать открытыми одновременно ограничено.

У вас не должно быть более одного потока, ассоциированного с одним файлом.

Чтобы открыть поток заново, его сначала надо ОБЯЗАТЕЛЬНО закрыть

Всегда закрывайте потоки ввода-вывода.