Теория 6

## Сетевое взаимодействие - клиент-серверная архитектура, основные протоколы, их сходства и отличия.

Сетевое взаимодействие — ключевая часть разработки многих приложений на Java, особенно веб-сервисов, клиент-серверных систем, облачных решений и т.д. Давай подробно разберем, что такое клиент-серверная архитектура, какие протоколы чаще всего применяются, и в чем их различия и сходства — всё с акцентом на Java.

**📌 1. Клиент-серверная архитектура в Java**

**🔹 Суть архитектуры:**

Клиент-серверная модель предполагает, что одна часть приложения (клиент) делает запросы, а другая (сервер) отвечает на них. Они взаимодействуют через сеть по определённым протоколам.

* **Клиент**: инициирует запрос, например, веб-браузер или Java-программа.
* **Сервер**: принимает запросы, обрабатывает и отправляет ответы.

**🔹 Пример на Java:**

**Сервер (на TCP сокетах):**

import java.io.\*;

import java.net.\*;

public class SimpleServer {

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

ServerSocket serverSocket = new ServerSocket(5000);

System.out.println("Сервер запущен на порту 5000");

Socket clientSocket = serverSocket.accept(); // ожидание клиента

BufferedReader in = new BufferedReader(new InputStreamReader(clientSocket.getInputStream()));

PrintWriter out = new PrintWriter(clientSocket.getOutputStream(), true);

String inputLine;

while ((inputLine = in.readLine()) != null) {

System.out.println("Получено: " + inputLine);

out.println("Ответ: " + inputLine.toUpperCase());

}

in.close();

out.close();

clientSocket.close();

serverSocket.close();

}

}

**Клиент:**

import java.io.\*;

import java.net.\*;

public class SimpleClient {

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

Socket socket = new Socket("localhost", 5000);

BufferedReader userInput = new BufferedReader(new InputStreamReader(System.in));

BufferedReader in = new BufferedReader(new InputStreamReader(socket.getInputStream()));

PrintWriter out = new PrintWriter(socket.getOutputStream(), true);

String input;

while ((input = userInput.readLine()) != null) {

out.println(input);

System.out.println("Сервер ответил: " + in.readLine());

}

in.close();

out.close();

socket.close();

}

}

**📌 2. Основные сетевые протоколы**

Java предоставляет поддержку для многих протоколов, но самые часто используемые:

| **Протокол** | **Назначение** | **Протокол уровня** | **Особенности** |
| --- | --- | --- | --- |
| **TCP** | Надежная передача данных | Транспортный | Гарантирует порядок и доставку |
| **UDP** | Быстрая передача без подтверждений | Транспортный | Нет гарантии доставки |
| **HTTP/HTTPS** | Веб-запросы и ответы | Прикладной | Основан на TCP, широко используется |
| **FTP/FTPS** | Передача файлов | Прикладной | FTP – незащищённый, FTPS – с TLS |
| **WebSocket** | Двустороннее соединение | Прикладной | Работает поверх HTTP |

**📌 3. Сходства и различия между TCP и UDP**

| **Характеристика** | **TCP** | **UDP** |
| --- | --- | --- |
| Надежность | Да (проверка доставки, контроль потока) | Нет |
| Скорость | Медленнее из-за надёжности | Быстрее |
| Подключение | Устанавливает соединение (3-way handshake) | Соединения нет |
| Использование | HTTP, FTP, SMTP, SSH | VoIP, игры, потоковое видео |
| Поддержка в Java | Socket, ServerSocket | DatagramSocket, DatagramPacket |

## Протокол TCP. Классы Socket и ServerSocket.

**🔌 Что такое TCP?**

**TCP (Transmission Control Protocol)** — это надёжный протокол транспортного уровня, который:

* Обеспечивает **надежную** доставку данных.
* Гарантирует **сохранение порядка** пакетов.
* Поддерживает **двустороннее соединение** между клиентом и сервером.
* Управляет **потоком** и **контролем ошибок**.

В Java поддержка TCP реализована через классы из пакета java.net, в первую очередь:

* ServerSocket — для создания сервера.
* Socket — для создания клиента и обработки подключения на сервере.

**📚 Основные классы**

**1. ServerSocket (серверная часть)**

Этот класс прослушивает определённый порт и принимает входящие TCP-соединения.

**Основные методы:**

| **Метод** | **Назначение** |
| --- | --- |
| accept() | Ожидает входящее соединение и возвращает Socket |
| close() | Закрывает сокет |
| getLocalPort() | Возвращает номер порта |

**2. Socket (клиентская часть и соединение на сервере)**

Этот класс представляет собой клиентское соединение или соединение, установленное через accept().

**Основные методы:**

| **Метод** | **Назначение** |
| --- | --- |
| getInputStream() | Получение входящего потока |
| getOutputStream() | Получение исходящего потока |
| close() | Закрытие соединения |

**📌 Принцип работы:**

1. Сервер запускается и слушает порт (например, 1234).
2. Клиент подключается по IP (или localhost) и порту.
3. Стороны обмениваются данными через потоки:
   * InputStream / BufferedReader — для чтения.
   * OutputStream / PrintWriter — для записи.
4. TCP гарантирует, что данные дойдут в правильном порядке и без потерь.

