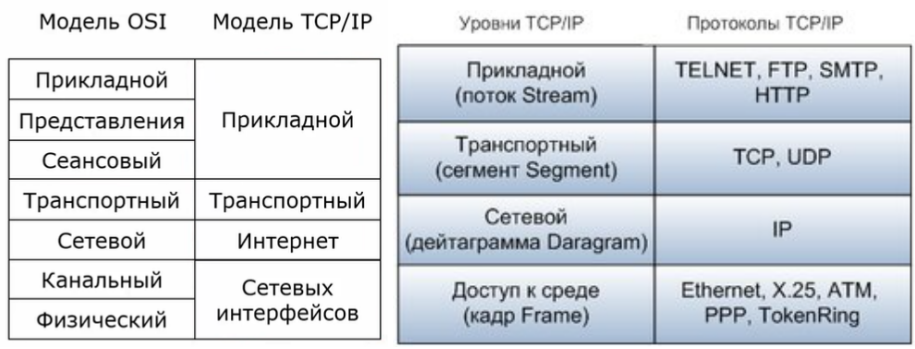
- **Сетевое взаимодействие - клиент-серверная архитектура, основные протоколы, их сходства и отличия.**

OSI - эталонный стандарт сетевого взаимодействия

  - 1) Физический уровень (Physical Layer): определяет метод передачи данных, какая среда используется (передача электрических сигналов, световых импульсов или радиоэфир), уровень напряжения, метод кодирования двоичных сигналов.
  - 2) Канальный уровень (Data Link Layer): он берет на себя задачу адресации в пределах локальной сети, обнаруживает ошибки, проверяет целостность данных. Если слышали про MAC-адреса и протокол «Ethernet», то они располагаются на этом уровне.
  - 3) Сетевой уровень (Network Layer): этот уровень берет на себя объединения участков сети и выбор оптимального пути (т.е. маршрутизация). Каждое сетевое устройство должно иметь уникальный сетевой адрес в сети. Думаю, многие слышали про протоколы IPv4 и IPv6. Эти протоколы работают на данном уровне.
  - 4) Транспортный уровень (Transport Layer): Этот уровень берет на себя функцию транспорта. К примеру, когда вы скачиваете файл с Интернета, файл в виде сегментов отправляется на Ваш компьютер. Также здесь вводятся понятия портов, которые нужны для указания назначения к конкретной службе. На этом уровне работают протоколы TCP (с установлением соединения) и UDP (без установления соединения).
  - 5) Сеансовый уровень (Session Layer): Роль этого уровня в установлении, управлении и разрыве соединения между двумя хостами. К примеру, когда открываете страницу на веб-сервере, то Вы не единственный посетитель на нем. И вот для того, чтобы поддерживать сеансы со всеми пользователями, нужен сеансовый уровень.
  - 6) Уровень представления (Presentation Layer): Он структурирует информацию в читабельный вид для прикладного уровня. Например, многие компьютеры используют таблицу кодировки ASCII для вывода текстовой информации или формат jpeg для вывода графического изображения.
  - 7) Прикладной уровень (Application Layer): Наверное, это самый понятный для всех уровень. Как раз на этом уроне работают привычные для нас приложения — e-mail, браузеры по протоколу HTTP, FTP и остальное.

TCP/IP



- **Протокол TCP. Классы Socket и ServerSocket.**

  Адрес подразумевает под собой идентификатор машины в пространстве сети Internet. Он может быть доменным именем, 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.

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

  Class Socket

  Класс Socket реализует идею сокета. Через его каналы ввода/вывода будут общаться клиент с сервером. Объявляется этот класс на стороне клиента, а сервер воссоздаёт его, получая сигнал на подключение.

  – Конструкторы
    * Socket(String имя хоста, int порт)
    * Socket(InetAddress IP-адрес, int порт)

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

    Порт — есть порт. Если в качестве номера порта будет указан 0, то система сама выделит свободный порт.

    Следует отметить тип адреса во втором конструкторе — InetAddress. Он приходит на помощь, например, когда нужно указать в качестве адреса доменное имя. Так же когда под доменом подразумевается несколько ip-адресов, то с помощью InetAddress можно получить их массив.

