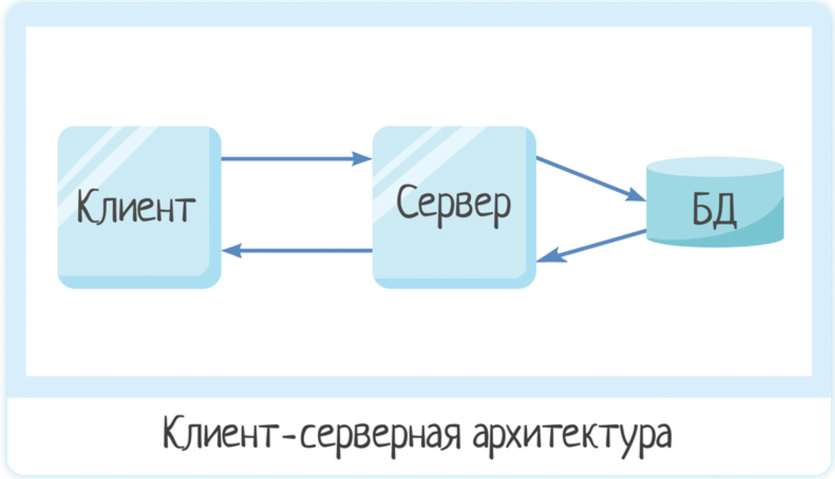
Теория 6 лаба

**1**

**Порт** — уникальный номер, с которым связан определённый сокет (этот термин будет рассмотрен далее), проще говоря, его занимает определённая служба для того что бы по нему могли связаться с ней.

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

Клиент – серверная архитектура



## **Плюсы архитектуры**

1. **Мощный сервер дешевле 100+ мощных клиентских машин** — если мы хотим, чтобы приложение не тормозило, нужна хорошая машина. Она у вас будет одна. Или несколько, если нагрузка большая, но явно меньше, чем количество клиентов.
2. **Нет дублирования кода** — основной код хранится на сервере, клиент отвечает только за «нарисовать красивенько» и простенькие проверки на полях «тут число, тут строка не длиннее 100 символов».
3. **Персональные данные в безопасности** — простой пользователь не видит лишнего.

## **Минусы архитектуры**

**Упало одно звено — все отдыхают**

Фактически [клиент](https://ru.wikipedia.org/wiki/%D0%9A%D0%BB%D0%B8%D0%B5%D0%BD%D1%82_(%D0%B8%D0%BD%D1%84%D0%BE%D1%80%D0%BC%D0%B0%D1%82%D0%B8%D0%BA%D0%B0)) и [сервер](https://ru.wikipedia.org/wiki/%D0%A1%D0%B5%D1%80%D0%B2%D0%B5%D1%80_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%BD%D0%BE%D0%B5_%D0%BE%D0%B1%D0%B5%D1%81%D0%BF%D0%B5%D1%87%D0%B5%D0%BD%D0%B8%D0%B5)) — это [программное обеспечение](https://ru.wikipedia.org/wiki/%D0%9F%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%BD%D0%BE%D0%B5_%D0%BE%D0%B1%D0%B5%D1%81%D0%BF%D0%B5%D1%87%D0%B5%D0%BD%D0%B8%D0%B5). Обычно эти программы расположены на разных [вычислительных машинах](https://ru.wikipedia.org/wiki/%D0%92%D1%8B%D1%87%D0%B8%D1%81%D0%BB%D0%B8%D1%82%D0%B5%D0%BB%D1%8C%D0%BD%D0%B0%D1%8F_%D0%BC%D0%B0%D1%88%D0%B8%D0%BD%D0%B0) и взаимодействуют между собой через [вычислительную сеть](https://ru.wikipedia.org/wiki/%D0%92%D1%8B%D1%87%D0%B8%D1%81%D0%BB%D0%B8%D1%82%D0%B5%D0%BB%D1%8C%D0%BD%D0%B0%D1%8F_%D1%81%D0%B5%D1%82%D1%8C) посредством [сетевых протоколов](https://ru.wikipedia.org/wiki/%D0%A1%D0%B5%D1%82%D0%B5%D0%B2%D0%BE%D0%B9_%D0%BF%D1%80%D0%BE%D1%82%D0%BE%D0%BA%D0%BE%D0%BB), но они могут быть расположены также и на одной машине.

Язык и правила общения определены в протоколе связи. Все протоколы клиент-серверной модели работают на уровне приложений. Протокол прикладного уровня определяет основные шаблоны диалога. Чтобы ещё больше формализовать обмен данными, сервер может реализовать интерфейс прикладного программирования ([API](https://ru.wikipedia.org/wiki/API)). API — это уровень абстракции для доступа к сервису. Ограничивая связь определённым форматом контента, он облегчает синтаксический анализ. Абстрагируя доступ, он облегчает межплатформенный обмен данными.

**Многоуровневая архитектура «клиент — сервер»** — разновидность архитектуры «клиент — сервер», в которой функция обработки данных вынесена на несколько отдельных серверов. Это позволяет разделить функции хранения, обработки и представления данных для более эффективного использования возможностей серверов и клиентов.

