образование ПОЛИТЕХ одноклассники экосистема

Исключения и потоки ввода/вывода

Галкин Александр Сергеевич

2022 г.

Оглавление

1. Исключения
    - Непроверяемые и проверяемые
    - Обработка
2. Файловая система
    - Path vs File
    - DirectoryStream
3. Блокирующие потоки
    - Потоки байтов
    - Потоки символов
4. Сериализация

Исключения

Ошибки

В любой программе может произойти ошибка, что можно с этим сделать?

- Выйти из программы
- Вернуть специальное значение
- Дополнительный метод на проверку ошибки
- Бросить исключение

Исключения

- Программа на каком-то уровне кидает исключение
- Уровнем выше можно обработать исключение
  - Сохранить данные и завершить программу
  - Обработать исключение и продолжить программу
  - Ничего не делать, бросить исключение дальше

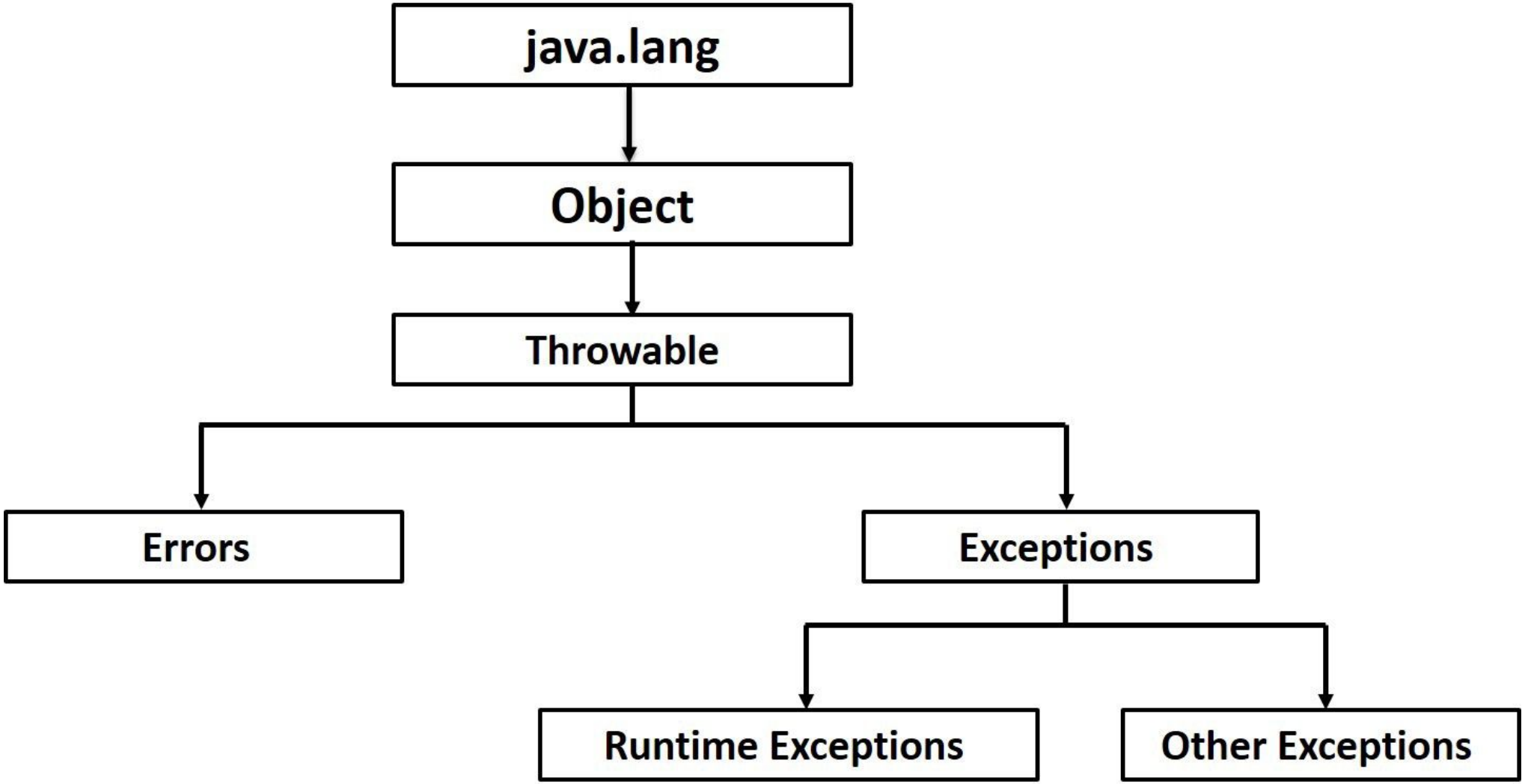
Throwable

- Исключение это объект, который надо создавать
- Все исключения в Java наследуются от класса ***Throwable***
- Исключения бросаются с помощью ключевого слова ***throw***

```
throw new NullPointerException();
```

- Исключение содержат в себе сообщение, стэк трэйс
- Также, исключение может содержать в себе другое исключение, если оно было вызвано им
- Часто, на каждый тип ошибки создают отдельный класс исключений

Иерархия исключений



Error

- Ошибки виртуальной машины
- Не нужно пытаться их обрабатывать
- Примеры
  - ***OutOfMemoryError***
  - ***NoClassDefFoundError***
  - ***StackOverflowError***

Exception. Проверяемые исключения

- Все подклассы ***Exception*** являются проверяемыми исключениями
- Проверяемые исключения необходимо декларировать и обрабатывать
- Если метод декларирован, как кидающий исключение, то все методы, которые его используют должны или обрабатывать это исключение или также декларировать его
- `public static void exampleIOE(String arg) throws IOException {`

RuntimeException. Непроверяемые исключения

- Все подклассы ***RuntimeException*** являются непроверяемыми исключениями
- Непроверяемые исключения не обязательно декларировать и обрабатывать, их можно кидать из любой точки программы
