

Что такое ООП	<p><b>ООП</b> - методология программирования, основанная на представлении программы в виде совокупности объектов, каждый из которых является экземпляром определенного класса, а классы образуют иерархию наследования.</p> <p>Согласно парадигме ООП программа состоит из объектов, обменивающихся сообщениями. Объекты могут обладать состоянием, единственный способ изменить состояние объекта - передать ему сообщение, в ответ на которое, объект может изменить собственное состояние.</p> <p>Класс — это описание еще не созданного объекта, как бы общий шаблон, состоящий из полей, методов и конструктора, а объект – экземпляр класса, созданный на основе этого описания.</p>
Какие преимущества у ООП	<p><b>Легко читается</b> - не нужно выискивать в коде функции и выяснять, за что они отвечают</p> <p><b>Быстро пишется</b> - можно быстро создать сущности, с которыми должна работать программа.</p> <p><b>Простота реализации большого функционала</b> - т.к. на написание кода уходит меньше времени, можно гораздо быстрее создать приложение с множеством возможностей</p> <p><b>Меньше повторений кода</b> - не нужно писать однотипные функции для разных сущностей</p>
Какие недостатки у ООП	<p><b>Меньше повторений кода</b> - не нужно писать однотипные функции для разных сущностей</p> <p><b>Снижает производительность</b> - многие вещи технически реализованы иначе, поэтому они используют больше ресурсов.</p> <p><b>Сложно начать</b> - парадигма ООП сложнее функционального программирования, поэтому на старт уходит больше времени</p>
Назовите основные принципы ООП	<p><b>Инкапсуляция</b>  <b>Наследование</b>  <b>Полиморфизм</b></p>
Что такое инкапсуляция? (С примером)	<p>Свойство системы, которое объединяет данные и методы, манипулирующие этими данными, а также защищает и то, и другое от внешнего вмешательства или неправильного использования. Инкапсуляция - это объединение данных и методов работы с этими данными в одной упаковке («капсуле»). Чтобы малейшее изменение в классе не влекло за собой изменение внешнего поведения класса</p>
Что такое наследование? (С примером)	<p><b>Свойство</b> системы, которое <b>позволяет описать</b> новый класс на основе уже существующего с частично или полностью заимствованной функциональностью.</p>
Что такое полиморфизм? (С примером)	<p>Полиморфизм – это свойство системы использовать объекты с одинаковым интерфейсом без информации о типе и внутренней структуре объекта.</p> <p>Преимуществом полиморфизма является то, что он помогает снижать сложность программ, разрешая использование одного и того же интерфейса для задания единого набора действий. Выбор же конкретного действия, в зависимости от ситуации, возлагается на компилятор языка программирования. Отсюда следует ключевая особенность полиморфизма - использование объекта производного класса, вместо объекта базового (потомки могут изменять родительское поведение, даже если обращение к ним будет производиться по ссылке родительского типа).</p> <p>Полиморфизм бывает динамическим (переопределение) и статическим (перегрузка).</p> <p>Полиморфная переменная, это переменная, которая может принимать значения разных типов, а полиморфная функция, это функция у которой хотя бы один аргумент является полиморфной переменной. Выделяют два вида полиморфных функций:</p> <p>ad hoc, функция ведет себя по разному для разных типов аргументов (например, функция draw() — рисует по разному фигуры разных типов); параметрический, функция ведет себя одинаково для аргументов разных типов (например, функция add() — одинаково кладет в контейнер элементы разных типов).</p>

<p>Что такое ассоциация</p>	<p><u>Есть два типа связи между объектами: ассоциация, которая делится на композицию и агрегацию, и наследование. Ассоциация - обозначает связь между объектами. Например, игрок играет в определенной команде.</u></p> <p><u>Ассоциация означает, что объекты двух классов могут ссылаться один на другой, иметь некоторую связь между друг другом. Например Менеджер может выписать Счет. Соответственно возникает ассоциация между Менеджером и Счетом. Еще пример — Преподаватель и Студент — т.е. какой-то Студент учится у какого-то Преподавателя. Ассоциация и есть описание связи между двумя объектами. Студент учится у Преподавателя. Идея достаточно простая — два объекта могут быть связаны между собой и это надо как-то описать.</u></p> <p><a href="http://java-course.ru/begin/relations/">http://java-course.ru/begin/relations/</a></p>
<p>Что такое композиция</p>	<p><b>Композиция</b> — еще более «жесткое отношение, когда объект не только является частью другого объекта, но и вообще не может принадлежат еще кому-то. Например Машина и Двигатель. Хотя двигатель может быть и без машины, но он вряд ли сможет быть в двух или трех машинах одновременно. В отличии от студента, который может входить и в другие группы тоже.</p> <p>Например, в класс автомобиля содержит объект класса электрического двигателя:</p> <pre>public class ElectricEngine{ }</pre> <pre>public class Car {     ElectricEngine engine;     public Car()     {         engine = new ElectricEngine();     } }</pre> <p>При этом класс автомобиля полностью управляет жизненным циклом объекта двигателя. При уничтожении объекта автомобиля в области памяти вместе с ним будет уничтожен и объект двигателя. И в этом плане объект автомобиля является главным, а объект двигателя - зависимой.</p>
<p>Что такое агрегация</p>	<p>Агрегация определяет отношение <b>HAS A</b>, но связь слабее чем в композиции, т.к. объекты равноправны.</p>
<p>Расскажите про раннее и позднее связывание.</p>	<p><b>Связывание есть наличие связи между вызываемым методом программы и написанным кодом.</b></p> <p><b>Раннее связывание</b> Если метод известен компилятору, то происходит раннее связывание на этапе компиляции (early binding), также называют статическим связыванием.</p> <p><b>Позднее связывание</b> (late binding) - вызов метода возможен только во время выполнения, т.к. у компилятора нет информации, чтобы проверить корректность такого вызова. В java это возможно при помощи рефлексии.</p> <p><b>Статическое связывание</b> используется для final, перегруженных, приватных, статических методов, в то время как динамическое связывание используется для разрешения переопределенных методов. Все абстрактные методы разрешаются при помощи динамического связывания.</p> <p>В случае статического связывания используются не конкретные объекты, а информация о типе, то есть используется тип ссылочной переменной. С другой стороны, при динамическом связывании для нахождения нужного метода используется конкретный объект.</p>

<b>SOLID</b>	<p><b>SOLID</b> — это акроним, образованный из заглавных букв первых пяти принципов ООП и проектирования.</p> <p><b>S(Single Responsibility Principle)</b> - принцип единственной ответственности - каждый класс выполняет лишь одну задачу. Легкая модификация в будущем, простое тестирование, класс не имеет зависимостей на другие классы.</p> <p><b>O(Open Closed Principle)</b> - принцип открытости/закрытости - программные сущности открыты для расширения и закрыты для модификации. Чтобы не сломать логику в классе-родителе, мы унаследуемся от него и реализуем что-то своё, и используем свой класс.</p> <p><b>L(Liskov's Substitution Principle)</b> - принцип подстановки барбары лисков - объекты в программе можно заменить их наследниками без изменения свойств программы.</p> <p><b>I(Interface Segregation Principle)</b> - принцип разделения интерфейса - много специализированных интерфейсов лучше, чем один общий</p> <p><b>D(Dependency Inversion Principle)</b> - принцип инверсии зависимостей - зависимость на абстракциях. Модули верхних уровней не должны зависеть от модулей нижних уровней. Оба типа модулей должны зависеть от абстракций. Абстракции не должны зависеть от деталей. Детали должны зависеть от абстракций.</p> <p>Использование: Создание интерфейсов и их реализаций. Пример: терминал оплаты(абстракция) и разные карты оплаты.</p>
<b>Java</b>	
<b>Какая основная идея языка?</b>	<p>«Написано однажды - работает везде».</p> <p>Идея основывается в написании одного кода, который будет работать на любой платформе.</p>
<b>За счет чего обеспечивается кроссплатформенность?</b>	<p>Кроссплатформенность была достигнута <b>за счёт</b> создания <b>виртуальной машина Java</b>. Java Virtual Machine или JVM - это программа, являющаяся <b>прослойкой между операционной системой и Java программой</b>. В среде виртуальной машины выполняются коды Java программ. Сама JVM реализована для разных ОС. Что байт код для JVM может исполняться везде где установлена JVM.</p> <p>Код не нужно перекомпилировать под каждую из платформ.</p>

Какие преимущества у java?	<p><b>Объектно-ориентированное программирование</b> - структура данных становится объектом, которым можно управлять для создания отношений между различными объектами.</p> <p><b>Язык высокого уровня с простым синтаксисом и плавной кривой обучения</b> - синтаксис Java основан на C ++, поэтому Java похожа на C. Тем не менее, синтаксис Java проще, что позволяет новичкам быстрее учиться и эффективнее использовать код для достижения конкретных результатов.</p> <p><b>Стандарт для корпоративных вычислительных систем</b> - корпоративные приложения — главное преимущество Java с 90-х годов, когда организации начали искать надежные инструменты программирования не на C.</p> <p><b>Безопасность</b> - благодаря отсутствию указателей и Security Manager (политика безопасности, в которой можно указать правила доступа, позволяет запускать приложения Java в "песочнице").</p> <p><b>Независимость от платформы</b> - Можно создать Java-приложение на Windows, скомпилировать его в байт-код и запустить его на любой другой платформе, поддерживающей виртуальную машину Java (JVM). Таким образом, JVM служит уровнем абстракции между кодом и оборудованием.</p> <p><b>Язык для распределенного программирования и комфортной удаленной совместной работы</b> - Специфическая для Java методология распределенных вычислений называется Remote Method Invocation (RMI). RMI позволяет использовать все преимущества Java: безопасность, независимость от платформы и объектно-ориентированное программирование для распределенных вычислений. Кроме того, Java также поддерживает программирование сокетов и методологию распределения CORBA для обмена объектами между программами, написанными на разных языках.</p> <p><b>Автоматическое управление памятью</b> Разработчикам Java не нужно вручную писать код для управления памятью благодаря автоматическому управлению памятью (AMM).</p> <p><b>Многопоточность</b> Поток — наименьшая единица обработки в программировании. Чтобы максимально эффективно использовать время процессора, Java позволяет запускать потоки одновременно, что называется многопоточностью.</p> <p><b>Стабильность и сообщество</b> Сообщество разработчиков Java не имеет себе равных. Около 45% респондентов опроса StackOverflow 2018 используют Java.</p>
Какие недостатки у java?	<p><b>Платное коммерческое использование (с 2019)</b></p> <p><b>Низкая производительность</b> из-за компиляции и абстракции с помощью виртуальной машины, а также приложение очистки памяти.</p> <p><b>Не развитые инструменты по созданию GUI приложений на чистой java.</b></p> <p><b>Многословный код</b> Java — это более легкая версия неприступного C ++, которая вынуждает программистов прописывать свои действия словами из английского языка. Это делает язык более понятным для неспециалистов, но менее компактным.</p>
Что такое JDK? Что в него входит?	<p><b>JDK (Java Development Kit) - включает JRE и набор инструментов разработчика</b> приложений на языке Java:</p> <ul style="list-style-type: none"> <li>- компилятор Java (javac)</li> <li>- стандартные библиотеки классов java</li> <li>- примеры</li> <li>- документацию</li> <li>- различные утилиты</li> </ul>
Что такое JRE? Что в него входит?	<p><b>JRE (java Runtime Environment) - минимально-необходимая реализация виртуальной машины для исполнения Java-приложений. Состоит из JVM, ClassLoader и стандартного набора библиотек</b> и классов Java</p>
Что такое JVM?	<p><b>JVM (Java Virtual Machine) - виртуальная машина Java</b> исполняет байт-код Java, предварительно созданный из кода JIT компилятором, с помощью встроенного интерпретатора байткода.</p> <p><b>HotSpot</b> представляет собой реализацию концепции JVM.</p>

Что такое byte code?	Байт-код Java — набор инструкций, скомпилированный компилятором, исполняемый JVM.
Что такое загрузчик классов (classloader)?	<p>Используется <b>для передачи в JVM скомпилированного байт-кода, хранится в файлах с расширением .class</b></p> <p>При запуске JVM, используются три загрузчика классов:</p> <ul style="list-style-type: none"> <li>- <b>Bootstrap ClassLoader - базовый загрузчик</b></li> <li>- загружает платформенные классы JDK из архива rt.jar</li> <li>- <b>AppClassLoader - системный загрузчик</b></li> <li>- загружает классы приложения, определенные в CLASSPATH</li> <li>- <b>Extension ClassLoader - загрузчик расширений после B Java9 выпилили</b></li> <li>- загружает классы расширений, которые по умолчанию находятся в каталоге jre/lib/ext.</li> </ul> <p>ClassLoader выполняет три основных действия в строгом порядке:</p> <ul style="list-style-type: none"> <li>• Загрузка: находит и импортирует двоичные данные для типа.</li> <li>• Связывание: выполняет проверку, подготовку и (необязательно) разрешение.</li> <li>- Проверка: обеспечивает правильность импортируемого типа.</li> <li>- Подготовка: выделяет память для переменных класса и инициализация памяти значениями по умолчанию.</li> <li>- Разрешение: преобразует символические ссылки из типа в прямые ссылки.</li> <li>• Инициализация: вызывает код Java, который инициализирует переменные класса их правильными начальными значениями.</li> </ul> <p>Каждый загрузчик хранит указатель на родительский, чтобы суметь передать загрузку если сам будет не в состоянии этого сделать.</p>
Что такое JIT?	<p><u>JIT (Just-in-time compilation) - компиляция на лету или динамическая компиляция - технология увеличения производительности программных систем, использующих байт-код, путем компиляции байт-кода в машинный код во время работы программы.</u></p> <p><u>В основном отвечает за оптимизацию производительности приложений во время выполнения.</u></p> <p><u><a href="https://javahelp.online/osnovy/voprosy-otvety-sobesedovanie-java">https://javahelp.online/osnovy/voprosy-otvety-sobesedovanie-java</a> (Q13)</u></p>
Что такое сборщик мусора? (Garbage collector)	<p>Сборщик мусора выполняет две задачи:</p> <ul style="list-style-type: none"> <li>- <b>поиск мусора;</b></li> <li>- <b>очистка мусора.</b></li> </ul> <p><b>Для обнаружения мусора есть два подхода:</b></p> <ul style="list-style-type: none"> <li>- <b>Учет ссылок (Reference counting);</b></li> </ul> <p>Учет ссылок - если объект не имеет ссылок, он считается мусором.          Проблема - не возможность выявить циклические ссылки, когда два объекта не имеют внешних ссылок, но ссылаются друг на друга -&gt; утечка памяти</p> <ul style="list-style-type: none"> <li>- <b>Трассировка (Tracing).</b> (используется в HotSpot)6</li> </ul> <p>Трассировка - до объекта можно добраться из Корневых точек (GC root).          До чего добраться нельзя - мусор.          Всё, что доступно из «живого» объекта, также является «живым».</p> <p><b>Типы корневых точек (GC Roots) java приложения:</b></p> <ul style="list-style-type: none"> <li>- объекты в статических полях классов</li> <li>- объекты, доступные из стека потоков</li> <li>- объекты из JNI(java native interface) ссылок в native методах</li> </ul>

Процессы сборки мусора разделяются несколько видов:

**minor GC (малая)** - частый и быстрый, работает только с областью памяти "young generation";

- приложение приостанавливается на начало сборки мусора (такие остановки называются stop-the-world);
- «живые» объекты из Eden перемещаются в область памяти «To»;
- «живые» объекты из «From» перемещаются в «To» или в «old generation», если они достаточно «старые»;
- Eden и «From» очищаются от мусора;
- «To» и «From» меняются местами;
- приложение возобновляет работу.

**major GC (старшая)** - редкий и более длительный, затрагивает объекты старшего поколения.

В принцип работы «major GC» добавляется процедура «уплотнения», позволяющая более эффективно использовать память. В процедуре живые объекты перемещаются в начало. Таким образом, мусор остается в конце памяти.

**full GC (полная)** - полный сборщик мусора сначала запускает Minor, а затем Major (хотя порядок может быть изменен, если старое поколение заполнено, и в этом случае он освобождается первым, чтобы позволить ему получать объекты от молодого поколения).

## Виды ссылок в Java

1) **StrongReference** — это самые обычные ссылки которые мы создаем каждый день, любая переменная ссылочного типа.

`StringBuilder builder = new StringBuilder();` - builder это и есть strong-ссылка на объект `StringBuilder`.

2) **SoftReference** — GC гарантировано удалит с кучи все объекты, доступные только по soft-ссылке, перед тем как бросит `OutOfMemoryError`. `SoftReference` это наш механизм кэширования объектов в памяти, но в критической ситуации, когда закончится доступная память, GC удалит не использующиеся объекты из памяти и тем самым попытается спасти JVM от завершения работы.

`StringBuilder builder = new StringBuilder();`  
`SoftReference<StringBuilder> softBuilder = new SoftReference(builder);`

`softBuilder.get()` — вернет strong-ссылку на объект `StringBuilder` в случае если GC не удалил этот объект из памяти. В другом случае вернется `null`.

`softBuilder.clear()` — удалит ссылку на объект `StringBuilder`

То же самое работает для **WeakReference**.

3) **WeakReference** — если GC видит, что объект доступен только через цепочку weak-ссылок (исчезнули strong-ссылки), то он удалит его из памяти.

4) **PhantomReference** — если GC видит что объект доступен только через цепочку phantom-ссылок, то он его удалит из памяти. После нескольких запусков GC. Особенностей у этого типа ссылок две.

Первая это то, что метод `get()` всегда возвращает `null`. Именно из-за этого `PhantomReference` имеет смысл использовать только вместе с `ReferenceQueue`.

Вторая особенность – в отличие от `SoftReference` и `WeakReference`, GC добавит phantom-ссылку в `ReferenceQueue` **после** того как выполниться метод `finalize()`.