2/3

**Протокол передачи данных** — набор соглашений [интерфейса](https://ru.wikipedia.org/wiki/%D0%98%D0%BD%D1%82%D0%B5%D1%80%D1%84%D0%B5%D0%B9%D1%81) *логического уровня*, которые определяют обмен данными между различными [программами](https://ru.wikipedia.org/wiki/%D0%9F%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%BD%D0%BE%D0%B5_%D0%BE%D0%B1%D0%B5%D1%81%D0%BF%D0%B5%D1%87%D0%B5%D0%BD%D0%B8%D0%B5). Эти соглашения задают единообразный способ передачи сообщений и [обработки ошибок](https://ru.wikipedia.org/wiki/%D0%9E%D0%B1%D0%BD%D0%B0%D1%80%D1%83%D0%B6%D0%B5%D0%BD%D0%B8%D0%B5_%D0%B8_%D0%B8%D1%81%D0%BF%D1%80%D0%B0%D0%B2%D0%BB%D0%B5%D0%BD%D0%B8%D0%B5_%D0%BE%D1%88%D0%B8%D0%B1%D0%BE%D0%BA) при взаимодействии программного обеспечения разнесённой в пространстве [аппаратуры](https://ru.wikipedia.org/wiki/%D0%90%D0%BF%D0%BF%D0%B0%D1%80%D0%B0%D1%82%D0%BD%D0%BE%D0%B5_%D0%BE%D0%B1%D0%B5%D1%81%D0%BF%D0%B5%D1%87%D0%B5%D0%BD%D0%B8%D0%B5), соединённой тем или иным интерфейсом.

**Сетево́й протоко́л** — набор правил и действий (очерёдности действий), позволяющий осуществлять соединение и обмен данными между двумя и более включёнными в [сеть](https://ru.wikipedia.org/wiki/%D0%9A%D0%BE%D0%BC%D0%BF%D1%8C%D1%8E%D1%82%D0%B5%D1%80%D0%BD%D0%B0%D1%8F_%D1%81%D0%B5%D1%82%D1%8C) устройствами.

[стек протоколов TCP/IP](https://ru.wikipedia.org/wiki/TCP/IP) — содержит 4 уровня:

* канальный уровень (link layer),
* сетевой уровень (Internet layer),
* транспортный уровень (transport layer),
* прикладной уровень (application layer).

Основные протоколы:

IP отвечает за поиск компьютеров в сети по их IP-адресам. Ещё он предоставляет стратегию маршрутизации, то есть составляет оптимальный маршрут для передачи данных.

DNS связывает IP-адрес с понятным для людей доменным именем, например, 178.79.181.169 превращается в htmlacademy.ru:

SSL — криптографический протокол шифрования запросов и ответов, он нужен для безопасного перемещения данных по интернету, а TLS — его продвинутая версия.

TCP и UDP

Оба протокола отвечают за передачу данных и работают поверх IP, но с небольшой разницей. TCP доставляет данные без потерь, поэтому его используют для передачи фотографий, сообщений и другой важной информации. А вот UDP не гарантирует доставку, но зато он гораздо быстрее. Его используют, когда скорость важнее надёжности, например, при передаче аудиосообщений или видеотрансляциях.

У протокола TCP есть несколько особенностей:

* **Система нумерации сегментов.** TCP отслеживает передаваемые и принимаемые сегменты, присваивая номера каждому из них. Байтам данных, которые должны быть переданы, присваивается определенный номер байта, в то время как сегментам присваиваются порядковые номера.
* **Управление потоком.** Функция ограничивает скорость, с которой отправитель передает данные. Это делается для обеспечения надежности доставки, в том числе чтобы компьютер не генерировал пакетов больше, чем может принять другое устройство. Если говорить простым языком, то получатель постоянно сообщает отправителю о том, какой объем данных может быть получен.
* **Контроль ошибок.** Функция реализуется для повышения надежности путем проверки байтов на целостность.
* **Контроль перегрузки сети.** Протокол TCP учитывает уровень перегрузки в сети, определяемый объемом данных, отправленных узлом.

UDP просто отправляет пакеты и не ждет ничего в ответ. Из-за этого достигается высокая скорость в ущерб надежности.

Ключевым различием между TCP и UDP является скорость, поскольку TCP сравнительно медленнее UDP. В целом, UDP является быстрым, простым и эффективным протоколом, однако повторная передача потерянных пакетов данных возможна только в TCP.

Еще одно заметное различие между TCP и UDP заключается в том, что первый обеспечивает упорядоченную доставку данных от пользователя к серверу (и наоборот). UDP, в свою очередь, не проверяет готовность получателя и может доставлять пакеты вразнобой.

Socket

Сокет. Оно обозначает точку, через которую происходит соединение. Проще говоря, сокет соединяет в сети две программы.

Изображение выглядит как текст

Автоматически созданное описание

«имя\_хоста» — подразумевает под собой определённый узел сети, ip-адрес.

**InetAddress** представляет IP-адрес

Создав объект типа **ServerSocket** необходимо выяснить, что с сервером кто-то хочет соединиться. Тут подключается метод accept(). Искомый ждёт пока кто-либо не захочет подсоединится к нему, и когда это происходит возвращает объект типа **Socket**, то есть воссозданный клиентский сокет. И вот когда сокет клиента создан на стороне сервера, можно начинать двухстороннее общение.

Создать объект типа **Socket** на стороне клиента и воссоздать его с помощью **ServerSocket** на стороне сервера – вот необходимый минимум для соединения.

сть два метода которые дают доступ к потокам **InputStream** и **OutputStream** объекта типа **Socket**

InputStream getInputStream()

OutputStream getOutputStream()

**bind()** — сокету, который определен, присваивается идентификатор и порт на работающей машине.

127.0.0.1 известен как адрес обратной связи, но вы можете увидеть его под именем «localhost». Когда вы указываете браузер на 127.0.0.1, он пытается подключиться к компьютеру, который вы используете прямо сейчас. Это удобно, когда вы хотите подключиться к серверу на своем компьютере.