**🧠 Особенности использования в Java:**

* Потоки (InputStream, OutputStream) блокируют выполнение до получения/отправки данных.
* Следует **закрывать соединения** и потоки после использования (лучше через try-with-resources).
* Не стоит использовать readLine() без проверки на null — это может вызвать NullPointerException, если соединение закрыто.
* TCP хорошо подходит для:
  + чат-приложений,
  + обмена файлами,
  + любых задач, требующих надежной доставки данных.

**🧩 Альтернатива: Java NIO**

Для высоконагруженных систем можно использовать **Java NIO** (java.nio.channels.SocketChannel, ServerSocketChannel) с неблокирующими сокетами и селекторами, но это более сложный подход.

## Протокол UDP. Классы DatagramSocket и DatagramPacket.

**📡 Что такое UDP?**

**UDP (User Datagram Protocol)** — это *ненадёжный*, *без соединения* протокол транспортного уровня:

**✨ Особенности UDP:**

* **Быстрый**, так как не устанавливает соединение.
* **Не гарантирует доставку** (нет подтверждений, контроля потока и порядка).
* Используется там, где **важна скорость**, а **потери не критичны**:
  + Онлайн-игры
  + Видеотрансляции
  + Голосовые вызовы (VoIP)
  + Сенсоры/телеметрия

**📚 Основные классы Java для UDP**

**1. DatagramSocket**

Представляет конечную точку обмена UDP-пакетами (для клиента или сервера).

| **Метод** | **Назначение** |
| --- | --- |
| send(DatagramPacket) | Отправка пакета |
| receive(DatagramPacket) | Получение пакета (блокирующий вызов) |
| close() | Закрытие сокета |

**2. DatagramPacket**

Представляет *один UDP-пакет* — и для отправки, и для приёма.

| **Конструктор** | **Назначение** |
| --- | --- |
| DatagramPacket(byte[] buf, int length) | Прием |
| DatagramPacket(byte[] buf, int length, InetAddress addr, int port) | Отправка |

**🔧 Пример: UDP клиент-сервер**

**✅ Сервер (слушает порт и печатает полученные сообщения)**

import java.net.\*;

public class UdpServer {

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

DatagramSocket socket = new DatagramSocket(9876); // порт сервера

byte[] buffer = new byte[1024];

System.out.println("Сервер запущен. Ожидание данных...");

while (true) {

DatagramPacket packet = new DatagramPacket(buffer, buffer.length);

socket.receive(packet); // блокирующий вызов

String received = new String(packet.getData(), 0, packet.getLength());

System.out.println("Получено от клиента: " + received);

}

}

}

**✅ Клиент (отправляет сообщение на сервер)**

import java.net.\*;

public class UdpClient {

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

DatagramSocket socket = new DatagramSocket();

String message = "Привет, сервер!";

byte[] data = message.getBytes();

InetAddress address = InetAddress.getByName("localhost"); // IP сервера

int port = 9876;

DatagramPacket packet = new DatagramPacket(data, data.length, address, port);

socket.send(packet);

System.out.println("Сообщение отправлено.");

socket.close();

}

}

**⚙️ Как это работает:**

1. **Сервер** создаёт DatagramSocket и слушает порт (например, 9876).
2. **Клиент** создает DatagramSocket, формирует DatagramPacket с данными и отправляет его на IP и порт сервера.
3. **Сервер** принимает пакет, извлекает данные и может их обработать или ответить.
4. Нет гарантии доставки — нужно реализовывать retry/подтверждения вручную, если важно.

**⚠️ Особенности и ограничения UDP:**

| **Особенность** | **Объяснение** |
| --- | --- |
| Максимальный размер пакета | Обычно до ~64Кб, но безопаснее использовать <= 512 байт |
| Потери пакетов | Возможны при перегрузке сети |
| Нет гарантии порядка | Пакеты могут прийти не по порядку |
| Блокирующий вызов receive() | Можно настроить таймаут через setSoTimeout(ms) |

**✅ Когда использовать UDP в Java?**

* Нужно **отправлять быстрые и маленькие пакеты**, без подтверждений.
* Протокол должен быть **лёгким и без перегрузки**.
* Пример: игра, пинг-утилита, датчики IoT, трансляция данных.

**🔄 UDP vs TCP — краткое сравнение:**

| **Характеристика** | **TCP** | **UDP** |
| --- | --- | --- |
| Установление соединения | Да | Нет |
| Гарантия доставки | Да | Нет |
| Порядок доставки | Гарантирован | Нет |
| Скорость | Ниже | Выше |
| Использование в Java | Socket, ServerSocket | DatagramSocket, DatagramPacket |

**📌 Заключение:**

Классы DatagramSocket и DatagramPacket позволяют реализовать обмен данными по UDP в Java — легко и быстро, но с пониманием ограничений. Если нужна скорость и допускается потеря данных — UDP подходит идеально.