- Примеры
  - ***NullPointerException***
  - ***ArrayIndexOutOfBoundsException***
  - ***ArithmeticException***

Свое исключение

- Придумываем название: по названию должно быть сразу понятно, какая ошибка случилась
- Выбираем тип: проверяемое или непроверяемое
- Создаем три конструктора:
  - Дефолтный
  - Со строкой, для дополнительного описания
  - Со строкой и ***Throwable*** для описания и вложения другой ошибки

Обработка исключений

- Исключения обрабатываются с помощью конструкции ***try-catch***
- В блоке ***try*** пишется код, который может кидать исключения
- В блоке ***catch*** пишется тип исключения (можно несколько, через знак |) , и что мы с ним делаем
- В блоке ***finally*** пишется код, который выполняется в любом случае, после ***try*** или ***catch*** (обычно там закрываются ресурсы или снимают блокировки)

try-catch. Пример

```
try {
    for (String arg : args) {
        validate(arg);
        doSomething(arg);
    }
} catch (ValidationException e) {
    System.out.println(e.getMessage() + ". Please try again");
    throw new IllegalArgumentException();
} catch (Throwable e) {
    System.out.println(e.getClass());
} finally {
    showResult();
}
```

try-catch-finally. Пример

```
InputStream is = new FileInputStream( name: "a.txt");
try {
    readFromInputStream(is);
} finally {
    try {
        is.close(); // тоже может бросить ошибку, потеряем исходную
    } catch (IOException e) {
        // игнорируем
    }
}
```

try-with-resources. Пример

```
try (InputStream is = new FileInputStream( name: "a.txt")) {
    readFromInputStream(is);
}
```

```
public interface AutoCloseable {
```

try-catch. Варианты обработки исключений

- Ошибка фатальная - закрываем программу
- Нет полной информации, или заворачиваем ошибку в другую, или прокидываем исходную (в последнем варианте стоит задуматься, а нужен ли там ***try-catch***?)
- Ничего страшного не случилось - продолжаем работать
- Сообщить, что надо повторить вызов
- **ГЛАВНОЕ**: во всех случаях обязательно залогировать ошибку!