DatagramPacket — класс, который представляет собой пакет датаграмм. Пакеты датаграмм используются для реализации службы доставки пакетов без подключения. Каждое сообщение направляется с одной машины на другую исключительно на основе информации, содержащейся в этом пакете.

#### Состав пакета java.net / Средства передачи по протоколу UDP

#### Для работы с **UDP** в пакете **java.net** определены следующие классы.

DatagramSocket — предназначен для посылки/приема **UDP** дейтаграмм.

Конструкторы класса:

DatagramSocket() — создаваемый сокет присоединяется к любому свободному порту на локальной машине;

DatagramSocket (**int** port) — создаваемый сокет присоединяется к порту port на локальной машине;

DatagramSocket(**int** port, InetAddress addr) — создаваемый сокет присоединяется к порту port; аргумент addr – IP-адрес сервера.

Методы класса:

send(DatagramPacket pack) — отправляет дейтаграмму, упакованную в пакет pack;

receive(DatagramPacket pack) — дожидается получения дейтаграммы и заносит ее в пакет pack;

setSoTimeout() — устанавливает тайм-аут для операций сокета;

close() — закрывает сокет.

DatagramPacket — предназначен для представления единичной дейтаграммы.

Конструкторы класса:

DatagramPacket(**byte**[] buf, **int** length) — служит для получения пакета длиной length, buf — буфер для получения дейтаграммы;

DatagramPacket(**byte**[] buf, **int** length, InetAddress addr, **int** port) — Создает пакет дейтаграммы длиной length, чтобы отправить его к указанному номеру порта (port) на указанном узле (address), buf — буфер, содержащий данные для передачи.

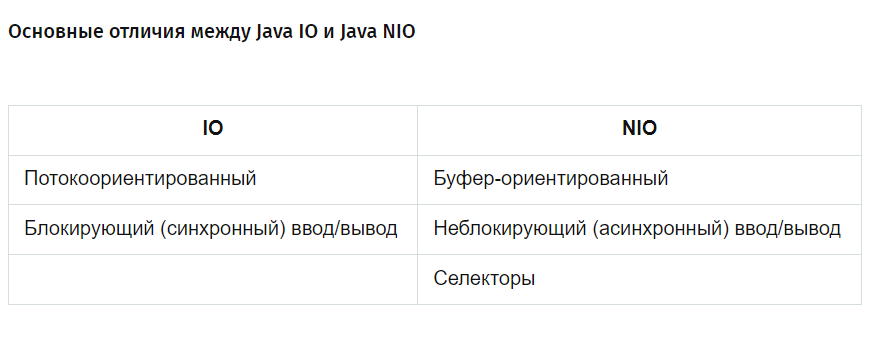
Кроме того пакет содержит вспомогательные методы:

connect(InetAddress addr, **int** port) — установить соединение;

disconnect() — разъединить;

InetAddress.getByName(host) — получить IP-адрес хоста;

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



Потокоориентированный ввод/вывод подразумевает чтение/запись из потока/в поток одного или нескольких байт в единицу времени поочередно. Данная информация нигде не кэшируются. Таким образом, невозможно произвольно двигаться по потоку данных вперед или назад. Если вы хотите произвести подобные манипуляции, вам придется сначала кэшировать данные в буфере.

Буферно-ориентированный: данные считываются в буфер, который обрабатываются позже, и при необходимости могут перемещаться туда и обратно в буфере. Это увеличивает гибкость процесса. Однако вам также необходимо проверить, содержит ли буфер все данные, которые вам нужно обработать. Более того, необходимо обеспечить, чтобы при считывании в буфер большего количества данных необработанные данные в буфере не перезаписывались.

#### **Блокирующий и неблокирующий ввод/вывод**

Потоки ввода/вывода (streams) в Java IO являются блокирующими. Это значит, что когда в потоке выполнения (tread) вызывается read() или write() метод любого класса из пакета java.io.\*, происходит блокировка до тех пор, пока данные не будут считаны или записаны. Поток выполнения в данный момент не может делать ничего другого.  
  
Неблокирующий режим Java NIO позволяет запрашивать считанные данные из канала (channel) и получать только то, что доступно на данный момент, или вообще ничего, если доступных данных пока нет. Вместо того, чтобы оставаться заблокированным пока данные не станут доступными для считывания, поток выполнения может заняться чем-то другим.

Таким образом неблокирующий режим Java NIO позволяет использовать один поток выполнения для решения нескольких задач вместо пустого прожигания времени на ожидание в заблокированном состояний.

Работа с сетевыми каналами.

Каналы – это логические (не физические) порталы, через которые осуществляется ввод/вывод данных, а буферы являются источниками или приёмниками этих переданных данных. При организации вывода, данные, которые вы хотите отправить, помещаются в буфер, а он передается в канал. При вводе, данные из канала помещаются в предоставленный вами буфер.

Отличия каналов от потока:

Каналы двунаправлены, доступны для чтения и записи, а чтение и запись потока - однонаправленные.

Каналы можно читать и записывать асинхронно.

Независимо от чтения и записи канал может взаимодействовать только с буфером.

#### **Селекторы** Селекторы в Java NIO позволяют одному потоку выполнения мониторить несколько каналов ввода. Вы можете зарегистрировать несколько каналов с селектором, а потом использовать один поток выполнения для обслуживания каналов, имеющих доступные для обработки данные, или для выбора каналов, готовых для записи.