**So in brief: Soft references try to keep the reference. Weak references don't try to keep the reference. Phantom references don't free the reference until cleared.**

**ReferenceQueue.** Он позволяет отслеживать момент, когда GC определит что объект более не нужен и его можно удалить. Именно сюда попадает `Reference` объект после того как объект на который он ссылается удален из памяти. При создании `Reference` мы можем передать в конструктор `ReferenceQueue`, в который будут помещаться ссылки после удаления.

Память процесса делится на Stack (стек) и Heap (куча) :

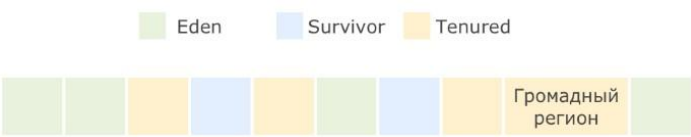
- Stack содержит stack frame'ы, они делятся на три части: параметры метода, указатель на предыдущий фрейм и локальные переменные.

- Структура Heap зависит от выбранного сборщика мусора. Читай про GC!

MetaSpace - специальное пространство кучи, отделенное от кучи основной памяти. JVM хранит здесь весь статический контент. Это включает в себя все статические методы, примитивные переменные и ссылки на статические объекты. Кроме того, он содержит данные о байт-коде, именах и JIT-информации . До Java 7 String Pool также был частью этой памяти.

Вкратце, при Serial/Parallel/CMS GC будет следующая структура:



<div>Stack и Heap</div>	<p>А при G1 GC:</p>  <p>С помощью опций Xms и Xmx можно настроить начальный и максимально допустимый размер кучи соответственно. Существуют опции для настройки величины стека.</p> <ul style="list-style-type: none"> <li>- Heap - используется всем приложением, Stack - одним потоком исполняемой программы.</li> <li>- Новый объект создается в heap, в stack размещается ссылка на него. В стеке размещаются локальные переменные примитивных типов.</li> <li>- Объекты в куче доступны из любого места программы, стековая память не доступна для других потоков.</li> <li>- Если память стека закончилась JRE вызовет исключение StackOverflowError, если куча заполнена OutOfMemoryError</li> <li>- Размер памяти стека, меньше памяти кучи. Стековая память быстрее памяти кучи.</li> <li>- В куче есть ссылки между объектами и их классами. На этом основана рефлексия.</li> </ul> <p>Обе области хранятся в RAM.</p>
<div>Процедурная Java</div>	
<div>Какие примитивные типы данных есть в Java?</div>	<p>Вещественные, целочисленные, логические и строковые.</p> <p>byte short int long float double char boolean</p>
<div>Что такое char?</div>	<p><b>16-разрядное беззнаковое целое</b>, представляющее собой символ UTF-16 (буквы и цифры)</p>
<div>Сколько памяти занимает boolean?</div>	<p><b>Зависит от реализации JVM</b> В стандартной реализации Sun JVM и Oracle HotSpot JVM тип boolean занимает 4 байта (32 бита), как и тип int. Однако, в определенных версиях JVM имеются реализации, где в массиве boolean каждое значение занимает по 1-му биту.</p>
<div>Что такое классы-обертки?</div>	<p>Обертка — это специальный <b>класс, который хранит внутри себя значение примитива</b>(объекты классов-оберток являются <b>неизменяемыми (Immutable)</b>). Нужны для реализации дженериков.</p>
<div>Что такое автоупаковка и автораспаковка?</div>	<p><b>Автоупаковка</b> - присвоение классу обертки значения примитивного типа;</p> <p><b>Автораспаковка</b> - присвоение переменной примитивного типа значение класса обертки.</p> <p>для присваивания ссылок-примитивов объектам их классов-оберток (и наоборот) не требуется ничего делать, все происходит автоматически. Для того, чтобы иметь возможность оперировать с простыми числами (и boolean) как с объектами были придуманы классы-обёртки.</p>

Что такое явное и неявное приведение типов? В каких случаях в java нужно использовать явное приведение?	<p><b>Неявное приведение</b> – автоматическое <b>расширение</b> типа переменной <b>от меньшего к большему</b>.</p> <p><b>Явное приведение</b> - <b>явное сужение от большего к меньшему</b>. Необходимо явно указать сужаемый тип.</p> <p>В случае с объектами мы можем делать неявное(автоматическое) приведение от наследника к родителю, но не наоборот, иначе получим ClassCastException.</p>
Что такое пул интов?	<p>В Java есть пул(pool) целых чисел в промежутке [-128;127], так как это самый часто встречающийся диапазон. Т.е. если мы создаем Integer в этом промежутке, то вместо того, чтобы каждый раз создавать новый объект, JVM берет их из пула.</p> <p>Изменить размер кэша в HotSpot вы можете, указав ключ -XX:AutoBoxCacheMax=&lt;размер&gt;.</p>
Какие нюансы у строк в Java?	<p>Класс <b>String</b> в Java - <b>неизменяемый</b> из-за модификатора final и отсутствия сеттера. Это нужно для реализации пула стрингов. <b>При редактировании</b> будет создаваться <b>новая строка</b>. <b>При копировании</b> новая строка не создается, а создается <b>ссылка на существующую</b> строку.</p>
Что такое пул строк?	<p><b>Область памяти где хранятся объекты строк.</b></p> <p><b>При создании</b> в пуле идет <b>поиск строки</b>:</p> <p><b>-если НЕ находит - создается строка, возвращается ссылка</b></p> <p><b>-если находит - возвращает ссылку найденной строки.</b></p> <p>При этом использование оператора new заставляет класс String создать новый объект, даже если такая строка уже есть в пуле. После этого можем использовать метод intern(), чтобы поместить этот объект в пул строк.</p> <p>Пул строк и Integer хранится в heap, но ссылки на объекты хранятся в stack.</p>
Почему не рекомендуется изменять строки в цикле? Что рекомендуется использовать?	<p>Т.к. <b>строка неизменяемый класс</b>, потребление ресурсов при редактировании, т.к. <b>каждую итерацию</b> при редактировании <b>будет создаваться новый объект</b> строки. <b>Рекомендуется</b> использовать <b>StringBuilder</b> или <b>StringBuffer</b>.</p>
Почему строки не рекомендуется использовать для хранения паролей?	<p><b>1. Пул строк</b></p> <p>Так как строки в Java хранятся в пуле строк, то ваш пароль в виде обычного текста будет доступен в памяти, пока сборщик мусора не очистит её. И поскольку String используются в String pool для повторного использования, существует довольно высокая вероятность того, что пароль останется в памяти надолго, что совсем не безопасно.</p> <p><b>2. Рекомендации авторов</b></p> <p>Java сама по себе рекомендует использовать метод getPassword () из класса JPasswordField, который возвращает char [].</p> <p><b>3. Случайная печать в логах</b></p> <p>С типом String всегда существует опасность того, что текст, хранящийся в строке будет напечатан в файле логов или в консоли. В то же время в случае использования Array, вы не будете печатать содержимое массива, а только его расположение в памяти.</p>



Почему String неизменяемый и финализированный класс?	<p>1. Для возможности <b>реализации строкового пула (String pool)</b></p> <p>Виртуальная машина имеет возможность сохранить много места в памяти (heap space) т.к. разные строковые переменные указывают на одну переменную в пуле. При изменении строк было бы невозможно реализовать интернирование, поскольку если какая-либо переменная изменит значение, это отразится также и на остальных переменных, ссылающихся на эту строку.</p> <p>2. <b>Безопасность</b></p> <p>Изменяемость строк несли бы в себе потенциальную угрозу безопасности приложения. Поскольку в Java строки используются для передачи параметров для авторизации, открытия файлов и т.д. — неизменяемость позволяет избежать проблем с доступом.</p> <p>3. <b>Для многопоточности. Неизменяемые строки потокобезопасны</b></p> <p>Так как строка неизменяемая то, она безопасна для многопоточности и один экземпляр строки может быть совместно использован различными потоками. Это позволяет избежать синхронизации для потокобезопасности. Таким образом, строки в Java полностью потокобезопасны.</p> <p>4. <b>Ключ для HashMap</b></p> <p>Поскольку строка неизменяемая, её hashCode кэшируется в момент создания и нет никакой необходимости рассчитывать его снова. Это делает строку отличным кандидатом для ключа в Map и его обработка будет быстрее, чем других ключей HashMap. Поэтому строка наиболее часто используется в качестве ключа HashMap.</p> <ul style="list-style-type: none"> <li>- можно передавать строку между потоками не опасаясь, что она будет изменена</li> <li>- отсутствуют проблемы с синхронизацией потоков</li> <li>- отсутствие проблем с утечкой памяти</li> <li>- отсутствие проблем с доступом и безопасностью при использовании строк для передачи параметров авторизации, открытия файлов и т.д.</li> <li>- кэширование hashCode</li> <li>- Экономия памяти при использовании пула строк для хранения повторяющихся строк.</li> </ul>
Почему строка является популярным ключом в HashMap в Java?	Поскольку <b>строки неизменны, их хэшкод кэшируется</b> в момент создания, и <b>не требует повторного пересчета</b> .
Что делает метод intern() в классе String?	Помещает строку в pool строк.
Можно ли использовать строки в конструкции switch?	<p><u>Да, начиная с Java 7 в операторе switch можно использовать строки, ранние версии Java не поддерживают этого.</u></p> <p><u>Более подробно: <a href="https://javarush.ru/groups/posts/759-java-string-voprosih-k-sobesedovaniju-i-otvetih-na-nikh-ch1">https://javarush.ru/groups/posts/759-java-string-voprosih-k-sobesedovaniju-i-otvetih-na-nikh-ch1</a> (10)</u></p> <p><u>При этом:</u></p> <ul style="list-style-type: none"> <li>- <u>участвующие строки чувствительны к регистру;</u></li> <li>- <u>использование строк в конструкции switch делает код читабельнее, убирая множественные цепи условий if-else</u></li> <li>- <u>оператор switch использует метод String.equals() для сравнения полученного значения со значениями case, поэтому добавьте проверку на NULL во избежание NullPointerException.</u></li> </ul>
Какая основная разница между String, StringBuffer, StringBuilder?	String - неизменяемый, потокобезопасный; StringBuffer - изменяемый, потокобезопасный; StringBuilder - изменяемый, потокобезопасный.
Существуют ли в java многомерные массивы?	Многомерные массивы в их классическом понимании в java не существуют. Многомерный массив всегда прямоугольный и неразрывен в памяти. А то, что в java считается многомерным - в других языках ещё называют "зубчатым массивом" или массивом массивов.

Какими значениями иницируются переменные по умолчанию?	byte 0 short 0 int 0 long 0L float 0.0f double 0.0d char '\u0000' boolean false Объекты null Локальные (в методе) переменные не имеют значений по умолчанию, их имеют поля класса. Не static-поле класса будет инициализировано после того, как будет создан объект этого класса. А static-поле будет инициализировано тогда, когда класс будет загружен виртуальной Java машиной.
Что такое сигнатура метода?	Это <b>имя метода</b> плюс <b>параметры</b> (порядок параметров имеет значение из-за множественной передачи данных через троеточие, которое должно располагаться последним). В сигнатуру метода не входит возвращаемое значение, а также бросааемые им исключения. А сигнатура метода в сочетании с типом возвращаемого значения и бросааемыми исключениями называется <b>контрактом метода</b> .
Расскажите про метод main	<b>Является, как правило, точкой входа в программу и вызывается JVM.</b> Как только <b>заканчивается выполнение</b> метода <b>main()</b> , так сразу же <b>завершается работа самой программы</b> . static - чтобы JVM смогла загрузить его во время компиляции. public static void и сигнатура - обязательное декларирование. Мэйнов может быть много и может не быть вообще. Может быть перегружен.
Каким образом переменные передаются в методы, по значению или по ссылке?	<b><u>Java передает параметры по значению. Всегда.</u></b> <u>С примитивами, мы получаем копию содержимого. Со ссылками мы тоже получаем копию ссылки.</u> <a href="https://javarush.ru/groups/posts/857-peredacha-parametrov-v-java">https://javarush.ru/groups/posts/857-peredacha-parametrov-v-java</a>
ООП в Java	
Какие виды классов есть в java?	1. Вложенные классы – нестатические классы внутри внешнего класса. 2. Вложенные статические классы – статические классы внутри внешнего класса. 3. Локальные классы Java – классы внутри методов. <a href="#">разница между локальным и внутренним</a> 4. Анонимные Java классы – классы, которые создаются на ходу. <a href="#">Анонимные классы доступно</a> 5. Final, abstract, enum - классы

<p>Расскажите про вложенные классы. В каких случаях они применяются?</p>	<p><b>Нужны для обслуживания внешних классов</b></p> <ol style="list-style-type: none"> <li><b>1. Статические вложенные классы (Static nested classes)</b> <ul style="list-style-type: none"> <li>○ Есть возможность обращения к внутренним статическим полям и методам класса обертки.</li> </ul> </li> <li><b>2. Вложенные классы</b> <ul style="list-style-type: none"> <li>○ Есть возможность обращения к внутренним полям и методам класса обертки.</li> <li>○ Не может иметь статических объявлений.</li> <li>○ Внутри такого класса нельзя объявить перечисления.</li> <li>○ Если нужно явно получить this внешнего класса — OuterClass.this</li> </ul> </li> <li><b>3. Локальный класс</b> <ul style="list-style-type: none"> <li>○ Видны только в пределах блока, в котором объявлены.</li> <li>○ Не могут быть объявлены как private/public/protected или static (по этой причине интерфейсы нельзя объявить локально).</li> <li>○ Не могут иметь внутри себя статических объявлений (полей, методов, классов), но могут иметь константы (static final)</li> <li>○ Имеют доступ к полям и методам обрамляющего класса.</li> <li>○ Можно обращаться к локальным переменным и параметрам метода, если они объявлены с модификатором final или являются effectively final.</li> </ul> </li> <li><b>4. Анонимные классы</b> <ul style="list-style-type: none"> <li>○ Локальный класс без имени.</li> </ul> </li> </ol>
<p>- Что такое «локальный класс»? Каковы его особенности?</p>	<p>Данные классы <b>объявляются внутри других методов</b>. Они <b>обладают всеми свойствами нестатического вложенного класса</b>, только <b>создавать их экземпляры можно только в методе</b>.</p> <p><b>Особенности:</b></p> <p>Локальные классы <b>способны работать</b> только с <b>final переменными</b> метода.</p> <p><b>С 8+ версий Java можно использовать не final переменные в локальных классах</b>, но только при условии, что они <b>не будут изменяться</b>.</p> <p>Локальные классы <b>нельзя объявлять с модификаторами доступа</b>.</p> <p>Локальные классы <b>обладают доступом к переменным метода</b>.</p> <p>Может быть создан внутри блоков инициализации.</p>
<p>Что такое «анонимные классы»? Где они применяются?</p>	<p>Это вложенный локальный класс без имени, который разрешено декларировать в любом месте обрамляющего класса, разрешающем размещение выражений. Создание экземпляра анонимного класса происходит одновременно с его объявлением. В зависимости от местоположения анонимный класс ведет себя как статический либо как нестатический вложенный класс - в нестатическом контексте появляется окружающий его экземпляр.</p> <p>Анонимные классы имеют несколько ограничений:</p> <p>Их использование разрешено только в одном месте программы - месте его создания;</p> <p>Применение возможно только в том случае, если после порождения экземпляра нет необходимости на него ссылаться;</p> <p>Реализует лишь методы своего интерфейса или суперкласса, т.е. не может объявлять каких-либо новых методов, так как для доступа к ним нет поименованного типа.</p> <p>Анонимные классы обычно применяются для:</p> <ul style="list-style-type: none"> <li>создания объекта функции (function object), например реализация интерфейса Comparator;</li> <li>создания объекта процесса (process object), такого как экземпляры классов Thread, Runnable и подобных;</li> <li>в статическом методе генерации;</li> <li>инициализации открытого статического поля final, которое соответствует сложному перечислению типов, когда для каждого экземпляра в перечислении требуется отдельный подкласс.</li> </ul> <p>Анонимные классы всегда являются конечными классами.</p> <p>Каждое объявление анонимного класса уникально. Видны только внутри того метода, в котором определены. <b>В документации Oracle приведена хорошая рекомендация: «Применяйте анонимные классы, если вам нужен локальный класс для одноразового использования».</b></p>

<p>- Каким образом из вложенного класса получить доступ к полю внешнего класса?</p>	<p>Статический вложенный класс имеет прямой доступ только к статическим полям обрамляющего класса. Простой вложенный класс, может обратиться к любому полю внешнего класса напрямую.</p> <p>В случае, если у вложенного класса уже существует поле с таким же литералом, то обращаться к внешнему полю следует через имя внешнего класса. Например: Outer.this.field.</p>
<p>Что такое перечисления (enum)?</p>	<p>Перечисления представляют <b>набор логически связанных констант</b>.</p> <p>Перечисление фактически представляет новый класс, поэтому мы можем определить переменную данного типа и использовать ее.</p> <p><b>Перечисления</b>, как и обычные классы, <b>могут определять конструкторы, поля и методы</b>. Следует отметить, что <b>конструктор по умолчанию приватный</b>. Также можно определять методы для отдельных констант. Методы: -<b>ordinal()</b> возвращает порядковый <b>номер определенной константы</b> (нумерация начинается с 0) -<b>values()</b> возвращает <b>массив всех констант</b> перечисления Enum имеет ряд преимуществ при использовании в сравнении с static final int. Главным отличием является то что <b>используя enum вы можете проверить тип данных</b>. <b>Недостатки</b> - К ним <b>не применимы операторы &gt;, &lt;, &gt;=, &lt;=</b> - enum также <b>требует больше памяти для хранения</b> чем обычная константа.</p> <p>Нужны для ограничения области допустимых значений: например, времена года, дни недели</p>
<p>Как проблема ромбовидного наследования решена в java?</p>	<p><b>В Java нет поддержки множественного наследования классов.</b></p> <p>Предположим, что SuperClass — это абстрактный класс, описывающий некоторый метод, а классы ClassA и ClassB — обычные классы наследники SuperClass, а класс ClassC наследуется от ClassA и ClassB одновременно. Вызов метода родительского класса приведет к неопределенности, так как компилятор не знает о том, метод какого именно суперкласса должен быть вызван. Это и есть основная причина, почему в Java нет поддержки множественного наследования классов.</p> <p><b>1. Классы всегда побеждают:</b> Определенный в классе / суперклассе метод всегда имеет высший приоритет перед дефолтными методами интерфейсов. <b>2. Если не срабатывает правило 1, то побеждают саб-интерфейсы (more specific).</b> Т.е. если интерфейс B наследует A, и у обоих есть методы с одинаковой сигнатурой, то побеждает B. <b>3. Если оба правила не работают, то класс, наследующий конфликтующие интерфейсы, должен явно через super определить,</b> какой именно метод вызвать, иначе компилятор будет сильно материться.</p>
<p>Что такое конструктор по умолчанию?</p>	<p>Если у какого-либо класса не определить конструктор, то компилятор сгенерирует конструктор без аргументов - так называемый «конструктор по умолчанию». Если у класса уже определен какой-либо конструктор, то конструктор по умолчанию создан не будет и, если он необходим, его нужно описывать явно.</p>
<p>Могут ли быть приватные конструкторы? Для чего они нужны?</p>	<p>Да, могут. Приватный конструктор <b>запрещает создание экземпляра класса вне методов самого класса</b>. Нужен для реализации паттернов, например singleton.</p>

