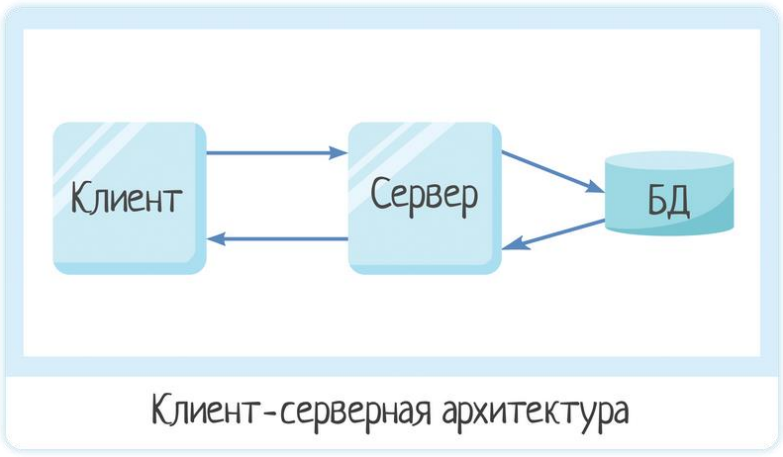
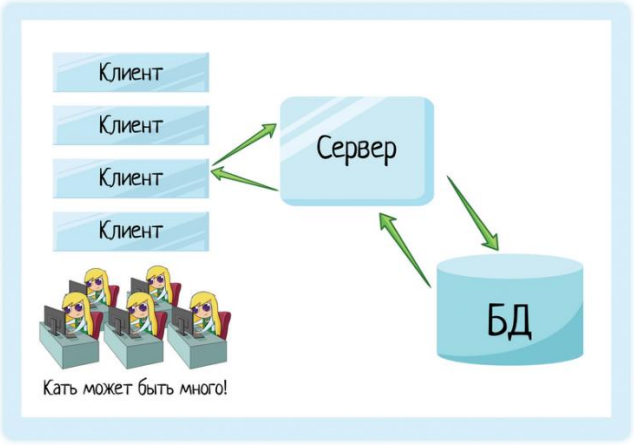
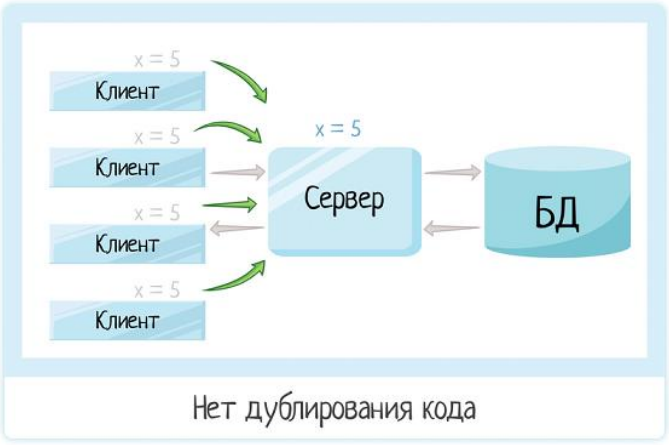
1. Сетевое взаимодействие - клиент-серверная архитектура, основные протоколы, их сходства и отличия.

К-С архитектура. Тут легче картинками

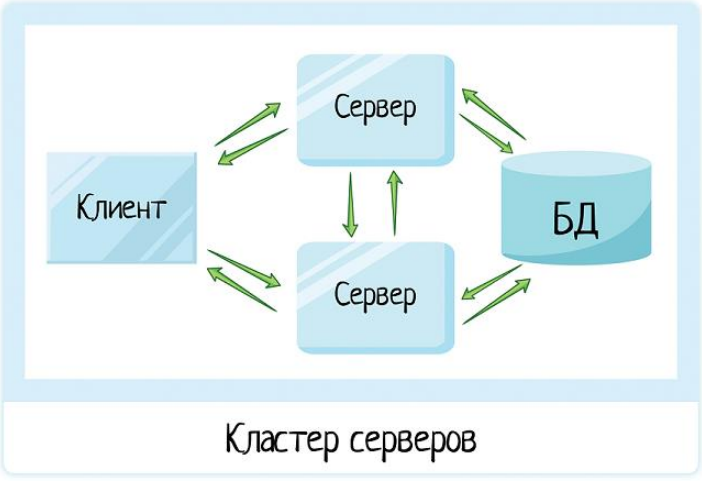


- ! на **клиенте** выполняются максимум маленькие проверки на правильность ввода данных! Всё остальное делает **сервер** (поэтому можно купить мощную машину для сервера (бугатти) и клиент может быть слабым там без разницы уже все будет работать (лада 9)
- это хорошо, потому что можно настроить доступ к данным у клиента (безопасность) (кейс: работник в банке дает кредит человеку , значит он сейчас выступает в роли клиента -> клиенту будет разрешено иметь доступ только к кредитной истории. Как плюс получаем сохранение конфиденциальной информации человека)
- можно обойтись и без БД если инфы мало и сервер потянет сам.
- клиент выступает в роли локальной машины



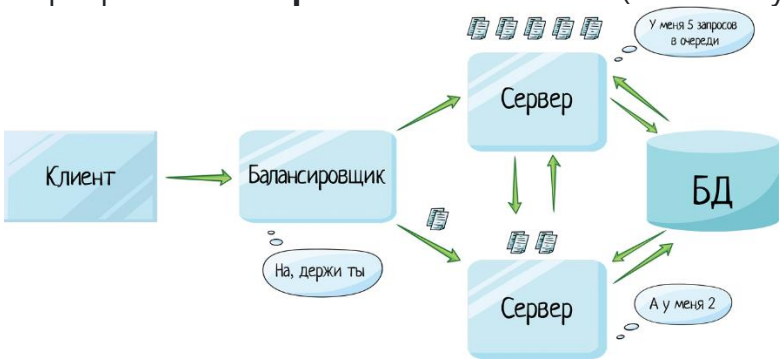


если бы все лежало на клиенте баги нужно было бы чинить на каждом клиенте индивидуально много раз, а так можно починить один раз на сервере – это круто и дешево