Работа с файловой системой

java.io.File

- Класс файл представляет файл или директорию в файловой системе
- Задается с помощью относительного или абсолютного пути
- `public static final String separator`
- ***File*** имеет методы для получения части пути и канонического пути: `String getName()`, `String getParent()`, `String getCanonicalPath()`

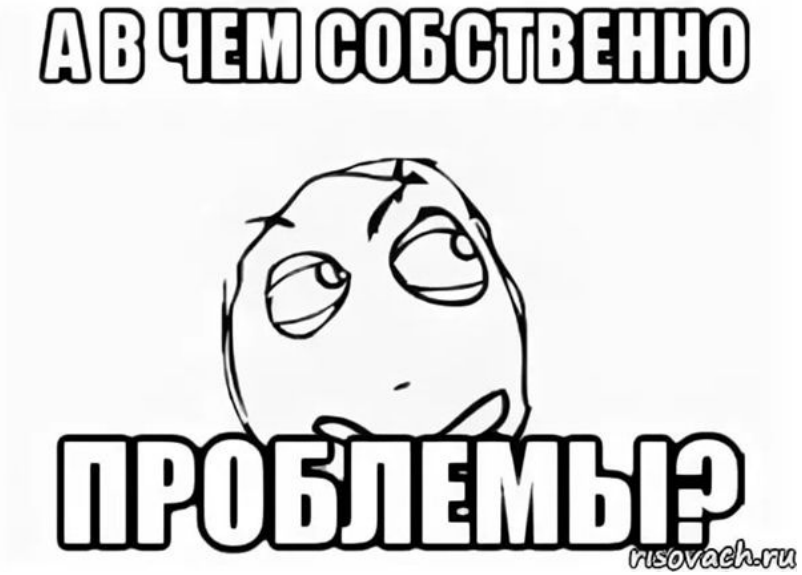
Работа с файлами

- Объекты класс ***File*** никак не привязаны к реальным директориям и файлам на диске
- Методы `boolean exists()`, `boolean isDirectory()`, `boolean isFile()` проверяют существование и тип объекта
- Для файла доступны методы: `long length()`, `long lastModified()`
- Для директории доступны методы: `String[] list(FilenameFilter filter)`
- Создание файла: `boolean createNewFile()` — кидает исключение
- Создание директории: `boolean mkdir()`, `boolean mkdirs()` — не кидают исключения

Работа с файлами. Дополнительные методы

- Удаление: `boolean delete()` — если удаляем директорию, то она должна быть пустой
- Переименование (и перенос в другую директорию): `boolean renameTo(File dest)`
- Метода копирования отсутствует
- Удаление не пустой папки надо писать самому с рекурсивным обходом

java.io.File. Недостатки



- Почти все методы возвращают ***true*** или ***false***, — не можем узнать причину ошибки
- Много методов не реализовано
- Обратная совместимость - нельзя просто взять и поменять контракты

Что с этим делать?

- Написать новый API

java.nio.file.Path

- ***Path*** это интерфейс, а не класс
- Не связан с файлам на диске
- По сути это просто строка с набором методов
- Создание: `Paths.get("polis");`
- Методы конвертации: `path.toFile(); file.toPath();`
- ***Path*** имеет аналоги всех синтаксических операций из класса ***File***
- Также многие методы доработаны, например

```
Path getName(int index)
```

- ***Path*** не имеет методов доступа к файловой системе в отличие от класса ***File***

Path. Доступ к файловой системе

- Почти любой доступ к файловой системе получается через статические методы класса ***Files***
- В ***Files*** есть аналоги всех операций из класса ***File*** и даже больше, например есть копирование:

```
Path copy(Path source, Path target,
CopyOption... options)
```

- Работа с существующими директориями осуществляется с помощью метода в классе ***Files*** и нового объекта:

```
DirectoryStream<Path> newDirectoryStream(
Path dir)
```

