


ПОТОКОВЫЙ ВВОД/ВЫВОД ДАННЫХ В Java

Потоковый ввод/вывод данных в Java

- ❑ **Потоки** являются абстракцией, которая связывает программу с физическим устройством с помощью системы ввода/вывода Java.

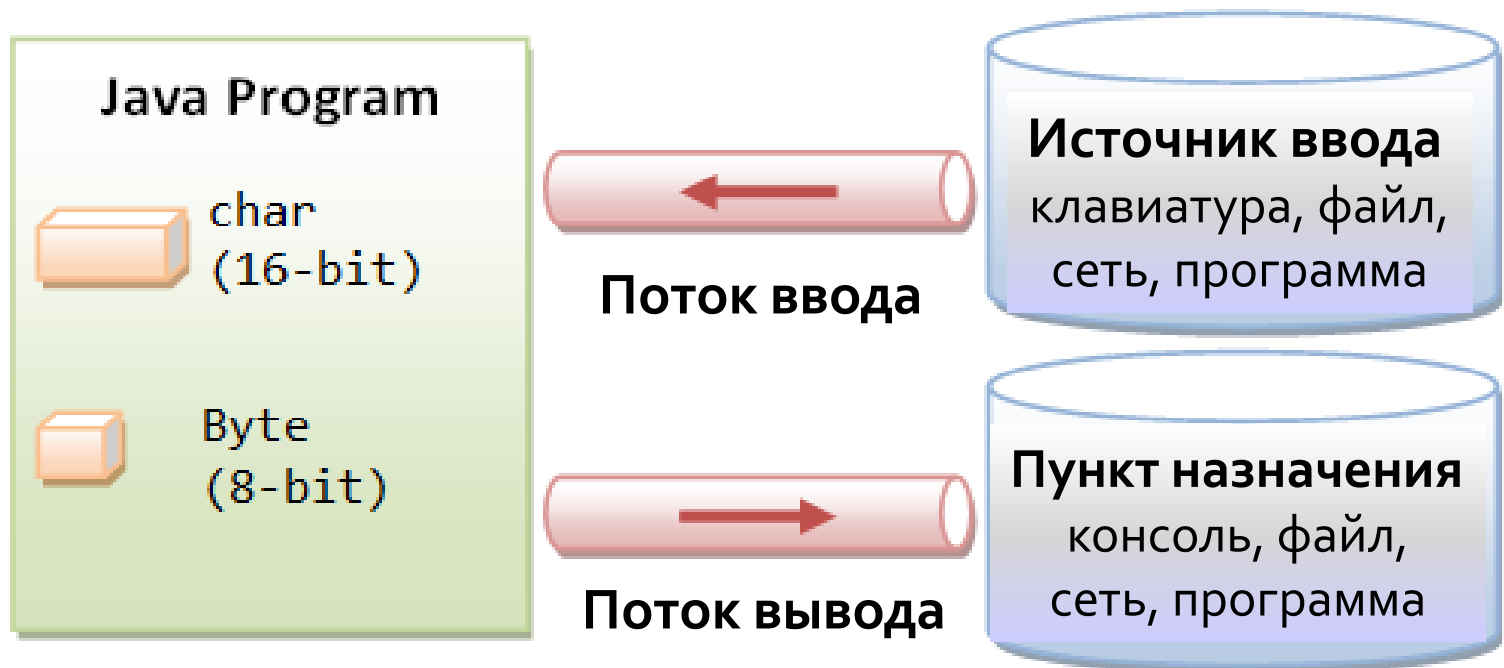
➤ **Поток** представляет собой последовательность данных.

[illegible]

Источник данных и пункт назначения данных могут быть всем, что *содержит, генерирует* или *потребляет* данные (файлы на диске; другая программа; периферийное устройство; сетевой разъем; массив и т.д.)

Потоковый ввод/вывод данных в Java

- ❑ Потоки поддерживают различные виды данных: простые байты, примитивные типы данных, локализованные символы и объекты;
- ❑ Некоторые потоки просто передают данные;
- ❑ Другие потоки управляют и преобразовывают данные в удобный вид.

