### Коллекции. Сортировка элементов коллекции. Интерфейсы java.util.Comparable и java.util.Comparator

В Java **коллекции** — это наборы объектов, которые можно хранить, управлять и обрабатывать. Все основные коллекции находятся в пакете java.util.

Примеры коллекций:

* List (например, ArrayList, LinkedList)
* Set (например, HashSet, TreeSet)
* Map (например, HashMap, TreeMap)
* Queue (например, PriorityQueue)

В Java можно сортировать:

* **по умолчанию** — через реализацию интерфейса Comparable;
* **по произвольному правилу** — через интерфейс Comparator

**Интерфейс Comparable<T>**

Этот интерфейс используется, чтобы объекты **сортировались по умолчанию**. Класс, реализующий Comparable, должен переопределить метод:

int compareTo(T o);

Метод возвращает:

* 0, если объекты равны;
* < 0, если текущий объект меньше;
* > 0, если больше.

**Интерфейс Comparator<T>**

Используется, когда нужно сортировать **по разным критериям** или **без изменения класса**.

Методы:

* int compare(T o1, T o2)
* static Comparator.comparing(...)

**Интерфейс Collection**

Это **корневой интерфейс** всех коллекций в Java (кроме Map). Он определяет основные операции с группой объектов:

* добавление: add(), addAll()
* удаление: remove(), clear()
* проверка: contains(), isEmpty(), size()
* перебор: через iterator() или enhanced-for

## Интерфейс List и ArrayList

List — это **упорядоченная коллекция**, допускающая **дубликаты**. Каждый элемент имеет индекс. Класс ArrayList — наиболее популярная реализация.

Особенности:

* Быстрый доступ по индексу
* Может хранить дубликаты
* Сохраняет порядок добавления

## HashSet

HashSet реализует интерфейс Set, который **не допускает дубликатов** и **не гарантирует порядок** элементов.

Особенности:

* Быстрая вставка и поиск (O(1) в среднем)
* Нет дубликатов
* Неупорядоченная структура
* Использует hashCode() и equals()

## TreeSet

TreeSet — это реализация Set, **сохраняющая элементы в отсортированном порядке** (по возрастанию или заданному компаратору).

Особенности:

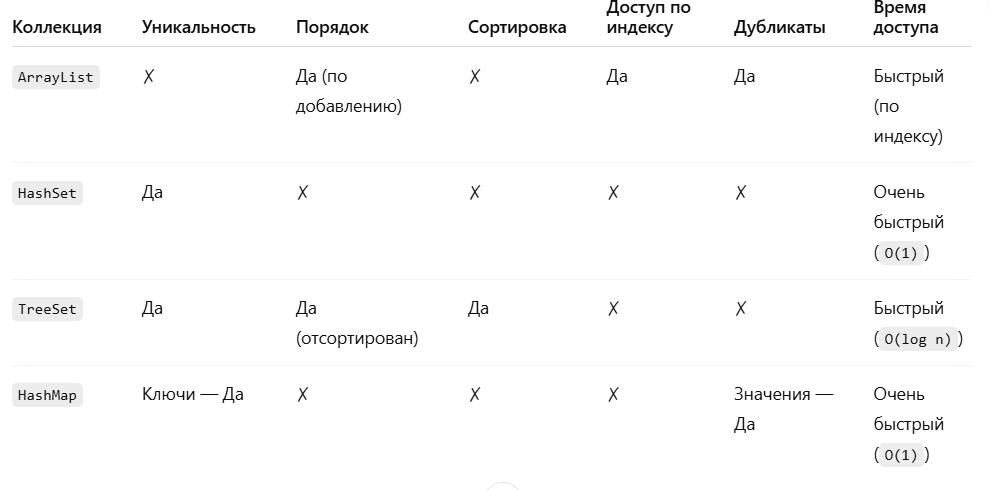
* Элементы сортируются автоматически (Comparable или Comparator)
* Нет дубликатов
* Работает медленнее HashSet (в среднем O(log n))

## HashMap

HashMap<K, V> — **ассоциативный массив** (ключ → значение). Ключи уникальны, значения — любые.