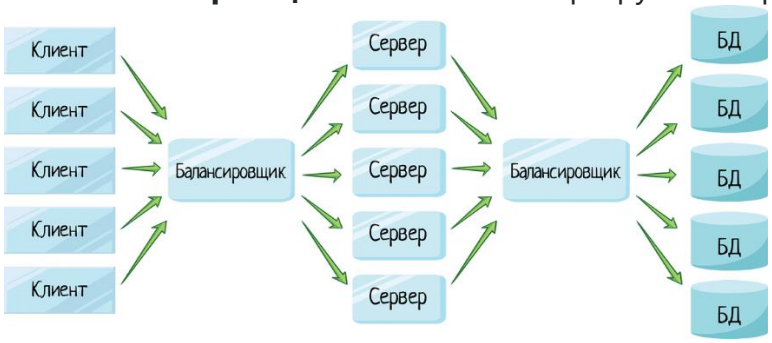


-тк как если упадет одно звено то работа прекратиться везде придумали такое (см выше)

это **горячий резерв** (то что у нас одновременно в работе 2 сервера) есть также **холодный** (второй сервер подключится только при падении первого)
-серверов в **кластере** может быть много (сколько купишь)



блок **балансировщик** помогает не перегружать серверы (можно также использовать для БД)



(базы данных можно например копировать чтобы при падении БД информация не потерялась и все работалао)

ВИДЫ СЕТЕВЫХ ПРОТОКОЛОВ

TCP/IP – совокупность протоколов передачи информации. TCP/IP – это особое обозначение всей сети, которая функционирует на основе протоколов TCP, а также IP.
TCP – вид протокола, который является связующим звеном для установки качественного соединения между 2 устройствами, передачи данных и верификации их получения.
IP – протокол, в функции которого входит корректность доставки сообщений по выбранному адресу. При этом информация делится на пакеты, которые могут поставляться по-разному.
MAC – вид протокола, на основании которого происходит процесс верификации сетевых устройств. Все устройства, которые подключены к сети Интернет, содержат свой оригинальный MAC-адрес.
ICMP – протокол, который ответственен за обмен данными, но не используется для процесса передачи информации.
UDP – протокол, управляющий передачей данных, но данные не проходят верификацию при получении. Этот протокол функционирует быстрее, чем протокол TCP.
HTTP – протокол для передачи информации (гипертекста), на базе которого функционируют все сегодняшние сайты. В его возможности входит процесс запрашивания необходимых данных у виртуально удаленной системы (файлы, веб-страницы и прочее).
FTP – протокол передачи информации из особого файлового сервера на ПК конечного пользователя.
POP3 – классический протокол простого почтового соединения, который ответственен за передачу почты.
SMTP – вид протокола, который может устанавливать правила для передачи виртуальной почты. Он ответственен за передачу и верификацию доставки, а также оповещения о возможных ошибках.

2. Протокол TCP. Классы `Socket` и `ServerSocket`.

https://coderlessons.com/tutorials/akademicheskii/izuchite-besprovodnuiu-sviaz/besprovodnaia-sviaz-tcp-ip - тут подробнее про TCP протокол и его уровни набора протоколов

Адрес подразумевает под собой идентификатор машины в пространстве сети Internet. Он может быть доменным именем, например, «itmo.ru», или обычным IP.
Порт — уникальный номер, с которым связан определённый сокет, проще говоря, его занимает определённая служба для того, что бы по нему могли связаться с ней.
Так что для того, что бы произошла встреча как минимум двух объектов на территории одного (сервера) — хозяин местности (сервер) должен занять конкретную квартиру (порт) на ней (машине), а второй должен найти место встречи зная адрес дома (домен или ip), и номер квартиры (порт).

Soket через каналы ввода/вывода осуществляет общение клиента с сервером

`Socket(String IP-адрес, int порт) throws UnknownHostException, IOException` (если класс сокета не смог преобразовать его в реальный, существующий, адрес) (при потере соединения)
`Socket(InetAddress IP-адрес, int порт) throws UnknownHostException` (тут InetAdress может быть доменным именем или массив IP-адресов)
Если порт указать 0, то система сама выделяет свободный порт.

`InetAddress getInetAddress()` – возвращает объект, содержащий данные о сокете. В случае если сокет не подключен – null
`int getPort()` – возвращает порт, по которому происходит соединение с сервером
`int getLocalPort()` – возвращает порт, к которому привязан сокет. Дело в том, что «общаться» клиент и сервер могут по одному порту, а порты, к которым они привязаны – могут быть совершенно другие
`boolean isConnected()` – возвращает true, если соединение установлено
`void connect(SocketAddress адрес)` – указывает новое соединение
`boolean isClosed()` – возвращает true, если сокет закрыт boolean
`isBound()` - возвращает true, если сокет действительно привязан к адресу
Для закрытия -> `.close()`

Как сервер понимает, что был передан запрос на соединение? Для это сервер имеет такой класс как ServerSocket, и метод accept() в нём. Его конструкторы представлены ниже:
```
ServerSocket() throws IOException
ServerSocket(int порт) throws IOException
ServerSocket(int порт, int максимум_подключений) throws IOException
ServerSocket(int порт, int максимум_подключений, InetAddress локальный_адрес) throws
IOException
```

При объявлении ServerSocket не нужно указывать адрес соединения, потому что общение происходит на машине сервера. Только при многоканальном хосте нужно указать к какому ip привязан сокет сервера.