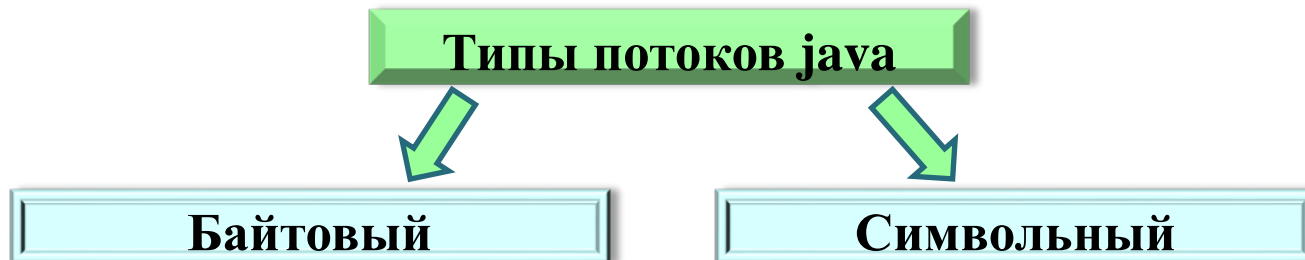


Потоковый ввод/вывод данных в Java

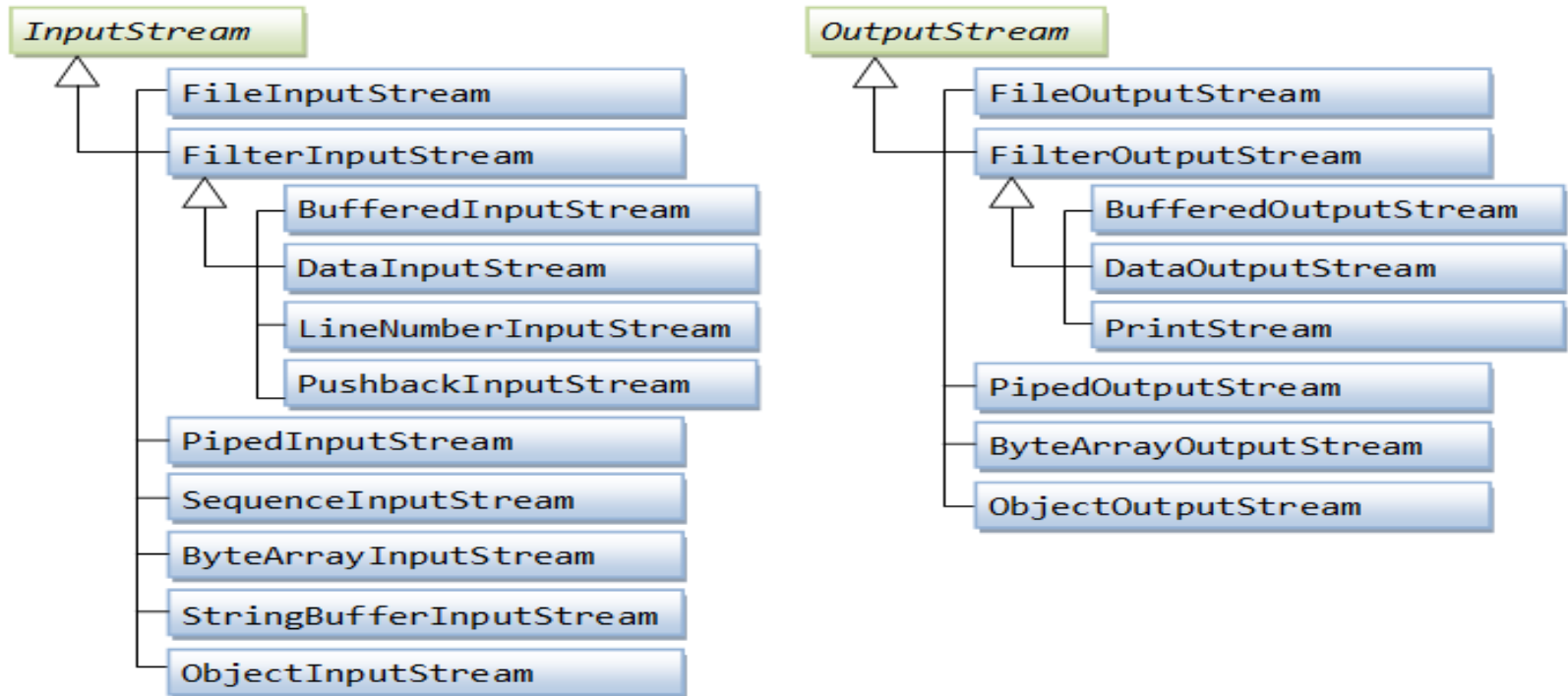
- ❑ Java внутри хранит символы (тип **char**) в 16-битном наборе символов UCS-2.
- ❑ Внешний источник данных может хранить символы в другом наборе символов (*например, US-ASCII, ISO-8859-x, UTF-8, UTF-16 и многие другие*), в фиксированной длине 8 бит или 16-бит, или в переменной длине от 1 до 4 байт.



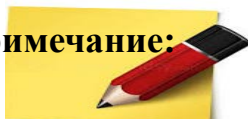
Как следствие, Java нужно различать ввод/вывод на основе байтов для обработки двоичных данных, и ввод/вывод на основе символов для обработки текста.



Классы байтового ввода/вывода



Примечание:



- ❑ **InputStream** и **OutputStream** являются абстрактными классами, которые определяют низкоуровневый интерфейс для всех байтовых потоков.
- ❑ Они содержат методы для чтения или записи неструктурированного потока данных на уровне байтов.

Потоковый ввод/вывод данных в Java

Методы класса InputStream

Модификатор и тип	Метод и описание
abstract int	read() Прочитать следующий байт из потока ввода
int	read(byte[]b) Прочитать некоторое количество байт из потока ввода и сохранить их в массиве <i>b</i>
int	read(byte[]b, int off, int len) Прочитать <i>len</i> байт из потока ввода и сохранить их в массиве <i>b</i> , начиная со смещения <i>off</i>
void	reset() Возвращает поток в состояние на момент вызова метода <i>mark()</i> для данного потока ввода
long	skip(long n) Пропустить и отбросить <i>n</i> байт из потока ввода

Методы класса InputStream

Модификатор и тип	Метод и описание
void	close() Закрывает поток ввода и освобождает системные ресурсы, связанные с этим потоком
void	mark(int readlimit) Отметить (маркировать) текущую позицию в потоке ввода
boolean	markSupported() Проверить, поддерживает ли поток ввода маркировку и сброс

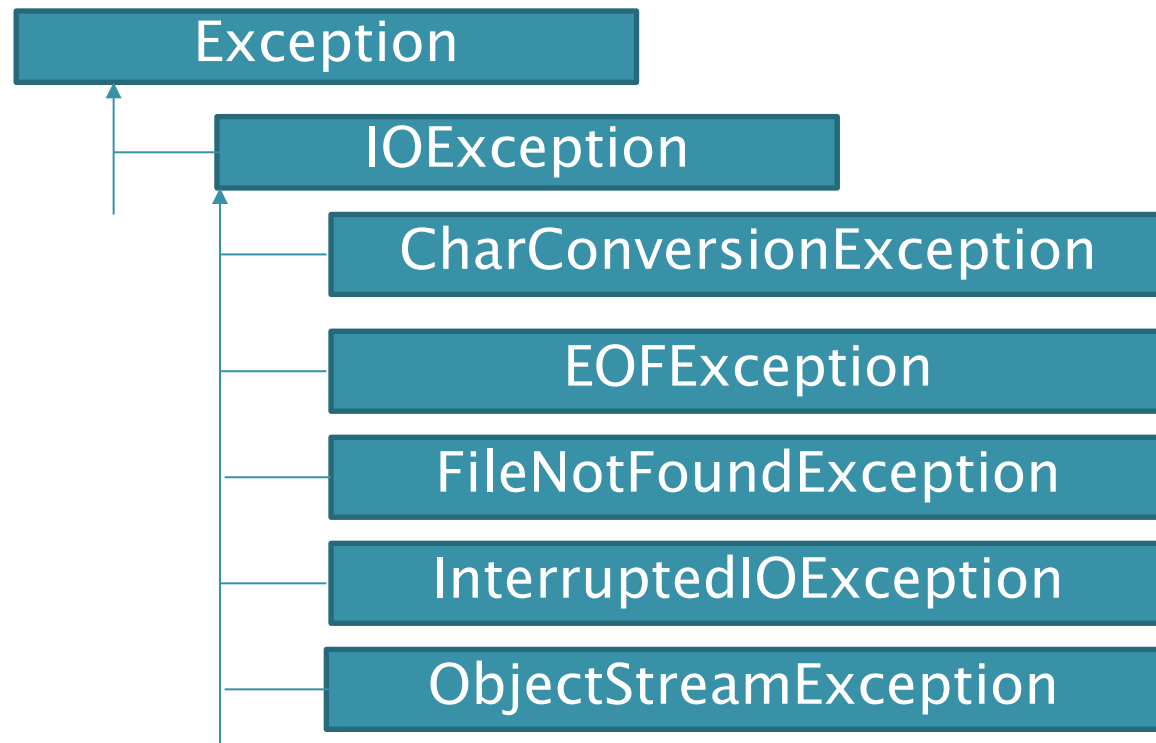
Потоковый ввод/вывод данных в Java

Методы класса OutputStream

Модификатор и тип	Метод и описание
void	close() Закрывает поток вывода и освобождает системные ресурсы, связанные с этим потоком
void	flush() Прочистить поток вывода и принудить записать данные
void	write(byte[]b) Записать <i>b.length</i> байт из массива <i>b</i> в поток вывода
void	write(byte[]b, int off, int len) Записать <i>len</i> байт из массива <i>b</i> , начиная с позиции <i>off</i> , в поток вывода
abstract void	write(int b) Записать указанный байт в поток вывода

Потоковый ввод/вывод данных в Java

- ❑ **IOException** – это общий класс исключений, произведенных неудачными или прерванными операциями ввода/вывода;
 - **IOException** сигнализирует, что произошел какой-то вид исключения ввода/вывода.