## Отличия блокирующего и неблокирующего ввода-вывода, их преимущества и недостатки. Работа с сетевыми каналами.

**📌 Блокирующий vs Неблокирующий ввод-вывод (I/O) в Java**

**🔹 Блокирующий ввод-вывод (Blocking I/O)**

Это классический подход, реализуемый через Socket, ServerSocket, InputStream, OutputStream.

**✅ Как работает:**

* Поток **ожидает завершения операции**: read() или accept() останавливает поток до получения данных.
* **Каждое соединение — отдельный поток.**

**📈 Пример:**

Socket socket = serverSocket.accept(); // блокирует до подключения клиента

InputStream in = socket.getInputStream();

int data = in.read(); // блокирует до получения байта

**✅ Преимущества:**

* Прост в реализации и отладке.
* Идеален для небольшого количества клиентов.

**❌ Недостатки:**

* Неэффективен при большом числе клиентов (каждому нужен поток).
* Потоки — дорогой ресурс.
* Трудно масштабировать.

**🔹 Неблокирующий ввод-вывод (Non-blocking I/O)**

Реализован в **Java NIO** (java.nio.channels). Вместо потоков — используются **каналы (Channels)** и **селекторы (Selector)**.

**✅ Как работает:**

* Операции read(), accept(), write() **не блокируют поток**.
* Один поток может **обслуживать множество соединений**.
* Канал регистрируется у **селектора**, который уведомляет, когда канал готов к операциям.

**📈 Пример:**

ServerSocketChannel serverChannel = ServerSocketChannel.open();

serverChannel.bind(new InetSocketAddress(1234));

serverChannel.configureBlocking(false); // неблокирующий режим

Selector selector = Selector.open();

serverChannel.register(selector, SelectionKey.OP\_ACCEPT);

while (true) {

selector.select(); // блокирует до готовности канала

Set<SelectionKey> keys = selector.selectedKeys();

for (SelectionKey key : keys) {

if (key.isAcceptable()) {

SocketChannel client = serverChannel.accept();

client.configureBlocking(false);

client.register(selector, SelectionKey.OP\_READ);

} else if (key.isReadable()) {

SocketChannel client = (SocketChannel) key.channel();

ByteBuffer buffer = ByteBuffer.allocate(256);

client.read(buffer);

System.out.println("Принято: " + new String(buffer.array()).trim());

}

}

keys.clear();

}

**📦 Сетевые каналы в Java (Java NIO)**

| **Класс** | **Описание** |
| --- | --- |
| ServerSocketChannel | Аналог ServerSocket, неблокирующий |
| SocketChannel | Аналог Socket, для клиента |
| DatagramChannel | Аналог DatagramSocket, для UDP |
| Selector | Мультиплексор событий (accept/read/write) |
| SelectionKey | Представляет регистрацию канала у селектора |

**⚖️ Сравнение: Blocking vs Non-Blocking I/O**

| **Критерий** | **Blocking I/O** | **Non-blocking I/O** |
| --- | --- | --- |
| Кол-во потоков | Один поток на соединение | Один поток на все |
| Производительность | Легко заблокировать CPU | Высокая масштабируемость |
| Сложность | Просто | Сложно |
| Подходит для | Простых серверов | Многоклиентских систем |
| Используемые API | Socket, ServerSocket | SocketChannel, Selector |

**✅ Где что использовать?**

* **Blocking I/O**:
  + Простые CLI-клиенты и чаты.
  + Приложения с небольшим числом пользователей.
* **Non-blocking I/O (NIO)**:
  + Высоконагруженные веб-серверы.
  + Мессенджеры, игры, брокеры сообщений.
  + Реализация собственного HTTP-сервера.

**💡 Вывод**

| Хочешь простоту? —> | Используй Socket/ServerSocket (blocking). |  
| Хочешь масштабируемость? —> | Используй NIO: Selector + Channel. |

## Классы SocketChannel и DatagramChannel.

Эти два класса являются частью **Java NIO (New I/O)** и предназначены для работы с сетевыми соединениями на более **низком и масштабируемом уровне**, чем традиционные Socket и DatagramSocket.

Разберём каждый класс подробно — с пояснениями, примерами и сравнением.

**📦 SocketChannel — TCP (аналог Socket)**

**🔹 Назначение:**

SocketChannel — это **неблокирующий канал**, основанный на TCP. Он может быть использован:

* как клиентом (инициирует соединение),
* так и сервером (получается через ServerSocketChannel.accept()).

**🔹 Особенности:**

* Может работать в **блокирующем и неблокирующем** режиме.
* Поддерживает чтение/запись через ByteBuffer.
* Совместим с Selector — ключевой элемент масштабируемых серверов.