Особенности:

* Нет порядка ключей
* Быстрый доступ по ключу (O(1) в среднем)
* Ключи уникальны, значения могут повторяться
* Использует hashCode() и equals() для ключей



### Категории коллекций - списки, множества. Интерфейс java.util.Map и его реализации.

В Java коллекции делятся на **две большие группы**:

| **Категория** | **Что такое** | **Примеры интерфейсов** |
| --- | --- | --- |
| **Списки (List)** | Упорядоченные коллекции, могут содержать дубликаты. | *List (ArrayList, LinkedList)* |
| **Множества (Set)** | Уникальные элементы без дубликатов. | *Set (HashSet, TreeSet)* |

**Важно:**  
Есть ещё отдельная группа — **отображения (Map)**, но Map **не** является частью Collection интерфейса!

**Списки (List)**

* Элементы имеют **индекс** (0, 1, 2, ...).
* Можно хранить **дубликаты**.
* Порядок **сохраняется**.

**Основные реализации:**

* **ArrayList** — быстрый доступ по индексу.
* **LinkedList** — оптимизирован для частых вставок и удалений.

**Множества (Set)**

* **Нет дубликатов**.
* Порядок может быть не определён или определён (в зависимости от реализации).

**Основные реализации:**

* **HashSet** — быстрое хранение без порядка.
* **TreeSet** — элементы автоматически сортируются.
* **LinkedHashSet** — сохраняет порядок вставки.

**Интерфейс Map<K, V>**

Map — это структура данных, которая хранит **пары ключ–значение**.

* Ключи **уникальны**.
* Значения могут **повторяться**.
* Map **НЕ** является Collection, но это часть Java Collections Framework.

## Методы интерфейса Map

| **Метод** | **Описание** |
| --- | --- |
| put(K key, V value) | Добавить пару ключ-значение |
| get(Object key) | Получить значение по ключу |
| remove(Object key) | Удалить по ключу |
| containsKey(Object key) | Проверить наличие ключа |
| containsValue(Object value) | Проверить наличие значения |
| keySet() | Получить множество всех ключей |
| values() | Получить коллекцию всех значений |
| entrySet() | Получить Set<Map.Entry<K,V>> — пары |

**Основные реализации Map**

| **Реализация** | **Особенности** |
| --- | --- |
| HashMap | Быстрая, нет порядка ключей. Основана на хэш-таблице. |
| LinkedHashMap | Сохраняет порядок добавления ключей. |
| TreeMap | Ключи автоматически сортируются (натуральный порядок или Comparator). |
| Hashtable | Устаревшая, синхронизированная версия HashMap. Не рекомендуется для нового кода. |

### Параметризованные типы. Создание параметризуемых классов. Wildcard-параметры.

**Что такое параметризованные типы (Generics)**

**Обобщения (Generics)** — это способ указать **тип данных как параметр** при создании классов, интерфейсов и методов.

**Пример:**

List<String> names = new ArrayList<>();

List<Integer> numbers = new ArrayList<>();

* List<T> — параметризованный тип.
* T заменяется на String, Integer и т. д.
* Компилятор проверяет типы **на этапе компиляции** → меньше ошибок.

Можно создать собственный обобщенный класс:

public class Box<T> {

private T value;

public void set(T value) {

this.value = value;

}

public T get() {

return value;

}

}

Обобщение можно делать не только в классах, но и в **отдельных методах**:

public class Util {

public static <T> void printTwice(T item) {

System.out.println(item);

System.out.println(item);

}

}

**Ограничения параметров (extends, super)**

Ты можешь ограничивать, какие типы допустимы в параметрах:

**🔹 Верхнее ограничение (extends) — только типы, наследующие от заданного:**

public class Box<T extends Number> {

private T value;

}

Теперь:

Box<Integer> b1 = new Box<>(); // ✅

Box<Double> b2 = new Box<>(); // ✅

Box<String> b3 = new Box<>(); // ❌ Ошибка компиляции

**🔹 Нижнее ограничение (super) — используется с wildcards, см. ниже**

**Wildcards (символ подстановки ?)**

Обобщения с ? позволяют **принимать типы без точного указания параметра**, с **определёнными ограничениями**.