#### 

#### **API обращений к классам ввода/вывода** Естественно, использование Java NIO серьезно отличается от использования Java IO. Так как, вместо чтения данных байт за байтом с использованием, например InputStream, данные для начала должны быть считаны в буфер и браться для обработки уже оттуда.

* Ссылка на статью, в случае если нужно больше инфы: https://habr.com/ru/articles/235585/

5.Классы SocketChannel и DatagramChannel.

**Реализация наиболее важных каналов в JavaNIO:**

FileChannel: Чтение и запись данных из файлов (только режим блокировки).

DatagramChannel: Чтение и запись данных в сети через UDP (блокирование и неблокирование по желанию).

SocketChannel: Чтение и запись данных в сети через TCP (блокирование и неблокирование по желанию).

**ServerSocketChannel**: Он может отслеживать новое входящее TCP-соединение, как веб-сервер. Для каждого нового входящего соединения создается SocketChannel.

* Ссылка на сайт про сокет каналы <https://russianblogs.com/article/19281613674/>

1. Передача данных

Обмен информацией между компьютерами (по проводному соединению или нет) происходит путем передачи пакетов (фрагментов) данных через сеть. Протокол передачи данных - это набор соглашений интерфейса логического уровня, которые определяют обмен данными между различными программами. Для передачи данных в сети Интернет используется стек протоколов TCP/IP.

Общий ход передачи информации выглядит следующим образом:

1. Данные от приложения отправляются протоколу транспортного уровня.
2. Получив данные от приложения, протокол разделяет всю информацию на небольшие блоки (пакеты). К каждому пакету добавляется адрес назначения, а затем пакет передается на следующий уровень - уровень протоколов Интернет (сетевой уровень).
3. На сетевом уровне пакет помещается в дейтаграмму протокола Интернет (IP), к которой добавляется заголовок и концевик. Протокол сетевого уровня определяет адрес следующего пункта назначения IP-дейтаграммы и отправляет его на уровень сетевого интерфейса.
4. Уровень сетевого интерфейса принимает IP-дейтаграмму и передает их в виде кадров с помощью аппаратного обеспечения (например, сетевой карты).

Сериализация — это преобразование объекта или дерева объектов в какой-либо формат с тем, чтобы потом эти объекты можно было восстановить из этого формата. Результат сериализации – xml json файл например

Сериализация имеющихся объектов – это процесс сохранения состояния объекта в последовательность байт.

Десериализация – восстановление объектов из байт, сохранение которых было произведено ранее.

Для того, чтобы сериализуемый набор объектов был «собран» в поток, необходимо применить class Object Output Stream (пишется слитно). Он отвечает за потоковую запись информации.

Пример сериализации из нашего кода

OutputStream send = socket1.getOutputStream();  
ObjectOutputStream objectSend = new ObjectOutputStream(send);  
objectSend.writeObject(request);

Чтобы объекты были сериализованы нужно class Person implements Serialazable

Интерфейс Serializable это интерфейс-маркер; в нём не задекларировано ни одного метода. Но говорит сериализующему механизму, что класс может быть сериализован. интерфейс Serializable говорит Java-машине: «все ок, если что, объекты этого класса можно сериализовать».

Объектный граф

Объекты имеют ссылки на другие объекты, которые, в свою очередь, могут иметь ссылки на другие объекты, включая начальный объект. Это создает граф объектов

При сериализации связанных объектов имеет место **граф сериализации**. Сериализуется объект, если он содержит ссылки или массивы ссылок на сериализуемые объекты, то они сериализуются рекурсивно. При этом сериализуемые объекты помечаются, что исключает зацикливание при обходе графа.

Кроме того, для сериализуемых объектов создаются уникальные идентификаторы, которые необходимы для восстановления ссылок при десериализации.

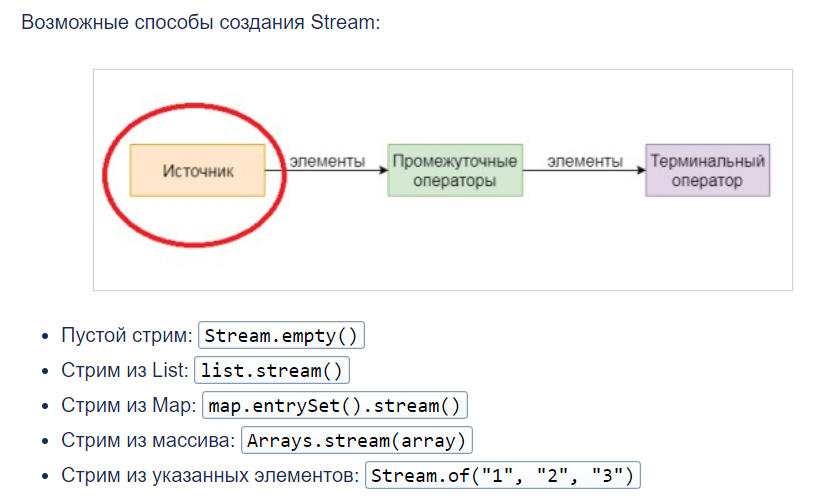
Если мы не хотим сериализовать поле то надо перед ним написать transient

Java Stream API. Создание конвейеров. Промежуточные и терминальные

**API** - **Stream** **API**. Его **задача** - упростить работу с наборами данных, в частности, упростить операции фильтрации, сортировки и другие манипуляции с данными. Вся **основная** функциональность данного **API** сосредоточена в пакете java.util.**stream**. Ключевым понятием в **Stream** **API** является поток данных.