**✅ Пример 1: TCP-клиент на SocketChannel**

import java.net.InetSocketAddress;

import java.nio.ByteBuffer;

import java.nio.channels.SocketChannel;

public class TcpNioClient {

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

SocketChannel client = SocketChannel.open(new InetSocketAddress("localhost", 1234));

ByteBuffer buffer = ByteBuffer.wrap("Привет, сервер!".getBytes());

client.write(buffer);

buffer.clear();

client.read(buffer);

System.out.println("Ответ: " + new String(buffer.array()).trim());

client.close();

}

}

**✅ Пример 2: TCP-сервер (упрощённо)**

import java.net.InetSocketAddress;

import java.nio.ByteBuffer;

import java.nio.channels.\*;

public class TcpNioServer {

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

ServerSocketChannel server = ServerSocketChannel.open();

server.bind(new InetSocketAddress(1234));

server.configureBlocking(false);

Selector selector = Selector.open();

server.register(selector, SelectionKey.OP\_ACCEPT);

while (true) {

selector.select();

for (SelectionKey key : selector.selectedKeys()) {

if (key.isAcceptable()) {

SocketChannel client = server.accept();

client.configureBlocking(false);

client.register(selector, SelectionKey.OP\_READ);

} else if (key.isReadable()) {

SocketChannel client = (SocketChannel) key.channel();

ByteBuffer buffer = ByteBuffer.allocate(256);

client.read(buffer);

buffer.flip();

client.write(buffer);

}

}

selector.selectedKeys().clear();

}

}

}

**📦 DatagramChannel — UDP (аналог DatagramSocket)**

**🔹 Назначение:**

DatagramChannel — это **канал для UDP**. Работает с датаграммами, не устанавливая соединения.

**🔹 Особенности:**

* Поддерживает **отправку и приём пакетов** по UDP.
* Может быть **неблокирующим**.
* Может регистрироваться у Selector (как и TCP-каналы).
* Использует ByteBuffer для передачи данных.

**✅ Пример 3: UDP-сервер на DatagramChannel**

import java.net.\*;

import java.nio.\*;

import java.nio.channels.\*;

public class UdpNioServer {

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

DatagramChannel channel = DatagramChannel.open();

channel.bind(new InetSocketAddress(9876));

ByteBuffer buffer = ByteBuffer.allocate(1024);

while (true) {

buffer.clear();

SocketAddress clientAddress = channel.receive(buffer); // блокирующая операция

buffer.flip();

System.out.println("Получено: " + new String(buffer.array(), 0, buffer.limit()));

}

}

}

**✅ Пример 4: UDP-клиент на DatagramChannel**

import java.net.\*;

import java.nio.\*;

import java.nio.channels.\*;

public class UdpNioClient {

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

DatagramChannel channel = DatagramChannel.open();

ByteBuffer buffer = ByteBuffer.wrap("Привет по UDP".getBytes());

InetSocketAddress address = new InetSocketAddress("localhost", 9876);

channel.send(buffer, address);

channel.close();

}

}

**🆚 Сравнение: SocketChannel vs DatagramChannel**

| **Свойство** | **SocketChannel** | **DatagramChannel** |
| --- | --- | --- |
| Протокол | TCP | UDP |
| Связь | Устанавливает соединение | Без соединения |
| Гарантия доставки | Есть | Нет |
| Использование | Надежные соединения (чаты, API) | Быстрые пакеты (игры, телеметрия) |
| Метод передачи | read(), write() | receive(), send() |
| Работает с Selector | ✅ | ✅ |
| Размер данных | Поток данных | Ограниченные пакеты |

**🔧 Поддержка селектора:**

Оба класса могут быть использованы с Selector:

channel.configureBlocking(false);

channel.register(selector, SelectionKey.OP\_READ);

Это позволяет **одним потоком обрабатывать сотни/тысячи соединений**, что критично для масштабируемых серверов.

**✅ Вывод:**

| Хочешь стабильное TCP-соединение? | → Используй SocketChannel. |  
| Нужен лёгкий, быстрый UDP-обмен? | → Используй DatagramChannel. |

Оба — часть **Java NIO**, мощного инструмента для высоконагруженных и неблокирующих приложений.

## Передача данных по сети. Сериализация объектов.

Тема **передачи данных по сети и сериализации объектов** — ключевая для разработки сетевых приложений на Java, особенно если нужно обмениваться не просто текстом, а **структурированными данными** (например, объектами класса User, Message, Product, и т.д.).

Разберёмся по шагам:

**📦 Что такое сериализация?**

**Сериализация** — это процесс преобразования объекта в **последовательность байтов**, чтобы его можно было:

* передать по сети,
* сохранить в файл,
* или записать в базу данных.

**Обратный процесс** — это **десериализация**.

**🔹 Интерфейс Serializable**

Чтобы объект можно было сериализовать, его класс должен **реализовать интерфейс**:

public class MyClass implements Serializable { ... }

🔸 Интерфейс Serializable — **маркерный** (не содержит методов).

🔸 Поля, которые не нужно сериализовать, помечаются transient.

**🔧 Передача объекта по сети через TCP (сериализация)**

Java позволяет **передавать объекты по TCP**, используя потоки:

* ObjectOutputStream — сериализация и отправка объекта.
* ObjectInputStream — получение и десериализация.