**✅ Пример: <?> — неизвестный тип**

public static void printList(List<?> list) {

for (Object obj : list) {

System.out.println(obj);

}

}

Используется, когда мы хотим читать элементы **любого типа**:

printList(List.of("one", "two"));

printList(List.of(1, 2, 3));

**🔼 Ограничение сверху: <? extends T>**

Позволяет передать **T или его подклассы** — **для чтения**, не для записи.

public static void printNumbers(List<? extends Number> list) {

for (Number n : list) {

System.out.println(n);

}

}

Можно передать List<Integer>, List<Double> и т. д.

**🔽 Ограничение снизу: <? super T>**

Позволяет передать **T или его суперклассы** — используется при **записи**.

public static void addNumbers(List<? super Integer> list) {

list.add(10); // можно

list.add(20); // можно

// list.add(3.14); // нельзя — Double не подойдёт

}

Можно передать List<Integer>, List<Number>, List<Object>

**PECS правило (для запоминания)**

**PECS = "Producer Extends, Consumer Super"**

* ? extends T → если объект **производит (выдаёт)** данные → читаем.
* ? super T → если объект **потребляет (принимает)** данные → пишем.

**Ограничения дженериков в Java**

* Нельзя создавать массивы: T[] arr = new T[10]; ❌
* Нельзя использовать обобщённые классы с примитивами: Box<int> ❌
* Нельзя делать instanceof с параметром типа: if (obj instanceof T) ❌

### Классы-оболочки. Назначение, область применения, преимущества и недостатки. Автоупаковка и автораспаковка.

**Классы-оболочки (wrapper classes)** — это объекты, которые оборачивают примитивные типы данных Java в полноценные объекты.

| **Примитив** | **Класс-оболочка** |
| --- | --- |
| int | Integer |
| double | Double |
| char | Character |
| boolean | Boolean |
| long | Long |
| float | Float |
| short | Short |
| byte | Byte |

**🔍 Пример:**

int num = 5;

Integer boxed = Integer.valueOf(num); // упаковка

int unboxed = boxed.intValue(); // распаковка

**Автоупаковка и автораспаковка (Autoboxing / Unboxing)**

Java позволяет **автоматически** преобразовывать примитивы и обратно — начиная с Java 5.

**✅ Пример: автоупаковка (autoboxing)**

List<Integer> list = new ArrayList<>();

list.add(10); // int → Integer (автоматически)

**✅ Пример: автораспаковка (unboxing)**

Integer x = 15;

int y = x; // Integer → int (автоматически)

**Назначение и область применения**

**🧱 Где нужны классы-оболочки:**

* 📦 **Коллекции**: нельзя хранить примитивы в List, Set, Map

List<int> list = ... // ❌ ошибка

List<Integer> list = new ArrayList<>(); // ✅

* 📊 **Обобщения (Generics)** — работают только с объектами, не с примитивами.
* 📚 **Reflection**, **Streams**, **Optional** и прочие API — требуют объекты.
* ✅ **Передача null** — примитивы не поддерживают null, объекты — да.
* 🔁 **Автоматическая упаковка/распаковка** — делает код чище, понятнее.

# Преимущества классов-оболочек

| **Преимущество** | **Описание** |
| --- | --- |
| ✅ Используются в обобщениях | Примитивы нельзя параметризовать |
| ✅ Совместимы с null | Можно использовать как "отсутствие значения" |
| ✅ Методы для работы | Например: Integer.parseInt(), Double.compare() |
| ✅ Простота с автопреобразованием | Java сама конвертирует при необходимости |

# ⚠️ 5. Недостатки классов-оболочек

| **Недостаток** | **Описание** |
| --- | --- |
| ❌ Больше памяти | Объекты требуют больше памяти, чем примитивы |
| ❌ Медленнее по производительности | Работа с объектами медленнее, чем с примитивами |
| ❌ Возможность NullPointerException | При автораспаковке null — исключение |
| ❌ Не всегда очевидные ошибки | Например: сравнение Integer через == |

**Пример ошибки:**

Integer a = 128;

Integer b = 128;

System.out.println(a == b); // ❌ false

Integer x = 100;

Integer y = 100;

System.out.println(x == y); // ✅ true

Почему так?  
Java кэширует значения -128 до 127 для Integer, но **значения за пределами не кэшируются**, и сравнение == работает **по ссылке**, а не по значению. Надо использовать .equals():

System.out.println(a.equals(b)); // ✅ true

### Потоки ввода-вывода в Java. Байтовые и символьные потоки. "Цепочки" потоков (Stream Chains).

**Потоки ввода-вывода (Java I/O Streams)**

Java I/O основан на **абстракции потока (stream)** — последовательности данных, которые читаются или записываются **побайтово или посимвольно**.