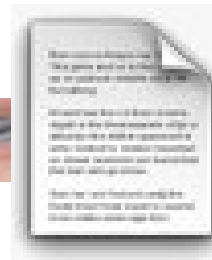


СЦЕПЛЕННЫЕ ПОТОКИ

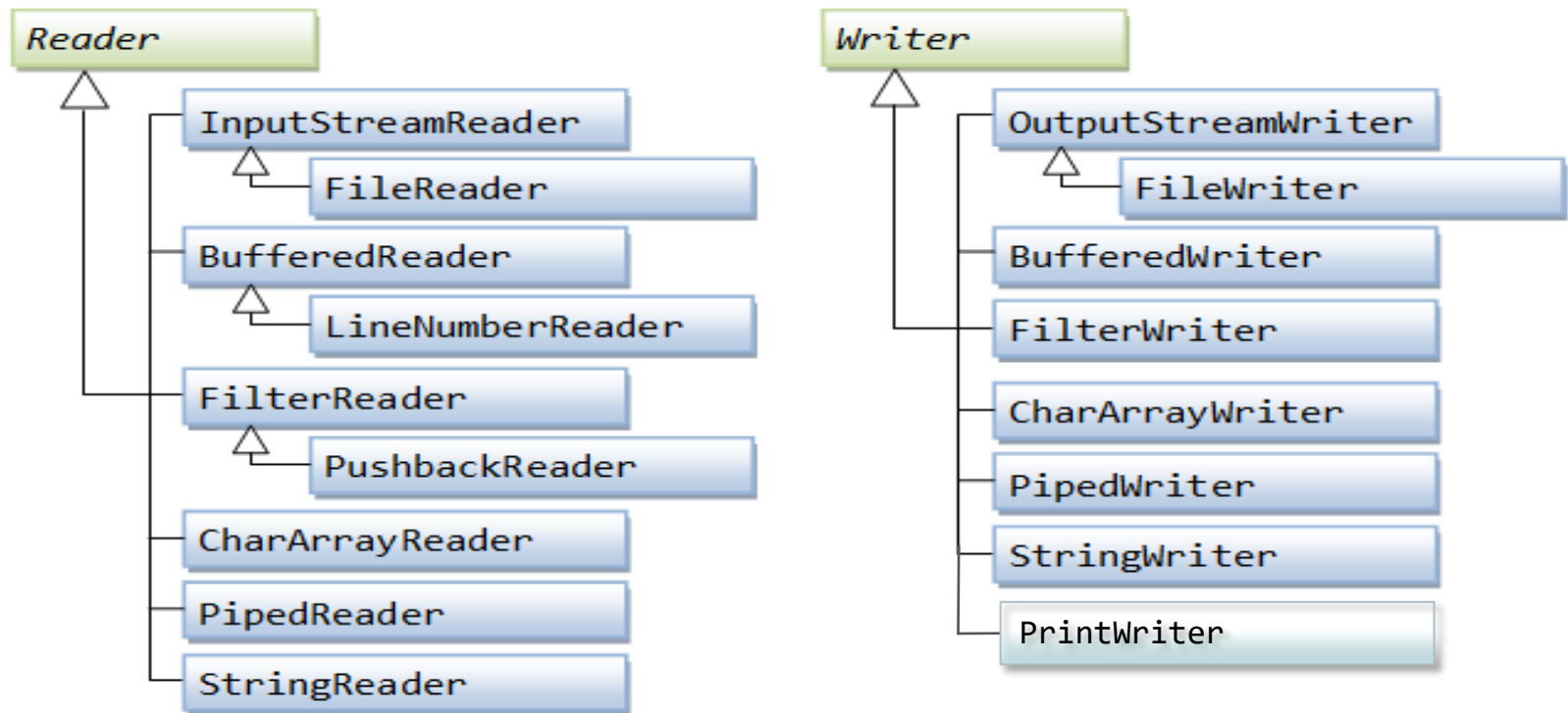
- ❑ В языке Java предусмотрен механизм разделения ответственности (одни потоки извлекают байты из файлов, другие объединяют байты в данные);
- ❑ Для использования возможностей разных потоков их необходимо объединять в сцепленные потоки).

Например,

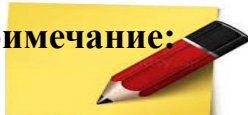
```
DataInputStream din = new DataInputStream(  
    new BufferedInputStream(  
        new FileInputStream("data.txt")));
```



Классы символьного ввода/вывода



Примечание:



- ❑ Класс **Reader** и **Writer** являются абстрактными классами;
- ❑ Класс **Reader** определяет методы *read()* и *readLine()* для чтения символа и строки символов соответственно;
- ❑ Класс **PrintWriter** определяет методы *print()* и *println()*

ПОТОКОВЫЙ ВВОД/ВЫВОД ДАННЫХ В Java

Пример 10:

```
public class BRReadLines {  
    public static void main(String [] args) throws IOException {  
        BufferedReader br = new BufferedReader(  
            new InputStreamReader(System.in));  
        String str;  
        System.out.println("Enter of string.");  
        System.out.println("Enter 'stop' for exit.");  
        do {  
            str = br.readLine();  
            System.out.println(str);  
        }  
        while (!str.toLowerCase().equals("stop"));  
    }  
}
```

Буферизи-
рованное
чтение

Преобразование
в символный
поток

ФАЙЛОВЫЕ ПОТОКИ ВВОДА/ВЫВОДА

- ❑ Для работы с файлами необходимо импортировать пакет **java.io**.
 - ❑ Для создания потока и открытия файла необходимо создать объект одного из классов:
 - `FileInputStream(String name);`
 - `FileReader(String name);`
 - `FileOutputStream(String name);`
 - `FileWriter(String name);`
- } Чтение из файла
- } Запись в файл



Особенности при работе с файлами

- ❑ Если при создании потока ввода (чтение) файл не существует, то генерируется ошибка открытия файла;
- ❑ Если при создании потока вывода (запись) файл не удастся создать (не существующий) или открыть, то генерируется ошибка открытия файла;
- ❑ По окончании работы с файлом его нужно закрыть для подтверждения всех произведенных изменений, а также для освобождения выделенных для него системных ресурсов.
 - Традиционный подход → вызов метода *close()*.

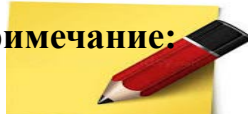
Потоковый ввод/вывод данных в Java

Пример 11: копирование файла

```
import java.io.*;
public class CopyFile {
    public static void main(String [] args) throws IOException {
        int i;
        FileInputStream fin = new FileInputStream(args[0]);
        FileOutputStream fout = new FileOutputStream(args[1]);
        while ( (i = fin.read()) != -1)
            fout.write(i);
        fin.close();
        fout.close();
    }
}
```

Имена файлов
передаются как
параметры
командной строки

Примечание:

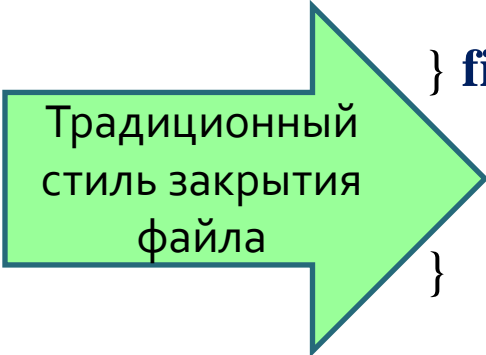


Код читает из входного потока и записывает в
выходной поток, один байт за раз!
Для уменьшения накладных расходов –
буферизация!