* *Промежуточные* (“intermediate”, ещё называют “lazy”) — обрабатывают поступающие элементы и возвращают стрим. Промежуточных операторов в цепочке обработки элементов может быть много.
* *Терминальные* (“terminal”, ещё называют “eager”) — обрабатывают элементы и завершают работу стрима, так что терминальный оператор в цепочке может быть только один.

Если нужны простые примеры с javarush - https://javarush.com/groups/posts/2203-stream-api



Потоковые операции делятся на промежуточные и терминальные операции и объединяются для формирования потоковых конвейеров. Конвейер потока состоит из источника (такого как Collection, массив, функция-генератор или канал ввода-вывода); за которым следует ноль или более промежуточных операций, таких как Stream.filter или Stream.map; и терминальной операции, такой как Stream.forEach or Stream.reduce.

Промежуточные операции далее подразделяются на операции без состояния и операции с сохранением состояния. Операции без состояния, такие как filter и map, не сохраняют состояние ранее просмотренного элемента при обработке нового элемента - каждый элемент может обрабатываться независимо от операций над другими элементами. Операции с сохранением состояния, такие как distinct и sorted, могут включать состояние ранее просмотренных элементов при обработке новых элементов.

Выражение System.out::println сокращенно от: x-> System.out.println(x)), которое в свою очередь проходит по всем элементам переданной ему коллекции и выводит её

# Промежуточные операции Stream API(Особенности, взятые из разных источников: возвращают типизированный интерфейс **Stream<>**, промежуточные операции не выполняются без терминальных)

**Преобразование**

Любое изменение исходного элемента можно делать с помощью метода **map()**. В качестве параметра метод принимает лямбда-выражение.

Stream.of(1, 2, 3, 4, 5, 6)  
        .map(n -> n \* 10) *// умножает каждый элемент на 10*

Stream.of(**"apple"**, **"orange"**)  
        .map(String::toUpperCase) *// преобразует буквы в каждом слове в верхний регистр*

Во втором случае мы воспользовались краткой записью лямбда-выражения через **method reference**.

**Фильтрация**

Отсеивание части объектов можно сделать с помощью метода **filter()**.

Stream.of(1, 2, 3, 4, 5, 6)  
        .filter(n -> n > 3) *// оставляет только те числа, которые больше 3*

**Доступ к элементу**

Если требуется получить доступ к элементу, никак его при этом не меняя, нам поможет метод **peek()**. Например, в целях логирования мы хотим вывести значение элемента в консоль.

Stream.of(1, 2, 3, 4, 5, 6)  
        .peek(System.out::println)

**Удаление дублей**

Если в стриме несколько элементов имеют одинаковые значения, а вы хотите получить только уникальные, используйте метод **distinct()**.

Stream.of(2, 2, 3, 2, 3, 3, 2, 2)  
        .distinct() *// в результате останется только два элемента: [2, 3]*

**Сортировка**

Для сортировки используйте метод **sorted()**. По умолчанию он использует «натуральный порядок», т.е. сортировку по возрастанию.

Stream.of(2, 1, 3, 5, 4, 6)  
        .sorted() *// получим [1, 2, 3, 4, 5, 6]*

Перегруженная версия этого метода принимает интерфейс **Comparator**, поэтому вы можете задать свои собственные правила сортировки или воспользоваться готовыми. Приведённый выше пример равносилен следующему:

Stream.of(2, 1, 3, 5, 4, 6)  
        .sorted(Comparator.naturalOrder())) *// получим [1, 2, 3, 4, 5, 6]*

Нетрудно догадаться, что обратную сортировку можно задать таким образом:

Stream.of(2, 1, 3, 5, 4, 6)  
        .sorted(Comparator.reverseOrder()) *// получим [6, 5, 4, 3, 2, 1]*

**Ограничение по количеству элементов**

Для получения первых N элементов используйте метод **limit()**. В данном случае мы берём первые 3 элемента:

Stream.of(1, 2, 3, 4, 5, 6)  
        .limit(3) *// [1, 2, 3]*

**Пропуск первых элементов**

Чтобы пропустить нужное количество первых элементов, используйте метод **skip()**. Здесь мы пропускаем первые 4 элемента и получаем оставшиеся два:

Stream.of(1, 2, 3, 4, 5, 6)  
        .skip(4) *// [5, 6]*

Обратите внимание, что комбинация методов **limit()** и **skip()** позволяет организовать постраничный вывод информации. Например, в пользовательском интерфейсе мы отображаем по две записи на странице. Тогда, если у нас запросят вторую страницу, то выражение будет выглядеть так:

Stream.of(1, 2, 3, 4, 5, 6)  
        .skip(2)  
        .limit(2) *// [3, 4]*

**Разворачивание многомерных структур**

Предположим, у вас есть многомерная структура (двумерный массив элементов типа **Integer**) и вы хотите его развернуть (сделать «плоским»), просто помещая каждый подмассив в конец результирующего. В этом нам поможет метод **flatMap()**:

Integer[][] array2d = **new** Integer[][]{{1, 2}, {3, 4}};  
Stream<Integer> stream =  
        Arrays.stream(array2d).flatMap(Arrays::stream); *// [1, 2, 3, 4]*

## [Терминальные операции Stream API](https://devmark.ru/article/stream-api-terminal-opers)

**Преобразование в коллекцию**

Самая распространённая терминальная операция **collect()**. Результатом может быть, например, список.