Все классы ввода-вывода находятся в пакете java.io.

**🚪 Основные категории потоков**

| **Тип** | **Примеры классов** | **Назначение** |
| --- | --- | --- |
| **Байтовые** | InputStream, OutputStream | Работа с **байтами** (файлы, изображения, сокеты) |
| **Символьные** | Reader, Writer | Работа с **текстом / символами** (т.е. поддержка кодировок) |

**🔸 2. Байтовые потоки**

Используются для **не-текстовой информации**: файлов, изображений, аудио, видео и т.д.

**✅ Основные классы:**

| **Класс** | **Назначение** |
| --- | --- |
| InputStream | Базовый класс для чтения байтов |
| OutputStream | Базовый класс для записи байтов |
| FileInputStream / FileOutputStream | Работа с файлами |
| BufferedInputStream / BufferedOutputStream | Буферизация для повышения производительности |

**🔤 3. Символьные потоки**

Используются для **текста** (поддерживают кодировки, например UTF-8).

**✅ Основные классы:**

| **Класс** | **Назначение** |
| --- | --- |
| Reader | Абстрактный класс для чтения символов |
| Writer | Абстрактный класс для записи символов |
| FileReader / FileWriter | Чтение/запись текста из/в файл |
| BufferedReader / BufferedWriter | Буферизация для строк |
| PrintWriter | Удобный вывод текста с print() и println() |

**📌 Пример:**

import java.io.\*;

public class CharExample {

public static void main(String[] args) throws IOException {

BufferedReader reader = new BufferedReader(new FileReader("input.txt"));

BufferedWriter writer = new BufferedWriter(new FileWriter("output.txt"));

String line;

while ((line = reader.readLine()) != null) {

writer.write(line.toUpperCase());

writer.newLine();

}

reader.close();

writer.close();

}

}

**🔗 4. "Цепочки" потоков (Stream Chaining)**

Цепочка потоков — это **обёртывание одного потока другим** для расширения функционала.

Например:

* FileInputStream читает байты из файла
* BufferedInputStream добавляет буферизацию
* DataInputStream позволяет читать числа, строки и т.д.

**📌 Пример байтовой цепочки:**

DataInputStream dis = new DataInputStream(

new BufferedInputStream(

new FileInputStream("data.bin")

)

);

int value = dis.readInt();

**📌 Пример символьной цепочки:**

BufferedReader reader = new BufferedReader(

new InputStreamReader(System.in)

);

String line = reader.readLine();

🧠 Здесь InputStreamReader преобразует байты в символы, а BufferedReader добавляет буфер и метод readLine().

**🧠 5. Отличия: байтовые vs. символьные**

| **Характеристика** | **Байтовые (InputStream/OutputStream)** | **Символьные (Reader/Writer)** |
| --- | --- | --- |
| Тип данных | Байт (byte) | Символ (char) |
| Предназначение | Бинарные данные (файлы, изображения) | Текстовые данные (файлы, строки) |
| Поддержка кодировки | ❌ | ✅ |

**🚧 6. Исключения и ресурсы**

При работе с I/O возможны исключения (например, FileNotFoundException, IOException), поэтому:

**✅ Используем try-with-resources (Java 7+):**

try (BufferedReader reader = new BufferedReader(new FileReader("input.txt"))) {

String line = reader.readLine();

System.out.println(line);

}

Ресурсы закроются автоматически, даже при ошибках.