**✅ Пример: передача объекта User от клиента серверу**

**1. 📘 Класс User**

import java.io.Serializable;

public class User implements Serializable {

private String name;

private int age;

public User(String name, int age) {

this.name = name;

this.age = age;

}

public String toString() {

return "User{name='" + name + "', age=" + age + "}";

}

}

**2. 🖥️ Сервер: получает объект по сети**

import java.io.\*;

import java.net.\*;

public class ObjectServer {

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

ServerSocket serverSocket = new ServerSocket(5000);

System.out.println("Ожидаем клиента...");

Socket socket = serverSocket.accept();

ObjectInputStream ois = new ObjectInputStream(socket.getInputStream());

User user = (User) ois.readObject();

System.out.println("Получен объект: " + user);

ois.close();

socket.close();

serverSocket.close();

}

}

**3. 💻 Клиент: отправляет объект**

import java.io.\*;

import java.net.\*;

public class ObjectClient {

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

Socket socket = new Socket("localhost", 5000);

ObjectOutputStream oos = new ObjectOutputStream(socket.getOutputStream());

User user = new User("Андрей", 25);

oos.writeObject(user);

System.out.println("Объект отправлен!");

oos.close();

socket.close();

}

}

**🛠 Что важно помнить:**

| **Особенность** | **Описание** |
| --- | --- |
| Serializable обязателен | Без него ObjectOutputStream выдаст NotSerializableException |
| Поля transient | Не сериализуются |
| Версия класса (serialVersionUID) | Указывается для стабильности при изменении класса |
| Безопасность | Сериализация может быть уязвимой для атак — в боевых системах лучше использовать JSON, protobuf и т.д. |

**📦 Альтернатива: JSON сериализация**

Во многих современных Java-приложениях вместо стандартной сериализации используют **JSON**, например:

import com.fasterxml.jackson.databind.ObjectMapper;

ObjectMapper mapper = new ObjectMapper();

String json = mapper.writeValueAsString(user); // сериализация

User u = mapper.readValue(json, User.class); // десериализация

➡ Это удобно для **веб-сервисов, REST API**, и **взаимодействия с другими языками (например, JS/Python)**.

**✅ Когда использовать что:**

| **Подход** | **Применение** |
| --- | --- |
| Serializable | Быстрая передача между Java-приложениями |
| JSON / XML | Веб-приложения, межъязыковое взаимодействие |
| Protocol Buffers, Avro, Kryo | Высокопроизводительные системы, микросервисы, big data |

**📌 Вывод:**

* **Сериализация** — мощный механизм Java, который позволяет передавать объекты по сети.
* Классы ObjectInputStream / ObjectOutputStream делают это просто.
* Однако в реальной практике часто переходят на **форматы, совместимые с другими системами** (JSON, protobuf и др.).

## Интерфейс Serializable. Объектный граф, сериализация и десериализация полей и методов.

**📘 1. Интерфейс Serializable**

Serializable — **маркерный интерфейс** из пакета java.io:

public interface Serializable { }

Он **не содержит методов**, а его наличие говорит JVM: **объекты этого класса можно сериализовать** с помощью ObjectOutputStream.

📌 Без реализации Serializable вы получите java.io.NotSerializableException.

**📦 2. Сериализация и десериализация**

**✅ Сериализация:**

Преобразование объекта в **байтовую последовательность**, которая может быть:

* сохранена в файл,
* передана по сети,
* закэширована.

**✅ Десериализация:**

Восстановление объекта из байтовой последовательности.

Примеры:

// Сериализация

ObjectOutputStream oos = new ObjectOutputStream(new FileOutputStream("data.ser"));

oos.writeObject(myObject);

oos.close();

// Десериализация

ObjectInputStream ois = new ObjectInputStream(new FileInputStream("data.ser"));

MyClass obj = (MyClass) ois.readObject();

ois.close();

**🧩 3. Объектный граф**

**🔹 Что это?**

**Объектный граф** — это все объекты, на которые ссылается сериализуемый объект.

➡ Когда вы сериализуете объект A, Java **автоматически сериализует все вложенные объекты**, если они тоже реализуют Serializable.

**📌 Пример:**

class Address implements Serializable {

String city;

}

class Person implements Serializable {

String name;

Address address; // часть графа

}

Если сериализуется Person, то сериализуется и Address.

**⚠️ 4. Особенности сериализации**

**🔹 transient поля**

Если не хотите сериализовать поле:

transient int secret;

— при десериализации оно будет иметь **значение по умолчанию** (0, null, false и т.п.).

**🔹 Статические поля**

**Не сериализуются**, потому что не являются частью конкретного объекта, а принадлежат классу.

**🔹 Методы не сериализуются**

Сериализация затрагивает **только поля**. Метод toString(), equals(), и т.д. не сериализуются — они сохраняются в байт-коде класса, а не в данных объекта.

**💡 5. Ключевая переменная serialVersionUID**

Эта переменная используется для контроля совместимости при десериализации.

private static final long serialVersionUID = 1L;

Если вы **измените класс**, не указав serialVersionUID, и попытаетесь десериализовать старый объект — получите InvalidClassException.