List<String> fruits = Stream.of(**"apple"**, **"banana"**, **"lemon"**, **"orange"**)  
        *// здесь могут быть ещё какие-то преобразования*  
        .collect(Collectors.toList());

А можно преобразовать стрим из строк в мапу, причём ключом сделать первую букву соответствующего слова:

Map<String, String> fruits = Stream.of(**"apple"**, **"banana"**, **"lemon"**, **"orange"**)  
        .collect(Collectors.toMap(e -> e.substring(0, 1), e -> e));  
        *// {a=apple, b=banana, l=lemon, o=orange}*

**Итерация по элементам**

Простой обход элементов стрима можно выполнить с помощью метода **forEach()**. Его смысл полностью аналогичен языковой конструкции **for**.

Stream.of(5, 3, 2, 10, 8).forEach(System.out::println);

Здесь мы просто выводим в консоль значение каждого элемента стрима.

**Подсчёт количества элементов**

Аналог метода **size()** в коллекциях и поля **length** в массивах. В стримах для этого используется метод **count()**. Обратите внимание, что метод возвращает не **int**, а **long**.

**long** count = Stream.of(**"apple"**, **"banana"**, **"lemon"**, **"orange"**)  
        .count(); *// 4*

**Минимальное и максимальное значения**

Как нетрудно догадаться, **min()** возвращает минимальное значение, а **max()** – максимальное. В качестве параметра они принимают класс **Comparator**.

Также обратите внимание, как мы обошли «одноразовость» потока. Чтобы не писать два раза конструкцию **Stream.of(...)**, мы использовали утилитарный класс **Supplier<>**. Вызов метода **get()** у него в действительности каждый раз создаёт новый поток с одними и теми же значениями.

Supplier<Stream<Integer>> streamSupplier = () -> Stream.of(5, 3, 2, 10, 8);  
Optional<Integer> min = streamSupplier.get().min(Comparator.naturalOrder()); *// Optional[2]*  
Optional<Integer> max = streamSupplier.get().max(Comparator.naturalOrder()); *// Optional[10]*

**Поиск первого подходящего элемента**

Метод **findFirst()** возвращает первый элемент стрима. Этот метод очень удобно использовать в паре с **filter()**.

Optional<Integer> first = Stream.of(1, 2, 3, 4, 5)  
        .filter(e -> e % 2 == 0)  
        .findFirst(); *// Optional[2]*

В данном случае мы вернём первое чётное число, т.е. 2.

**Соответствие всех элементов**

Чтобы проверить все элементы стрима на соответствие некоторому условию, используйте метод **allMatch()**. Его результатом может быть только **true** или **false**.

**boolean** isAllPositive = Stream.of(1, 2, 3, 4, 5)  
        .allMatch(e -> e > 0); *// true*

В этом примере мы проверяем, что все элементы положительные.

**Полное несоответствие**

Если нужно проверить, что ни один элемент не удовлетворяет условию, воспользуйтесь методом **noneMatch()**. Перепишем предыдущий пример так:

**boolean** isAllPositive = Stream.of(1, 2, 3, 4, 5)  
        .noneMatch(e -> e < 0); *// true*

**Соответствие хотя бы одного элемента**

Чтобы проверить наличие хотя бы одного элемента, удовлетворяющего условию, используйте метод **anyMatch()**.

**boolean** hasNegative = Stream.of(1, 2, 3, 4, 5)  
        .anyMatch(e -> e < 0); *// false*

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

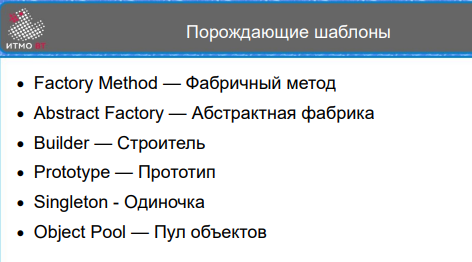


Паттерн представляет определенный способ построения программного кода для решения часто встречающихся проблем проектирования.

Категории паттернов:

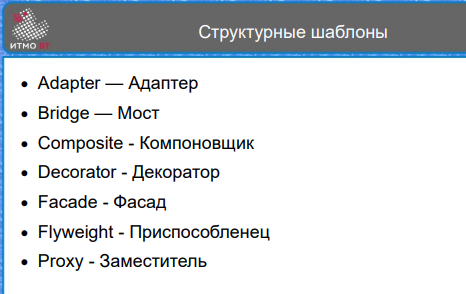
## **Порождающие** (creational patterns)

Эти паттерны решают проблемы обеспечения гибкости создания объектов

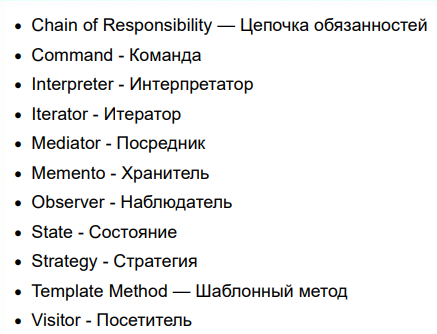


* **Структурные**

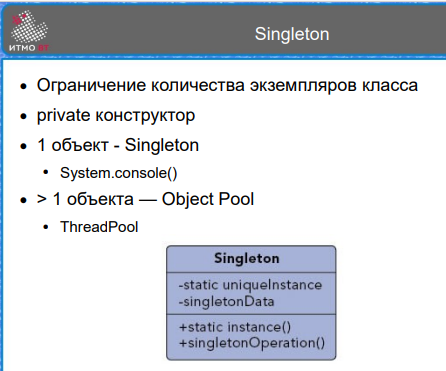
Эти паттерны решают проблемы эффективного построения связей между объектами



* **Поведенческие**

Эти паттерны решают проблемы эффективного взаимодействия между объектами 

Паттерн Singleton

****

**Одиночка** (Singleton) - порождающий паттерн, который гарантирует, что для определенного класса будет создан только один объект, а также предоставит к этому объекту точку доступа.