Потоковый ввод/вывод данных в Java

Пример 12: *буферизированное копирование файла:*

```
BufferedInputStream brin = null;  
BufferedOutputStream brout = null;  
try {  
    int i;  
    byte[] buf = new byte[1024];  
    brin = new BufferedInputStream(  
        new FileInputStream(args[0]));  
    brout = new BufferedOutputStream(  
        new FileOutputStream(args[1]));  
    while ( (i = brin.read (buf)) != -1)  
        brout.write(buf);  
} finally {  
    if ( brin != null) brin.close();  
    if ( brout != null) brout.close();  
}  
  
//.....
```



Традиционный
стиль закрытия
файла

Потоковый ввод/вывод данных в Java

Пример 13: *чтения файла через символьный файловый поток*

```
import java.io.*;
public class Demo {
    public static void main(String a[]) throws IOException {
        BufferedReader br = null;
        try {
            br = new BufferedReader(
                new FileReader("Demo.java"));
            String str;
            while ( (str = br.readLine()) != null)
                System.out.println(str);
        } finally {
            if (br != null)
                br.close();
        }
    }
}
```

ПОТОКИ ДЛЯ РАБОТЫ С ПРИМИТИВНЫМИ ТИПАМИ

- ❑ Потоки данных поддерживают бинарный ввод/вывод значений примитивных типов данных ;
- ❑ Эти потоки данных реализуют либо интерфейс **DataInput** либо интерфейс **DataOutput**;
- ❑ Наиболее широко используемые реализации этих интерфейсов: **DataInputStream** и **DataOutputStream**.

Потоковый ввод/вывод данных в Java

Пример 14: запись в файл данных различных типов:

```
try {  
    DataOutputStream out = new DataOutputStream(  
        new BufferedOutputStream(  
            new FileOutputStream("dataout.dat")));  
    out.writeShort(1200);  
    out.writeInt(50000);  
    out.writeLong(12345678L);  
    out.writeDouble(55.66);  
    out.writeBoolean(true);  
    out.writeUTF("Hello!!!");  
    out.flush();  
} catch (IOException ex) {  
    //...  
}  
    //...
```

Принуждение
записи в файл (из
потока вывода в
пункт назначения)

ПОТОКОВЫЙ ВВОД/ВЫВОД ДАННЫХ В Java

Пример 15: чтение из файла данных различных типов:

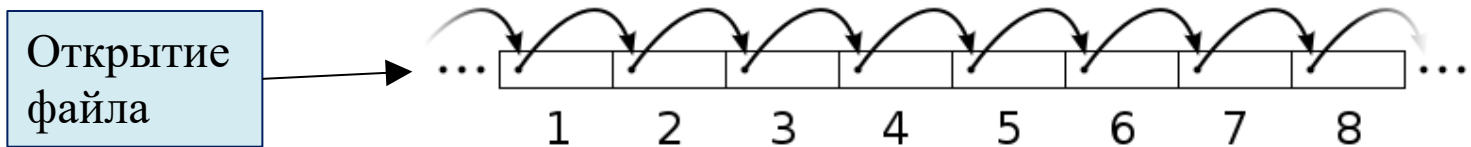
```
try {  
    DataInputStream in = new DataInputStream(  
        new BufferedInputStream(  
            new FileInputStream("dataout.dat")));  
    System.out.println("short: " + in.readShort());  
    System.out.println("int:   " + in.readInt());  
    System.out.println("long:  " + in.readLong());  
    System.out.println("double: " + in.readDouble());  
    System.out.println("boolean: " + in.readBoolean());  
    System.out.println("String UTF: " + in.readUTF());  
    System.out.println();  
} catch (IOException ex) {  
    //...  
}  
//...
```

Вывод в консоли:

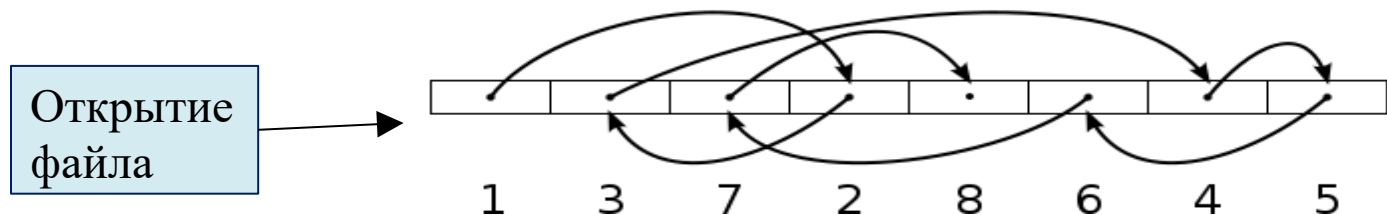
```
short: 1200  
int:   50000  
long:  12345678  
double: 55.66  
boolean: true  
String UTF: Hello!!!
```

ПРОИЗВОЛЬНЫЙ ДОСТУП К ФАЙЛУ

- Классы **FileInputStream**, **FileOutputStream**, **FileReader**, **FileWriter** используют последовательный доступ к содержимому файла.



- Для обеспечения чтения/записи данных в любой позиции файла используется класс **RandomAccessFile**.



Произвольный доступ к файлам работает подобно большому массиву байтов, хранящихся в файловой системе.

Потоковый ввод/вывод данных в Java

- ❑ Поток с произвольным доступом к содержимому файла всегда создается с режимом доступа:
 - "r" - только для чтения;
 - "rw" - для чтения и записи;
 - "rws" - для чтения и записи с записью на устройство.
- ❑ Произвольный доступ имеет курсор, называемый указатель файла;
- ❑ Операции ввода/вывода начинаются от указателя файла и продвигают его к концу файла;
- ❑ Текущая позиция указателя файла может быть получена методом *getFilePointer()* и установлена методом *seek(long pos)*.

ПОТОКОВЫЙ ВВОД/ВЫВОД ДАННЫХ В Java

Пример 16:

```
RandomAccessFile raf = null;
```

```
try {
```

```
//...
```

```
    raf = new RandomAccessFile("C:\\test.txt", "rw");
```

```
    raf.write(new byte[] {0,1,2,3,4,5,6,7,8,9});
```

```
    raf.seek(5);
```

```
    raf.write(new byte[] {66,77,88});
```

```
    raf.seek(0);
```

```
    byte[] buf = new byte[10];
```

```
    int n = raf.read(buf,0,10);
```

```
    System.out.println(Arrays.toString(buf));
```

```
    raf.close();
```

```
//...
```

Вывод в консоли:

[0, 1, 2, 3, 4, 66, 77, 88, 8, 9]



КЛАСС File

- ❑ Класс **File** – это класс, который не предназначен для работы с потоками;
- ❑ Класс **File** непосредственно общается с файлами и файловой системой:
 - Он описывает свойства файла или директории.