**🧠 6. Кастомизация сериализации (методы writeObject, readObject)**

Вы можете сами управлять сериализацией:

private void writeObject(ObjectOutputStream out) throws IOException {

out.defaultWriteObject(); // сериализует стандартные поля

out.writeInt(secret); // сериализуем вручную

}

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

in.defaultReadObject(); // десериализует стандартные поля

secret = in.readInt(); // читаем вручную

}

**🧪 Пример: сериализация с объектным графом**

import java.io.\*;

class Engine implements Serializable {

String type = "V8";

}

class Car implements Serializable {

String model = "BMW";

Engine engine = new Engine(); // часть графа

}

public class TestSerialization {

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

Car car = new Car();

// Сериализация

ObjectOutputStream oos = new ObjectOutputStream(new FileOutputStream("car.ser"));

oos.writeObject(car);

oos.close();

// Десериализация

ObjectInputStream ois = new ObjectInputStream(new FileInputStream("car.ser"));

Car car2 = (Car) ois.readObject();

ois.close();

System.out.println("Восстановлен: " + car2.model + ", " + car2.engine.type);

}

}

**🔒 7. Безопасность сериализации**

**ОСТОРОЖНО**: десериализация может быть уязвима к атаке, если вы читаете данные от ненадежного источника. Возможна **атака с внедрением вредоносных классов**.

🔐 Рекомендовано: использовать **JSON, protobuf, XML**, или библиотеки вроде **Kryo**, если нужна безопасная и быстрая сериализация.

**✅ Резюме**

| **Пункт** | **Кратко** |
| --- | --- |
| Serializable | Обязателен для сериализации |
| transient | Исключает поле |
| Статические поля | Не сериализуются |
| Методы | Не сериализуются |
| Объектный граф | Сериализуется весь, если все объекты реализуют Serializable |
| serialVersionUID | Используется для совместимости |
| Кастомизация | writeObject и readObject позволяют вручную контролировать сериализацию |

Если вы пытаетесь сериализовать объект, **у которого есть поле-ссылка на не сериализуемый объект**, то при вызове ObjectOutputStream.writeObject() произойдёт:

java.io.NotSerializableException: имя\_класса

## Java Stream API. Создание конвейеров. Промежуточные и терминальные операции.

**Java Stream API** — это мощный инструмент для обработки коллекций данных в декларативном, функциональном стиле. Он позволяет строить **конвейеры операций**, что делает код **чище, короче и более выразительным**.

Разберём ключевые моменты:

**🔹 Что такое Stream API?**

Stream — это последовательность данных (элементов), **которая может быть обработана поэтапно (поточно)**. Поток не хранит данные — он **перебирает источник** (список, массив, файл и т.д.).

List<String> names = List.of("Аня", "Оля", "Коля");

names.stream()

.filter(name -> name.startsWith("А"))

.map(String::toUpperCase)

.forEach(System.out::println);

**🔧 Создание Stream'ов**

| **Источник данных** | **Пример** |
| --- | --- |
| Коллекция | list.stream() |
| Массив | Arrays.stream(array) |
| Диапазон чисел | IntStream.range(0, 10) |
| Генератор | Stream.generate(() -> Math.random()) |
| Файл | Files.lines(Path) (может бросать исключение) |

**🛠️ Промежуточные операции (intermediate)**

**Они возвращают новый Stream** и **не выполняются сразу**, а откладываются до терминальной операции (ленивые).

**Примеры:**

| **Метод** | **Описание** |
| --- | --- |
| filter(Predicate) | фильтрация |
| map(Function) | преобразование элементов |
| distinct() | удаление дубликатов |
| sorted() | сортировка |
| limit(n) | ограничение количества |
| skip(n) | пропустить первые n |

📌 Все промежуточные операции **не изменяют исходную коллекцию**, а создают **новый поток**.

**✅ Пример: промежуточные операции**

List<String> words = List.of("java", "stream", "api", "java");

List<String> result = words.stream()

.filter(w -> w.length() > 3)

.map(String::toUpperCase)

.distinct()

.sorted()

.collect(Collectors.toList());

System.out.println(result); // [JAVA, STREAM]

**🧾 Терминальные операции (terminal)**

**Запускают выполнение стрима** и завершают конвейер. После терминальной операции поток **больше не доступен**.

**Примеры:**

| **Метод** | **Описание** |
| --- | --- |
| forEach(Consumer) | выполнить действие для каждого |
| collect(...) | собрать результат (в список, set, строку и т.д.) |
| count() | количество элементов |
| anyMatch(Predicate) | хотя бы один удовлетворяет |
| allMatch(Predicate) | все удовлетворяют |
| noneMatch(Predicate) | никто не удовлетворяет |
| findFirst() / findAny() | найти первый / любой элемент |
| reduce(...) | свёртка значений в одно (например, сумма) |

**✅ Пример: терминальные операции**

List<Integer> numbers = List.of(1, 2, 3, 4, 5);

int sum = numbers.stream()

.filter(n -> n % 2 == 0)

.map(n -> n \* n)

.reduce(0, Integer::sum);

System.out.println(sum); // (2\*2 + 4\*4) = 4 + 16 = 20

**🔄 Конвейер: Пример полного потока**

List<String> names = List.of("Анна", "Борис", "Алекс", "Виктор", "Артем");

List<String> filtered = names.stream() // создать поток

.filter(n -> n.startsWith("А")) // промежуточно: только на "А"

.map(String::toUpperCase) // промежуточно: в верхний регистр

.sorted() // промежуточно: сортировка

.collect(Collectors.toList()); // терминально: собрать в List

System.out.println(filtered); // [АЛЕКС, АННА, АРТЕМ]

**💡 Особенности:**

* Потоки **одноразовые** — нельзя использовать повторно.
* Потоки **ленивые** — пока нет терминальной операции, промежуточные не выполняются.
* Потоки можно **параллелить**: stream().parallel() — эффективно, но с осторожностью.