Расскажите про классы-загрузчики и про динамическую загрузку классов.	<p>При запуске JVM, используются три загрузчика классов:</p> <ul style="list-style-type: none"> <li>- <b>Bootstrap ClassLoader</b> - главный загрузчик</li> <li>- загружает платформенные классы JDK из архива rt.jar</li> <li>- <b>AppClassLoader</b> - системный загрузчик</li> <li>- загружает классы приложения, определенные в CLASSPATH</li> <li>- <b>Extension ClassLoader</b> - загрузчик расширений</li> <li>- загружает классы расширений, которые по умолчанию находятся в каталоге jre/lib/ext.</li> </ul> <p><b>Динамическая загрузка</b> происходит "на лету" <b>в ходе выполнения программы</b> с помощью статического метода <b>класса Class.forName</b>(имя класса). Для чего нужна динамическая загрузка? Например мы не знаем какой класс нам понадобится и принимаем решение в ходе выполнения программы передавая имя класса в статический метод <b>forName()</b>.</p>
Чем отличаются конструкторы по умолчанию, конструктор копирования и конструктор с параметрами?	<ul style="list-style-type: none"> <li>-У конструктора по умолчанию отсутствуют какие-либо аргументы.</li> <li>-Конструктор копирования принимает в качестве аргумента уже существующий объект класса для последующего создания его клона.</li> <li>-Конструктор с параметрами имеет в своей сигнатуре аргументы (обычно необходимые для инициализации полей класса).</li> </ul>
Какие модификаторы доступа есть в Java? Какие применимы к классам?	<p><b>Private</b> – доступ к компоненту только <b>из этого класса</b>, в котором объявлен.</p> <p><b>Default</b> – Переменная или метод будут доступны <b>для любого другого класса в том же пакете</b>.</p> <p><b>Protected</b> – Поля <b>protected</b> доступны <b>всем классам внутри пакета</b>, а также <b>всем классам-наследникам вне пакета</b>.</p> <p><b>Public</b> – доступ к компоненту из экземпляра <b>любого класса и любого пакета</b>.</p> <p>Класс может быть объявлен с модификатором <b>public</b> и <b>default</b>.</p>
Что означает модификатор static?	<p>Статическая <b>переменная</b> - это переменная, <b>принадлежащая классу</b>, а не объекту.</p> <p>А статический класс- это вложенный <b>класс</b>, который <b>может обращаться только к статическим полям обертывающего его класса</b>.</p> <p>Внутри <b>static метода</b> <b>нельзя вызвать не статический метод по имени класса</b>.</p>
Может ли статический метод быть переопределён или перегружен?	<p><b>Нельзя переопределять статические методы</b>.</p> <p>Если вы объявите такой же метод в классе-наследнике (subclass), т.е. метод с таким же именем и сигнатурой, вы лишь «спрячете» метод суперкласса вместо переопределения. Это явление известно как <b>сокрытие методов</b> (hiding methods).</p> <p><b>Перегружен - да</b>. Всё работает точно так же как и с обычными методами - 2 статических метода могут иметь одинаковое имя, если количество их параметров или типов различается.</p>
Могут ли нестатические методы перегрузить статические?	Да. Это будут просто два разных метода для программы. Статический будет доступен по имени класса.
Можно ли сузить уровень доступа/тип возвращаемого значения при переопределении метода?	<p>При переопределении метода <b>нельзя сузить модификатор доступа к методу</b> (например, с <b>public</b> до <b>private</b>), но можно расширить.</p> <p><b>Изменить тип возвращаемого значения нельзя</b>, но <b>можно сузить</b> возвращаемое значение, <b>если они совместимы</b>. Например, если метод возвращает объект класса, а переопределенный метод возвращает класс-наследник.</p>

Что можно изменить в сигнатуре метода при переопределении? Можно ли менять модификаторы (throws и тп)?	<p>В сигнатуре(имя + параметры) менять ничего нельзя.</p> <p><b>Возможно расширение уровня доступа.</b>  <b>Изменять тип возвращаемого значения</b> при переопределении метода разрешено только <b>в сторону сужения типа</b> (вместо родительского класса - наследника).  <b>Секцию throws</b> метода можно не указывать, но стоит помнить, что <b>она остаётся действительной, если уже определена у метода родительского класса.</b>          Так же, возможно добавлять новые исключения, являющиеся наследниками от уже объявленных или исключения RuntimeException. Порядок следования таких элементов при переопределении значения не имеет.</p>
Могут ли классы быть статическими?	<p><b>Класс можно объявить статическим за исключением классов верхнего уровня.</b>          Такие классы известны как «вложенные статические классы» (nested static class).</p>
Что означает модификатор final? К чему он может быть применим?	<p>Для класса это означает, что <b>класс не сможет иметь подклассов</b>, т.е. запрещено наследование.          Следует также отметить, что к <b>abstract-классам нельзя применить модификатор final</b>, т.к. это взаимоисключающие понятия.          Для <b>переменных примитивного типа</b> это означает, что <b>однажды присвоенное значение не может быть изменено</b>          Для <b>ссылочных переменных</b> это означает, что после присвоения объекта, нельзя изменить ссылку на данный объект. Важно: <b>Ссылку изменить нельзя, но состояние объекта изменять можно.</b>          Т.к. массив – это объект, то final означает, что после присвоения ссылки на объект, уже нельзя ее изменить, но можно изменять состояние объекта.</p>
Что такое абстрактные классы? Чем они отличаются от обычных?	<p>Абстрактным называется класс, на основе которого не могут создаваться объекты.          Как обычный класс, но <b>с абстрактными методами.</b>  <b>Нельзя создать</b> объект или <b>экземпляр абстрактного класса.</b></p> <p>Наследниками абстрактного класса могут быть другие абстрактные классы</p>
Может ли быть абстрактный класс без абстрактных методов?	Класс <b>может</b> быть абстрактным без единого абстрактного метода, если у него указан модификатор abstract.
Могут ли быть конструкторы у абстрактных классов? Для чего они нужны?	<p><b>Да. Необходимы для наследников.</b></p> <p>В абстрактном классе в Java можно объявить и определить конструкторы. Даже если вы не объявили никакого конструктора, компилятор добавит в абстрактный класс конструктор по умолчанию без аргументов. Абстрактные конструкторы будут часто использоваться для обеспечения ограничений класса или инвариантов, таких как минимальные поля, необходимые для настройки класса.</p>
Что такое интерфейсы? Какие модификаторы по умолчанию имеют поля и методы интерфейсов?	<p><b>Интерфейс описывает поведение, которым должны обладать классы, реализующие этот интерфейс. «Поведение» — это совокупность методов.</b>  <b>Интерфейс</b> — это план класса или, можно сказать, набор абстрактных методов и статических констант. В интерфейсе каждый метод является открытым и абстрактным, но не содержит конструктора. Таким образом, интерфейс в основном представляет собой группу связанных методов с пустыми телами. Другими словами, <b>интерфейс определяет как элементы будут взаимодействовать между собой.</b></p> <p>- <b>методы интерфейса</b> являются публичными (<b>public</b>) и абстрактными (<b>abstract</b>),          - <b>поля</b> — <b>public static final</b>.</p>

Чем интерфейсы отличаются от абстрактных классов? В каких случаях следует использовать абстрактный класс, а в каких интерфейс?	<p>1. <u>Интерфейс описывает</u> только <b>поведение</b> (методы) объекта, а вот <b>состояний</b> (полей) у него <b>нет</b> (кроме public static final), в то время как у абстрактного класса они могут быть.</p> <p>2. Мы <b>можем наследовать</b> только <b>один класс</b>, а <b>реализовать интерфейсов — сколько угодно</b>. Интерфейс может наследовать (extends) другой интерфейс/интерфейсы.</p> <p>3. <b>Абстрактные классы используются</b>, когда <b>есть</b> отношение "is-a", то есть класс-наследник расширяет базовый абстрактный класс, а <b>интерфейсы могут быть реализованы разными классами, вовсе не связанными друг с другом</b>.</p> <p>4. <b>Абстрактный класс может реализовывать методы</b>; <b>интерфейс</b> может реализовывать дефолтные методы <b>начиная с 8й версии</b>.</p> <p><a href="https://javahelp.online/osnovy/voprosy-otvety-sobesedovanie-java">https://javahelp.online/osnovy/voprosy-otvety-sobesedovanie-java</a> (Q5)</p>
Может ли один интерфейс наследоваться от другого? От двух других?	Да, может. Используется ключевое слово <b>extends</b>
Что такое дефолтные методы интерфейсов? Для чего они нужны?	В <b>JDK 8</b> была добавлена такая функциональность как <b>методы по умолчанию с модификатором default</b> . И теперь интерфейсы могут иметь их реализацию по умолчанию, которая используется, если класс, реализующий данный интерфейс, не реализует метод. Это <b>нужно для обратной совместимости</b> . (Если один или несколько методов добавляются к интерфейсу, все реализации также будут вынуждены их реализовывать. Методы интерфейса по умолчанию являются эффективным способом решения этой проблемы.)
Как решается проблема ромбовидного наследования при наследовании интерфейсов при наличии default методов?	класс, наследующий конфликтующие интерфейсы, <b>должен явно через super определить</b> , какой именно метод вызвать: <code>InterfaceB.super.method();</code>
Каков порядок вызова конструкторов и блоков инициализации с учётом иерархии классов?	<p>1. <b>Статические блоки</b> от первого до последнего предка(от предка до наследника)</p> <p>2. <b>Попарно динамической блок инициализации и конструктор</b> от первого до последнего предка</p>
Зачем нужны и какие бывают блоки инициализации?	Инициализация - это когда мы впервые задаем переменной какое-либо значение. Существуют <b>статические</b> и <b>нестатические</b> блоки инициализации.
Для чего в Java используются статические блоки инициализации?	Статические блоки инициализации используются <b>для выполнения кода, который должен выполняться один раз при инициализации класса загрузчиком классов</b> , в момент предшествующий созданию объектов этого класса при помощи конструктора. Такой блок принадлежит только самому классу.
Что произойдет, если в блоке инициализации возникнет исключительная ситуация?	<p>Для <b>нестатических</b> блоков инициализации, <b>если выбрасывание исключения прописано явным образом, требуется</b>, чтобы <b>объявления</b> этих исключений были перечислены <b>в throws всех конструкторов класса</b>. Иначе будет <b>ошибка компиляции</b>.</p> <p>Для <b>статического</b> блока выбрасывание исключения в явном виде, приводит к <b>ошибке компиляции</b>.</p>



<p>Какое исключение выбрасывается при возникновении ошибки в блоке инициализации класса?</p>	<p>Если возникшее исключение - <b>наследник RuntimeException</b>:</p> <ul style="list-style-type: none"> <li>-для <b>статических</b> блоков инициализации будет выброшено <code>java.lang.ExceptionInInitializerError</code>;</li> <li>-для <b>нестатических</b> будет выброшено <b>исключение-источник</b>.</li> </ul> <p>Если возникшее исключение - <b>наследник Error</b>, то в обоих случаях будет выброшено <code>java.lang.Error</code>.</p> <p>Если исключение: <code>java.lang.ThreadDeath</code> - смерть потока. В этом случае никакое <b>исключение</b> выброшено <b>не будет</b>.</p>
<p>Что такое класс Object?</p>	<p><b>Базовый класс для всех остальных объектов в Java. Любой класс наследуется от Object и, соответственно, наследуют его методы</b></p> <p><b>Все классы являются наследниками</b> суперкласса <b>Object</b>. Это не нужно указывать явно. В результате объект <b>Object</b> может ссылаться на объект любого другого класса.</p> <p><b>Рефлексия</b> (от позднелат. reflexio - обращение назад) - это <b>механизм исследования данных</b> о программе <b>во время её выполнения</b>.</p>
<p>Какие методы есть у класса Object (перечислить все)? Что они делают?</p>	<ul style="list-style-type: none"> <li>- <b>equals()</b> - проверка на равенство двух объектов</li> <li>- <b>hashCode()</b> - изначально случайно число int</li> <li>- <b>toString()</b> - представления данного объекта в виде строки.</li> <li>- <b>getClass()</b> - получение типа данного объекта</li> <li>- <b>clone()</b> - клонирует объект методом.</li> <li>- <b>finalize()</b> - deprecated, вызывается GC перед удалением. (нет гарантии что будет вызван)</li> </ul> <p>для многопоточки</p> <ul style="list-style-type: none"> <li>- <b>notify()</b> - «размораживает» одну случайную нить</li> <li>- <b>notifyAll()</b> - «размораживает» все нити данного монитора</li> <li>- <b>wait()</b> - нить освобождает монитор и «становится на паузу»</li> <li>- <b>wait(long timeOut)</b> - нить освобождает монитор и «становится на паузу»,принимает максимальное время ожидания в миллисекундах.</li> <li>- <b>wait(long timeOut, int nanos)</b> - нить освобождает монитор и «становится на паузу»,принимает максимальное время ожидания в миллисекундах, дополнительное время, в диапазоне наносекунд 0-999999.</li> </ul>
<p>Расскажите про equals и hashCode</p>	<p><b>Хеш-код</b> — это <b>целочисленный результат работы метода, которому</b> в качестве входного параметра <b>передан объект</b>.  <b>Если более точно, то это битовая строка фиксированной длины, полученная из массива произвольной длины.</b></p> <p><b>Equals</b> - это метод, определенный в Object, который служит для сравнения объектов. <b>При сравнении объектов при помощи == идет сравнение по ссылкам. При сравнении по equals() идет сравнение по состояниям</b> объектов.</p> <p><b>Свойства equals():</b></p> <ul style="list-style-type: none"> <li>• <b>Симметричность:</b> Для двух ссылок, a и b, <code>a.equals(b)</code> тогда и только тогда, когда <code>b.equals(a)</code></li> <li>• <b>Рефлексивность:</b> для любого заданного значения x, выражение <code>x.equals(x)</code> должно возвращать <code>true</code>.  Заданного — имеется в виду такого, что <code>x != null</code></li> <li>• <b>Постоянство:</b> повторный вызов метода <code>equals()</code> должен возвращать одно и тоже значение до тех пор, пока какое-либо значение свойств объекта не будет изменено.</li> <li>• <b>Транзитивность:</b> Если <code>a.equals(b)</code> и <code>b.equals(c)</code>, то тогда <code>a.equals(c)</code></li> <li>• <b>Совместимость с hashCode():</b> Два тождественно равных объекта должны иметь одно и то же значение <code>hashCode()</code></li> </ul> <p>При переопределении <code>equals()</code> обязательно нужно переопределить метод <code>hashCode()</code>. Равные объекты должны возвращать одинаковые хэш коды.</p>



<p><b>Каким образом реализованы методы hashCode() и equals() в классе Object?</b></p>	<p>1 - Реализация метода Object.equals() сводится к проверке на равенство двух ссылок:</p> <pre>public boolean equals(Object obj) {     return (this == obj); }</pre> <p>2 - hashCode реализован таким образом, что для одного и того же входного объекта, хеш-код всегда будет одинаковым. Реализация метода Object.hashCode() описана как native, т.е. написана не на Java. Непереопределенный hashCode возвращает идентификационный хеш, основанный на состоянии потока, объединённого с xorshift (в OpenJDK8). А вообще, функция предлагает шесть методов на базе значения переменной hashCode.</p> <ol style="list-style-type: none"> <li>0. Случайно сгенерированное число.</li> <li>1. Функция адреса объекта в памяти.</li> <li>2. Жёстко запрограммированное значение 1 (используется при тестировании на чувствительность (sensitivity testing)).</li> <li>3. Последовательность.</li> <li>4. Адрес объекта в памяти, приведённый к целочисленному значению.</li> <li>5. Состояние потока, объединённое с xorshift (<a href="https://en.wikipedia.org/wiki/Xorshift">https://en.wikipedia.org/wiki/Xorshift</a>)</li> </ol> <pre>public native int hashCode();</pre> <p>Ситуация, когда у разных объектов одинаковые хеш-коды называется — коллизией. Вероятность возникновения коллизии зависит от используемого алгоритма генерации хеш-кода.</p>
<p><b>Зачем нужен equals(). Чем он отличается от операции ==?</b></p>	<p><b>equals()</b> - сравнение <b>по состоянию</b>, <b>==</b> - <b>по ссылкам</b></p>
<p><b>Правила переопределения equals()</b></p>	<ul style="list-style-type: none"> <li>-<b>Рефлексивность</b>: Объект должен равняться себе самому.</li> <li>-<b>Симметричность</b>: если a.equals(b) возвращает true, то b.equals(a) должен тоже вернуть true.</li> <li>-<b>Транзитивность</b>: если a.equals(b) возвращает true и b.equals(c) тоже возвращает true, то c.equals(a) тоже должен возвращать true.</li> <li>-<b>Согласованность</b>: повторный вызов метода equals() должен возвращать одно и тоже значение до тех пор, пока какое-либо значение свойств объекта не будет изменено. То есть, если два объекта равны в Java, то они будут равны пока их свойства остаются неизменными.</li> <li>-<b>Неравенство с null</b>: объект должны быть проверен на null. Если объект равен null, то метод должен вернуть false, а не NullPointerException. Например, a.equals(null) должен вернуть false.</li> </ul> <pre>@Override public boolean equals(Object o) {     if (this == o) return true;     if (o == null    getClass() != o.getClass()) return false;     Man man = (Man) o;     return dnaCode == man.dnaCode; }</pre>
<p><b>что будет если переопределить equals() и не переопределить hashCode()</b></p>	<p>Нарушится контракт. Классы и методы, которые использовали правила этого контракта могут некорректно работать. Так для объекта HashMap это может привести к тому, что пара, которая была помещена в Map возможно не будет найдена в ней при обращении к Map, если используется новый экземпляр ключа.</p>