3. Протокол UDP. Классы `DatagramSocket` и `DatagramPacket`.

Java NIO дейтаграмма используется в качестве канала, который может отправлять и получать UDP-пакеты по протоколу без установления соединения. По умолчанию канал дейтаграмм блокируется, и его можно использовать в неблокирующем режиме. Чтобы сделать его неблокирующим, мы можем использовать `configureBlocking(false)` метод. Канал DataGram можно открыть, вызвав его один из статических методов с именем `open()`, который также может принимать IP-адрес в качестве параметра, чтобы его можно было использовать для многоадресной передачи.
Мы можем проверить состояние соединения канала дейтаграмм, вызвав его метод `isConnected()`. После подключения канал дейтаграмм остается подключенным до тех пор, пока он не будет отключен или закрыт. Каналы дейтаграмм являются поточно-ориентированными и поддерживают многопоточность и параллелизм одновременно.

Важные методы дейтаграммы канала
`bind (SocketAddress local)` – этот метод используется для привязки сокета канала дейтаграммы к

локальному адресу, который предоставляется в качестве параметра для этого метода.
`connect (SocketAddress remote)` – этот метод используется для подключения сокета к удаленному адресу.
`disconnect ()` – этот метод используется для отключения сокета от удаленного адреса.
`getRemoteAddress ()` – Этот метод возвращает адрес удаленного местоположения, к которому подключен сокет канала.
`isConnected ()` – как уже упоминалось, этот метод возвращает статус подключения канала дейтаграммы, т. е. подключен он или нет.
`open ()` и `open` – используется метод Open, который открывает канал дейтаграммы для одного адреса, в то время как параметризованный метод Open метод открытия канала для нескольких адресов, представленных в виде семейства протоколов.
`read (ByteBuffer dst)` – этот метод используется для чтения данных из данного буфера через канал дейтаграммы.
`receive (ByteBuffer dst)` – этот метод используется для получения дейтаграммы через этот канал.
`send (ByteBuffer src, SocketAddress target)` – этот метод используется для отправки дейтаграммы по этому каналу.

Для посылки дейтаграмм отправитель и получатель создают сокеты дейта-граммного типа. В Java их представляет класс **DatagramSocket**. В классе три конструктора:

- `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 СОКСТ при соединяется к порту port; аргумент addr — один из адресов локальной машины.

Класс содержит массу методов доступа к параметрам сокета и, кроме того, методы отправки и приема дейтаграмм:

- `send(DatagramPacket pack)` — отправляет дейтаграмму, упакованную в пакет pack;
- `receive (DatagramPacket pack)` — дожидается получения дейтаграммы и заносит ее в пакет pack.

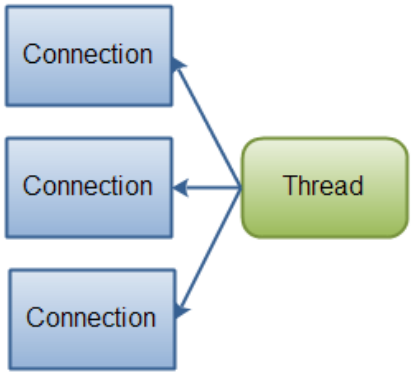
При обмене дейтаграммами соединение обычно не устанавливается, дейтаграммы посылаются наудачу, в расчете на то, что получатель ожидает их. Но можно установить соединение методом
При этом устанавливается только одностороннее соединение с хостом по адресу addr и номером порта port — или на отправку или на прием дейтаграмм. Потом соединение можно разорвать методом
При посылке дейтаграммы по протоколу JJDP сначала создается сообщение в виде массива байтов, например,
Потом записывается адрес — объект класса inetAddress
Затем сообщение упаковывается в пакет — объект класса **DatagramPacket**. При этом указывается массив данных, его длина, адрес и номер порта
Далее создается дейтаграммный сокет и дейтаграмма отправляется
После посылки всех дейтаграмм сокет закрывается, не дожидаясь какой-либо реакции со стороны получателя
Прием и распаковка дейтаграмм производится в обратном порядке, вместо метода send () применяется метод receive (DatagramPacket pack)
http://xprogramming.narod.ru/books/Java/Glava19/Index3.htm - эта же статья, но подробнее и с примерами

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

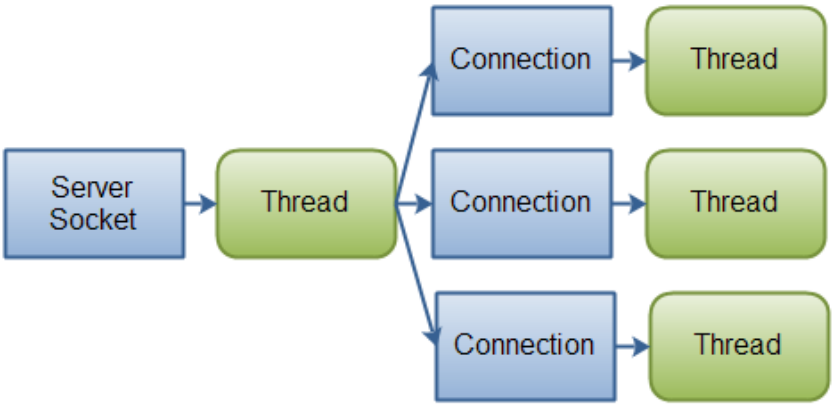
Потоки ввода/вывода (streams) в Java IO являются блокирующими. Это значит, что когда в потоке выполнения (tread) вызывается read() или write() метод любого класса из пакета java.io.*, происходит блокировка до тех пор, пока данные не будут считаны или записаны. Поток выполнения в данный момент не может 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.
Java NIO позволяет управлять несколькими каналами (сетевыми соединениями или файлами) используя минимальное число потоков выполнения. Однако ценой такого подхода является более сложный, чем при использовании блокирующих потоков, парсинг данных.