**💡 7. Когда и что использовать**

| **Задача** | **Класс** |
| --- | --- |
| Прочитать текст из файла | BufferedReader + FileReader |
| Прочитать бинарный файл | BufferedInputStream + FileInputStream |
| Записать строку в файл | BufferedWriter, PrintWriter |
| Записать числа в файл | DataOutputStream |
| Читать с консоли | BufferedReader + InputStreamReader(System.in) |

### Работа с файлами в Java. Класс java.io.File

**Класс java.io.File**

Класс File **не представляет файл как содержимое**, а как **абстракцию пути** к файлу или папке. Через него можно **проверять существование**, **получать свойства**, **создавать и удалять** файлы и директории.

📦 Находится в пакете: java.io

**🔧 2. Создание объекта File**

import java.io.File;

File file1 = new File("test.txt"); // относительный путь

File file2 = new File("C:/Users/User/test.txt"); // абсолютный путь

❗ Это **не создаёт файл на диске**, а лишь объект, ссылающийся на путь.

**🔍 3. Проверка существования и свойств**

if (file1.exists()) {

System.out.println("Файл существует");

if (file1.isFile()) {

System.out.println("Это обычный файл");

} else if (file1.isDirectory()) {

System.out.println("Это папка");

}

System.out.println("Имя файла: " + file1.getName());

System.out.println("Путь: " + file1.getPath());

System.out.println("Абсолютный путь: " + file1.getAbsolutePath());

System.out.println("Размер (байт): " + file1.length());

System.out.println("Можно читать? " + file1.canRead());

System.out.println("Можно писать? " + file1.canWrite());

}

**📝 4. Создание файла и папки**

**📄 Создание файла:**

import java.io.\*;

File file = new File("newfile.txt");

if (file.createNewFile()) {

System.out.println("Файл создан");

} else {

System.out.println("Файл уже существует");

}

Метод createNewFile() выбрасывает IOException, если не удалось создать файл.

**📁 Создание папки:**

File folder = new File("myFolder");

if (folder.mkdir()) {

System.out.println("Папка создана");

}

📌 Если нужно создать **вложенные директории**:

new File("a/b/c").mkdirs(); // создаёт всё дерево

**🗑️ 5. Удаление файла или папки**

File file = new File("newfile.txt");

if (file.delete()) {

System.out.println("Файл удалён");

}

Папка должна быть **пустой**, иначе delete() не сработает.

**📂 6. Получение списка файлов**

File folder = new File("myFolder");

File[] files = folder.listFiles();

if (files != null) {

for (File f : files) {

System.out.println(f.getName() + (f.isDirectory() ? " [папка]" : " [файл]"));

}

}

Можно фильтровать файлы:

File[] txtFiles = folder.listFiles((dir, name) -> name.endsWith(".txt"));

**📤 7. Перемещение и переименование**

File oldFile = new File("old.txt");

File newFile = new File("renamed.txt");

if (oldFile.renameTo(newFile)) {

System.out.println("Файл переименован/перемещён");

}

**💥 8. Пример: создание, запись, чтение, удаление**

import java.io.\*;

public class FileWorkflow {

public static void main(String[] args) throws IOException {

File file = new File("example.txt");

// Создание

if (file.createNewFile()) {

System.out.println("Файл создан.");

}

// Запись

FileWriter writer = new FileWriter(file);

writer.write("Привет, файл!");

writer.close();

// Чтение

BufferedReader reader = new BufferedReader(new FileReader(file));

String line = reader.readLine();

System.out.println("Содержимое: " + line);

reader.close();

// Удаление

if (file.delete()) {

System.out.println("Файл удалён.");

}

}

}

**⚠️ 9. Советы по безопасности**

| **Что делать** | **Почему** |
| --- | --- |
| Проверяй exists() | Иначе будут ошибки при доступе к несуществующим файлам |
| Используй try-with-resources | Автоматическое закрытие ресурсов |
| Проверяй canRead() / canWrite() | Некоторые файлы защищены |
| Не используй File для содержимого — только для работы с путями | Для чтения/записи используй FileReader, Scanner, Files, BufferedReader и т.д. |

### Пакет java.nio - назначение, основные классы и интерфейсы.

**Что такое java.nio?**

java.nio (New I/O) — это **альтернативный и улучшенный API для ввода-вывода**, появившийся в Java 1.4 и значительно расширенный в Java 7 (через java.nio.file).

java.nio — это **огромный набор API**, не только для файлов, но и для **высокопроизводительного I/O, работы с буферами, каналами, памятью, сокетами** и многого другого. Он разработан как альтернатива старому java.io, особенно в задачах, где важна **производительность, масштабируемость и работа с неблокирующими потоками**.