Какой контракт между hashCode() и equals()?	1) Если два объекта возвращают разные значения hashCode(), то они не могут быть равны 2) Если equals объектов true, то и хэшкоды должны быть равны. 3) Переопределив equals, всегда переопределять и hashCode.
Для чего нужен метод hashCode()?	вычисляет целочисленное значение для конкретного элемента класса, чтобы использовать его для быстрого поиска и доступа к этому элементу в hash-структурах данных, например, HashMap, HashSet и прочих.
- Правила переопределения метода hashCode().	Если хеш-коды разные, то и входные объекты гарантированно разные. Если хеш-коды равны, то входные объекты не всегда равны. При вычислении хэш-кода следует использовать те же поля, которые сравниваются в equals и которые не вычисляются на основе других значений.  - вызов метода hashCode один и более раз над одним и тем же объектом должен возвращать одно и то же хэш-значение, при условии что поля объекта, участвующие в вычислении значения, не изменялись. - вызов метода hashCode над двумя объектами должен всегда возвращать одно и то же число, если эти объекты равны (вызов метода equals для этих объектов возвращает true). - вызов метода hashCode над двумя неравными между собой объектами должен возвращать разные хэш-значения. Хотя это требование и не является обязательным, следует учитывать, что его выполнение положительно повлияет на производительность работы хэш-таблиц.
- Есть ли какие-либо рекомендации о том, какие поля следует использовать при подсчете hashCode()?	Выбирать поля, которые с большой долей вероятности будут различаться. Для этого необходимо использовать уникальные, лучше всего примитивные поля, например такие как id, uuid. При этом нужно следовать правилу, если поля задействованы при вычислении hashCode(), то они должны быть задействованы и при выполнении equals().
Могут ли у разных объектов быть одинаковые hashCode()?	Когда у разных объектов одинаковые хеш-коды называется — <b>коллизией</b> .
Почему нельзя реализовать hashCode() который будет гарантированно уникальным для каждого объекта?	В Java множество возможных хэш кодов ограничено типом int, а множество объектов ничем не ограничено. Из-за этого, вполне возможна ситуация, что хэш коды разных объектов могут совпасть
Есть класс Point{int x, y;}. Почему хэш-код в виде $31 * x + y$ предпочтительнее чем $x + y$ ?	Множитель создает зависимость значения хэш-кода от очередности обработки полей, а это дает гораздо лучшую хэш-функцию.
Чем a.getClass().equals(A.class) отличается от a instanceof A.class	getClass() получает только класс, а оператор instanceof проверяет является ли объект экземпляром класса или его потомком
Исключения	
Что такое исключения?	Исключение — это ошибка (является объектом), возникающая во время выполнения программы.

Опишите иерархию исключений.	<p>1. класс <b>Throwable</b> (checked)</p> <p>2. от <b>Throwable</b> -&gt; <b>Error</b> (ошибки JVM) и <b>Exception</b> (checked общие)</p> <p>3. от <b>Exception</b></p> <ul style="list-style-type: none"> <li>- &gt; <b>RuntimeException</b> (unchecked)</li> <li>- &gt; <b>IOException</b>, <b>SQLException</b>, <b>ReflectiveOperationException</b> (checked)</li> </ul> <p>4. <b>RuntimeException</b> (unchecked):</p> <ul style="list-style-type: none"> <li><b>ClassCastException</b></li> <li><b>IndexOutOfBoundsException</b></li> <li><b>ArithmeticException</b></li> <li><b>NullPointerException</b></li> </ul> <p>checked - зависит от программиста, unchecked - от программиста не зависит</p>
Расскажите про обрабатываемые и необрабатываемые исключения	<p>1. Checked исключения, это те, которые должны обрабатываться блоком catch или описываться в сигнатуре метода. Unchecked могут не обрабатываться и не быть описанными.</p> <p>2. Unchecked исключения в Java — наследованные от RuntimeException, checked — от Exception.</p> <p>Checked исключения отличаются от Unchecked исключения в Java, тем что наличие\обработка Checked исключения проверяются компилятором на этапе компиляции. Наличие\обработка Unchecked исключения происходит на этапе выполнения.</p>
Можно ли обработать необрабатываемые исключения?	Можно, чтобы в некоторых случаях программа не прекратила работу
Какой оператор позволяет принудительно выбросить исключение?	Throw
О чем говорит ключевое слово throws?	Метод потенциально может выбросить исключение с указанным типом. Передаёт обработку исключения вышестоящему методу.
Как создать собственное («пользовательское») исключение?	Необходимо унаследоваться от базового класса требуемого типа исключений (например, от Exception или RuntimeException). и переопределит методы
Расскажите про механизм обработки исключений в java (Try-catch-finally)	<p><b>Try</b> - блок в котором может появиться исключение;</p> <p><b>Catch</b> - блок в котором мы указываем исключение и логику его обработки;</p> <p><b>Finally</b> - блок который обязательно отработает</p>
Возможно ли использование блока try-finally (без catch)?	try может быть в паре с finally, без catch. Работает это точно так же - после выхода из блока try выполняется блок finally
Может ли один блок catch отлавливать сразу несколько исключений?	Да

Всегда ли выполняется блок <b>finally</b> ? Существуют ли ситуации, когда блок <b>finally</b> не будет выполнен?	<p>Да, кроме случаев завершения работы программы или JVM:</p> <ol style="list-style-type: none"> <li>1 - Finally может не выполниться в случае если в блоке try вызывает System.exit(0),</li> <li>2 - Runtime.getRuntime().exit(0), Runtime.getRuntime().halt(0) и если во время исполнения блока try виртуальная машина выполнила недопустимую операцию и будет закрыта.</li> <li>3 - В блоке try{} бесконечный цикл.</li> </ol>
Может ли метод <b>main()</b> выбросить исключение во вне и если да, то где будет происходить обработка данного исключения?	<p>Может и оно будет передано в виртуальную машину Java (JVM). Для случая с методом <b>main</b> произойдет две вещи:</p> <ul style="list-style-type: none"> <li>- будет завершен главный поток приложения;</li> <li>- будет вызван <b>ThreadGroup.uncaughtException</b>.</li> </ul>
В каком порядке следует обрабатывать исключения в <b>catch</b> блоках?	От наследника к предку
Что такое механизм <b>try-with-resources</b> ?	<p>Дает возможность объявлять один или несколько ресурсов в блоке <b>try</b>, которые будут закрыты автоматически без использования <b>finally</b> блока. В качестве ресурса можно использовать любой объект, класс которого реализует интерфейс <b>java.lang.AutoCloseable</b> или <b>java.io.Closeable</b>.</p>
Что произойдет если исключение будет выброшено из блока <b>catch</b> после чего другое исключение будет выброшено из блока <b>finally</b> ?	<b>finally</b> -секция может «перебить» <b>throw/return</b> при помощи другого <b>throw/return</b>
Что произойдет если исключение будет выброшено из блока <b>catch</b> после чего другое исключение будет выброшено из метода <b>close()</b> при использовании <b>try-with-resources</b> ?	В <b>try-with-resources</b> добавлена возможность хранения "подавленных" исключений, и брошенное try-блоком исключение имеет больший приоритет, чем исключения получившиеся во время закрытия.
Сериализация и копирование	
Что такое сериализация и как она реализована в Java?	<p>Сериализация это процесс сохранения состояния объекта в последовательность байт;</p> <p>Реализована через интерфейс - маркер <b>Serializable</b>.</p>
Для чего нужна сериализация?	Для компактного сохранения состояния объекта и считывание этого состояния.

Опишите процесс сериализации/десериализации с использованием Serializable.	<p>1) Класс объекта должен реализовывать интерфейс Serializable</p> <p>2) Создать поток ObjectOutputStream (oos), который записывает объект в переданный OutputStream.</p> <p>3) Записать в поток: oos.writeObject(Object);</p> <p>4) Сделать oos.flush() и oos.close()</p>
Как изменить стандартное поведение сериализации/десериализации?	<p>Использовать интерфейс <b>Externalizable</b>.</p> <p>Переопределить методы  writeExternal(ObjectOutput out) throws IOException  readExternal(ObjectInput in) throws IOException, ClassNotFoundException</p>
Какие поля не будут сериализованы при сериализации? Будет ли сериализовано final поле?	<p>1) Добавить к полю модификатор <b>transient</b>. В таком случае после восстановления его значение будет null.</p> <p>2) Сделать поле <b>static</b>. Значения статических полей автоматически не сохраняются.</p> <p>3) Поля с модификатором <b>final</b> сериализуются <b>как и обычные</b>. За одним <b>исключением</b> – их <b>невозможно десериализовать при</b> использовании <b>Externalizable</b>, поскольку final-поля должны быть инициализированы в конструкторе, а после этого в readExternal изменить значение этого поля будет невозможно. Соответственно, если необходимо сериализовать объект с final-полем необходимо использовать только стандартную сериализацию.</p>
Как создать собственный протокол сериализации?	<p>Для создания собственного протокола нужно просто <b>переопределить writeExternal() и readExternal()</b>.</p> <p>В отличие от двух других вариантов сериализации, здесь ничего не делается автоматически. Протокол полностью в ваших руках.</p>
Какая роль поля serialVersionUID в сериализации?	<p>Поле private static final long serialVersionUID содержит <b>уникальный идентификатор</b> версии сериализованного класса. Оно вычисляется по содержимому класса - полям, их порядку объявления, методам, их порядку объявления. Соответственно, <b>при любом изменении в классе это поле поменяет свое значение</b>. Если мы не объявляем его явно, Java делает это за нас.</p>
Когда стоит изменять значение поля serialVersionUID?	<p>Вы должны изменить serialVersionUID только тогда, <b>когда вы сознательно хотите нарушить совместимость со всеми существующими сериализациями</b>, например, когда изменения в вашем классе сделают его настолько семантически отличным, что у вас не будет выбора - в этом случае вы действительно должны несколько раз подумать о том, что вы на самом деле делаете.</p>
В чем проблема сериализации Singleton?	<p><b>- Проблема -</b>  в том что <b>после десериализации мы получим другой объект</b>.</p> <p>Таким образом, сериализация дает возможность создать Singleton еще раз, что не совсем нужно.</p> <p><b>- Решение -</b>  <b>В классе определяется метод с сигнатурой "Object readResolve() throws ObjectStreamException"</b></p> <p><b>- Назначение -</b>  этого метода - <b>возвращать замещающий объект вместо объекта, на котором он вызван</b>.</p>

<p><b>Расскажите про клонирование объектов.</b></p>	<p>в Java, есть 3 способа клонирования объекта:</p> <ol style="list-style-type: none"> <li><b>1. С использованием интерфейса Cloneable;</b> Первый способ подразумевает, что вы будете использовать механизм так называемого «поверхностного клонирования» и сами позаботитесь о клонировании полей-объектов. Метод clone() в родительском классе Object является protected, поэтому требуется переопределение его с объявлением как public. Он возвращает экземпляр объекта с копированными полями-примитивами и ссылками. И получается что у оригинала и его клона поля-ссылки указывают на одни и те же объекты.</li> <li><b>2. С использованием конструктора клонирования объекта;</b> В классе описывается конструктор, который принимает объект этого же класса и инициализирует значениями его полей поля нового объекта.</li> <li><b>3. С использованием сериализации.</b> Он заключается в сохранении объекта в поток байтов с последующей десериализацией его от туда.</li> </ol>
<p><b>В чем отличие между поверхностным и глубоким клонированием?</b></p>	<p><b>Поверхностное копирование копирует настолько малую часть информации, насколько это возможно. По умолчанию, клонирование в Java является поверхностным</b>, т.е. Object class не знает о структуре класса, которого он копирует. <b>Глубокое копирование дублирует все.</b> Глубокое копирование — это <b>две коллекции</b>, в <b>одну</b> из которых <b>дублируются все элементы оригинальной коллекции</b>.</p>
<p><b>Какой способ клонирования предпочтительней?</b></p>	<p>Наиболее безопасным и следовательно предпочтительным способом клонирования является использование специализированного конструктора копирования: Отсутствие ошибок наследования (не нужно беспокоиться, что у наследников появятся новые поля, которые не будут скопированы через метод clone()); Поля для клонирования указываются явно; Возможность клонировать даже final поля.</p>
<p><b>Почему метод clone() объявлен в классе Object, а не в интерфейсе Cloneable?</b></p>	<p>Метод clone() объявлен в классе Object с сигнатурой native, чтобы обеспечить доступ к стандартному механизму "поверхностного копирования" объектов (копируются значения всех полей, включая ссылки на сторонние объекты); он объявлен, как protected, чтобы нельзя было вызвать этот метод у не переопределивших его объектов.</p>
<p><b>Как создать глубокую копию объекта? (2 способа)</b></p>	<p><b>1 Сериализация</b> – это еще один способ глубокого копирования. Мы просто сериализуем нужный объект и десериализуем его. Очевидно, объект должен поддерживать интерфейс Serializable. Мы сохраняем объект в массив байт и потом прочитать из него.</p> <p><b>2 При помощи библиотеки DeepCloneable</b> Глубокое клонирование с этой библиотекой сводится к двум строкам кода:  <pre>Cloner cloner = new Cloner(); DeepCloneable clone = cloner.deepClone(this);</pre> <ul style="list-style-type: none"> <li>- Переопределение метода clone() и реализация интерфейса Cloneable();</li> <li>- Механизм сериализации - сохранение и последующее восстановление объекта в/из потока байтов.</li> <li>- Конструктор копирования - в классе описывается конструктор, который принимает объект этого же класса и инициализирует поля создаваемого объекта значениями полей переданного;</li> </ul> </p>

## Core-2

### Что такое дженерики?

"Дженерики – это параметризованные типы. С их помощью можно объявлять классы, интерфейсы и методы, в которых тип данных указан в виде параметра.

Используя дженерики, можно создать единственный класс, который будет автоматически работать с разными типами данных. Эта информация доступна только на этапе компиляции и стирается в runtime, и в байт код попадет только информация о том, что в программе есть некий список `List<Object> list` вместо `List<String> list`, например. Появились в версии 1.5

### Для чего нужны дженерики?

Для строгой типизации и проверки на этапе компиляции.

Дженерики позволяют передавать тип объекта компилятору в форме `<тип>`. Таким образом, компилятор может выполнить все необходимые действия по проверке типов во время компиляции, обеспечивая безопасность по приведению типов во время выполнения.

Какую проблему они решают.

- Типобезопасность (Typesafe). Позволяют создавать листы или коллекции определенных типов, которые содержат только определенные элементы, и позволяют находить ошибки на уровне компиляции.

- Повторное использование кода (Reusable code). Позволяют не создавать похожие классы, методы, похожий код, а использовать Generics.

Что можно типизировать.

Параметризованные типы позволяют объявлять классы (кроме класса Enum, Анонимные, и Экспешены), интерфейсы, методы, (конструкторы это тоже методы их не выделять отдельно) и поля, НО только те где тип данных, которыми они оперируют, указан в виде параметра.

Чему эквивалентно `<?>`.

`<?>` wildcard с неограниченным символом подстановки. Мы просто ставим `<?>`, без ключевых слов `super` или `extends`. На самом деле такой «неограниченный» wildcard все-таки ограничен, сверху. `Collection<?>` — это тоже символ подстановки, как и `"? extends Object"`.

### Что такое стирание?

Его суть заключается в том, что внутри класса не хранится никакой информации о его типе-параметре. Эта информация доступна только на этапе компиляции и стирается (становится недоступной) в runtime.

Когда весь написанный тобой Java-код превратится в байт-код, в нем не будет информации о типах-параметрах.

----- Тип параметра `<T>` преобразуется в `Object`

----- Тип параметра `<T extends Comparable>` в `Comparable`

----- Запись вида `List<String>` преобразуется в `List` и т.д.

Стирание типов возникло потому, что при разработке Java важна обратная совместимость. Виртуальная машина не умеет работать с генериками изначально, поэтому их нужно "убирать" после компиляции.