**✅ Когда использовать Stream API**

| **Подходит** | **Не подходит** |
| --- | --- |
| Преобразование коллекций | Сложная логика с побочными эффектами |
| Фильтрация, сортировка, группировка | Модификация исходных данных |
| Чтение файлов, агрегация данных | Алгоритмы, требующие сложной структуры (например, графы) |

**📚 Бонус: collect() и Collectors**

collect() — самая мощная терминальная операция.

| **Collector** | **Описание** |
| --- | --- |
| toList() | собрать в List |
| toSet() | в Set |
| joining() | склеить строки |
| groupingBy() | сгруппировать по какому-то признаку |
| partitioningBy() | разделить на 2 группы по условию |

**Пример:**

List<String> names = List.of("Аня", "Оля", "Олег", "Алексей");

Map<Boolean, List<String>> partitioned = names.stream()

.collect(Collectors.partitioningBy(n -> n.startsWith("О")));

**✅ Вывод**

| **Категория** | **Суть** |
| --- | --- |
| **Stream** | Поток элементов |
| **Промежуточные операции** | filter, map, sorted — создают новый Stream |
| **Терминальные операции** | forEach, collect, reduce — завершают работу |
| **Ленивость** | Промежуточные операции не выполняются до терминальной |
| **Конвейер** | Stream API строится в виде цепочки (pipeline) |

## Шаблоны проектирования: Decorator, Iterator, Factory method, Command, Flyweight, Interpreter, Singleton, Strategy, Adapter, Facade, Proxy.

Шаблоны проектирования (Design Patterns) — это проверенные решения типовых задач объектно-ориентированного проектирования. В Java они особенно важны, так как язык активно их поддерживает и поощряет.

Разберём **12 классических шаблонов** (в духе “Gang of Four”) — с кратким объяснением и примерами на Java.

**✅ 1. Decorator — *добавление поведения без изменения класса***

Позволяет оборачивать объект другими объектами, добавляя поведение **в рантайме**, не меняя исходный код.

**🔧 Пример:**

interface Coffee {

String getDescription();

}

class BasicCoffee implements Coffee {

public String getDescription() {

return "Обычный кофе";

}

}

class MilkDecorator implements Coffee {

private Coffee coffee;

public MilkDecorator(Coffee coffee) {

this.coffee = coffee;

}

public String getDescription() {

return coffee.getDescription() + ", с молоком";

}

}

**✅ 2. Iterator — *поэтапный доступ к элементам коллекции***

Позволяет перебирать элементы коллекции без раскрытия её внутренней структуры.

**🔧 Пример:**

List<String> names = List.of("Анна", "Борис");

Iterator<String> it = names.iterator();

while (it.hasNext()) {

System.out.println(it.next());

}

**✅ 3. Factory Method — *делегирует создание объектов подклассам***

Определяет интерфейс создания объекта, но оставляет подклассам решение о том, **какой класс инстанцировать**.

**🔧 Пример:**

abstract class Dialog {

public void renderWindow() {

Button okButton = createButton();

okButton.render();

}

abstract Button createButton();

}

class WindowsDialog extends Dialog {

public Button createButton() {

return new WindowsButton();

}

}

**✅ 4. Command — *запаковывает запрос как объект***

Полезен для **отмены**, **повтора**, **очереди команд** и **логгирования**.

**🔧 Пример:**

interface Command {

void execute();

}

class LightOnCommand implements Command {

public void execute() {

System.out.println("Свет включен");

}

}

class RemoteControl {

private Command command;

public void setCommand(Command command) {

this.command = command;

}

public void pressButton() {

command.execute();

}

}

**✅ 5. Flyweight — *разделяет общие данные между множеством объектов***

Используется для **экономии памяти**, когда есть много объектов с одинаковыми данными.

**🔧 Пример:**

class TreeType {

String name;

public TreeType(String name) {

this.name = name;

}

}

class TreeFactory {

static Map<String, TreeType> types = new HashMap<>();

public static TreeType getTreeType(String name) {

return types.computeIfAbsent(name, TreeType::new);

}

}

**✅ 6. Interpreter — *создание языка и интерпретация выражений***

Определяет грамматику и интерпретирует предложения в этом языке.

**🔧 Пример:**

interface Expression {

boolean interpret(String context);

}

class TerminalExpression implements Expression {

private String word;

public TerminalExpression(String word) { this.word = word; }

public boolean interpret(String context) {

return context.contains(word);

}

}

**✅ 7. Singleton — *один экземпляр класса на всё приложение***

Контролирует создание объекта, гарантируя, что будет создан только **один экземпляр**.