**🎯 Основные цели java.nio:**

* Работа с **буферами** и **каналами** (для производительного I/O)
* Работа с **файловой системой** и **путями** (в java.nio.file)
* Упрощённый и безопасный доступ к **файлам, директориям**
* Поддержка **символических ссылок**, **атрибутов**, **наблюдения за изменениями**
* Совместимость с современными ОС

**🗂️ Основные пакеты**

| **Подсистема** | **Назначение** |
| --- | --- |
| **java.nio.file** | Работа с путями, файлами, директориями |
| **java.nio.channels** | Каналы ввода-вывода (вместо потоков) |
| **java.nio.charset** | Кодировки символов |
| **java.nio.ByteBuffer и друзья** | Буферы для хранения данных |
| **java.nio.MappedByteBuffer** | Маппинг файлов в память (memory-mapped I/O) |
| **java.nio.file.attribute** | Доступ к POSIX и DOS-атрибутам файлов |

**📁 Основные классы и интерфейсы в java.nio.file**

**🔹 Path**

Объект, представляющий путь к файлу или папке (аналог File, но лучше).

Path path = Paths.get("example.txt");

* Поддерживает вложенные пути: Paths.get("dir", "file.txt")
* Может быть относительным или абсолютным
* Не создаёт файл — просто представляет путь

**🔹 Files**

Утилитный класс со **статическими методами** для работы с файлами и директориями.

**Примеры:**

Files.exists(path); // проверка существования

Files.createFile(path); // создать файл

Files.createDirectories(path); // создать директории

Files.delete(path); // удалить файл/директорию

Files.copy(source, target); // копировать

Files.move(source, target); // переместить

Files.readAllLines(path); // чтение строк

Files.write(path, lines); // запись строк

**🔹 Paths**

Фабричный класс: предоставляет метод get(...) для создания Path объектов.

Path path = Paths.get("data", "text.txt");

**🔹 DirectoryStream**

Позволяет безопасно перебирать содержимое директории.

try (DirectoryStream<Path> stream = Files.newDirectoryStream(Paths.get("folder"))) {

for (Path file : stream) {

System.out.println(file.getFileName());

}

}

**🔹 StandardOpenOption**

Перечисление для указания флагов открытия файла (APPEND, CREATE, WRITE, TRUNCATE\_EXISTING, и т.д.)

**📌 Пример: создание, запись и чтение файла**

import java.nio.file.\*;

import java.io.IOException;

import java.util.List;

public class NioExample {

public static void main(String[] args) throws IOException {

Path path = Paths.get("example.txt");

// Создание файла

if (!Files.exists(path)) {

Files.createFile(path);

}

// Запись

Files.write(path, List.of("Привет", "Мир!"));

// Чтение

List<String> lines = Files.readAllLines(path);

lines.forEach(System.out::println);

}

}

**🧠 Преимущества java.nio.file перед java.io.File**

| **Возможность** | **java.io.File** | **java.nio.file.Path + Files** |
| --- | --- | --- |
| Поддержка Unicode путей | Частично | ✅ Полная |
| Поддержка символьных ссылок | ❌ | ✅ |
| Современные API и безопасность | ❌ | ✅ |
| Простота чтения/записи строк | ❌ (нужно много кода) | ✅ Files.readAllLines() |
| Наблюдение за файлами (WatchService) | ❌ | ✅ |

**🔁 Пример: копирование и перемещение**

Path source = Paths.get("file1.txt");

Path target = Paths.get("backup", "file1\_backup.txt");

Files.copy(source, target, StandardCopyOption.REPLACE\_EXISTING);

Files.move(target, Paths.get("archive", "file1\_final.txt"));

**👀 Наблюдение за директорией: WatchService**

Позволяет отслеживать **изменения в директории** (создание, удаление, модификация файлов):

WatchService watchService = FileSystems.getDefault().newWatchService();