<p><b>Что такое сырые типы (raw type)?</b></p>	<p>Raw type - это не параметризированный объект. В Java так называют generic-типы без указания типа-параметра. Такая языковая конструкция валидна, но в большинстве случаев приводит к предупреждению компилятора.</p> <p>К чему приводит использование raw type.          Теряется безопасность типов, если использовать необработанный тип. В то время как Джeneralик явно сообщил компилятору, что он способен удерживать объекты любого типа, сырой тип отказался от проверки общих типов.          При использовании необработанного типа Java предполагает, что тип элементов в этом списке равен Object . Таким образом, list.get(0) вернет ссылку типа Object .</p> <p>Хоть где-то можно и нужно использовать Raw Types в своём коде без риска получить ошибку?          Джeneralики Java не являются овеществленными, есть два исключения, когда необработанные типы (Raw Types) должны использоваться в новом коде:          - Литералы класса, например List.class, а не List&lt;String&gt;.class.          - Операнд instanceof , например o instanceof Set, а не o instanceof Set&lt;String&gt;</p>
<p><b>Что такое вайлдкарды?</b></p>	<p>Запись вида "? extends ..." или "? super ..." — называется wildcard или символом подстановки, с верхней границей (extends) или с нижней границей (super).</p> <p>Для решения проблемы совместимости используется Wildcard («?»). Он не имеет ограничения в использовании (то есть имеет соответствие с любым типом) и в этом его плюсы. Мы можем описать "неизвестный тип" символом вопроса, так называемого question mark.</p> <p>Благодаря Wildcard &lt;?&gt; можно сделать универсальный метод (или переменную), работающий с разными типами данных. Wildcard — удобный инструмент, чтобы смягчить некоторые ограничения дженериков. Джeneralики инвариантны. Это значит что хотя все классы являются наследниками (подтипами, subtypes) типа Object, List&lt;любой тип&gt; не является подтипом List&lt;Object&gt;, но, List&lt;любой тип&gt; является подтипом List&lt;?&gt;.</p> <p>Как и обычные дженерики, дженерики с wildcard могут быть ограничены. Ограничение по верхней границе (Upper bounded wildcard - &lt;? extends Number&gt;) и по нижней границе (Lower bound wildcard - &lt;? super Integer&gt;)</p>
<p><b>Расскажите про принцип PECS</b></p>	<p>Producer Extends Consumer Super</p> <ul style="list-style-type: none"> <li>• Если мы объявили wildcard с extends, то это producer. Он только «продюсирует», предоставляет элемент из контейнера, а сам ничего не принимает. (аргумент - Производитель)</li> <li>• Если же мы объявили wildcard с super — то это consumer. Он только принимает, а предоставить ничего не может. (аргумент - Потребитель)</li> </ul> <p>Если метод читает данные из аргумента, то этот аргумент - производитель, а если метод передаёт данные в аргумент, то аргумент является потребителем. Важно заметить, что, определяя производителя или потребителя, мы рассматриваем только данные типа T.</p> <p>Из одного только читать, а в другой только записывать (исключением является возможность записать null для extends и прочитать Object для super).</p>



## Коллекции

### Что такое «коллекция»?

Коллекция – это объект, который содержит набор объектов одного типа. Каждый из этих объектов в коллекции называется элементом.

Коллекции - это хранилища или контейнеры, поддерживающие различные способы накопления и упорядочения объектов с целью обеспечения возможностей эффективного доступа к ним.

Они представляют собой реализацию абстрактных структур данных, поддерживающих

три основные операции:

добавление нового элемента в коллекцию;

удаление элемента из коллекции;

изменение элемента в коллекции.

Какие есть типы коллекций? Как они характеризуются?

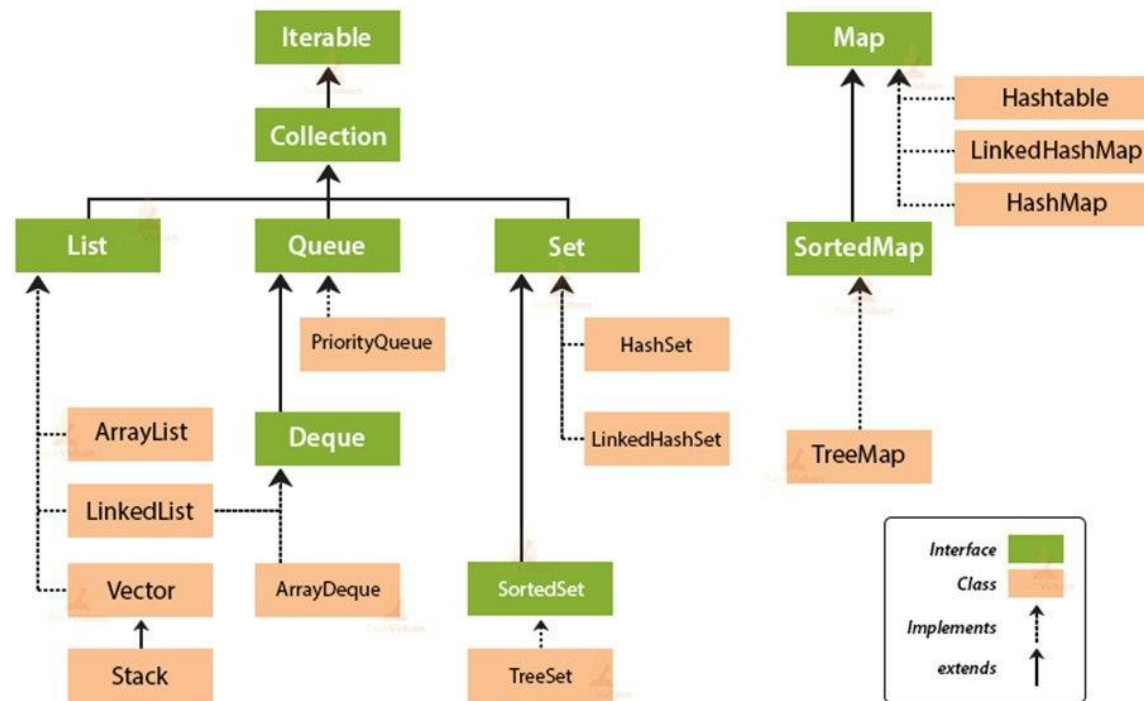
- Set - неупорядоченное множество уникальных (по equals) объектов. Set откажется добавлять объект, если такой объект в нем уже есть.

- List - упорядоченный список объектов (в том числе одинаковых).

- Queue - очередь ждущих обработки объектов. По запросу выдает один самый приоритетный объект. Бывает очередь FIFO, бывает Priority.

- Map - множество объектов-пар вида ключ --> значение.

## Collection Framework Hierarchy in Java



Расскажите про иерархию коллекций

Почему Map — это не Collection, в то время как List и Set являются Collection?

Коллекция (List и Set) представляет собой совокупность некоторых элементов (обычно экземпляров одного класса). Map -это совокупность пар "ключ"- "значение".  
У map нет итерабл, не понятно по чему проводить итерацию

Отличие коллекции от массива.

1. Массивы имеют фиксированный размер при создании, коллекции динамически расширяются.
2. У массивов нет защиты от изменений. final действует на ссылку, а не на массив.
3. Коллекции могут разрешать чтение, но запрещать изменение содержимого.
4. Коллекции имеют полное разнообразие методов.
5. Коллекции работают только с элементами ссылочного типа, потому что они автоматически параметризованы.

<p><b>В чем разница между <code>java.util.Collection</code> и <code>java.util.Collections</code>?</b></p>	<p>Класс <code>java.util.Collections</code> содержит исключительно статические методы для работы с коллекциями. В них входят методы, реализующие полиморфные алгоритмы (такие алгоритмы, использование которых возможно с разными видами структур данных), "оболочки", возвращающие новую коллекцию с инкапсулированной указанной структурой данных и некоторые другие методы.</p> <p><code>java.util.Collection</code> - это корневой интерфейс <b>Java Collections Framework</b>. Этот интерфейс в основном применяется там, где требуется высокий уровень абстракции, например, в классе <code>java.util.Collections</code>.</p>
<p><b>Какая разница между итераторами с <code>fail-fast</code> и <code>fail-safe</code> поведением? (С примерами)</b></p>	<p><b>Итератор <code>fail-safe</code> не вызывает исключений при изменении структуры коллекции, потому что работает с её клоном.</b> Пример <code>fail-safe</code> - <code>CopyOnWriteArrayList</code> и итератор <code>keySet</code> коллекции <code>ConcurrentHashMap</code>.</p> <p><b>Итератор <code>fail-fast</code> генерирует исключение <code>ConcurrentModificationException</code>, если коллекция меняется во время итерации, но работает быстрее.</b> Пример <code>fail-fast</code> - <code>Vector</code> и <code>Hashtable</code>.</p>
<p><b>Чем различаются <code>Enumeration</code> и <code>Iterator</code>?</b></p>	<p><code>Iterator</code> имеет больше методов работы с коллекциями и был специально введен в <code>java2</code>, вместо <code>Enumeration</code>(interface). Рекомендуется юзать <code>Iterator</code>.</p> <p>Оба интерфейса предназначены для обхода коллекции, но есть различия:</p> <ul style="list-style-type: none"> <li>-с помощью <code>Enumeration</code> нельзя добавлять/удалять элементы;</li> <li>-в <code>Iterator</code> исправлены имена методов для повышения читаемости кода (<code>Enumeration.hasMoreElements()</code> соответствует <code>Iterator.hasNext()</code>, <code>Enumeration.nextElement()</code> соответствует <code>Iterator.next()</code> и т.д);</li> <li>-<code>Enumeration</code> присутствуют в устаревших классах, таких как <code>Vector/Stack</code>, тогда как <code>Iterator</code> есть во всех современных коллекциях.</li> </ul>
<p><b>Как между собой связаны <code>Iterable</code>, <code>Iterator</code> и «for-each»?</b></p>	<ul style="list-style-type: none"> <li>- Классы, реализующие интерфейс <code>Iterable</code>, могут применяться в конструкции <code>forEach</code>, которая использует <code>Iterator</code>.</li> <li>- При работе с <code>forEach</code> нельзя одновременно «идти по коллекции циклом» и удалять из неё элементы. Это всё из-за устройства итератора.</li> <li>- В цикле <code>forEach</code> использование итератора скрыто полностью. (позволяет сделать код лаконичнее)</li> <li>- Цикл <code>forEach</code> можно использовать для любых объектов, которые поддерживают итератор. Т.е. ты можешь написать свой класс, добавить ему метод <code>iterator()</code> и сможешь использовать его объекты в правой части конструкции <code>forEach</code>.</li> </ul>
<p><b>Можно ли итерируясь по <code>ArrayList</code> удалить элемент? Какое вылетит исключение?</b></p>	<p>Можно, но нужно использовать <code>iterator.remove()</code>. Иначе при прохождении по <code>ArrayList</code> в цикле <code>for</code> сразу после удаления элемента будет <code>ConcurrentModificationException</code>.</p>

<b>listIterator - что это, в чём отличие от обычного.</b>	<ul style="list-style-type: none"> <li>- ListIterator расширяет интерфейс Iterator</li> <li>- ListIterator может быть использован только для перебора элементов коллекции List;</li> <li>- Iterator позволяет перебирать элементы только в одном направлении, при помощи метода next(). Тогда как ListIterator позволяет перебирать список в обоих направлениях, при помощи методов next() и previous();</li> <li>- ListIterator не указывает на конкретный элемент: его текущая позиция располагается между элементами, которые возвращают методы previous() и next().</li> <li>- При помощи ListIterator можно модифицировать список, добавляя/удаляя элементы с помощью методов add() и remove(). Iterator не поддерживает данного функционала.</li> </ul>
<b>В чём разница между Iterable и Iterator.</b>	<p>Iterable - Он не имеет никакого состояния итерации, такого как "текущий элемент". Проходит все. Вместо этого, он имеет один метод, который производит Iterator. forEach ()</p> <p>Iterator - это интерфейс с состоянием итерации. Это позволяет проверить, если он имеет больше элементов с помощью hasNext() и перейти к следующему элементу с помощью next().</p> <p>Iterable должен быть в состоянии произвести любое количество действующих iterators.</p> <p>Интерфейс Iterable содержит только один абстрактный метод.</p>
<b>Как поведёт себя коллекция, если вызвать iterator.remove()?</b>	<p>Если вызову iterator.remove() предшествовал вызов iterator.next(), то iterator.remove() удалит элемент коллекции, на который указывает итератор, в противном случае будет выброшено IllegalStateException().</p> <p>Попытка удаления элемента при итерации с помощью цикла приведет к исключению.</p>
<b>Чем Set отличается от List?</b>	<p>1) List позволяет дублировать элементы. Set содержит только уникальные элементы.</p> <p>2) List - упорядоченная последовательность элементов (LinkedList, ArrayList, Vector), тогда как</p> <p>3) Set — это отдельный список неупорядоченных элементов (HashSet, LinkedHashSet, TreeSet).</p> <p>Хотя Set предоставляет другую альтернативу SortedSet, которая может хранить элементы Set в определенном порядке сортировки, определенные методами Comparable и Comparator для объектов, хранящихся в Set.</p> <p>ГЕТ</p>
<b>Расскажите про интерфейс Set.</b>	<p>Интерфейс Set расширяет интерфейс Collection.</p> <p>Set не добавляет новых методов, только вносит изменения унаследованные.</p> <p>Set - неупорядоченный набор неповторяющихся элементов</p> <p>В частности, метод add() добавляет элемент в коллекцию и возвращает true, если не было такого элемента.</p> <p>Разрешено наличие только одной ссылки типа null.</p>
<b>Расскажите про реализации интерфейса Set</b>	<p>В HashSet порядок добавления элементов будет непредсказуемым - используется хэширование для ускорения выборки.</p> <p>В TreeSet объекты хранятся отсортированными по возрастанию из-за применения к/ч дерева.</p> <p>LinkedHashSet хранит элементы в порядке добавления.</p>
<b>В чем отличия TreeSet и HashSet?</b>	<p>HashSet быстрее, чем TreeSet .</p> <p>В HashSet элементы в случайном порядке, в TreeSet в отсортированном.</p> <p>HashSet обеспечивает постоянную производительность - O(1) - для большинства операций, таких как add () , remove () и contains () , по сравнению с временем log(n), предлагаемым TreeSet.</p>

Чем <code>LinkedHashSet</code> отличается от <code>HashSet</code> ?	Основное различие в том, что <code>LinkedHashSet</code> сохраняет порядок вставки элементов, а <code>HashSet</code> - нет. В основе <code>LinkedHashSet</code> лежит <code>LinkedHashMap</code> вместо <code>HashMap</code> . Благодаря этому порядок элементов при обходе коллекции является идентичным порядку добавления элементов																								
Что будет, если добавлять элементы в <code>TreeSet</code> по возрастанию?	<code>TreeSet</code> все равно в каком порядке вы добавляете в него элементы, так как в основе <code>TreeSet</code> лежит красно-черное дерево, которое умеет само себя балансировать и хранить элементы по возрастанию.																								
Как устроен <code>HashSet</code> , сложность основных операций.	<table><tr><th colspan="4">HashSet — временная сложность основных операций</th></tr><tr><th></th><th>Поиск</th><th>Вставка</th><th>Удаление</th></tr><tr><td>Метод</td><td><code>contains(object)</code></td><td><code>add(object)</code></td><td><code>remove(index)</code></td></tr><tr><td>Среднее</td><td><math>O(1)</math></td><td><math>O(1)</math></td><td><math>O(1)</math></td></tr><tr><td>Худшее (до Java 8)</td><td><math>O(n)</math></td><td><math>O(n)</math></td><td><math>O(n)</math></td></tr><tr><td>Худшее (Java 8+)</td><td><math>O(\log_2(n))</math></td><td><math>O(\log_2(n))</math></td><td><math>O(\log_2(n))</math></td></tr></table>	HashSet — временная сложность основных операций					Поиск	Вставка	Удаление	Метод	<code>contains(object)</code>	<code>add(object)</code>	<code>remove(index)</code>	Среднее	$O(1)$	$O(1)$	$O(1)$	Худшее (до Java 8)	$O(n)$	$O(n)$	$O(n)$	Худшее (Java 8+)	$O(\log_2(n))$	$O(\log_2(n))$	$O(\log_2(n))$
	HashSet — временная сложность основных операций																								
	Поиск	Вставка	Удаление																						
Метод	<code>contains(object)</code>	<code>add(object)</code>	<code>remove(index)</code>																						
Среднее	$O(1)$	$O(1)$	$O(1)$																						
Худшее (до Java 8)	$O(n)$	$O(n)$	$O(n)$																						
Худшее (Java 8+)	$O(\log_2(n))$	$O(\log_2(n))$	$O(\log_2(n))$																						
	Все классы, реализующие интерфейс <code>Set</code> , внутренне поддерживаются реализациями <code>Map</code> . <code>HashSet</code> хранит элементы с помощью <code>HashMap</code> . Значение, которые мы передаем в <code>HashSet</code> , является ключом к объекту <code>HashMap</code> , а в качестве значения используется <code>Object</code> .																								
Как устроен <code>LinkedHashSet</code> , сложность основных операций.	<table><tr><th colspan="4">LinkedHashSet — временная сложность основных операций</th></tr><tr><th></th><th>Поиск</th><th>Вставка</th><th>Удаление</th></tr><tr><td>Метод</td><td><code>contains(object)</code></td><td><code>add(object)</code></td><td><code>remove(index)</code></td></tr><tr><td>Среднее</td><td><math>O(1)</math></td><td><math>O(1)</math></td><td><math>O(1)</math></td></tr><tr><td>Худшее (до Java 8)</td><td><math>O(n)</math></td><td><math>O(n)</math></td><td><math>O(n)</math></td></tr><tr><td>Худшее (Java 8+)</td><td><math>O(\log_2(n))</math></td><td><math>O(\log_2(n))</math></td><td><math>O(\log_2(n))</math></td></tr></table>	LinkedHashSet — временная сложность основных операций					Поиск	Вставка	Удаление	Метод	<code>contains(object)</code>	<code>add(object)</code>	<code>remove(index)</code>	Среднее	$O(1)$	$O(1)$	$O(1)$	Худшее (до Java 8)	$O(n)$	$O(n)$	$O(n)$	Худшее (Java 8+)	$O(\log_2(n))$	$O(\log_2(n))$	$O(\log_2(n))$
	LinkedHashSet — временная сложность основных операций																								
	Поиск	Вставка	Удаление																						
Метод	<code>contains(object)</code>	<code>add(object)</code>	<code>remove(index)</code>																						
Среднее	$O(1)$	$O(1)$	$O(1)$																						
Худшее (до Java 8)	$O(n)$	$O(n)$	$O(n)$																						
Худшее (Java 8+)	$O(\log_2(n))$	$O(\log_2(n))$	$O(\log_2(n))$																						
	В его основе лежит <code>LinkedHashMap</code> . Благодаря этому порядок элементов при обходе коллекции является идентичным порядку добавления элементов																								

Как устроен TreeSet, сложность основных операций.

TreeSet — временная сложность основных операций			
	Поиск	Вставка	Удаление
Метод	contains(object)	add(object)	remove(index)
Среднее	$O(\log_2(n))$	$O(\log_2(n))$	$O(\log_2(n))$
Худшее	$O(\log_2(n))$	$O(\log_2(n))$	$O(\log_2(n))$

Время для базовых операций - Логарифмическое время.