    При инициализации объекта типа Socket, клиент, которому тот принадлежит, объявляет в сети, что хочет соединиться с сервером про определённому адресу и номеру порта

  – Основные методы
    * 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

Class ServerSocket

Предоставляет механизм серверной программы для прослушивания клиентов и установления соединений с ними.

- Конструкторы
    * ServerSocket()
    * ServerSocket(int порт)
    * ServerSocket(int порт, int максимум подключений)
    * ServerSocket(int порт, int максимум подключений, InetAddress локальный адрес)
- Основные методы
    * Socket accept() Ожидание подключения клиента
    * void bind(SocketAddress endpoint) Связывание ServerSocket с определенным адресом (IP-адрес и порт)
    * void close() Закрытие сокета
    * ServerSocketChannel getChannel() Получение объекта ServerSocketChannel, связанного с сокетом
    * InetAddress getInetAddress() Получение локального адреса сокета сервера int getLocalPort() Получение номера порта, который серверный сокет слушает SocketAddress
    * getLocalSocketAddress() Получение адреса серверного сокета в виде объекта SocketAddress int getReceiveBufferSize() Получение размера буфера серверного сокета
    * boolean isClosed() Проверка, закрыт ли серверный сокет
    * void setReceiveBufferSize(int size) Определение размера буфера серверного сок

- **Протокол UDP. Классы DatagramSocket и DatagramPacket.**

Class DatagramSocket

-это точка отправки или получения для службы доставки пакетов. Каждый пакет, отправленный или полученный через сокет дейтаграммы, адресуется и маршрутизируется индивидуально.

- Конструкторы
  - public DatagramSocket(int port);
  - public DatagramSocket();
- Прием и передача данных на датаграммном сокете выполняется с помощью методов receive и send, соответственно:
  - public void receive(DatagramPacket p);
  - public void send(DatagramPacket p);

В качестве параметра этим методам передается ссылка на пакет данных (соответственно, принимаемый и передаваемый), определенный как объект класса DatagramPacket.

DatagramPacket

-используются для реализации службы доставки пакетов без установления соединения. Каждое сообщение перенаправляется с одной машины на другую исключительно на основе информации, содержащейся в этом пакете. Множественные пакеты, отправленные с одного компьютера на другой, могут маршрутизироваться по-разному и приходить в любом порядке. Доставка пакетов не гарантируется.

- Конструкторы
  - DatagramPacket(byte[] buf, int length)
    Создает a DatagramPacketдля приема пакетов большой длины length.
  - DatagramPacket(byte[] buf, int length, InetAddress address, int port)
    Создает пакет дейтаграммы для отправки пакетов определенной длины lengthна указанный номер порта на указанном хосте.
  - DatagramPacket(byte[] buf, int offset, int length)
    Создает a DatagramPacketдля приема пакетов длины length, определяя смещение в буфере.
  - DatagramPacket(byte[] buf, int offset, int length, InetAddress address, int port)
    Создает пакет дейтаграммы для отправки пакетов длины lengthсо смещением ioffsetна указанный номер порта на указанном хосте.
  - DatagramPacket(byte[] buf, int offset, int length, SocketAddress address)
    Создает пакет дейтаграммы для отправки пакетов длины lengthсо смещением ioffset на указанный номер порта на указанном хосте.

* DatagramPacket(byte[] buf, int length, SocketAddress address)
  Создает пакет дейтаграммы для отправки пакетов определенной длины lengthна указанный номер порта на указанном хосте.

Таким образом, информация о том, в какой узел и на какой порт необходимо доставить пакет данных, хранится не в сокете, а в пакете, то есть в объекте класса DatagramPacket.

Помимо только что описанных конструкторов, в классе DatagramPacket определены четыре метода, позволяющие получить данные и информацию об адресе узла, из которого пришел пакет, или для которого предназначен пакет.