DirectoryStream<T>

- ***DirectoryStream*** — это интерфейс, представляющий директорию открытую на чтение, поэтому его нужно закрывать и освобождать ресурсы
- Поэтому ***DirectoryStream*** должен использоваться в блоке ***try*** с ресурсами
- Имеет всего два метода: `void close()` и `Iterator<T> iterator()`
- Загружает директории последовательно, что удобно, т. к. директории могут быть очень большими

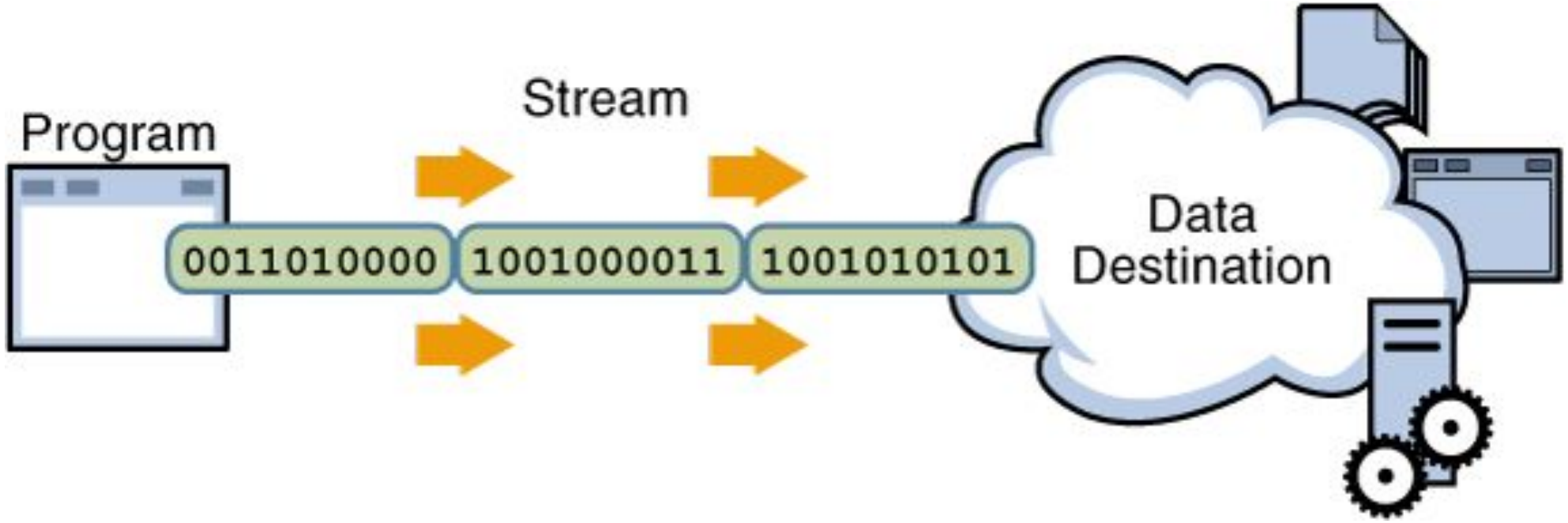
java.nio.file. Рекурсивный обход директорий

- Рекурсивного удаления все также нет
- Добавлен новый механизм рекурсивного обхода:
  ```
  Path walkFileTree(Path start, FileVisitor<?
  super Path> visitor)
  ```
- Интерфейс ***FileVisitor*** содержит методы, в которых описывается что делать перед входом в ***dir***, после выхода из ***dir***, после нахождения файла и после того, как файл нашли но не смогли прочитать атрибуты
- `SimpleFileVisitor<T>` — простейшая реализация из заглушек

Потоки байт

Блокирующий ввод/вывод

- Данные читаются/записываются из потока/в поток без какого либо кэширования
- Надо сделать буффер, из/в которого/который мы будем читать/писать