* + ***Когда надо использовать Синглтон?***
    - Когда необходимо, чтобы для класса существовал только один экземпляр

**Общие сведения**  
Паттерн Singleton гарантирует, что у класса есть только один экземпляр, и предоставляет к нему глобальную точку доступа.

**Область применения**  
1.) В системе должно существовать не более одного экземпляра заданного класса.  
2.) Экземпляр должен быть легко доступен для всех клиентов данного класса.  
3.) Создание объекта on demand, то есть, когда он понадобится первый раз, а не во время инициализации системы.

## Структурные паттерны (structural patterns)

С созданием объектов стало понятнее. И теперь самое время посмотреть на структурные паттерны. Их цель — построение удобных в поддержке иерархий классов и их взаимосвязей. Одним из первых и всем известных паттернов — "**Заместитель**" (Proxy). Заместитель имеет тот же интерфейс, что и реальный объект, поэтому для клиента нет разницы — работать через заместителя или напрямую. Самым простым примером является [java.lang.reflect.Proxy](https://docs.oracle.com/javase/8/docs/technotes/guides/reflection/proxy.html):

**import** java.util.\*;

**import** java.lang.reflect.\*;

**class** Main {

**public** **static** **void** main(String[] arguments) {

**final** Map<String, String> original = **new** HashMap<>();

InvocationHandler proxy = (obj, method, args) -> {

System.out.println("Invoked: " + method.getName());

**return** method.invoke(original, args);

};

Map<String, String> proxyInstance = (Map) Proxy.newProxyInstance(

original.getClass().getClassLoader(),

original.getClass().getInterfaces(),

proxy);

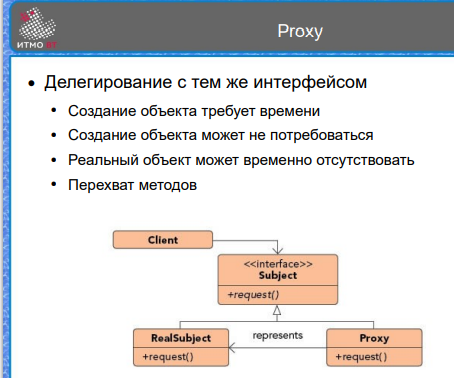
proxyInstance.put("key", "value");

System.out.println(proxyInstance.get("key"));

}

}

Как видно, в примере у нас есть original — это HashMap, который реализует интерфейс Map. Мы далее создаём прокси, который замещает оригинальную HashMap для клиентской части, которая вызывает методы put и get, добавляя во время вызова свою логику.

****

Как мы видим, взаимодействие в паттерне идёт через интерфейсы. Но иногда заместителя недостаточно. И тогда может быть использован паттерн "**Декоратор**" (Decorator). Декоратор ещё называют обёрткой или враппером (Wrapper). Прокси и декоратор очень похожи, но если посмотреть на пример — будет видна разница:

**import** java.util.\*;

**class** Main {

**public** **static** **void** main(String[] arguments) {

List<String> list = **new** ArrayList<>();

List<String> decorated = Collections.checkedList(list, String.**class**);

decorated.add("2");

list.add("3");

System.out.println(decorated);

}

}

В отличии от прокси, декоратор оборачивается вокруг чего-то, что передали на вход. Прокси же может как принимать то, что нужно проксировать, так и сам управлять жизнью проксируемого объекта (например, создавать проксируемый объект).



"**Адаптер**" (adapter). Он похож на декоратор — на вход декоратор принимает один объект и возвращает обёртку над этим объектом. Отличие в том, что цель у этого не изменение функционала, а адаптация одного интерфейса к другому.

**import** java.awt.\*;

**class** Main {

**public** **static** **void** main(String[] arguments) {

Container container = **new** Container();

Component component = **new** java.awt.Component(){};

System.out.println(component.getComponentOrientation().isLeftToRight());

container.add(component);

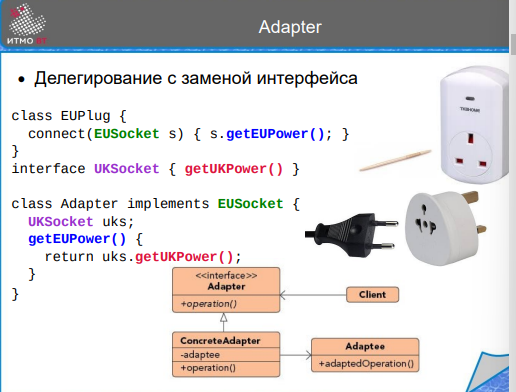
container.applyComponentOrientation(ComponentOrientation.RIGHT\_TO\_LEFT);

System.out.println(component.getComponentOrientation().isLeftToRight());

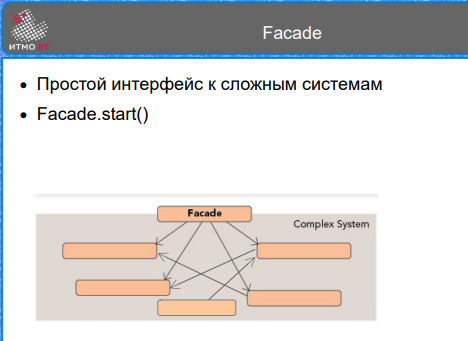
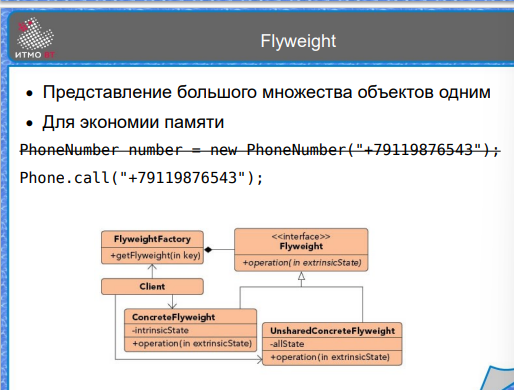
}

}

Как мы видим, мы добавили в контейнер компонент. А потом попросили контейнер применить новую ориентацию компонентов. И контейнер, зная из каких компонентов он состоит, делегировал выполнение этой команды всем дочерним компонентам.