- ❑ Класс **File** можно использовать для:
 - определения, что файл или директория существует;
 - создания директории, если она не существует;
 - определения длины файла в байтах;
 - переименования или перемещения файла;
 - удаления файла;
 - определения, что есть путь к файлу или директории;
 - чтения списка файлов в директории.

<https://docs.oracle.com/javase/7/docs/api/java/io/File.html>

Потоковый ввод/вывод данных в Java

Пример 17:

```
File file = new File("c:\\testfile.txt");
```

```
boolean isDirectory = file.isDirectory();
```

← Проверка - это директория

```
boolean fileExists = file.exists();
```

← Проверка – существует ли файл

```
long length = file.length();
```

← Получить размер файла

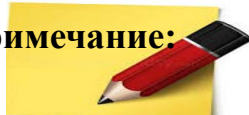
```
boolean sucMov =  
    file.renameTo(new File("c:\\newfile.txt"));
```

← Переименовать

```
boolean successDel = file.delete();
```

← Удалить файл

Примечание:



Объект типа **File** можно использовать при создании потока чтения/записи в файл:

```
new FileReader(file);
```

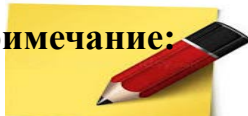
АВТОМАТИЧЕСКОЕ УПРАВЛЕНИЕ РЕСУРСАМИ

- ❑ С Java 7 введен новый механизм обработки исключений под названием "*try с ресурсами*";
- ❑ Этот механизм обработки исключений ориентирован на обработку исключений, когда вы используете ресурсы, которые должны быть закрыты после использования, *например, InputStream, OutputStream* и т.д.
- ❑ Синтаксис оператора **try-with-resources**:

```
try (<спецификация ресурса>) {  
    <код, использующий ресурс>  
}
```

<спецификация ресурса> - это оператор, который объявляет и инициализирует ресурс, такой как файловый поток данных.

Примечание:



По завершении блока **try** ресурс автоматически освобождается, а в случае файла – он закрывается.



Особенности применения

- ❑ Ресурсами могут быть экземпляры классов, которые реализуют интерфейс **AutoCloseable**, определенный в пакете **java.lang** (интерфейс содержит только один метод *close()*, который и вызывается по завершению блока *try*);
- ❑ Ресурс, объявленный в блоке *try*, неявно является **final** (т.е. нельзя изменить ресурс после того, как он был создан; область видимости ресурса ограничивается блоком *try*);
- ❑ Можно управлять несколькими ресурсами, перечислив их в объявлении через точку с запятой (т.е. один и тот же блок *try* может использоваться для контроля нескольких ресурсов);
- ❑ Вторичные исключения подавляются и добавляются в список подавленных (можно посмотреть методом *getSuppressed()*, определенном в классе **Throwable**).

ПОТОКОВЫЙ ВВОД/ВЫВОД ДАННЫХ В Java

Пример 18: до оператора «*try-c-песчсами*»

```
static String readFirstLineFromFileWithFinallyBlock(String path)
                                throws IOException {
    BufferedReader br = new BufferedReader(
        new FileReader(path));
    try {        return br.readLine();    }
    finally {    if (br != null) br.close();    }
}
```

С оператором *try-c-песчсами*

```
static String readFirstLineFromFile(String path)
                                throws IOException {
    try (BufferedReader br = new BufferedReader(
        new FileReader(path))) {
        return br.readLine();
    }
}
```

Потоковый ввод/вывод данных в Java

Измененный пример 11 копирования содержимого файла:

```
import java.io.*;
public class Test_CopyFile {
    public static void main(String [] args)
                                throws IOException {
        int i;
        try (FileInputStream fin = new FileInputStream(args[0]);
            FileOutputStream fout = new FileOutputStream(args[1])) {
            while ((i = fin.read()) != -1)
                fout.write(i);
        }
    }
}
```

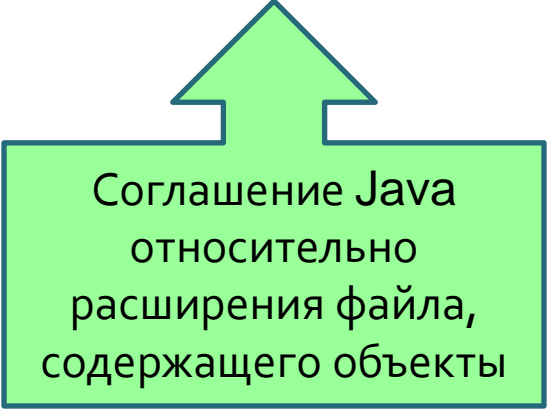
ОБЪЕКТНЫЕ ПОТОКИ

- ❑ Java для передачи между программой и источником/пунктом назначения данных в виде объектов использует специальные *объектные потоки*.
- ❑ Для записи в пункт назначения необходимо:
 - а) создать объект класса **ObjectOutputStream**;
 - б) вызвать метод **writeObject()**.
- ❑ Для чтения из источника данных необходимо:
 - а) создать объект класса **ObjectInputStream**;
 - б) вызвать метод **readObject()**.

Потоковый ввод/вывод данных в Java

Пример 19:

```
try (ObjectOutputStream out = new ObjectOutputStream(  
    new FileOutputStream("data.ser"))) {  
    Integer A = 55;  
    Float F = new Float(5.5f);  
    out.writeObject(A);  
    out.writeObject(F);  
} catch (FileNotFoundException e) {  
    System.out.println("File not found!");  
} catch (IOException e) {  
    e.printStackTrace();  
}
```