Path dir = Paths.get("watched\_dir");

dir.register(watchService, StandardWatchEventKinds.ENTRY\_CREATE);

while (true) {

WatchKey key = watchService.take();

for (WatchEvent<?> event : key.pollEvents()) {

System.out.println("Событие: " + event.kind() + " — " + event.context());

}

key.reset();

}

### Утилита javadoc. Особенности автоматического документирования кода в Java.

**javadoc** — это **утилита командной строки**, встроенная в JDK, которая **создаёт HTML-документацию** на основе специальных комментариев в исходном коде Java

**Зачем нужен javadoc?**

* 📖 Автоматически создаёт читаемую **документацию API**
* 🤝 Помогает другим понять твой код без чтения реализации
* 📦 Обязательно для библиотек, фреймворков, SDK
* ✅ Используется IDE (Eclipse, IntelliJ) для всплывающих подсказок

# Как писать javadoc-комментарии

Javadoc-комментарии начинаются с /\*\* и пишутся **перед классом, методом, полем и т.д.**

/\*\*

\* Утилитный класс для работы с числами.

\*/

public class MathUtils {

/\*\*

\* Считает сумму двух целых чисел.

\*

\* @param a первое число

\* @param b второе число

\* @return сумма чисел

\*/

public static int add(int a, int b) {

return a + b;

}

}

**Основные теги Javadoc**

| **Тег** | **Назначение** |
| --- | --- |
| @param | Описание параметра метода |
| @return | Что возвращает метод |
| @throws / @exception | Какие исключения может выбросить |
| @see | Ссылка на другой класс/метод |
| @since | С какой версии доступно |
| @author | Автор |
| @version | Версия |
| @deprecated | Метод устарел |
| {@link} | Встроенная ссылка внутри описания |
| {@code} | Форматированный фрагмент кода |

### Scanner

Scanner — это удобный класс из пакета java.util, предназначенный для **чтения и разбора текста (input)** из различных источников:

* с **консоли** (System.in)
* из **файлов**
* из **строк**
* из **сетевых потоков** (например, InputStream)

Scanner разбивает входной поток по **разделителям (по умолчанию — пробелы, табы, переносы строк)** и позволяет считывать данные как числа, строки, слова и т.д.

**✅ Пример 1: Чтение с консоли**

import java.util.Scanner;

public class ConsoleInputExample {

public static void main(String[] args) {

Scanner scanner = new Scanner(System.in);

System.out.print("Введите имя: ");

String name = scanner.nextLine();

System.out.print("Введите возраст: ");

int age = scanner.nextInt();

System.out.println("Привет, " + name + "! Тебе " + age + " лет.");

}

}

**🧠 Основные методы Scanner**

| **Метод** | **Описание** |
| --- | --- |
| next() | Считывает **одно слово** |
| nextLine() | Считывает **всю строку до \n** |
| nextInt() | Считывает **целое число** |
| nextDouble() | Считывает **число с плавающей точкой** |
| hasNext() | Есть ли ещё токены? |
| hasNextInt() | Есть ли следующее число? |

📌 **Важно!** После nextInt() или next() надо вызывать nextLine() чтобы "съесть" символ новой строки \n.

**📂 Пример 2: Чтение из файла**

import java.io.File;

import java.io.FileNotFoundException;

import java.util.Scanner;

public class FileScannerExample {

public static void main(String[] args) throws FileNotFoundException {

File file = new File("data.txt");

Scanner scanner = new Scanner(file);

while (scanner.hasNextLine()) {

String line = scanner.nextLine();

System.out.println("Прочитано: " + line);

}

scanner.close();

}

}

**🔧 Пример 3: Чтение чисел из строки**

String data = "100 200 300";

Scanner scanner = new Scanner(data);

while (scanner.hasNextInt()) {

int num = scanner.nextInt();

System.out.println("Число: " + num);

}

forEach(...) — метод из Iterable, пробегает по всем элементам.

collection::add — **ссылка на метод** (method reference), короткий способ записи:

for (Movie m : loaded) {

collection.add(m);

}

javadoc -d docs -classpath json-20250107.jar Main.java core\\*.java collection\\*.java commands\\*.java input\\*.java input\basic\\*.java model\\*.java storage\\*.java -encoding UTF-8 -charset UTF-8