Среди структурных паттернов так же хочется отметить паттерн "**Фасад**" (facade). Суть его в том, чтобы за удобным и лаконичным интерфейсом спрятать сложность использования библиотек/фрэймворков, стоящих за этим API.

"**Легковес**" (Flyweight). Его суть заключается в том, что если у разных объектов есть одинаковое состояние, то его можно обобщить и хранить не в каждом объекте, а в одном месте. И тогда каждый объект сможет ссылаться на общую часть, что позволит сократить расходы памяти на хранение. Часто работа данного паттерна связана с предварительным кэшированием или с поддержанием пула объектов.

При помощи паттерна "Стратегия" мы можем внутри объекта хранить то, каким образом мы будем выполнять действие, т.е. объект внутри хранит стратегию, которая может быть изменена в том числе во врем выполнения кода. Этот паттерн мы часто используем, когда применяем компаратор:

**import** java.util.\*;

**class** Main {

**public** **static** **void** main(String[] args) {

List<String> data = Arrays.asList("Moscow", "Paris", "NYC");

Comparator<String> comparator = Comparator.comparingInt(String::length);

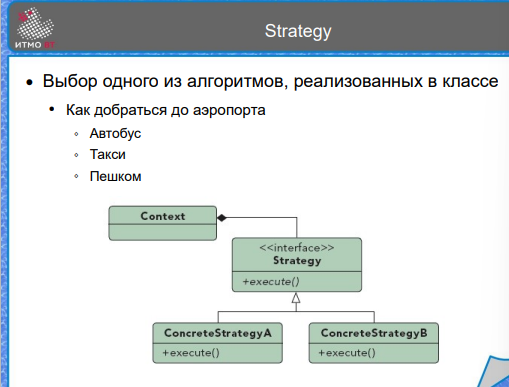
Set dataSet = **new** TreeSet(comparator);

dataSet.addAll(data);

System.out.println("Dataset : " + dataSet);

}

}



паттерн "**Комманда**". Этот паттерн проектирования говорит о том, что различные команды можно представлять в виде разных классов. Данный паттерн очень похож на паттерн "Стратегия". Но в паттерне "Стратегия" мы переопределяли то, как будет выполняться конкретное действие (например, сортировка в TreeSet). В паттерне "Комманда" же мы переопределяем то, какое вообще действие будет выполнено. Паттерн комманда с нами каждый день, когда мы используем потоки:

**import** java.util.\*;

**class** Main {

**public** **static** **void** main(String[] args) {

Runnable command = () -> {

System.out.println("Command action");

};

Thread th = **new** Thread(command);

th.start();

}

}



Как видим, command определяет действие или комманду, которая будет выполнена в новом потоке.

Ещё один паттерн, который мы видим каждый день "**Итератор**". Суть его заключается в том, чтобы разделить коллекцию объектов (т.к. класс, представляющий структуру данных. Например, List) и обход этой коллекции.

**import** java.util.\*;

**class** Main {

**public** **static** **void** main(String[] args) {

List<String> data = Arrays.asList("Moscow", "Paris", "NYC");

Iterator<String> iterator = data.iterator();

**while** (iterator.hasNext()) {

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

}

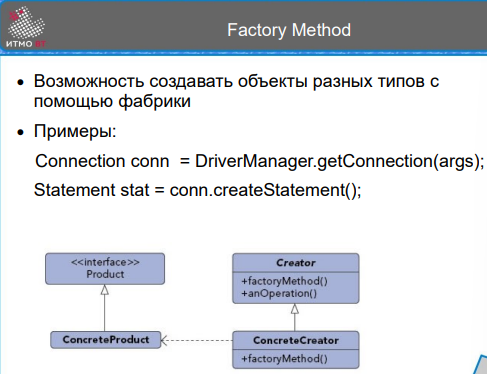
}

}

Как видно, итератор — не является частью коллекции, а представлен отдельным классом, который обходит коллекцию. Использующий итератор даже может не знать про то, по какой коллекции он итерируется, т.е. какую коллекцию он обходит.



**Фабричный метод** (Factory Method) - это паттерн, который определяет интерфейс для создания объектов некоторого класса, но непосредственное решение о том, объект какого класса создавать происходит в подклассах. То есть паттерн предполагает, что базовый класс делегирует создание объектов классам-наследникам.

* ***Когда надо применять паттерн?***
  + Когда заранее неизвестно, объекты каких типов необходимо создавать
  + Когда система должна быть независимой от процесса создания новых объектов и расширяемой: в нее можно легко вводить новые классы, объекты которых система должна создавать.
  + Когда создание новых объектов необходимо делегировать из базового класса классам наследникам
  + 
* Полезная ссылка - <https://github.com/georgedem975/patterns/blob/master/description/PATTERNS.md>