**🔧 Пример:**

class Singleton {

private static final Singleton instance = new Singleton();

private Singleton() {}

public static Singleton getInstance() {

return instance;

}

}

**✅ 8. Strategy — *выбор поведения во время выполнения***

Инкапсулирует поведение и позволяет **подменять алгоритмы на лету**.

**🔧 Пример:**

interface SortStrategy {

void sort(List<Integer> list);

}

class QuickSort implements SortStrategy {

public void sort(List<Integer> list) {

Collections.sort(list);

System.out.println("QuickSort");

}

}

class Sorter {

private SortStrategy strategy;

public Sorter(SortStrategy strategy) {

this.strategy = strategy;

}

public void sort(List<Integer> list) {

strategy.sort(list);

}

}

**✅ 9. Adapter — *приведение несовместимых интерфейсов к совместимым***

Позволяет объектам с разными интерфейсами **взаимодействовать**.

**🔧 Пример:**

interface Target {

void request();

}

class Adaptee {

public void specificRequest() {

System.out.println("Специфический запрос");

}

}

class Adapter implements Target {

private Adaptee adaptee;

public Adapter(Adaptee adaptee) {

this.adaptee = adaptee;

}

public void request() {

adaptee.specificRequest();

}

}

**✅ 10. Facade — *упрощённый интерфейс к подсистеме***

Скрывает сложность системы за **простым API**.

**🔧 Пример:**

class CPU { void start() {} }

class Memory { void load() {} }

class Computer {

private CPU cpu = new CPU();

private Memory memory = new Memory();

public void start() {

cpu.start();

memory.load();

}

}

**✅ 11. Proxy — *заместитель объекта для контроля доступа***

Используется для **логирования, кеширования, безопасности**, ленивой загрузки.

**🔧 Пример:**

interface Image {

void display();

}

class RealImage implements Image {

public void display() {

System.out.println("Показ изображения");

}

}

class ProxyImage implements Image {

private RealImage realImage;

public void display() {

if (realImage == null) realImage = new RealImage();

realImage.display();

}

}

**✅ 12. Chain of Responsibility (бонус)**

Позволяет передавать запрос по цепочке обработчиков.

abstract class Handler {

protected Handler next;

public void setNext(Handler next) { this.next = next; }

public abstract void handle(String request);

}

class AuthHandler extends Handler {

public void handle(String request) {

if (request.equals("auth")) System.out.println("Авторизация");

else if (next != null) next.handle(request);

}

}

**🧩 Краткая таблица по категориям**

| **Категория** | **Шаблоны** |
| --- | --- |
| **Порождающие** | Singleton, Factory Method |
| **Структурные** | Decorator, Adapter, Facade, Proxy, Flyweight |
| **Поведенческие** | Strategy, Command, Iterator, Interpreter |

Это **уникальный идентификатор версии класса**:

private static final long serialVersionUID = 1L;

Он нужен, чтобы Java могла проверить, **что сериализованный объект соответствует текущей версии класса** при десериализации.

@Serial

Это **аннотация**, которая говорит компилятору:

"Этот элемент (поле, метод) — часть механизма сериализации"

**new Thread(server::run).start();**

**Что она делает:**

1. **Создаёт новый поток (Thread)**
2. Передаёт в него **ссылку на метод run() объекта server**
3. **Запускает** поток с помощью .start()

Runtime.*getRuntime*().addShutdownHook(new Thread(() -> {  
 try {  
 fileManager.saveToFile(fileName, collection.getMovies());  
 System.*out*.println("\n[Shutdown] Коллекция сохранена перед выходом.");  
 } catch (IOException e) {  
 System.*err*.println("[Shutdown] Ошибка при сохранении: " + e.getMessage());  
 }  
}));

Он **регистрирует специальный "крючок завершения" (shutdown hook)** — код, который **выполнится, когда программа завершается**.

**Runtime.getRuntime()**

* Возвращает **объект текущей виртуальной машины Java (JVM)**.
* Через него ты можешь управлять завершением, выделением памяти и т.д.

**🔹 .addShutdownHook(...)**

* Метод, который **регистрирует "хук"** — код, который сработает при завершении JVM:
  + по Ctrl+C,
  + по System.exit(0),
  + при завершении процесса ОС.

📌 Аргументом передаётся **объект класса Thread**.

**🔹 new Thread(() -> { ... })**

* Создаётся новый поток.
* В него передаётся **лямбда** () -> { ... } — тело выполняемой задачи.
* Это **что будет сделано перед выходом**.

javadoc -d docs -classpath json-20250107.jar logback-classic-1.5.18.jar logback-core-1.5.18.jar slf4j-api-2.0.16.jar client\core\\*.java server\collection\\*.java server\commands\\*.java client\input\\*.java client\input\basic\\*.java common\model\\*.java server\storage\\*.java server\network\\*.java server\core\\*.java client\core\\*.jar -encoding UTF-8 -charset UTF-8