Гарантирует порядок элементов - в основе лежит красно-черное дерево, которое умеет само себя балансировать.

Не предоставляет каких-либо параметров для настройки производительности

Предоставляет дополнительные методы для упорядоченного списка: first(), last(), headSet(), tailSet()

Расскажите про интерфейс List

Контейнер List хранит элементы в порядке добавления. Интерфейс List дополняет Collection несколькими методами, обеспечивающими вставку и удаление элементов в середине списка.

Как устроен ArrayList, сложность основных операций.

ArrayList — временная сложность основных операций				
	Индекс	Поиск	Вставка	Удаление
Метод	get(i)	contains(object)	add(object)	remove(index)
Среднее	$O(1)$	$O(n)$	$O(n)$	$O(n)$
Худшее	$O(1)$	$O(n)$	$O(n)$	$O(n)$

ArrayList реализован внутри в виде обычного массива. Поэтому при вставке элемента в середину, приходится сначала сдвигать на один все элементы после него, а уже затем в освободившееся место вставлять новый элемент.

Механизм автоматического «расширения» массива существует, а вот автоматического «сжатия» нет, можно только явно выполнить «сжатие» командой trimToSize()

Что такое Queue?

Queue - коллекция, предназначенная для хранения элементов в порядке, нужном для их обработки. Очереди обычно, но не обязательно, упорядочивают элементы в FIFO (first-in-first-out) порядке.

Что такое Deque? Чем отличается от Queue?

Deque - двухсторонняя очередь, расширяет queue. Он отличается от Queue тем, что можно добавлять и удалять элементы как в хвосте так и в голове. Количество методов удваивается. Пример:

```
addFirst(E e);  
addLast(E e);
```

Помимо этого реализации интерфейса Deque могут строиться по принципу FIFO, либо LIFO.

Реализации и Deque, и Queue обычно не переопределяют методы equals() и hashCode(), а используются методы класса Object, основанные на сравнении ссылок.

Рекомендуется использовать вместо устаревшего Stack.

Приведите пример реализации Deque.	Linked list, Stack, ArrayDeque																								
Какая коллекция реализует FIFO?	Queue																								
Какая коллекция реализует LIFO?	Vector, ArrayDeque																								
Оцените количество памяти на хранение одного примитива типа byte в LinkedList?	<p>Каждый элемент LinkedList хранит ссылку на предыдущий элемент, следующий элемент и ссылку на данные. Для x32 систем каждая ссылка занимает 32 бита (4 байта). Сам объект типа Node занимает приблизительно 8 байт. Размер каждого объекта в Java кратен 8, соответственно получаем 24 байта.</p> <p>Примитив типа byte занимает 1 байт памяти, но в списке примитивы упаковываются, соответственно получаем еще 8 байт. Таким образом, в x32 JVM около 32 байтов выделяется для хранения одного значения типа byte в LinkedList.</p> <p>Для 64-битной JVM каждая ссылка занимает 64 бита (8 байт). Вычисления аналогичны. (32 ответ) Посчитать на других переменных!</p>																								
Оцените количество памяти на хранение одного примитива типа byte в ArrayList?	ArrayList основан на массиве. Каждый элемент массива хранит примитивный тип данных - byte, размер которого 1 байт.																								
Какие существуют реализации Map?	TreeMap, HashMap, HashTable, LinkedHashMap																								
	<table><tr><th colspan="4">HashMap — временная сложность основных операций</th></tr><tr><th></th><th>Поиск</th><th>Вставка</th><th>Удаление</th></tr><tr><td>Метод</td><td>get/contains(key)</td><td>put(key, object)</td><td>remove(key)</td></tr><tr><td>Среднее</td><td>O(1)</td><td>O(1)</td><td>O(1)</td></tr><tr><td>Худшее (до Java 8)</td><td>O(n)</td><td>O(n)</td><td>O(n)</td></tr><tr><td>Худшее (Java 8+)</td><td>O(log<sub>2</sub>(n))</td><td>O(log<sub>2</sub>(n))</td><td>O(log<sub>2</sub>(n))</td></tr></table>	HashMap — временная сложность основных операций					Поиск	Вставка	Удаление	Метод	get/contains(key)	put(key, object)	remove(key)	Среднее	O(1)	O(1)	O(1)	Худшее (до Java 8)	O(n)	O(n)	O(n)	Худшее (Java 8+)	O(log <sub>2</sub> (n))	O(log <sub>2</sub> (n))	O(log <sub>2</sub> (n))
HashMap — временная сложность основных операций																									
	Поиск	Вставка	Удаление																						
Метод	get/contains(key)	put(key, object)	remove(key)																						
Среднее	O(1)	O(1)	O(1)																						
Худшее (до Java 8)	O(n)	O(n)	O(n)																						
Худшее (Java 8+)	O(log <sub>2</sub> (n))	O(log <sub>2</sub> (n))	O(log <sub>2</sub> (n))																						

Как устроена HashMap, сложность основных операций? (Расскажите про принцип корзин)

HashMap – внутри состоит из корзин и списка элементов, на которые ссылаются корзины.

Корзины – массив

Элементы(Node) – связанный список, то есть каждый элемент списка имеет указатель на следующий элемент.

При добавлении нового элемента, хэш-код ключа определяет корзину для элемента с помощью `hashFunction()`, который принимает `hashCode` ключа и возвращает номер корзины. В корзине есть ссылка на связанный список, в который будет положен наш объект.

Идет проверка, есть ли элементы в этом списке. Если нету, то корзина получает ссылку нового элемента, если есть, то происходит прохождение по списку элементов и сравнение элементов в списке. Проверяется равенство `hashCode`. Зная о коллизии, проводится еще сравнение ключей методом `equals`.

Если оба равны: идет перезапись

Если не равен `equals`: добавляется элемент в конец списка

HashMap имеет поле `loadFactor`. Оно может быть задано через конструктор. По умолчанию - 0.75. Его произведение на количество корзин дает нам необходимое число объектов которое нужно добавить чтобы состоялось удвоение количества корзин.

Например если у нас мапка с 16-ю(default) корзинами, а `loadFactor` равняется 0.75, то расширение произойдет когда мы добавим  $16 * 0.75 = 12$  объектов.

После удвоения все объекты будут перераспределены с учетом нового количества корзин

Как устроена TreeMap, сложность основных операций?

TreeMap — временная сложность основных операций			
	Поиск	Вставка	Удаление
Метод	<code>get/contains(key)</code>	<code>put(key, object)</code>	<code>remove(key)</code>
Среднее	$O(\log_2(n))$	$O(\log_2(n))$	$O(\log_2(n))$
Худшее	$O(\log_2(n))$	$O(\log_2(n))$	$O(\log_2(n))$

Класс `TreeMap<K, V>` представляет отображение в виде дерева. Он наследуется от класса `AbstractMap` и реализует интерфейс `NavigableMap`, а следовательно, также и интерфейс `SortedMap`. Поэтому в отличие от коллекции `HashMap` в `TreeMap` все объекты автоматически сортируются по возрастанию их ключей.

Как работает HashMap при попытке сохранить в него два элемента по ключам с одинаковым `hashCode()`, но для которых `equals() == false`?

По значению `hashCode()` вычисляется индекс ячейки массива, в список которой этот элемент будет добавлен. Перед добавлением осуществляется проверка на наличие элементов в этой ячейке. Если элементы с таким `hashCode()` уже присутствует, но их `equals()` методы не равны, то элемент будет добавлен в конец списка.



<p><b>Что будет, если мы кладем в HashMap ключ, у которого equals и hashCode определены некорректно?</b></p>	<p>Объект скорее всего добавится, но обратно мы не сможем получить его.</p>
<p><b>Возможна ли ситуация, когда HashMap вырождается в список даже с ключами имеющими разные hashCode()?</b></p>	<p>Это возможно в случае, если метод, определяющий номер корзины будет возвращать одинаковые значения.</p>
<p><b>Почему нельзя использовать byte[] в качестве ключа в HashMap?</b></p>	<p>Хэш-код массива не зависит от хранимых в нем элементов, а присваивается при создании массива (метод вычисления хэш-кода массива не переопределен и вычисляется по стандартному Object.hashCode() на основании адреса массива). Также у массивов не переопределен equals и выполняется сравнение указателей. Это приводит к тому, что обратиться к сохраненному с ключом-массивом элементу не получится при использовании другого массива такого же размера и с такими же элементами, доступ можно осуществить лишь в одном случае — при использовании той же самой ссылки на массив, что использовалась для сохранения элемента.</p>
<p><b>Что такое коллизия. Что происходит при коллизии. Как будет разрешаться коллизия.</b></p>	<p>1) Коллизия в Map - это когда два и более элемента попадают в одну "корзину" - ячейку внутреннего хранилища Map. увеличивается время работы.  2 причины?  - Значение хеша ограничено диапазоном значений типа int (порядка 4 млрд.). В hashCode контракте, неравные объекты могут иметь один и тот же код hash.  - Корзина, может быть, повторно использована даже при наличии большого количества пустых корзин при добавлении двух (неравных) объектов, которые попадают в одну корзину (хэш-код % buckets.length). Когда корзины заполняются, у нового хешируемого объекта, больше шансов оказаться в непустой корзине.</p> <p>Как будет разрешаться коллизия?  Полученное новое значение также нужно куда-то записать, и для этого нужно определить, куда именно оно будет записано. Это называется решением коллизии.  Существует два подхода:  - external chaining или метод цепочек (реализован в HashMap) — т.е. в ячейке на самом деле содержится список (chain). А уже в списке может содержаться несколько значений (не обязательно с одинаковым хеш-кодом).  - linear probing или метод открытой адресации (реализован в IdentityHashMap) – заключается в поиске первой пустой ячейки после той, на которую указала хеш-функция;  В итоге, добавление элементов при коллизии (в пределах одной корзины) выглядит следующим образом:  - проверяем с помощью методов hashCode() и equals(), что оба ключа одинаковы.  - если ключи одинаковы, заменить текущее значение новым.  - иначе связать новый и старый объекты с помощью структуры данных "связный список", указав ссылку на следующий объект в текущем и сохранить оба под нужным индексом; либо осуществить переход к древовидной структуре.</p>

Будет ли работать HashMap, если все добавляемые ключи будут иметь одинаковый hashCode()?	Да, будет, но в этом случае HashMap вырождается в связный список и теряет свои преимущества.																																																																																																			
Какое худшее время работы метода get(key) для ключа, которого нет в HashMap?	O(N). Худший случай - это поиск ключа в таблице, вырожденной в список, перебор ключей которой занимает линейно пропорциональное время количеству хранимых элементов.																																																																																																			
Какое худшее время работы метода get(key) для ключа, который есть в HashMap?	O(N) - линейное																																																																																																			
Начальная ёмкость коллекций	<table><tr><th>Name</th><th>Base Class</th><th>Base Interface</th><th>AD</th><th>AN</th><th>Inserted Order?</th><th>Sorted Order?</th><th>Synch-roniz-ed</th><th>Random Access</th><th>Default Capacity</th><th>Description</th></tr><tr><td>ArrayList</td><td>AbstractList</td><td>List</td><td>Yes</td><td>Yes</td><td>Yes</td><td>No</td><td>No</td><td>Yes</td><td>10</td><td>It supports dynamic arrays that can grow as needed.</td></tr><tr><td>LinkedList</td><td>Abstract SequentialList</td><td>List, Deque, Queue</td><td>Yes</td><td>Yes</td><td>Yes</td><td>No</td><td>No</td><td>Yes</td><td>0</td><td>It provides a Linked List data structure.</td></tr><tr><td>HashSet</td><td>AbstractSet</td><td>Set</td><td>No</td><td>Yes</td><td>No</td><td>No</td><td>No</td><td>No</td><td>16</td><td>It creates a collection that uses hash table for storage.</td></tr><tr><td>LinkedHashSet</td><td>HashSet</td><td>-</td><td>No</td><td>Yes</td><td>Yes</td><td>No</td><td>No</td><td>No</td><td>16</td><td>It creates a Linked List with no duplicate elements.</td></tr><tr><td>TreeSet</td><td>AbstractSet</td><td>Navigable Set</td><td>No</td><td>No</td><td>No</td><td>Yes</td><td>No</td><td>No</td><td>16</td><td>It creates a collection that uses tree for storage.By default, objects are stored in ascending order.</td></tr><tr><td>PriorityQueue</td><td>Abstract Queue</td><td>Queue</td><td>Yes</td><td>No</td><td>No</td><td>Yes</td><td>No</td><td>No</td><td>11</td><td>It creates a queue that is prioritized based on queue's comparator.</td></tr><tr><td>ArrayDeque</td><td>Abstract Collection</td><td>Deque</td><td>Yes</td><td>No</td><td>Yes</td><td>No</td><td>No</td><td>No</td><td>16</td><td>It creates a dynamic array.</td></tr><tr><td>EnumSet</td><td>AbstractSet</td><td>Set</td><td>No</td><td>No</td><td>Yes</td><td>No</td><td>No</td><td></td><td></td><td>It is specifically for use with elements of enum type.</td></tr></table>	Name	Base Class	Base Interface	AD	AN	Inserted Order?	Sorted Order?	Synch-roniz-ed	Random Access	Default Capacity	Description	ArrayList	AbstractList	List	Yes	Yes	Yes	No	No	Yes	10	It supports dynamic arrays that can grow as needed.	LinkedList	Abstract SequentialList	List, Deque, Queue	Yes	Yes	Yes	No	No	Yes	0	It provides a Linked List data structure.	HashSet	AbstractSet	Set	No	Yes	No	No	No	No	16	It creates a collection that uses hash table for storage.	LinkedHashSet	HashSet	-	No	Yes	Yes	No	No	No	16	It creates a Linked List with no duplicate elements.	TreeSet	AbstractSet	Navigable Set	No	No	No	Yes	No	No	16	It creates a collection that uses tree for storage.By default, objects are stored in ascending order.	PriorityQueue	Abstract Queue	Queue	Yes	No	No	Yes	No	No	11	It creates a queue that is prioritized based on queue's comparator.	ArrayDeque	Abstract Collection	Deque	Yes	No	Yes	No	No	No	16	It creates a dynamic array.	EnumSet	AbstractSet	Set	No	No	Yes	No	No			It is specifically for use with elements of enum type.
	Name	Base Class	Base Interface	AD	AN	Inserted Order?	Sorted Order?	Synch-roniz-ed	Random Access	Default Capacity	Description																																																																																									
	ArrayList	AbstractList	List	Yes	Yes	Yes	No	No	Yes	10	It supports dynamic arrays that can grow as needed.																																																																																									
	LinkedList	Abstract SequentialList	List, Deque, Queue	Yes	Yes	Yes	No	No	Yes	0	It provides a Linked List data structure.																																																																																									
	HashSet	AbstractSet	Set	No	Yes	No	No	No	No	16	It creates a collection that uses hash table for storage.																																																																																									
	LinkedHashSet	HashSet	-	No	Yes	Yes	No	No	No	16	It creates a Linked List with no duplicate elements.																																																																																									
	TreeSet	AbstractSet	Navigable Set	No	No	No	Yes	No	No	16	It creates a collection that uses tree for storage.By default, objects are stored in ascending order.																																																																																									
	PriorityQueue	Abstract Queue	Queue	Yes	No	No	Yes	No	No	11	It creates a queue that is prioritized based on queue's comparator.																																																																																									
	ArrayDeque	Abstract Collection	Deque	Yes	No	Yes	No	No	No	16	It creates a dynamic array.																																																																																									
	EnumSet	AbstractSet	Set	No	No	Yes	No	No			It is specifically for use with elements of enum type.																																																																																									
Функциональные интерфейсы																																																																																																				

**Что такое  
функциональный  
интерфейс?**

Это интерфейс, который содержит только 1 абстрактный метод.

Интерфейс может включать сколько угодно default (и static) методов и при этом оставаться функциональным, потому что default методы - не абстрактные.

Могут ли быть поля в интерфейсах? (Могут, поля константы)

Интерфейсы могут содержать поля, так же, как и обычные классы, но с несколькими отличиями:

- Поля должны быть проинициализированы.
- Поля считаются публичными статическими финальными.
- Модификаторы public, static и final не нужно указывать явно (они «проставляются» по умолчанию)

Что такое абстрактные методы?

- У абстрактных методов нет тела.
- Реализация абстрактных методов предоставляется классам, реализующими данный интерфейс.
- Абстрактные методы считаются публичными и абстрактными даже, если это не задано явно.
- Абстрактные методы не могут быть финальными, поскольку в Java комбинация модификаторов abstract и final запрещена.

Для чего в функциональных интерфейсах Static методы?

Static – методы в интерфейсе — это тоже, что и static методы в абстрактном классе.

- Статические методы в интерфейсе являются частью интерфейса, мы не можем использовать его для объектов класса реализации.
- Статические методы в интерфейсе хороши для обеспечения вспомогательных методов, например, проверки на null, сортировки коллекций и т.д.
- Статические методы в интерфейсе помогают обеспечивать безопасность, не позволяя классам, которые реализуют интерфейс, переопределить их.
- Мы не можем определить статические методы для методов Object, потому что получим ошибку компиляции.

**Для чего нужна  
аннотация  
@FunctionalInterface?**

Нужно чтобы точно определить интерфейс как функциональный. Она обозначит замысел и не даст определить второй абстрактный метод в интерфейсе.

Какие встроенные функциональные интерфейсы вы знаете?

**Predicate<T>** - реализуется функция, получающая на вход экземпляр класса T и возвращающая на выходе значение типа boolean

and()-возвращает составной предикат логического И: sout(A.and(B).test("ABCD")); //true  
or()-возвращает составной предикат логического ИЛИ: sout(A.or(B).test("A")); //true  
sout(A.or(B).test("C")); //false  
negate()-возвращает предикат, представляющий логическое отрицание этого предиката.  
isEqual()-возвращает предикат, который проверяет, равны ли два аргумента.  
test(T t)-оценивает этот предикат для данного аргумента Predicate<Integer> negativ = x -> x<0;  
System.out.println(negativ.test(6)); //false

**Consumer<T>** - реализуется функция, которая получает на вход экземпляр класса T, производит с ним некоторое действие и ничего не возвращает

accept(T t)Выполняет операцию с заданным аргументом.  
andThen(Consumer<? super T> after)возвращает состав, Consumer который последовательно выполняет операцию, за которой следует afterоперация.