Потоки байт

- ***java.io.InputStream*** — поток байтов из которого можно читать
- ***java.io.OutputStream*** — поток байтов в который можно писать
- Это абстрактные классы реализующие интерфейс ***Closeable***, т.е. экземпляры наследников данных классов нужно закрывать
- Неизвестно откуда мы читаем и куда мы пишем, эта информация зависит от наследников

java.io.InputStream

- `int read()` — читает 1 байт из потока и сдвигается на 1 байт
- `int read(byte b[])` — читает ***b.length*** байт и записывает их в массив, возвращает количество прочитанных байт
- `int read(byte b[], int off, int len)` — читает ***len*** байт и пишет в массив начиная с индекса ***off***
- `long skip(long n)` — пропускает ***n*** байт
- `void mark(int readlimit),void reset()` — метод ***mark*** помечает байт с которого начнется чтение, если вызвать метод ***reset***
- Все методы могут бросить ***IOException***

java.io.OutputStream

- `void write(int b)` — пишем 1 байт в поток
- `void write(byte b[])` — пишем массив байт в поток
- `void write(byte b[], int off, int len)` — пишем ***len*** байт из массива в поток, с позиции ***off***
- `void flush()` — сбрасывает промежуточные буфера, где хранятся данные перед передачей их операционной системе
- Просто вызов метода ***write*** не гарантирует, что операционная система получит данные, но метод close внутри себя вызывает метод ***flush***
- Все методы могут бросить ***IOException***

Стримы для работы с файлами

- ***FileInputStream***, ***FileOutputStream*** — для работы со старым API (конструктор принимает или строку - путь до файла, или экземпляр класса File)
- `InputStream newInputStream(Path path, OpenOption... options)`, `OutputStream newOutputStream(Path path, OpenOption... options)` — методы для получения стримов с помощью нового API

Дополнительные варианты стримов

- `OutputStream outputStream = socket.getOutputStream();`
  `InputStream inputStream = socket.getInputStream();`
  — стримы сетевого соединения
- ***ByteArrayInputStream***, ***ByteArrayOutputStream*** — стрим который можно создать с помощью существующего массива байт и стрим из которого этот массив байт можно получить

Дополнительные стримы

- ***BufferedInputStream***, ***BufferedOutputStream*** — содержат буфер для ввода/выводы сразу нескольких байтов
- ***PushbackInputStream*** — умеет записывать данные обратно в поток
- ***SequenceInputStream*** — читает из нескольких стримов по очереди
- ***DataInputStream***, ***DataOutputStream*** — добавляет методы для чтения/записи примитивов и строк

java.io.PrintStream

- Добавляет методы для записи примитивов, строк и объектов
- Делает внутри себя два преобразования:
- Объект в строку
- Строку в последовательность байт
- Не бросает исключений, выставляет флаг

java.io.RandomAccessFile

- ***DataInput***, ***DataOutput*** — реализуемые интерфейсы, НЕ стримы
- Позволяет позиционироваться внутри файла

Потоки символов

Потоки символов

- ***java.io.Reader*** — поток символов из которого можно читать
- ***java.io.Writer*** — поток символов в который можно писать
- Это абстрактные классы реализующие интерфейс ***Closeable***, т.е. экземпляры наследников данных классов нужно закрывать
- Неизвестно откуда мы читаем и куда мы пишем, эта информация зависит от наследников

Конвертация потока байт в поток символов

- `public InputStreamReader(InputStream in, String charsetName)`
  здесь ***charsetName*** это название кодировки, например UTF-8
- `public OutputStreamWriter(OutputStream out, Charset cs)`
  здесь ***cs*** и есть объект кодировки
- Основные кодировки лежат в классе ***StandardCharsets***
- Если не указывать ничего кроме стрима, будет использовать кодировка по умолчанию

Файл как поток символов

- `FileReader(String fileName)`, `FileWriter(File file)` — классы для строкового чтения и записи
- `Reader reader = new InputStreamReader( new FileInputStream(fileName), StandardCharsets.UTF_8);`
- `Writer writer = new OutputStreamWriter( new FileOutputStream(fileName), StandardCharsets.UTF_8);`
- Второй способ предпочтительнее, так как можно указать кодировку