– Методы
  * Метод getData возвращает ссылку на массив данных пакета:
  * Размер пакета, данные из которого хранятся в этом массиве, легко определить с помощью метода getLength:
  * Методы getAddress и getPort позволяют определить адрес и номер порта узла, откуда пришел пакет, или узла, для которого предназначен пакет: связи.

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

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. Наиболее частой практикой является использование сэкономленного времени работы потока выполнения на обслуживание операций ввода/вывода в другом или других каналах.

Работа с сетевыми каналами

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. Тоесть Вы проверяете каналы сокета с помощью Selector и указываете какую операцию вы запрашивали. Затем у вас есть поток, который ждет Selector пока не поступит одна из регистрируемых операций. Поток «узнает», когда такая операция поступила и может ее выполнить, а потом будет ждать следующие операции. Посмотрим каждый шаг. Чтобы создать объект Selector, вы вызываете метод open (не вызывая конструктор класса):

- **Классы SocketChannel и DatagramChannel.**

  Канал представляет открытое соединение с источником или адресатом ввода-вывода. Классы каналов реализуют интерфейс Channel, расширяющий интерфейс Closeable, а также интерфейс AutoCloseable.

  Конкретный тип возвращаемого канала зависит от типа объекта, для которого вызывается метод getChannel (). Соответственно при вызове данного метода для объектов типа Socket, он возвращает канал типа SocketChannel и соответственно datagram - DatagramChanel.

  Все каналы померживают дополнительные методы, предоставляющие доступ к каналу и позволяющие управлять им. Например, канал типа SocketChannel поддерживает среди прочего методы для получения и установки текущей позиции, передачи данных между файловыми каналами, получения текущего размера канала и его блокировки. В классе SocketChannel предоставляется статический метод open (), который открывает файл и возвращает для него канал. Такой результат достигается другим способом получения канала. В классе SocketChannel предоставляется также метод map (),с помощью которого можно отобразить файл в буфер.

- **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.

- **Интерфейс Serializable. Объектный граф, сериализация и десериализация полей и методов**

  Java Serialization API предоставляет стандартный механизм для создания сериализуемых объектов. Процесс сериализации заключается в сериализации каждого поля объекта, но только в том случае, если это поле не имеет спецификатора static или transient. Поля, помеченные ими не могут быть предметом сериализации.

  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.

Методы Serializable

- Метод writeObject отвечает за запись состояния объекта для его конкретного класса, чтобы соответствующий метод readObject мог его восстановить. Механизм по умолчанию для сохранения полей объекта можно вызвать, вызвав out.defaultWriteObject. Метод не должен заботиться о состоянии, принадлежащем его суперклассам или подклассам. Состояние сохраняется путем записи отдельных полей в ObjectOutputStream с использованием метода writeObject или с помощью методов для примитивных типов данных, поддерживаемых DataOutput.
- Метод readObject отвечает за чтение из потока и восстановление полей классов. Он может вызвать in.defaultReadObject для вызова механизма по умолчанию для восстановления нестатических и непереходных полей объекта. Метод defaultReadObject использует информацию в потоке для назначения полей объекта, сохраненного в потоке, соответствующим именованным полям в текущем объекте. Это относится к случаю, когда класс эволюционировал, чтобы добавить новые поля. Метод не должен заботиться о состоянии, принадлежащем его суперклассам или подклассам. Состояние сохраняется путем записи отдельных полей в ObjectOutputStream с использованием метода writeObject или с помощью методов для примитивных типов данных, поддерживаемых DataOutput.
- Метод readObjectNoData отвечает за инициализацию состояния объекта для его конкретного класса в том случае, если поток сериализации не перечисляет данный класс как суперкласс десериализуемого объекта. Это может происходить в тех случаях, когда принимающая сторона использует другую версию класса десериализованного экземпляра, чем отправляющая сторона, а версия получателя расширяет классы, которые не расширяются версией отправителя.

- **Java Stream API. Создание конвейеров. Промежуточные и терминальные операции.**

  Stream - Поток - это «последовательность элементов из источника, поддерживающая агрегированные операции». Давайте разберемся:

  - Последовательность элементов: поток предоставляет интерфейс для упорядоченного набора значений определенного типа элемента. Однако потоки на самом деле не хранят элементы; они вычисляются по запросу.