**Function<T,R>** - реализуется функция, получающая на вход экземпляр класса T и возвращающая на выходе экземпляр класса R

andThen(Function<? super R,? extends V> after)  
Возвращает составную функцию, которая сначала применяет эту функцию к своему входу, а затем применяет after функцию к результату.  
apply(T t) Применяет эту функцию к заданному аргументу.  
compose(Function<? super V,? extends T> before)  
Возвращает составную функцию, которая сначала применяет before функцию к ее входу, а затем применяет эту функцию к результату.  
identity() Возвращает функцию, которая всегда возвращает свой входной аргумент.

**Supplier<T>** - реализуется функция, ничего не принимающая на вход, но возвращающая на выход результат класса T

get() Получает результат.  
String t = "One";  
Supplier<String> supplierStr = () -> t.toUpperCase();  
System.out.println(supplierStr.get());

**UnaryOperator<T>** - принимает в качестве параметра объект типа T, выполняет над ними операции и возвращает результат операций в виде объекта типа T

identity() возвращает унарный оператор, который всегда возвращает свой входной аргумент.  
andThen, apply, compose методы унаследованные от интерфейса Function

**BinaryOperator<T, T>** - реализуется функция, получающая на вход два экземпляра класса T и возвращающая на выходе экземпляр класса T

Какие дополнительные, не относящиеся к этим семействам ФИ вы знаете? (Встроенные ФИ)

Runnable, Comparator, Cloneable

## Что такое ссылка на метод?

Ссылки на методы (Method References) - это компактные лямбда выражения для методов у которых уже есть имя. Если лямбда-выражения связываются с функциональным интерфейсом, то методы также могут быть связаны с функциональным интерфейсом. **Связь метода с функциональным интерфейсом осуществляется с помощью ссылки на метод.** Если лямбда-выражение может быть передано в некоторый метод как параметр, то ссылка на метод также может быть передана в качестве параметра. С помощью этой ссылки можно обращаться к методу не вызывая его.

**В Java различают 4 вида ссылок на методы:**

- ссылки на статические методы;
- ссылки на методы экземпляра;
- ссылки на конструкторы;
- ссылки на обобщенные (шаблонные) методы.

Ссылка на статический метод - `ContainingClass::staticMethodName`

Ссылка на нестатический метод конкретного объекта - `containingObject::instanceMethodName`

Ссылка на конструктор - `ClassName::new`

Ссылка на метод - это сокращенный синтаксис выражения лямбда, который выполняет только один метод. Это позволяет нам сослаться на конструкторы или методы, не выполняя их.

**Что такое лямбда-выражение? Чем его можно заменить?**

Лямбда-выражение - упрощённая запись анонимного класса, реализующего функциональный интерфейс

Они позволяют написать метод и сразу же использовать его. Особенно полезно в случае однократного вызова метода, т.к. сокращает время на объявление и написание метода без необходимости создавать класс. Lambda-выражения в Java обычно имеют следующий синтаксис (параметры) -> (тело)

(int a, int b) -> { return a + b; }

() -> System.out.println("Hello World");

(String s) -> { System.out.println(s); }

() -> 42

() -> { return 3.1415 ;}

Структура Lambda-выражений:

1) Lambda-выражения могут иметь от 0 и более входных параметров.

2) Тип параметров можно указывать явно либо может быть получен из контекста.

Например (int a) можно записать и так (a)

3) Параметры заключаются в круглые скобки и разделяются запятыми. Например (a, b) или (int a, int b) или (String a, int b, float c)

4) Если параметров нет, то нужно использовать пустые круглые скобки. Например () -> 42

5) Когда параметр один, если тип не указывается явно, скобки можно опустить. Пример: a -> return a\*a

6) Тело Lambda-выражения может содержать от 0 и более выражений.

7) Если тело состоит из одного оператора, его можно не заключать в фигурные скобки, а возвращаемое значение можно указывать без ключевого слова return.

8) В противном случае фигурные скобки обязательны (блок кода), а в конце надо указывать возвращаемое значение с использованием ключевого слова return (в противном случае типом возвращаемого значения будет void).

9) Каждое lambda-выражение может быть неявно привязано к какому-нибудь функциональному интерфейсу. Например, можно создать ссылку на Runnable интерфейс:

Runnable r = () -> System.out.println("hello world");

Как взаимосвязаны лямбда и функциональный интерфейс?

Чтобы объявить и использовать лямбда-выражение, основная программа разбивается на ряд этапов:

1.Определение ссылки на функциональный интерфейс: Operationable operation;

2.Создание лямбда-выражения: operation = (x,y) → x+y

К каким переменным есть доступ из лямбда-выражения?

Лямбда-выражение может использовать переменные, которые объявлены на уровне класса или метода, в котором лямбда-выражение определено.

- Значение глобальных переменных уровня класса (Static) изменить в лямбда выражении можно.

- Значение локальных переменных на уровне метода изменить в лямбда выражении нельзя, потому что они воспринимаются как константа (effectively final).

- Инициализированные переменные интерфейса (константы). В лямбда выражении их изменить нельзя.

<p><b>Что такое Stream API? Для чего нужны стримы?</b></p>	<p>Stream API — это средства потоковой обработки данных в функциональном стиле. Они не имеют ничего общего (кроме названия) с потоками ввода-вывода. Типичные применения – конвертация, переупаковка, и агрегация данных.</p> <p>Три основных понятия Java Stream API – источник данных, промежуточная (intermediate), и терминальная (terminal) операции.</p> <p><b>Источником</b> может быть заранее заданный набор данных(массив, коллекция), или динамический генератор, возможно даже бесконечный. Сам источник никогда не модифицируется последующими операциями.</p> <p><b>Промежуточные операции</b> модифицируют стрим. На одном потоке можно вызвать сколько угодно промежуточных операций.</p> <p><b>Терминальная операция</b> «потребляет» поток. Она может быть только одна, в конце работы с отдельно взятым стримом. Стримы работают лениво – вся цепочка промежуточных операций не начнет выполняться до вызова терминальной.</p> <p>Пакет java.util.stream – это средства потоковой обработки данных в функциональном стиле. Они не имеют ничего общего (кроме названия) с потоками ввода-вывода. Типичные применения – конвертация, переупаковка, и агрегация данных.</p> <p>Его задача - упростить работу с наборами данных, в частности, упростить операции фильтрации, сортировки и другие манипуляции с данными.</p>
<p><b>Почему Stream называют ленивым?</b></p>	<p>Методы не будут выполняться пока не будет вызван терминальный метод</p>
<p><b>Какие существуют способы создания стрима?</b></p>	<p>Пустой стрим: Stream.empty()  Стрим из List: list.stream()  Стрим из Map: map.entrySet().stream()  Стрим из массива: Arrays.stream(array)  Стрим из указанных элементов: Stream.of("1", "2", "3")</p> <ul style="list-style-type: none"> <li>-Можно получить из BufferedReader при помощи метода lines(), который вернет поток строк из потока символов.</li> <li>-Из директории на диске при помощи методов Files.list() и Files.walk()</li> <li>-Можно получить из строки методом chars(), будет IntStream с символами.</li> <li>-Можно порождать динамически, генерировать при помощи supplier.</li> <li>-Итерированием какой-то функции</li> <li>-Можно получить диапазон чисел в виде стрима range и rangeClosed</li> <li>-Конкатенацией других стримов</li> </ul>
<p><b>Как из коллекции создать стрим?</b></p>	<pre>Collection&lt;String&gt; collection = Arrays.asList("a1", "a2", "a3"); Stream&lt;String&gt; streamFromCollection = collection.stream();</pre>

<p><b>Какие промежуточные методы в стримах вы знаете?</b></p>	<p>Filter - Отфильтровывает записи, возвращает только записи, соответствующие условию</p> <p>Skip - Позволяет пропустить N первых элементов</p> <p>Distinct - Возвращает стрим без дубликатов</p> <p>Map - Преобразует каждый элемент стрима</p> <p>flatMap – для работы с элементами элементов коллекций</p> <p>Peek - Возвращает тот же стрим, но применяет функцию к каждому элементу стрима</p> <p>Limit - Позволяет ограничить выборку определенным количеством первых элементов</p> <p>Sorted - Позволяет сортировать значения либо в натуральном порядке, либо задавая Comparator</p> <p>mapToInt, mapToDouble, mapToLong - Аналог map, но возвращает числовой стрим (то есть стрим из числовых примитивов)</p> <p>takeWhile(Predicate predicate) Появился в Java 9. Возвращает элементы до тех пор, пока они удовлетворяют условию, то есть функция-предикат возвращает true. Это как limit, только не с числом, а с условием.</p> <p>dropWhile(Predicate predicate) Появился в Java 9. Пропускает элементы до тех пор, пока они удовлетворяют условию, затем возвращает оставшуюся часть стрима. Если предикат вернул для первого элемента false, то ни единого элемента не будет пропущено. Оператор подобен skip, только работает по условию.</p> <p>boxed() - Преобразует примитивный стрим в объектный.</p>
<p><b>Расскажите про метод peek().</b></p>	<p>Предполагается, что map() получает на вход один объект, а возвращает другой. Возможно, того же типа, но другой. peek() - это частный случай map(), который возвращает тот же самый объект, который получил на входе, возможно, с изменённым внутренним состоянием. Конечно, можно использовать для этого map(), но есть нюансы. Во-первых, peek() на одну строчку короче - не нужно писать return, Java и так знает, что нужно возвращать. Во-вторых, вы боитесь от ошибок - из peek() невозможно вернуть не тот объект, который пришёл на вход.</p>
<p><b>Расскажите про метод map().</b></p>	<p>Метод map() заданным образом преобразует каждый элемент стрима, потом преобразует все объекты в итоговый стрим.</p>
<p><b>Расскажите про метод flatMap().</b></p>	<p>flatMap возвращает по стриму для каждого объекта в первоначальном стриме, а затем результирующие потоки объединяются в исходный стрим.</p>
<p><b>Чем отличаются методы map() и flatMap().</b></p>	<p>map для каждого объекта в стриме возвращает по 1 объекту, потом преобразует все объекты в итоговый стрим. flatMap возвращает по стриму для каждого объекта в первоначальном стриме, а затем результирующие потоки объединяются в исходный стрим.</p>



<b>Расскажите про метод filter()</b>	фильтрует стрим, возвращая только те элементы, что проходят по условию (Predicate) Проверяет значение на "true" и "false"
<b>Расскажите про метод limit()</b>	limit(n) - возвращает новый поток, ограниченный n-результатами
<b>Расскажите про метод skip()</b>	skip(n) - возвращает новый поток, пропуская первые n элементов
<b>Расскажите про метод sorted()</b>	sorted() - возвращает отсортированный поток
<b>Расскажите про метод distinct()</b>	distinct() - возвращает поток равнозначный исходному, но без дубликатов
<b>Какие терминальные методы в стримах вы знаете?</b>	<p>-forEach – принимает consumer, которому будут выведены элементы стрима.          -forEachOrdered – как и forEach, но гарантирует порядок.          -count() - подсчет всех значений          -max() - возвращает максимальный элемент          -min() - возвращает минимальный элемент          -findAny() - находится вхождение – сразу возвращает результат          -anyMatch() проверяет на наличие совпадения          -allMatch() – возвращает boolean          -noneMatch() – возвращает boolean          -findFirst – возвращает первый элемент из стрима, возвращается OptionalInt          -collect – собирает элементы в новое хранилище          -reduce – результат применения бинарного оператора к каждой паре элементов стрима, пока не останется один элемент.          -toArray - возвращает массив</p> <p>Терминальный метод можно вызвать только один раз.          Все оконечные методы возвращают Optional - оболочка ответа          (этот специальный тип ввели чтобы не возвращать null)</p>
<b>Расскажите про метод collect()</b>	Stream.collect () является одним из терминальных методов. Это позволяет выполнять изменяемые операции свертывания (переупаковка элементов в некоторые структуры данных и применение некоторой дополнительной логики, объединение их и т. Д.) Преобразует стрим в коллекцию
<b>Расскажите про метод reduce()</b>	<p>позволяет выполнять агрегатные функции и возвращать один результат.</p> <p>-          Результат применения бинарного оператора к каждой паре элементов стрима, пока не останется один элемент.</p>
<b>Расскажите про класс Collectors и его методы.</b>	<p>Нужен для того, чтобы упаковывать стримы в коллекции:</p> <p>toList() - преобразует поток в список — List&lt;T&gt;          toSet() - преобразует поток в список — Set&lt;T&gt;          toMap() - преобразует поток в список — Map&lt;K, V&gt;          Используются в методе collect().</p>

<b>Расскажите о параллельной обработке в Java 8.</b>	Чтобы сделать обычный последовательный поток параллельным, надо вызвать у объекта Stream метод parallel. А обратный метод - sequential(). Кроме того, можно также использовать блокирующий метод parallelStream() интерфейса Collection для создания параллельного потока из коллекции. В то же время если рабочая машина не является многоядерной, то поток будет выполняться как последовательный. Работает на фреймворке fork/join.
<b>Что такое IntStream и DoubleStream?</b>	В Java 8 создание Stream-ов примитивов напрямую невозможно, из-за дженериков. Но разработчики сделали 3 Stream-а примитивов : IntStream, LongStream, DoubleStream. Работает быстрее, чем стрим с классами-обертками. Поддерживают дополнительные терминальные методы sum(), average(), mapToObj()
<b>Java 8</b>	
<b>Какие нововведения появились в java 8?</b>	<ol style="list-style-type: none"> <li>1. Полноценная поддержка лямбда-выражений</li> <li>2. Ссылки на методы ::</li> <li>3. Функциональные интерфейсы</li> <li>4. default методы в интерфейсах</li> <li>5. Потоки для работы с коллекциями</li> <li>6. Новое api для работы с датами</li> <li>7. Nashorn движок JavaScript, разрабатываемый полностью на Java компанией Oracle.</li> <li>8. Кодировщик/декодировщик.</li> <li>9. Новые методы для Map - putIfAbsent(), computeIfAbsent()\computeIfPresent(), Remove(), GetOrDefault(), Merge()</li> <li>10. Metaspaces пришла на замену PermGen</li> </ol>
<b>Какие новые классы для работы с датами появились в java 8?</b>	LocalDate , LocalTime, LocalDateTime, ZonedDateTime, Period, Duration
<b>Расскажите про класс Optional</b>	Optional - новый класс в пакете java.util, является контейнером (оберткой) для значений которая также может безопасно содержать null. Благодаря опциональным типам можно забыть про проверки на null и NullPointerException.
<b>Что такое Nashorn?</b>	В Java 8, Nashorn, представлен значительно улучшенный движок javascript для замены существующего Rhino. Nashorn обеспечивает в 2-10 раз лучшую производительность, так как он напрямую компилирует код в памяти и передает байт-код в JVM. Nashorn использует функцию динамического вызова, представленную в Java 7, для повышения производительности. * Nashorn — немецкое слово (Носорог)
<b>Что такое jjs?</b>	Инструмент командной строки для выполнения JavaScript-кодов на консоли.
<b>Какой класс появился в Java 8 для кодирования/декодирования данных?</b>	public static class Base64.Encoder /public static class Base64.Decoder
<b>Как создать Base64 кодировщик и декодировщик?</b>	Используя метод getDecoder() класса Base64 он возвращает декодировщик Base64.Decoder, который декодирует данные с помощью схемы кодирования base64.