Высокоуровневые стримы

- Стримы можно заворачивать друг в друга (внутренний - низкоуровневый, внешний реализует высокоуровневые методы)
- `public BufferedReader(Reader in)` — оборачивает некоторый стрим и добавляет методы для чтение целой строки
- Аналогично `public BufferedWriter(Writer out)`
- Также эти классы пишут и читают не по одному символу, а сразу большими блоками

Высокоуровневое чтение файлов

- Метод для чтения с помощью ***BufferdReader*** в классе ***Files***

```
BufferedReader newBufferedReader(Path path,
Charset cs)
```

- Метод для чтения небольших файлов

```
List<String> readAllLines(Path path, Charset cs)
```

- Метод для записи с помощью ***BufferdWriter*** в классе ***Files***

```
BufferedWriter newBufferedWriter(Path path,
Charset cs, OpenOption... options)
```

- Метод для записи нескольких строк в файл

```
Path write(Path path,
Iterable<? extends CharSequence> lines,
Charset cs, OpenOption... options)
```

java.io.PrintWriter

- `PrintWriter(OutputStream out)` — создаем удобный ***Writer*** на основе заданного символьного стрима
- `void print(int i)` — пишем целое число в поток
- `void print(String s)` — пишем строку в поток
- `void print(Object obj)` — пишем объект как строку (используется метод ***toString()***)
- `PrintWriter printf(String format, Object ... args)` — аналог записи строки с параметрами из C++
- Методы НЕ кидают исключения, они просто устанавливают флаг ошибки, который можно прочитать с помощью метода boolean ***checkError()***

java.io.PrintStream

- `PrintStream(OutputStream out)` — создаем удобный ***OutputStream*** на основе заданного байтового стрима
- Имеет все те же методы, что и ***PrintWriter***
- Делает внутри себя два преобразования:
  - Объект в строку
  - Строку в последовательность байт

java.util.Scanner

- Аналог ***PrintWriter*** только для чтения
- Не является ***Reader***-ом и ***InputStream***-ом
- `Scanner(InputStream source)` — можно создать на основе стрима и ридера
- `String next()` — возвращает следующую строку
- `double nextDouble()` — возвращает следующий дабл
- `boolean hasNextLine()` — возвращает истину, если дальше есть еще строка
- `Scanner useDelimiter(Pattern pattern)` — устанавливает разделить между элементами, по умолчанию это пробелы

Стандартные потоки ввода и вывода

- ***System.in*** — ***InputStream***, для работы с текстом - удобно обернуть в ***Scanner***
- ***System.out***, ***System.err*** — ***PrintStream***, можно выводить двоичные данные (метод ***write***) и текстовые (метод ***print***)

Сериализация объектов

Сериализация объектов

- `interface Serializable` — интерфейс, который говорит *jvm*, что этот объект можно сериализовать в поток байт. *Jvm* это делает сама
- Если какое-то поле объекта мы не хотим сериализовать, его надо пометить идентификатором ***transient***
- Все поля объекта ***Serializable***, которые не помечены как ***transient***, должны быть также ***Serializable*** или примитивами
- `interface Externalizable extends java.io.Serializable` — ручная сериализация с помощью методов

```
void writeExternal(ObjectOutput out)
void readExternal(ObjectInput in)
```

Сериализация объектов. Стримы

Для сериализации и десериализации объектов, помеченных как ***Serializable***, используются следующие стримы

- `class ObjectOutputStream` и метод `void writeObject(Object obj)`
- `class ObjectInputStream` и метод `Object readObject()`

Сериализация. Порядок записи

- Основной класс и его поля с типами
- Суперклассы и их поля с типами
- Сами данные в обратном порядке (от супер класса к исходному)
- Если в данных содержится объект, то работаем с ним с первого шага

Автоматическая сериализация

Ручная сериализация

- `interface Serializable`
  - `void writeObject(ObjectOutputStream out)`
  - `void readObject(ObjectInputStream out)`
- `interface Externalizable`
  - `void writeExternal(ObjectOutput out)`
  - `void readExternal(ObjectInput in)`
- Руками в отдельном методе записываем поля в заданном порядке

Спасибо!

образование

ПОЛИТЕХ

одноклассники
экосистема