Если вам необходимо управлять тысячами открытых соединений одновременно, причем каждое из них передает лишь незначительный объем данных, выбор Java NIO для вашего приложения может дать преимущество. Дизайн такого типа схематически изображен на следующем рисунке:



Если вы имеете меньшее количество соединений, по которым передаются большие объемы данных, то лучшим выбором станет классический дизайн системы ввода/вывода:



https://habr.com/ru/post/235585/ - тут подробнее

5. Классы `SocketChannel` и `DatagramChannel`.

Каналы пришли на замену InputStream и OutputStream и имеют ряд существенных отличий от предшественников:
• Канал двунапрваленный – из него можно читать и одновременно в него писать.
• Каналы позволяют асинхронные чтение и запись.
• Канал пишет данные в буфер и читает из буфера.

Основные реализации каналов в java.nio:
• FileChannel – работает с файлами.
• **DatagramChannel** – передает данные по UDP.
• **SocketChannel** – передает данные по TCP.
• ServerSocketChannel – принимает входящие TCP соединения. На каждое новое соединение создается SocketChannel для дальнейшего общения с клиентом.

SocketChannel – это канал для работы с TCP соединениями. Как и обычный Socket, его можно создать вручную и привязать к удаленному серверу либо получить на сервере из метода `accept()`. Поскольку в nio конструкторов нет, вручную канал создается статическим методом `open()`.

```
SocketChannel channel = SocketChannel.open();
channel.connect(new InetSocketAddress("http://aptu.ru", 80));
```

Теперь мы можем читать и писать через буферы. Пример записи.

```
String newData = "New String to write to file..." + System.currentTimeMillis();
ByteBuffer buf = ByteBuffer.allocate(48);
buf.clear();
buf.put(newData.getBytes());
buf.flip();

while (buf.hasRemaining()) {
  channel.write(buf);
}
```

После использования канал, как и сокет и файл, надо закрыть методом close(). В неблокирующем режиме метод `connect()` завершается быстро, даже если еще не успел подключиться. Чтобы узнать, когда канал уже подключился, есть метод `finishConnect()`. Выглядит это так.

```
SocketChannel channel = SocketChannel.open();
channel.configureBlocking(false);
channel.connect(new InetSocketAddress("vk.com", 80));

while (!socket.finishConnect()) {
  // ждем или делаем еще что-то
}
```

Методы `read()` и `write()` в неблокирующем режиме могут прочитать/записать данные не до конца и завершиться. Поэтому их надо вызывать в цикле, пока они не прочитают/запишут столько, сколько нужно.

DatagramChannel – это канал для работы с UDP соединениями. Пример. Обратите внимание, что чтобы привязать канал к порту, надо вызвать метод `socket()`, хотя формально у UDP никаких сокетов нет.

```
DatagramChannel channel = DatagramChannel.open();
channel.socket().bind(new InetSocketAddress(9999));

ByteBuffer buf = ByteBuffer.allocate(48);
buf.clear();

channel.receive(buf);
....
String newData = "New String to write to file..." + System.currentTimeMillis();
buf.clear();
buf.put(newData.getBytes());
buf.flip();

int bytesSent = channel.send(buf, new InetSocketAddress("aptu.ru", 80));
```

Неблокирующий режим для DatagramChannel есть, но он бесполезен, так как, во-первых, пакеты могут

теряться, а во-вторых, все клиенты шлют пакеты на один порт, поэтому их все можно обрабатывать в одном потоке.

Ниже опять сокеты можно не читать

UDP-Socket
При создании сокета, ОС выделяет порт и закрепляет за этим сокетом.
Рассмотрим пример отправки UDP-запросов (клиент).

```
try (DatagramSocket s = new DatagramSocket()) {
  DatagramPacket p = new DatagramPacket(buf, buf.length, remoteAddress);
  s.send(p);
}
```

Создаем сокет и пакет, который хотим передать. В качестве параметров конструктор **DatagramPacket** принимает массив байтов, длину массива и адрес получателя. Адрес включает в себя IP-адрес и порт получателя. В нашем примере порт на нашем компьютере выбирается случайно. Если важно использовать конкретный порт, его номер можно передать в конструктор сокета. Теперь рассмотрим пример сервера.

```
try (DatagramSocket s = new DatagramSocket(port)) {
  byte[] buf = new byte [1024];
  DatagramPacket p = new DatagramPacket(buf, buf.length);
  s.receive(p);
}
```

Что будет, если придет пакет размера больше buf.length байтов? Тогда первые buf.length байта считаются, а остальные останутся в сокете, и их можно будет прочитать при следующем вызове receive(). Далее у пакета можно спросить много разной информации – кто отправил, какая длина, итд. В конце надо из полученных байтов как-то восстановить данные, которые мы ожидали получить.

TCP-Socket
В отличие от UDP, в TCP мы не создаем пакет, а отправляем данные в сокет, и он всё формирует и отправляет за нас. Рассмотрим процесс общения по TCP. Клиент хочет создать соединение с vk.com. Он посылает запрос на сервер, на 80 порт. Потом приходит второй клиент. TCP-соединение необходимо постоянно поддерживать, а значит нельзя со всеми клиентами общаться через один порт (иначе буфер, в который приходят запросы, будет переполняться). Поэтому, когда сервер получает запрос на 80 порт, он в ответ высылает номер другого порта, через который данный клиент должен продолжить общение с сервером. В UDP такой проблемы нет, поэтому там все клиенты общаются через один порт.
39 из 50 Java, второй семестр Сети в Java Рассмотрим пример использования TCP сокета (клиент).

```
Socket socket = new Socket (''localhost'', 11111);

OutputStream os = socket.getOutputStream();
os.write(requestBytes);
os.flush();

InputStream is = socket.getInputStream();
is.read(responseBytes);
```