Соглашение Java
относительно
расширения файла,
содержащего объекты

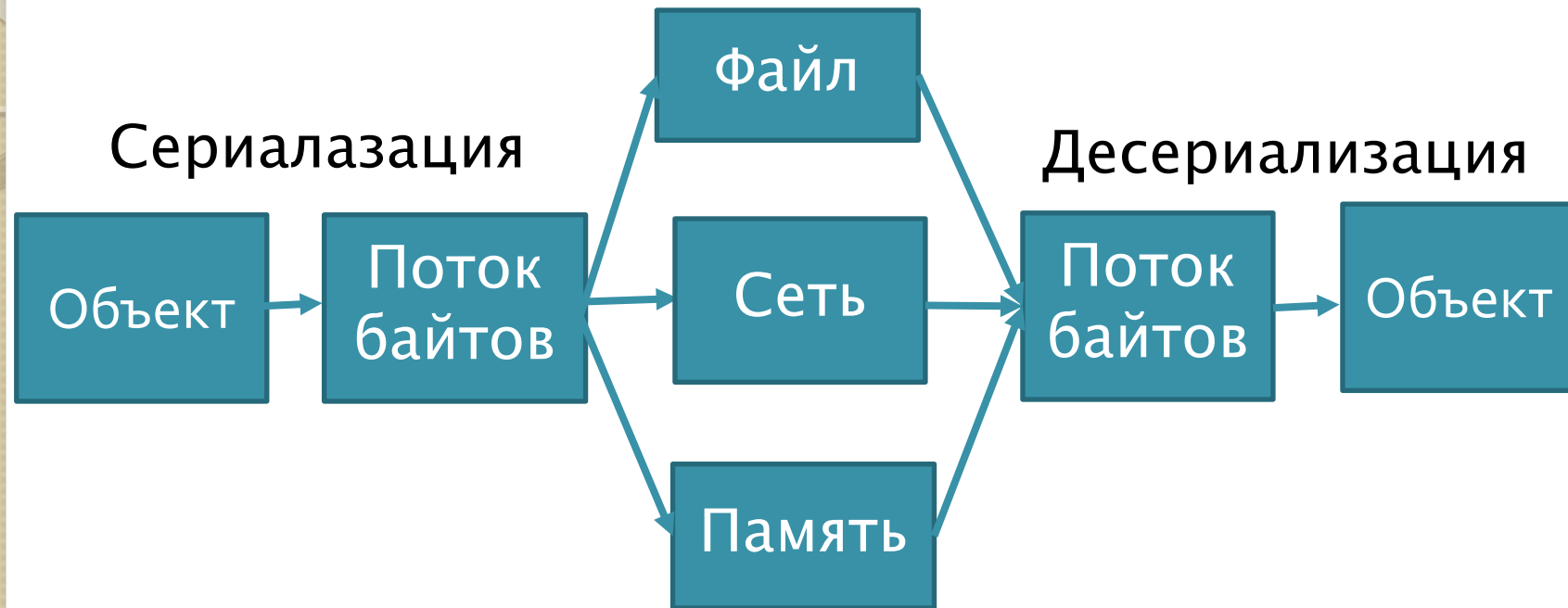


Сериализация в Java

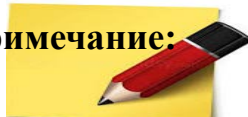
Сериализация в Java

- ❑ **Сериализация** – это процесс преобразования состояния объекта и его метаданных (*например*, имя класса и имена атрибутов) в формат, в котором они могут быть сохранены (*например*, в файле, или буфере памяти, или переданы через сетевое соединение) и восстановлены в этой же или другой компьютерной среде.
 - *Сериализация* используется для облегчения переносимости и связи через сокеты или удаленные вызовы методов Java;
 - *Сериализация* позволяет записывать и читать из потока данные с помощью существующего протокола, чтобы обеспечить совместимость с механизмами записи и чтения по умолчанию.

Сериализация в Java



Примечание:



Десериализация – это процесс восстановления (воссоздания копии) объекта по информации из двоичного потока.



Для того, чтобы объекты класса были сериализуемы, классу необходимо реализовывать интерфейс **java.io.Serializable**

Сериализация в Java

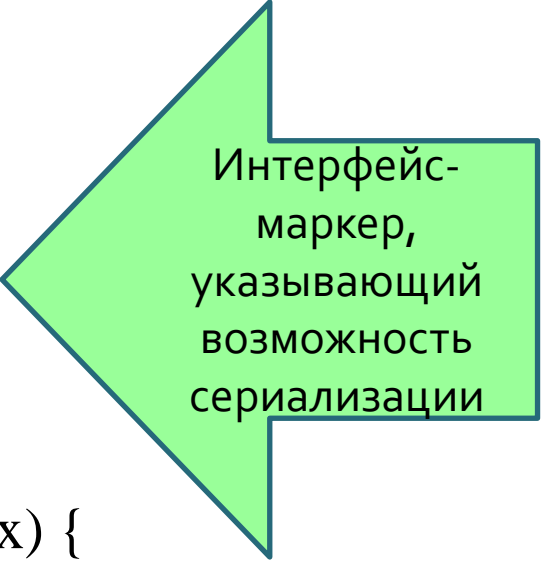
Пример 20:

```
import java.io.*;

class DemoSerial implements Serializable {
    private long number;
    double x;

    public DemoSerial(long number, double x) {
        this.number = number;
        this.x = x;
    }

    public String toString() {
        return "number = " + number + "; x = " + x;
    }
}
```



Интерфейс-
маркер,
указывающий
возможность
сериализации

Сериализация в Java

Продолжение примера 20:

```
public class DSF {  
    public static void main(String[] args)  
        throws IOException, ClassNotFoundException {  
        DemoSerial ds = new DemoSerial(100L, 1.1);  
        File fp = new File("d:\\temp\\demo.ser");  
        try (ObjectOutputStream ostream = new ObjectOutputStream(  
            new FileOutputStream(fp));  
            ObjectInputStream istream = new ObjectInputStream(  
                new FileInputStream(fp))) {  
            ostream.writeObject(ds);  
            ds = null;  
            DemoSerial obj = (DemoSerial)istream.readObject();  
            System.out.println(obj);  
        }  
    }  
}
```

Вывод в консоли:

number = 100; x= 1.1



Особенности сериализации

- ❑ Сериализация и десериализация происходят с учетом версии объекта:
 - Java использует свойство **serialVersionUID**, которое не нужно объявлять и которое вычисляется на основе атрибутов класса, его имени и положения в локальном кластере.
 - Если у источника и приемника различные **serialVersionUID**, то среда исполнения считает, что это разные классы и выбросит **InvalidClassException**.
 - Свойство **serialVersionUID** изменяется всякий раз, когда:
 - ✓ добавляется или удаляется элемент класса;
 - ✓ меняется порядок описания элементов класса.



Особенности сериализации

- ❑ Возможно зафиксировать значение свойства **serialVersionUID**, объявив в классе его как поле типа *private static final long*
- ❑ При попытке сериализовать объект класса, который реализует интерфейс **Serializable**, но объект включает ссылку на несериализуемый класс, тогда будет сгенерировано **NotSerializableException**.
- ❑ Процесс десериализации происходит с использованием рефлексии, т.е. при восстановлении объекта конструктор не вызывается.

Сериализация в Java

Пример 21: сериализация объекта:

```
public class Student implements Serializable {  
    private static final long serialVersionUID = 10L;  
    private String firstName;  
    private int group;  
    private int age;  
    public Student(String firstName, int group, int age) {  
        this.firstName = firstName;  
        this.group = group;  
        this.age = age;  
        System.out.println("Constructor");  
    }  
    public String toString() {  
        return "FirstName = " + firstName +  
            ", Group = " + group + ", Age = " + age;  
    }  
}
```

Фиксация версии
объекта

Сериализация в Java

Продолжение примера 21:

```
Student stud = new Student("Alex", 217, 20);  
try (ObjectOutputStream oos = new ObjectOutputStream(  
    new FileOutputStream("C:\\student.ser"));  
    ObjectInputStream ois = new ObjectInputStream(  
        new FileInputStream("C:\\student.ser"))) {  
    oos.writeObject(stud);  
    stud = null;  
    Student ss = (Student)ois.readObject();  
    System.out.println(ss);  
} catch(IOException | ClassNotFoundException e) {  
    e.printStackTrace();  
}
```

Может возникнуть
при десериализации

Вызов конструктора
только при создании
объекта

Вывод в консоли:

Constructor
FirstName = Alex, Group = 217, Age = 20

СЕРИАЛИЗАЦИЯ СЛОЖНЫХ ОБЪЕКТОВ

- ❑ *Сложный (составной) объект* – это объект, который имеет в своем составе ссылки на другие объекты.