- Источник: потоки потребляют из источника, предоставляющего данные, такого как коллекции, массивы или ресурсы ввода-вывода.
- Совокупные операции: Потоки поддержка SQL-подобные операциям и общие операциям с функциональных языков программирования, такие как filter, map, reduce, find, match, sorted, и так далее.

Разница между коллекциями и потоками связана с тем, когда что-то вычисляется. Коллекция - это структура данных в памяти, которая содержит все значения, которые структура данных в настоящее время имеет - каждый элемент в коллекции должен быть вычислен, прежде чем его можно будет добавить в коллекцию. Напротив, поток - это концептуально фиксированная структура данных, в которой элементы вычисляются по запросу.

Методы Stream

- Потоковые операции, которые могут быть связаны, называются промежуточными операциями . Их можно соединить вместе, потому что их возвращаемый тип - это Stream.
  - filter - Принимает предикат в качестве аргумента и возвращает поток, включающий все элементы, соответствующие данному предикату. - filter(t -> t.getType() == Transaction.GROCERY)
  - distinct - Возвращает поток с уникальными элементами
  - any match - шаблон обработки данных определяет, соответствуют ли некоторые элементы заданному свойству
  - Find first - для извлечения произвольных элементов из потока
  - Map - принимает в качестве аргумента функцию () для проецирования элементов потока в другую форму. Функция применяется к каждому элементу, «сопоставляя» его с новым элементом. map(String::length)
  - sorted
  - map
- Операции, закрывающие конвейер потока, называются терминальными операциями . Они производят результат из конвейера, такого как a List
  - collect

Конвейерная обработка: многие операции с потоками сами возвращают поток. Это позволяет объединять операции в более крупный конвейер. Это обеспечивает определенные оптимизации, такие как лень и короткое замыкание.

- Короткое замыкание -
- Ленивые - промежуточные операции не выполняют никакой обработки до тех пор, пока терминальная операция не будет вызвана в конвейере потока; они ленивые."Это связано с тем, что промежуточные операции обычно могут быть «объединены» и обработаны за один проход операцией терминала.

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

  - **Decorator** позволяет динамически добавлять объектам новую функциональность, оборачивая их в полезные «обёртки». Декоратор имеет альтернативное название — обёртка. Оно более точно описывает суть паттерна: вы помещаете целевой объект в другой объект-обёртку, который запускает базовое поведение объекта, а затем добавляет к результату что-то своё.

    Оба объекта имеют общий интерфейс, поэтому для пользователя нет никакой разницы, с каким объектом работать — чистым или обёрнутым. Вы можете использовать несколько разных обёрток одновременно — результат будет иметь объединённое поведение всех обёрток сразу. - Блины с добавками в Теремке

  - **Iterator** - даёт возможность последовательно обходить элементы составных объектов, не раскрывая их внутреннего представления.

    Идея паттерна Итератор состоит в том, чтобы вынести поведение обхода коллекции из самой коллекции в отдельный класс. Объект-итератор будет отслеживать состояние обхода, текущую позицию в коллекции и сколько элементов ещё осталось обойти. Одну и ту же коллекцию смогут одновременно обходить различные итераторы, а сама коллекция не будет даже знать об этом.

    К тому же, если вам понадобится добавить новый способ обхода, вы сможете создать отдельный класс итератора, не изменяя существующий код коллекции.

  - **Factory method** - определяет общий интерфейс для создания объектов в суперклассе, позволяя подклассам изменять тип создаваемых объектов.

    Паттерн Фабричный метод предлагает создавать объекты не напрямую, используя оператор new, а через вызов особого фабричного метода. Не пугайтесь, объекты всё равно будут создаваться при помощи new, но делать это будет фабричный метод.

    На первый взгляд, это может показаться бессмысленным: мы просто переместили вызов конструктора из одного конца программы в другой. Но теперь вы сможете переопределить фабричный метод в подклассе, чтобы изменить тип создаваемого продукта.