Создаем сокет, передаем ему адрес и порт, с которыми соединяемся, запрашиваем у сокета OutputStream и InputStream. Метод flush() сбрасывает всё, что накопилось в буфере и пытается отправить это по сети.
Варианты конструктора для сокета:
• `Socket(String host, int port) throws UnknownHostException, IOException` – получает имя хоста и номер порта.
• `Socket(InetAddress host, int port) throws IOException` – то же самое, только вместо имени хоста – адрес.
• `Socket(String host, int port, InetAddress localAddress, int localPort) throws IOException` – дополнительно получает локальный адрес и порт. Локальный адрес бывает надо передать, если у компьютера несколько IP-адресов, так как у него есть разные сетевые интерфейсы. Такой сокет может слушать, например, только Wi-fi.

• `Socket(InetAddress host, int port, InetAddress localAddress, int localPort) throws IOException`.
• `Socket()` – неподключенный сокет. Чтобы его подключить, надо вызвать метод connect(). Поскольку мы не знаем, сколько байтов нам хотят прислать, есть соглашение первым делом отправлять количество данных, а затем сами данные. Рассмотрим пример серверного TCP-сокета.

```
ServerSocket server = new ServerSocket(11111);
Socket socket = server.accept();

InputStream is = socket.getInputStream();
is.read(requestBytes);

OutputStream os = socket.getOutputStream();
os.write(responseBytes);
os.flush();
```

ServerSocket – это как раз тот сокет, который не общается с клиентом, а отправляет его на другой сокет. Ему в качестве параметра передается порт, на который приходят запросы соединения. Далее блокирующий метод accept() ждет, пока не придет клиент, а потом создает и возвращает сокет для общения. 40 из 50 Java, второй семестр Сети в Java
Варианты конструктора серверного сокета:
• `ServerSocket(int port) throws IOException` – слушает заданный порт.
• `ServerSocket(int port, int backlog) throws IOException` – получает максимальное количество клиентов в очереди на соединение.
• `ServerSocket(int port, int backlog, InetAddress address) throws IOException` – слушает только запросы, поступившие на заданный адрес.
• `ServerSocket() throws IOException` – не привязанный сокет, привязывается методом `bind()`.

6. Передача данных по сети. Сериализация объектов.

Сериализация (Serialization) — это процесс, который переводит объект в последовательность байтов, по которой затем его можно полностью восстановить. Зачем это нужно? Дело в том, при обычном выполнении программы максимальный срок жизни любого объекта известен — от запуска программы до ее окончания. Сериализация позволяет расширить эти рамки и «дать жизнь» объекту так же между запусками программы.
Реализовать механизм сериализации довольно просто. Необходимо, чтобы ваш класс реализовывал интерфейс ***Serializable***. Это интерфейс — идентификатор, который не имеет методов, но он указывает jvm, что объекты этого класса могут быть сериализованы. Так как механизм сериализации связан с базовой системой ввода/вывода и переводит объект в поток байтов, для его выполнения необходимо создать выходной поток *OutputStream*, упаковать его в *ObjectOutputStream* и вызвать метод *writeObject()*. Для восстановления объекта нужно упаковать *InputStream* в *ObjectInputStream* и вызвать метод *readObject()*.

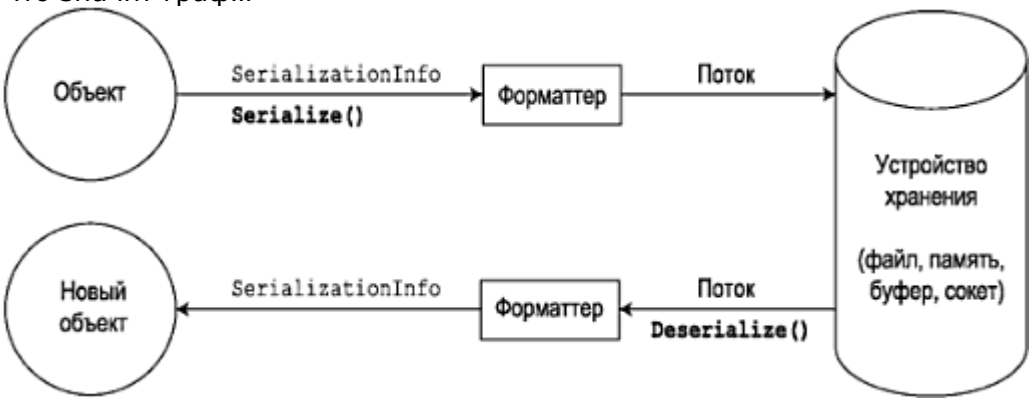
7. Интерфейс `Serializable`. Объектный граф, сериализация и десериализация полей и методов.

Serializable – см выше
При использовании **Serializable** применяется стандартный алгоритм сериализации, который с помощью рефлексии (Reflection API) выполняет

- запись в поток метаданных о классе, ассоциированном с объектом (имя класса, идентификатор SerialVersionUID, идентификаторы полей класса),
- рекурсивную запись в поток описания суперклассов до класса java.lang.Object (не включительно),
- запись примитивных значений полей сериализуемого экземпляра, начиная с полей самого верхнего суперкласса,
- рекурсивную запись объектов, которые являются полями сериализуемого объекта.

При этом ранее сериализованные объекты повторно не сериализуются, что позволяет алгоритму корректно работать с циклическими ссылками.
Для выполнения десериализации под объект выделяется память, после чего его поля заполняются значениями из потока. Конструктор объекта при этом не вызывается. Однако при десериализации будет вызван конструктор без параметров родительского несериализуемого класса, а его отсутствие повлечёт ошибку десериализации.