Например, добавим в описание класса **Student** ссылку на объект класса **Elective** и методы *getElective()* и *setElective()*.

```
public class Elective {  
    private long id;  
    private String name;  
    public Elective(long id, String name) {  
        this.id = id;  
        this.name = name;  
    }  
  
    public String toString() {  
        return "id = " + id + ", name = " + name;  
    }  
}
```

Сериализация в Java

Используем сериализацию (продолжение примера 21):

```
Student stud = new Student("Alex", 217, 20);
Elective elect = new Elective(111, "Java EE");
stud.setElective(elect);
try (ObjectOutputStream oos = new ObjectOutputStream(
    new FileOutputStream("C:\\student.ser"));
    ObjectInputStream ois = new ObjectInputStream(
    new FileInputStream("C:\\student.ser"))) {
    oos.writeObject(stud); ← java.io.NotSerializableException: Elective
    Student ss = (Student)ois.readObject();
    System.out.println(ss);
    System.out.println(ss.getElective());
} catch (IOException | ClassNotFoundException e) {
    e.printStackTrace();
}
```

Сериализация в Java



Для правильной записи объекта сериализуемого класса необходимо, чтобы он содержал:

- ссылки на объекты сериализуемых классов;
- ссылки не подлежащие сериализации.

Первый вариант:

```
public class Elective implements Serializable {  
    private long id;  
    private String name;  
    public Elective(long id, String name) {  
        this.id = id;  
        this.name = name;  
    }  
  
    public String toString() {  
        return "id = " + id + ", name = " + name;  
    }  
}
```

Вывод в консоли:

Constructor

FirstName = Alex, Group = 217, Age = 20
id = 111, name = Java EE

ОТКАЗ ОТ СЕРИАЛИЗАЦИИ ЗНАЧЕНИЙ

- Для отказа от сериализации поля класса можно использовать ключевое слово **transient**.

Например,

```
public class Student implements Serializable {  
    private static final long serialVersionUID = 10L;  
    private String firstName;  
    private int group;  
    private int age;  
    private transient Elective elective;  
    // .....
```

Информация о факультативе не была записана в поток

Вывод в консоли:

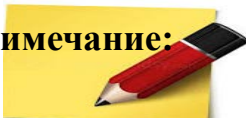
Constructor

FirstName = Alex, Group = 217, Age = 20
null

ПРИМЕНЕНИЕ КЛЮЧЕВОГО СЛОВА **transient**

- ❑ Когда значение поля вычисляется на основе значений остальных полей, то из соображений экономии времени и трафика имеет смысл воздержаться от сериализации поля;
- ❑ Когда значение поля корректно только в рамках текущего контекста (поле, хранит ссылку на ресурс системы);
- ❑ Из соображений безопасности (поле хранит пароль).

Примечание:



Поскольку статические поля класса не сериализуются, то не имеет смысла одновременное использование модификаторов **static** и **transient**.

Сериализация в Java

Пример 22: *сериализация со статическим полем класса,*

```
class A implements Serializable {  
    private static int n = 0;  
    private int i;  
    A(int i) {  
        this.i = i;  
        n = i;  
    }  
    public String toString(){  
        return i + " " + n;  
    }  
}  
// .....
```

Сериализация в Java

Продолжение примера 22:

```
class Temp {  
    public static void main(String []args) throws IOException,  
                                                ClassNotFoundException {  
  
        A a = new A(5);  
        System.out.println("initial -> " + a);  
        ObjectOutputStream oos = new ObjectOutputStream(  
                                new FileOutputStream("serial.ser"));  
        oos.writeObject(a);  
        a = new A(10);  
        System.out.println("second -> " + a);  
        ObjectInputStream ois = new ObjectInputStream(  
                                new FileInputStream("serial.ser"));  
        a = (A)ois.readObject();  
        System.out.println("rebuilt -> " + a);  
    }  
}
```

Вывод в консоли:

```
initial -> 5, 5  
second -> 10, 10  
rebuilt -> 5, 10
```

Сериализация в Java



Если есть необходимость все-таки записывать значение статического поля, тогда запись проводится явно. *Например,*

```
class A implements Serializable {  
    private static int n = 0;  
    private int i;  
    // ....  
    public static void serializeStatic(ObjectOutputStream oos)  
        throws IOException {  
        oos.writeInt(n);  
    }  
    public static void deserializeStatic(ObjectInputStream ois)  
        throws IOException{  
        n = ois.readInt();  
    }  
}
```


Сериализация в Java

```
class Temp {  
    public static void main(String []args) throws IOException,  
                                                ClassNotFoundException {  
        A a = new A(5);  
        System.out.println("initial -> " + a);  
        ObjectOutputStream oos = new ObjectOutputStream(  
            new FileOutputStream("serial.ser"));  
        A.serializeStatic(oos);  
        oos.writeObject(a);  
        a = new A(10);  
        System.out.println("second -> " + a);  
        ObjectInputStream ois = new ObjectInputStream(  
            new FileInputStream("serial.ser"));  
        A.deserializeStatic(ois);  
        a = (A)ois.readObject();  
        System.out.println("rebuilt -> " + a);  
    }  
}
```

Явная запись
и чтение

Вывод в консоли:

```
initial -> 5, 5  
second -> 10, 10  
rebuilt -> 5, 5
```


СЕРИАЛИЗАЦИЯ И НАСЛЕДОВАНИЕ

- ❑ При десериализации используется механизм рефлексии, с помощью которого воссоздается информация о классе и под объект выделяется память, после чего его поля заполняются значениями из потока (конструктор объекта при этом *не вызывается*);
- ❑ При десериализации объекта подкласса, если родительский класс был не сериализуемый, то вызывается конструктор без параметров для родительского класса;
- ❑ При отсутствии конструктора без параметров у родительского класса при десериализации объекта возникнет ошибка типа **InvalidClassException**.

Сериализация в Java

Пример 23: введем для класса **Student** родителя:

```
public class Person {  
    protected String firstName;  
    protected int age;  
    public Person() {  
        System.out.println("Person");  
    }  
    //...  
}
```




Перенесено описание
имени и возраста из
класса **Student**

Сериализация в Java

Изменим описание класса **Student**:

```
public class Student extends Person
                                implements Serializable {
    private static final long serialVersionUID = 10L;
    private int group;
    public Student(String firstName, int group, int age) {
        this.firstName = firstName;
        this.group = group;
        this.age = age;
        System.out.println("Constructor");
    }
    // .....
}
```



Сериализация в Java

Продолжение примера 23:

```
Student stud = new Student("Alex", 217, 20);  
try (ObjectOutputStream oos = new ObjectOutputStream(  
    new FileOutputStream("C:\\student.ser")) ;  
    ObjectInputStream ois = new ObjectInputStream(  
        new FileInputStream("C:\\student.ser"))) {  
    oos.writeObject(stud);  
    stud = null;  
    Student ss = (Student)ois.readObject();  
    System.out.println(ss);  
} catch(IOException | ClassNotFoundException e) {  
    e.printStackTrace();  
}
```

} Вызов конструктора
при создании объекта

Вызов конструктора
при десериализации

Вывод в консоли:

Person
Constructor
Person
FirstName = null, Group = 217, Age = 0

Сериализация в Java

- ❑ Изменим описание класса **Person**, реализовав конструктор с параметрами:

```
public class Person {  
    protected String firstName;  
    protected int age;  
    public Person(String firstName, int age) {  
        this.firstName = firstName;  
        this.age = age;  
        System.out.println("Person");  
    }  
    //...  
}
```

Сериализация в Java

- ❑ Изменим описание конструктора класса **Student**:

```
public Student(String firstName, int group, int age) {  
    super(firstName, age);  
    this.group = group;  
    System.out.println("Constructor");  
}
```

Добавить явный вызов конструктора

- ❑ Выполним сериализацию и десериализацию объекта класса **Student**:

Вывод в консоли:

Person

Constructor

java.io.InvalidClassException: Student; no valid constructor

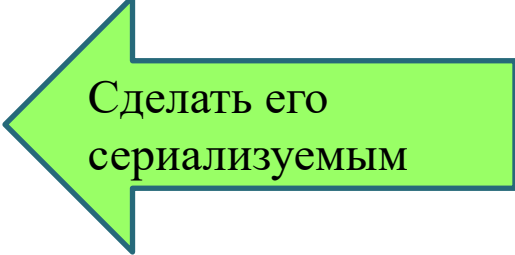
Ошибка в процессе десериализации

Сериализация в Java

- ❑ Если же родительский класс тоже будет сериализуемым, то тогда для его восстановления в процессе десериализации будет использоваться рефлексия.

Например, изменим снова описание класса **Person**:

```
public class Person implements Serializable {  
    protected String firstName;  
    protected int age;  
    public Person() {  
        System.out.println("Person");  
    }  
    //...  
}
```



Сделать его
сериализуемым

Вывод в консоли:

Person

Constructor

FirstName = Alex, Group = 217, Age = 20

ПОЛЬЗОВАТЕЛЬСКАЯ СЕРИАЛИЗАЦИЯ

I) Модификация сериализации по умолчанию

- ❑ *Для реализации пользовательской сериализации необходимо определить методы `writeObject()` и/или `readObject()`.*
 - Это не переопределение методов потоков **ObjectOutputStream** и **ObjectInputStream**:
виртуальная машина проверяет и вызывает эти методы от средств рефлексии.
 - Эти методы должны быть описаны как закрытые, чтобы гарантировать не возможность их переопределения или перегрузки.

Сериализация в Java

Пример 24, проверка вызова методов:

```
public class Student implements Serializable {  
    //...  
    private void writeObject(ObjectOutputStream out)  
                                throws IOException {  
        System.out.println("Custom Serialization!");  
        throw new NotSerializableException("ERROR!");  
    }  
    private void readObject(ObjectInputStream in)  
                                throws IOException {  
        System.out.println("Custom Deserialization!");  
        throw new NotSerializableException("ERROR!");  
    }  
}
```

Вывод в консоли:

Person

Constructor

Custom Serialization!

java.io.NotSerializableException: ERROR!

Сериализация в Java

Пример 24, пользовательский порядок записи:

```
public class Student implements Serializable {  
    //...  
    private void writeObject(ObjectOutputStream out)  
                                throws IOException {  
        out.writeUTF(firstName);  
        out.writeInt(age);  
        out.writeInt(group);  
    }  
    private void readObject(ObjectInputStream in)  
                                throws IOException {  
        firstName = in.readUTF();  
        age = in.readInt();  
        group = in.readInt();  
    }  
}
```

II) Собственная сериализация

- ❑ Для этого используется интерфейс ***Externalizable***, который содержит два метода и которые нужно переопределить:
 - **public void *writeExternal*(ObjectOutput out) throws**
IOException;
 - **public void *readExternal*(ObjectInput in) throws**
IOException, *ClassNotFoundException*;
- ❑ Необходимо, чтобы сериализуемый класс имел конструктор без параметров, так как при десериализации вызывается сначала этот конструктор, а затем метод *readExternal()*;
- ❑ Все наследники такого класса тоже будут считаться реализующими интерфейс ***Externalizable***, и у них тоже должен быть конструктор без параметров!



Сериализация в Java

Что дает использование интерфейса *Externalizable*

- ❑ Полный контроль над процессом сериализации:
 - Сериализация по умолчанию (поток *ObjectOutputStream*) сохраняет ссылки на объекты, которые в него записываются (т.е. если состояние объекта, который уже был записан, будет записываться снова, то новое состояние не сохраняется):

```
MyObject obj = new MyObject();
```

```
obj.setState(100);
```

Установка состояния

```
out.writeObject(obj);
```

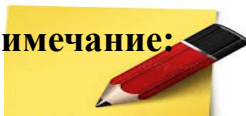
```
obj.setState(200);
```

Изменение состояния

```
out.writeObject(obj);
```

Новое состояние не будет сохранено!

Примечание:



Собственная сериализация позволит избежать такой ситуации!

Сериализация в Java

- Контроль версией хорошо работает до тех пор, пока вносимые изменения совместимы (т.е. при добавлении и/или удалении методов можно зафиксировать значение свойства *serialVersionUID*, чтобы не получить ошибку **InvalidClassException**; при несовместимых изменениях — например, изменение иерархии объектов или прекращение реализации интерфейса *Serializable* — это делать не рекомендуется);

Собственная сериализация не работает со свойством *serialVersionUID*

- Сериализация по умолчанию является простой, но менее производительной.

Собственная сериализация может повысить производительность до 50%.



Особенности собственной сериализации

- ❑ Если поле в классе объявлено с ключевым словом *transient*, то при использовании интерфейса **Externalizable** такое поле все равно можно и записывать и читать из потока;
- ❑ Если поле в классе объявлено как статическое, то при использовании интерфейса **Externalizable** такое поле все равно можно и записывать и читать из потока;
- ❑ Если поле в классе объявлено с модификтором **final**, то десериализовать его не нельзя (**final**-поля должны быть инициализированы в конструкторе, а после этого изменить значение этого поля в методе *readExternal()* будет невозможно).

Сериализация в Java

Пример 26, применения интерфейса *Externalizable*:

```
public class User implements Externalizable {  
    private int id;   private String username;  
    public User() {   }  
    public User(int id, String username) {  
        this.id = id;   this.username = username;  
    }  
    @Override  
    public void writeExternal(ObjectOutput out) throws IOException {  
        out.writeInt(id);   out.writeObject(username);  
    }  
    @Override  
    public void readExternal(ObjectInput in) throws IOException,  
        ClassNotFoundException {  
        id = in.readInt();   username = (String) in.readObject();  
    }  
}
```


Сериализация в Java

Продолжение примера 26:

```
User userWrite = new User(1, "AlexUser");
try (ObjectOutputStream oos = new
        FileOutputStream("userfile.ser"));
    ObjectInputStream ois = new ObjectInputStream(
        new FileInputStream("userfile.ser"))) {
    oos.writeObject(userWrite);
    User userRead = (User) ois.readObject();
    System.out.println("User: " + userRead);
} catch (IOException | ClassNotFoundException e) {
    e.printStackTrace();
}
```

Вывод в консоли:

User: id = 1, username = AlexUser