    Чтобы эта система заработала, все возвращаемые объекты должны иметь общий интерфейс. Подклассы смогут производить объекты различных классов, следующих одному и тому же интерфейсу.

  - **Command** - превращает запросы в объекты, позволяя передавать их как аргументы при вызове методов, ставить запросы в очередь, логировать их, а также поддерживать отмену операций.

    В реальности это выглядит так: один из объектов интерфейса напрямую вызывает метод одного из объектов бизнес-логики, передавая в него какие-то параметры. Паттерн Команда предлага-

ет больше не отправлять такие вызовы напрямую. Вместо этого каждый вызов, отличающийся от других, следует завернуть в собственный класс с единственным методом, который и будет осуществлять вызов. Такие объекты называют командами.

К объекту интерфейса можно будет привязать объект команды, который знает, кому и в каком виде следует отправлять запросы. Когда объект интерфейса будет готов передать запрос, он вызовет метод команды, а та — позаботится обо всём остальном.

- **Flyweight** - 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.

  Паттерн Легковес предлагает не хранить в классе внешнее состояние, а передавать его в те или иные методы через параметры. Таким образом, одни и те же объекты можно будет повторно использовать в различных контекстах. Но главное — понадобится гораздо меньше объектов, ведь теперь они будут отличаться только внутренним состоянием, а оно имеет не так много вариаций.

- **Interpreter**

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

  Все реализации одиночки сводятся к тому, чтобы скрыть конструктор по умолчанию и создать публичный статический метод, который и будет контролировать жизненный цикл объекта-одиночки.

  Если у вас есть доступ к классу одиночки, значит, будет доступ и к этому статическому методу. Из какой точки кода вы бы его ни вызвали, он всегда будет отдавать один и тот же объект.

  * Ограничение количества экземпляров класса
  * private конструктор
  * 1 объект - Singleton
  * $>$ 1 объекта — Object Pool

- **Strategy** - определяет семейство схожих алгоритмов и помещает каждый из них в собственный класс, после чего алгоритмы можно взаимозаменять прямо во время исполнения программы.

  Вместо того, чтобы изначальный класс сам выполнял тот или иной алгоритм, он будет играть роль контекста, ссылаясь на одну из стратегий и делегируя ей выполнение работы. Чтобы сменить алгоритм, вам будет достаточно подставить в контекст другой объект-стратегию.

  Важно, чтобы все стратегии имели общий интерфейс. Используя этот интерфейс, контекст будет независимым от конкретных классов стратегий. С другой стороны, вы сможете изменять и добавлять новые виды алгоритмов, не трогая код контекста.

- **Adapter** позволяет объектам с несовместимыми интерфейсами работать вместе.

Вы можете создать адаптер. Это объект-переводчик, который трансформирует интерфейс или данные одного объекта в такой вид, чтобы он стал понятен другому объекту.

При этом адаптер оборачивает один из объектов, так что другой объект даже не знает о наличии первого. Например, вы можете обернуть объект, работающий в метрах, адаптером, который бы конвертировал данные в футы.

– **Facade** - предоставляет простой интерфейс к сложной системе классов, библиотеке или фреймворку.

  Фасад — это простой интерфейс для работы со сложной подсистемой, содержащей множество классов. Фасад может иметь урезанный интерфейс, не имеющий 100

  Фасад полезен, если вы используете какую-то сложную библиотеку со множеством подвижных частей, но вам нужна только часть её возможностей.

– **Proxy** - 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.

  Паттерн Заместитель предлагает создать новый класс-дублёр, имеющий тот же интерфейс, что и оригинальный служебный объект. При получении запроса от клиента объект-заместитель сам бы создавал экземпляр служебного объекта и переадресовывал бы ему всю реальную работу.

  Но в чём же здесь польза? Вы могли бы поместить в класс заместителя какую-то промежуточную логику, которая выполнялась бы до (или после) вызовов этих же методов в настоящем объекте. А благодаря одинаковому интерфейсу, объект-заместитель можно передать в любой код, ожидающий сервисный объект.