Что значит граф…



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

Все операции **Stream** делятся на промежуточные и терминальные операции и объединяются для формирования конвейеров потоков.
Конвейер потока состоит из источника (такого как Коллекция , массив, функция генератора, канал ввода-вывода или генератор бесконечной последовательности); за ним следуют ноль или более промежуточных операций и терминальная операция.
Виды и примеры операций тут -
https://github.com/drucoder/javalearn/blob/master/src/test/java/letscode/javalearn/Streams.java

Промежуточные Операции
Промежуточные операции не выполняются, вызывается некоторая терминальная операция.
Они состоят из конвейера выполнения Stream.
Все Промежуточные операции являются ленивыми, поэтому они не выполняются до тех пор, пока результат обработки действительно не понадобится.
В принципе, промежуточные операции возвращают новый поток. Выполнение промежуточной операции фактически не выполняет никакой операции, а вместо этого создает новый поток, который при обходе содержит элементы исходного потока, соответствующие данному предикату.
Как таковой, обход Течение не начинается до тех пор, пока терминал эксплуатация трубопровода выполнена.
Это очень важное свойство, особенно важное для бесконечных потоков, поскольку оно позволяет нам создавать потоки, которые будут фактически вызываться только при вызове операции Terminal .

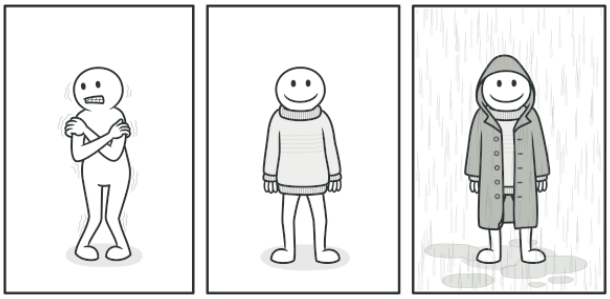
Терминальные операции
Операции терминала могут пересекать поток для получения результата или побочного эффекта.

После выполнения терминальной операции потоковый трубопровод считается израсходованным и больше не может использоваться. Почти во всех случаях терминальные перации выполняются с нетерпением, завершая обход источника данных и обработку конвейера перед возвращением.
Рвение терминальной операции важно в отношении бесконечных потоков , потому что в момент обработки нам нужно тщательно подумать, правильно ли наш поток ограничен , например, преобразованием limit () . Каждая из терминальная операция вызовет выполнение всех промежуточных операций.

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

Чтобы открыть ссылки включи впн😆😋

Декоратор — это структурный паттерн проектирования, который позволяет динамически добавлять объектам новую функциональность, оборачивая их в полезные «обёртки». ПРИМЕР ИЗ ЖИЗНИ Одежду можно надевать слоями, получая комбинированный эффект.



⚖ Преимущества и недостатки

- ✓ Большая гибкость, чем у наследования.
- ✓ Позволяет добавлять обязанности на лету.
- ✓ Можно добавлять несколько новых обязанностей сразу.
- ✓ Позволяет иметь несколько мелких объектов вместо одного объекта на все случаи жизни.
- ✗ Трудно конфигурировать многократно обёрнутые объекты.
- ✗ Обилие крошечных классов.

https://refactoring.guru/ru/design-patterns/decorator

Итератор — это поведенческий паттерн проектирования, который даёт возможность последовательно обходить элементы составных объектов, не раскрывая их внутреннего представления. ПРИМЕР ИЗ ЖИЗНИ Ты приехал в Рим и хочешь обойти все достопримечательности. Но приехав, вы можете долго петлять узкими улочками, пытаясь найти Колизей. Если у вас ограниченный бюджет — не беда. Вы можете воспользоваться виртуальным гидом, скачанным на телефон, который позволит отфильтровать только интересные вам точки. А можете плюнуть и нанять локального гида, который хоть и обойдётся в копеечку, но знает город как свои пять пальцев, и сможет посвятить вас во все городские легенды. Таким образом, Рим выступает коллекцией достопримечательностей, а ваш мозг, навигатор или гид — итератором по коллекции. Вы, как клиентский код, можете выбрать один из итераторов, отталкиваясь от решаемой задачи и доступных ресурсов.

⚖ Преимущества и недостатки

- ✓ Упрощает классы хранения данных.
- ✓ Позволяет реализовать различные способы обхода структуры данных.
- ✓ Позволяет одновременно перемещаться по структуре данных в разные стороны.
- ✗ Не оправдан, если можно обойтись простым циклом.

https://refactoring.guru/ru/design-patterns/iterator

Фабричный метод — это порождающий паттерн проектирования, который определяет общий интерфейс для создания объектов в суперклассе, позволяя подклассам изменять тип

создаваемых объектов.

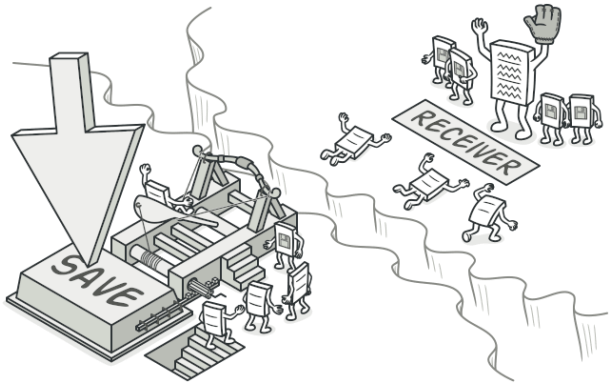
Преимущества и недостатки

- ✓ Избавляет класс от привязки к конкретным классам продуктов.
- ✓ Выделяет код производства продуктов в одно место, упрощая поддержку кода.
- ✓ Упрощает добавление новых продуктов в программу.
- ✓ Реализует *принцип открытости/закрытости.*
- ✗ Может привести к созданию больших **параллельных иерархий классов**, так как для каждого класса продукта надо создать свой подкласс создателя.

https://refactoring.guru/ru/design-patterns/factory-method

Команда — это поведенческий паттерн проектирования, который превращает запросы в объекты, позволяя передавать их как аргументы при вызове методов, ставить запросы в очередь, логировать их, а также поддерживать отмену операций. ПРИМЕР ИЗ ЖИЗНИ Вы заходите в ресторан и садитесь у окна. К вам подходит вежливый официант и принимает заказ, записывая все пожелания в блокнот. Откланявшись, он уходит на кухню, где вырывает лист из блокнота и клеит на стену. Далее лист оказывается в руках повара, который читает содержание заказа и готовит заказанные блюда. В этом примере вы являетесь *отправителем*, официант с блокнотом — *командой*, а повар — *получателем*. Как и в паттерне, вы не соприкасаетесь напрямую с поваром. Вместо этого вы отправляете заказ с официантом, который самостоятельно «настраивает» повара на работу. С другой стороны, повар не знает, кто конкретно послал ему заказ. Но это ему безразлично, так как вся необходимая информация есть в листе заказа.

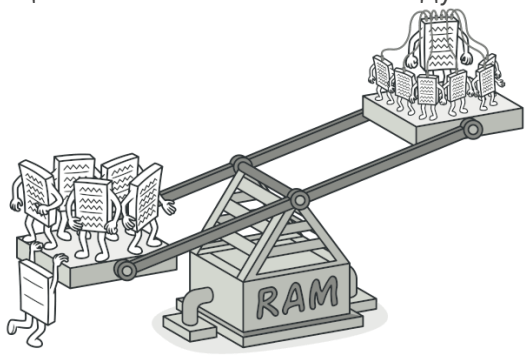
Преимущества и недостатки



- ✓ Убирает прямую зависимость между объектами, вызывающими операции, и объектами, которые их непосредственно выполняют.
- ✓ Позволяет реализовать простую отмену и повтор операций.
- ✓ Позволяет реализовать отложенный запуск операций.
- ✓ Позволяет собирать сложные команды из простых.
- ✓ Реализует *принцип открытости/закрытости.*
- ✗ Усложняет код программы из-за введения множества дополнительных классов.

https://refactoring.guru/ru/design-patterns/command

Легковес — это структурный паттерн проектирования, который позволяет вместить бóльшее количество объектов в отведённую оперативную память. Легковес экономит память, разделяя общее состояние объектов между собой, вместо хранения одинаковых данных в каждом объекте.



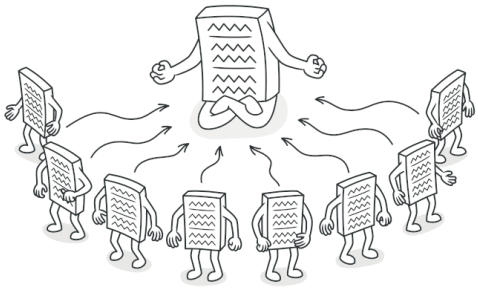
Преимущества и недостатки

- ✓ Экономит оперативную память.
- ✗ Расходует процессорное время на поиск/вычисление контекста.
- ✗ Усложняет код программы из-за введения множества дополнительных классов.

https://refactoring.guru/ru/design-patterns/flyweight

Интерпретатор — это поведенческий паттерн. Для заданного языка определяет представление его грамматики, а также интерпретатор предложений этого языка. (я ничего не поняла)

Одиночка — это порождающий паттерн проектирования, который гарантирует, что у класса есть только один экземпляр, и предоставляет к нему глобальную точку доступа. ПРИМЕР ИЗ ЖИЗНИ Правительство государства — хороший пример одиночки. В государстве может быть только одно официальное правительство. Вне зависимости от того, кто конкретно заседает в правительстве, оно имеет глобальную точку доступа «Правительство страны N».



Преимущества и недостатки

- ✓ Гарантирует наличие единственного экземпляра класса.
- ✓ Предоставляет к нему глобальную точку доступа.
- ✓ Реализует отложенную инициализацию объекта-одиночки.
- ✗ Нарушает *принцип единственной ответственности класса.*
- ✗ Маскирует плохой дизайн.
- ✗ Проблемы мультипоточности.
- ✗ Требует постоянного создания Mock-объектов при юнит-тестировании.

https://refactoring.guru/ru/design-patterns/singleton

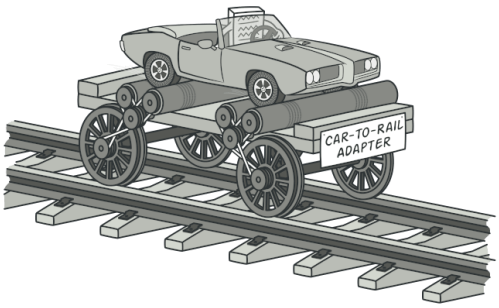
Стратегия — это поведенческий паттерн проектирования, который определяет семейство схожих алгоритмов и помещает каждый из них в собственный класс, после чего алгоритмы можно взаимозаменять прямо во время исполнения программы. ПРИМЕР ИЗ ЖИЗНИ Вам нужно добраться до аэропорта. Можно доехать на автобусе, такси или велосипеде. Здесь вид транспорта является стратегией. Вы выбираете конкретную стратегию в зависимости от контекста — наличия денег или времени до отлёта.

Преимущества и недостатки

- ✓ Горячая замена алгоритмов на лету.
- ✓ Изолирует код и данные алгоритмов от остальных классов.
- ✓ Уход от наследования к делегированию.
- ✓ Реализует *принцип открытости/закрытости.*
- ✗ Усложняет программу за счёт дополнительных классов.
- ✗ Клиент должен знать, в чём состоит разница между стратегиями, чтобы выбрать подходящую.

https://refactoring.guru/ru/design-patterns/strategy

Адаптер — это структурный паттерн проектирования, который позволяет объектам с несовместимыми интерфейсами работать вместе. ПРИМЕР ИЗ ЖИЗНИ Когда вы в первый раз летите за границу, вас может ждать сюрприз при попытке зарядить ноутбук. Стандарты розеток в разных странах отличаются. Ваша европейская зарядка будет бесполезна в США без специального адаптера, позволяющего подключиться к розетке другого типа.

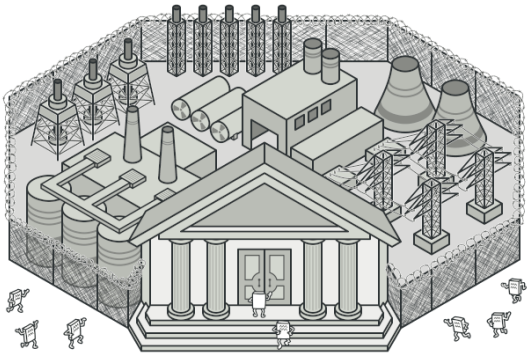


Преимущества и недостатки

- ✓ Отделяет и скрывает от клиента подробности преобразования различных интерфейсов.
- ✗ Усложняет код программы из-за введения дополнительных классов.

https://refactoring.guru/ru/design-patterns/adapter

Фасад — это структурный паттерн проектирования, который предоставляет простой интерфейс к сложной системе классов, библиотеке или фреймворку. ПРИМЕР ИЗ ЖИЗНИ Когда вы звоните в магазин и делаете заказ по телефону, сотрудник службы поддержки является вашим фасадом ко всем службам и отделам магазина. Он предоставляет вам упрощённый интерфейс к системе создания заказа, платёжной системе и отделу доставки



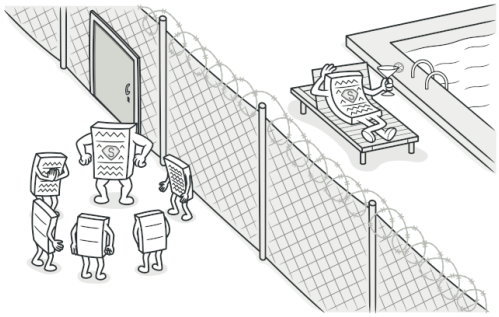
⚖ Преимущества и недостатки

✓ Изолирует клиентов от компонентов сложной подсистемы.

✗ Фасад рискует стать **божественным объектом**, привязанным ко всем классам программы.

Боже́ственный объе́кт (англ. God object) — антипаттерн объектно-ориентированного программирования, описывающий объект, который хранит в себе «слишком много» или делает «слишком много».
https://refactoring.guru/ru/design-patterns/facade

Заместитель — это структурный паттерн проектирования, который позволяет подставлять вместо реальных объектов специальные объекты-заменители. Эти объекты перехватывают вызовы к оригинальному объекту, позволяя сделать что-то *до* или *после* передачи вызова оригиналу. ПРИМЕР ИЗ ЖИЗНИ Платёжная карточка — это заместитель пачки наличных. И карточка, и наличные имеют общий интерфейс — ими можно оплачивать товары. Для покупателя польза в том, что не надо таскать с собой тонны наличных, а владелец магазина рад, что ему не нужно делать дорогостоящую инкассацию наличности в банк — деньги поступают к нему на счёт напрямую.



⚖ Преимущества и недостатки

✓ Позволяет контролировать сервисный объект незаметно для клиента.

✓ Может работать, даже если сервисный объект ещё не создан.

✓ Может контролировать жизненный цикл служебного объекта.

✗ Усложняет код программы из-за введения дополнительных классов.

✗ Увеличивает время отклика от сервиса.

https://refactoring.guru/ru/design-patterns/proxy

10. log4j

log4j – это надежная, быстрая и гибкая среда ведения журналов (API), написанная на Java. log4j легко настраивается с помощью внешних файлов конфигурации во время выполнения. Он рассматривает процесс ведения журнала с точки зрения уровней приоритетов и предлагает механизмы для направления информации регистрации в самые разные пункты назначения, такие как база данных, файл, консоль, системный журнал UNIX ..

log4j состоит из трех основных компонентов:
Регистраторы : Ответственный за сбор информации журнала.
appenders : Отвечает за публикацию информации о регистрации в различных предпочтительных местах назначения.
макеты : отвечает за форматирование информации журнала в разных стилях.

Регистраторы : Ответственный за сбор информации журнала.
appenders : Отвечает за публикацию информации о регистрации в различных предпочтительных местах назначения.
макеты : отвечает за форматирование информации журнала в разных стилях.

Особенности log4j
-Это потокобезопасно.

-Это оптимизировано для скорости.
-Он основан на именованной иерархии логгеров.
-Он поддерживает несколько выходных приложений для каждого регистратора.
-Это не ограничено предопределенным набором средств.
-Поведение ведения журнала может быть установлено во время выполнения с помощью файла конфигурации.
-Он предназначен для обработки исключений Java с самого начала.
-Он использует несколько уровней, а именно ALL, TRACE, DEBUG, INFO, WARN, ERROR и FATAL.
-Формат вывода журнала можно легко изменить, расширив класс *Layout* .
-Цель вывода журнала, а также стратегия записи могут быть изменены реализациями интерфейса Appender.
-Это аварийный останов. Однако, хотя он, безусловно, стремится обеспечить доставку, log4j не гарантирует, что каждый оператор журнала будет доставлен по назначению.

Про уязвимость тут - https://habr.com/ru/company/acronis/blog/598473/