<p>Какие дополнительные методы для работы с ассоциативными массивами (maps) появились в Java 8?</p>	<p><b>putIfAbsent()</b> добавляет пару «ключ-значение», только если ключ отсутствовал:  map.putIfAbsent("a", "Aa");</p> <p><b>forEach()</b> принимает функцию, которая производит операцию над каждым элементом:  map.forEach((k, v) -&gt; System.out.println(v));</p> <p><b>compute()</b> создаёт или обновляет текущее значение на полученное в результате вычисления (возможно использовать ключ и текущее значение):  map.compute("a", (k, v) -&gt; String.valueOf(k).concat(v)); //[ "a", "aAa"]</p> <p><b>computeIfPresent()</b> если ключ существует, обновляет текущее значение на полученное в результате вычисления (возможно использовать ключ и текущее значение):  map.computeIfPresent("a", (k, v) -&gt; k.concat(v));</p> <p><b>computeIfAbsent()</b> если ключ отсутствует, создаёт его со значением, которое вычисляется (возможно использовать ключ):  map.computeIfAbsent("a", k -&gt; "A".concat(k)); //[ "a", "Aa"]</p> <p><b>getOrDefault()</b> в случае отсутствия ключа, возвращает переданное значение по-умолчанию:  map.getOrDefault("a", "not found");</p> <p><b>merge()</b> принимает ключ, значение и функцию, которая объединяет передаваемое и текущее значения. Если под заданным ключом значение отсутствует, то записывает туда передаваемое значение.</p> <p><b>- map.remove(key, value) - Если такое ключ-значение есть в map, то удаляем</b></p>
<p>Что такое <b>LocalDateTime</b>?</p>	<p><b>LocalDateTime</b> объединяет вместе <b>LocalDate</b> и <b>LocalTime</b>, содержит дату и время в календарной системе ISO-8601 без привязки к часовому поясу. Время хранится с точностью до наносекунды. Содержит множество удобных методов, таких как <b>plusMinutes</b>, <b>plusHours</b>, <b>isAfter</b>, <b>toSecondOfDay</b> и т.д.</p>
<p>Что такое <b>ZonedDateTime</b>?</p>	<p><b>java.time.ZonedDateTime</b> — аналог <b>java.util.Calendar</b>, класс с самым полным объемом информации о временном контексте в календарной системе ISO-8601. Включает объект <b>ZoneId</b> - временную зону(в <b>ZoneId</b> 599 зон), поэтому все операции с временными сдвигами этот класс проводит с её учётом.</p>

<p><b>Чем процесс отличается от потока?</b></p>	<p><b>Процесс</b> — экземпляр программы во время выполнения, независимый объект, которому выделены системные ресурсы (например, процессорное время и память). <b>Каждый процесс выполняется в отдельном адресном пространстве: один процесс не может получить доступ к переменным и структурам данных другого.</b></p> <p>Если процесс хочет получить доступ к чужим ресурсам, необходимо использовать межпроцессное взаимодействие. Это могут быть конвейеры, файлы, каналы связи между компьютерами и многое другое.</p> <p>Для каждого процесса ОС создает так называемое «виртуальное адресное пространство», к которому процесс имеет прямой доступ. Это пространство принадлежит процессу, содержит только его данные и находится в полном его распоряжении. Операционная система же отвечает за то, как виртуальное пространство процесса проецируется на физическую память.</p> <p><b>Поток(thread) — способ выполнения процесса, определяющий последовательность исполнения кода в процессе. Потоки всегда создаются в контексте какого-либо процесса, и вся их жизнь проходит только в его границах.</b></p> <p>Потоки могут исполнять один и тот же код и манипулировать одними и теми же данными, а также совместно использовать описатели объектов ядра, поскольку таблица описателей создается не в отдельных потоках, а в процессах. Так как потоки расходуют существенно меньше ресурсов, чем процессы, в процессе выполнения работы выгоднее создавать дополнительные потоки и избегать создания новых процессов.</p> <p><b>Основное отличие процесса от потока в том, что "содержимое" потока просто "движется" (оставаясь тем же самым), а процесс это взаимодействие его "содержимого" с изменением соотношения его "частей" и свойств (внешне процесс может выглядеть и "неподвижным").</b></p>
<p><b>Чем Thread отличается от Runnable? Когда нужно использовать Thread, а когда Runnable? (Ответ что тред - это класс, а ранбл интерфейс - считается не полным, нужно рассказать подробно)</b></p>	<p>Thread - это класс, некоторая надстройка над физическим потоком. Runnable - это интерфейс, представляющий абстракцию над выполняемой задачей. Помимо того, что Runnable помогает разрешить проблему множественного наследования, несомненный плюс от его использования состоит в том, что он позволяет логически отделить логику выполнения задачи от непосредственного управления потоком.</p> <p>В классе Thread имеется несколько методов, которые можно переопределить в порожденном классе. Из них обязательному переопределению подлежит только метод run(). Этот же метод, безусловно, должен быть определен и при реализации интерфейса Runnable.</p> <p>Некоторые программисты считают, что создавать подкласс, порожденный от класса Thread, следует только в том случае, если нужно дополнить его новыми функциями. Так, если переопределять любые другие методы из класса Thread не нужно, то можно ограничиться только реализацией интерфейса Runnable. Кроме того, реализация интерфейса Runnable позволяет создаваемому потоку наследовать класс, отличающийся от Thread</p>
<p><b>Что такое монитор? Как монитор реализован в java?</b></p>	<p><b>Монитор</b> - механизм синхронизации потоков, обеспечивающий доступ к неразделяемым ресурсам. Частью монитора является mutex, который встроен в класс Object и имеется у каждого объекта.</p> <p>Удобно представлять mutex как id захватившего его объекта. Если этот id равен 0 – ресурс свободен. Если не 0 – ресурс занят. Можно встать в очередь и ждать его освобождения.</p> <p>В Java <b>монитор реализован</b> с помощью ключевого слова <b>synchronized</b>.</p>

<p><b>Что такое синхронизация? Какие способы синхронизации существуют в java?</b></p>	<p>Синхронизация это <b>процесс, который позволяет выполнять потоки параллельно</b>. В Java все объекты имеют блокировку, благодаря которой только один поток одновременно может получить доступ к критическому коду в объекте. Такая синхронизация помогает предотвратить повреждение состояния объекта.</p> <p>Способы синхронизации в Java: <b>Системная синхронизация с использованием wait()/notify()</b>. Поток, который ждет выполнения каких-либо условий, вызывает у этого объекта метод wait(), предварительно захватив его монитор. На этом его работа приостанавливается. Другой поток может вызвать на этом же самом объекте метод notify() (опять же, предварительно захватив монитор объекта), в результате чего, ждущий на объекте поток «просыпается» и продолжает свое выполнение. В обоих случаях монитор надо захватывать в явном виде, через synchronized-блок, потому как методы wait()/notify() не синхронизированы! <b>Системная синхронизация с использованием join()</b>. Метод join(), вызванный у экземпляра класса Thread, позволяет текущему потоку остановиться до того момента, как поток, связанный с этим экземпляром, закончит работу. <b>Использование классов из пакета java.util.concurrent.Locks</b> - механизмы синхронизации потоков, альтернативы базовым synchronized, wait, notify, notifyAll: <b>Lock, Condition, ReadWriteLock</b>.</p>
<p><b>Как работают методы wait(), notify() и notifyAll()?</b></p>	<p><b>wait(): освобождает монитор и переводит вызывающий поток в состояние ожидания</b> до тех пор, <b>пока другой поток не вызовет метод notify()/notifyAll()</b>; <b>notify(): продолжает работу потока</b>, у которого ранее был вызван метод wait(); <b>notifyAll(): возобновляет работу всех потоков</b>, у которых ранее был вызван метод wait(). Когда вызван метод wait(), поток освобождает блокировку на объекте и переходит из состояния Работающий (Running) в состояние Ожидания (Waiting). Метод <b>notify()</b> <b>подаёт сигнал одному из потоков, ожидающих на объекте, чтобы перейти в состояние Работоспособный (Runnable)</b>. При этом невозможно определить, какой из ожидающих потоков должен стать работоспособным. Метод notifyAll() заставляет все ожидающие потоки для объекта вернуться в состояние Работоспособный (Runnable). <b>Если ни один поток не находится в ожидании на методе wait(), то при вызове notify() или notifyAll() ничего не происходит.</b> wait(), notify() и notifyAll() <b>должны вызываться только из синхронизированного кода</b>.</p>
<p><b>В каких состояниях может находиться поток?</b></p>	<p><b>New</b> - объект класса Thread создан, но еще не запущен. Он еще не является потоком выполнения и естественно не выполняется. <b>Runnable</b> - поток готов к выполнению, но планировщик еще не выбрал его. <b>Running</b> – поток выполняется. <b>Waiting/blocked/sleeping</b> - поток блокирован или поток ждет окончания работы другого потока. <b>Dead</b> - поток завершен. Будет выброшено исключение при попытке вызвать метод start() для dead потока. <b>public enum State</b> (У класса Thread есть внутренний класс State - состояние, а также метод public State getState().) {   <b>NEW</b>, — поток создан, но еще не запущен;   <b>RUNNABLE</b>, — поток выполняется;   <b>BLOCKED</b>, — поток блокирован;   <b>WAITING</b>, — поток ждет окончания работы другого потока;   <b>TIMED_WAITING</b>, — поток некоторое время ждет окончания другого потока;   <b>TERMINATED</b>; — поток завершен. }</p>

<p><b>Что такое семафор? Как он реализован в Java?</b></p>	<p><b>Semaphore – это новый тип синхронизатора: семафор со счётчиком, реализующий шаблон синхронизации Семафор.</b> Доступ управляется с помощью счётчика: изначальное значение счетчика задается в конструкторе при создании синхронизатора, когда поток заходит в заданный блок кода, то значение счетчика уменьшается на единицу, когда поток его покидает, то увеличивается. Если значение счетчика равно нулю, то текущий поток блокируется, пока кто-нибудь не выйдет из защищаемого блока. Semaphore используется для защиты дорогих ресурсов, которые доступны в ограниченном количестве, например подключение к базе данных в пуле.</p>
<p><b>Что означает ключевое слово volatile? Почему операции над volatile переменными не атомарны?</b></p>	<p><b>Переменная volatile является атомарной для чтения, но операции над переменной НЕ являются атомарными. Поля, для которых неприемлемо увидеть «несвежее» (stale) значение в результате кэширования или переупорядочения.</b></p> <p>Если происходит какая-то операция, например, инкремент, то атомарность уже не обеспечивается, потому что сначала выполняется чтение(1), потом изменение(2) в локальной памяти, а затем запись(3). Такая операция не является атомарной и в неё может вклиниться поток по середине.</p> <p>Атомарная операция выглядит единой и неделимой командой процессора.</p> <p><b>Переменная volatile находится в хипе, а не в кэше стека .</b></p> <p><b>Атомарная операция — это операция, которую невозможно наблюдать в промежуточном состоянии, она либо выполнена либо нет. Атомарные операции могут состоять из нескольких операций.</b></p>
<p><b>Для чего нужны Atomic типы данных? Чем отличаются от volatile?</b></p>	<p>volatile не гарантирует атомарность. Например, операция count++ не станет атомарной просто потому что count объявлена volatile. С другой стороны class AtomicInteger предоставляет атомарный метод для выполнения таких комплексных операций атомарно, например getAndIncrement() – атомарная замена оператора инкремента, его можно использовать, чтобы атомарно увеличить текущее значение на один. Похожим образом сконструированы атомарные версии и для других типов данных.</p> <p>volatile обеспечивает только видимость изменений, а классы Atomic* дают еще и атомарность изменений.</p> <p>Простой пример - вам нужно проинкрементить счетчик и вернуть значение. Если поле счетчика будет обычным volatile int - возможна ситуация, когда два разных потока сначала проведут инкремент, а потом оба заберут результат двух инкрементов.</p> <p>Если же взять AtomicInteger, будет гарантирована атомарность, и каждый поток получит правильный результат.</p> <p>Типичное применение volatile:</p> <ul style="list-style-type: none"> <li>- флаги (например, флаг выполнения потока);</li> <li>- поля в POJO, которые используются только для хранения данных, когда по какой-то причине нет возможности использовать final-поля.</li> </ul>
<p><b>Что такое потоки демоны? Для чего они нужны? Как создать поток-демон?</b></p>	<p>Потоки-демоны <b>работают в фоновом режиме вместе с программой</b>, но не являются неотъемлемой частью программы. <b>Если какой-либо процесс может выполняться на фоне работы основных потоков выполнения и его деятельность заключается в обслуживании основных потоков приложения</b>, то такой процесс может быть запущен как поток-демон с помощью метода <b>setDaemon(boolean value)</b>, вызванного у потока до его запуска. Метод boolean isDaemon() позволяет определить, является ли указанный поток демоном или нет. <b>Основной поток приложения может завершить выполнение потока-демона</b> (в отличие от обычных потоков) <b>с окончанием кода метода main(), не обращая внимания, что поток-демон еще работает.</b></p> <p>Поток демон можно сделать только если он еще не запущен. Пример демона - GC.</p>

<p><b>Что такое приоритет потока? На что он влияет? Какой приоритет у потоков по умолчанию?</b></p>	<p>Приоритеты потоков используются планировщиком потоков для принятия решений о том, когда какому из потоков будет разрешено работать. Теоретически высокоприоритетные потоки получают больше времени процессора, чем низкоприоритетные. Практически объем времени процессора, который получает поток, часто зависит от нескольких факторов помимо его приоритета (является ли поток демоном).</p> <p>Чтобы установить приоритет потока, используется метод класса Thread: <code>final void setPriority(int level)</code>. Значение <code>level</code> изменяется в пределах от <code>Thread.MIN_PRIORITY = 1</code> до <code>Thread.MAX_PRIORITY = 10</code>. Приоритет по умолчанию - <code>Thread.NORM_PRIORITY = 5</code>.</p> <p>Получить текущее значение приоритета потока можно вызвав метод: <code>final int getPriority()</code> у экземпляра класса Thread.</p> <p>Метод <code>yield()</code> можно использовать для того чтобы принудить планировщик выполнить другой поток, который ожидает своей очереди.</p>
<p><b>Как работает Thread.join()? Для чего он нужен?</b></p>	<p>Когда поток вызывает <code>join()</code>, он будет ждать пока поток, к которому он присоединяется, будет завершён, либо отработает переданное время:</p> <pre>void join() void join(long millis) - с временем ожидания void join(long millis, int nanos)</pre> <p>Применение: при распараллелили вычисления, вам надо дождаться результатов, чтобы собрать их в кучу и продолжить выполнение.</p>
<p><b>Чем отличаются методы wait() и sleep()?</b></p>	<p>метод <code>sleep()</code> - приостанавливает поток на указанное время. Состояние меняется на <code>WAITING</code>, по истечению - <code>RUNNABLE</code>.</p> <p>метод <code>wait()</code> - меняет состояние потока на <code>WAITING</code>. Может быть вызван только у объекта владеющего блокировкой, в противном случае выкинется исключение <code>IllegalMonitorStateException</code></p>
<p><b>Можно ли вызвать start() для одного потока дважды?</b></p>	<p>Нельзя стартовать поток больше, чем единожды. В частности, поток не может быть перезапущен, если он уже завершил выполнение. Выдает: <code>IllegalThreadStateException</code></p>
<p><b>Как правильно остановить поток? Для чего нужны</b></p>	<p>Как остановить поток?</p> <p>На данный момент в Java принят уведомительный порядок остановки потока (хотя JDK 1.0 и имеет несколько управляющих выполнением потока методов, например <code>stop()</code>, <code>suspend()</code> и <code>resume()</code> - в следующих версиях JDK все они были помечены как <code>deprecated</code> из-за потенциальных угроз взаимной блокировки).</p> <p><b>Для корректной остановки потока</b> можно использовать метод класса Thread - <b><code>interrupt()</code></b>. Этот метод выставляет внутренний флаг-статус прерывания. В дальнейшем состояние этого флага можно проверить с помощью метода <code>isInterrupted()</code> или <code>Thread.interrupted()</code> (для текущего потока). <b>Метод <code>interrupt()</code> также способен вывести поток из состояния ожидания или спячки.</b> Т.е. если у потока были вызваны методы <code>sleep()</code> или <code>wait()</code> – текущее состояние прервется и будет выброшено исключение <code>InterruptedException</code>. Флаг в этом случае не выставляется.</p> <p><b>Схема действия</b> при этом получается следующей:</p> <p><b>Реализовать поток.</b></p> <p><b>В потоке периодически проводить проверку статуса прерывания через вызов <code>isInterrupted()</code>.</b></p> <p>Если состояние флага изменилось или было выброшено исключение во время ожидания/спячки, следовательно поток пытаются остановить извне.</p> <p>Принять решение – продолжить работу (если по каким-то причинам остановиться невозможно) или освободить заблокированные потоком ресурсы и закончить выполнение.</p> <p>Возможная проблема, которая присутствует в этом подходе – блокировки на потоковом вводе-выводе. Если поток заблокирован на чтении данных - вызов <code>interrupt()</code> из этого состояния его не выведет. Решения тут различаются в зависимости от типа источника данных. Если чтение идет из файла – долговременная блокировка крайне маловероятна и тогда можно просто дождаться выхода из метода <code>read()</code>. Если же чтение каким-то образом связано с сетью – стоит использовать неблокирующий ввод-вывод из Java NIO.</p>



<p>для чего нужны методы <code>.stop()</code>, <code>.interrupt()</code>, <code>.interrupted()</code>, <code>.isInterrupted()</code>.</p>	<p>Второй вариант реализации метода остановки (а также и приостановки) – сделать собственный аналог <code>interrupt()</code>. Т.е. объявить в классе потока флаги – на остановку и/или приостановку и выставлять их путем вызова заранее определённых методов извне. Методика действия при этом остаётся прежней – проверять установку флагов и принимать решения при их изменении. Недостатки такого подхода. Во-первых, потоки в состоянии ожидания таким способом не «оживить». Во-вторых, выставление флага одним потоком совсем не означает, что второй поток тут же его увидит. Для увеличения производительности виртуальная машина использует кеш данных потока, в результате чего обновление переменной у второго потока может произойти через неопределенный промежуток времени (хотя допустимым решением будет объявить переменную-флаг как <code>volatile</code>).</p> <p><b>Почему не рекомендуется использовать метод <code>Thread.stop()</code>?</b></p> <p>При принудительной остановке (приостановке) потока, <code>stop()</code> прерывает поток в недетерминированном месте выполнения, в результате становится совершенно непонятно, что делать с принадлежащими ему ресурсами. Поток может открыть сетевое соединение - что в таком случае делать с данными, которые еще не вычитаны? Где гарантия, что после дальнейшего запуска потока (в случае приостановки) он сможет их дочитать? Если поток блокировал разделяемый ресурс, то как снять эту блокировку и не переведёт ли принудительное снятие к нарушению консистентности системы? То же самое можно расширить и на случай соединения с базой данных: если поток остановят посередине транзакции, то кто ее будет закрывать? Кто и как будет разблокировать ресурсы?</p> <p><b>В чем разница между <code>interrupted()</code> и <code>isInterrupted()</code>?</b></p> <p>Механизм прерывания работы потока в Java реализован с использованием внутреннего флага, известного как статус прерывания. Прерывание потока вызовом <code>Thread.interrupt()</code> устанавливает этот флаг. Методы <code>Thread.interrupted()</code> и <code>isInterrupted()</code> позволяют проверить, является ли поток прерванным.</p> <p>Когда прерванный поток проверяет статус прерывания, вызывая статический метод <code>Thread.interrupted()</code>, статус прерывания сбрасывается.</p> <p>Нестатический метод <code>isInterrupted()</code> используется одним потоком для проверки статуса прерывания у другого потока, не изменяя флаг прерывания.</p>
<p>Чем <code>Runnable</code> отличается от <code>Callable</code>?</p>	<p>Интерфейс <code>Runnable</code> появился в Java 1.0, а интерфейс <code>Callable</code> был введен в Java 5.0 в составе библиотеки <code>java.util.concurrent</code>;</p> <p><b>Классы, реализующие интерфейс <code>Runnable</code> для выполнения задачи должны реализовывать метод <code>run()</code>. Классы, реализующие интерфейс <code>Callable</code> - метод <code>call()</code>;</b></p> <p>Метод <code>Runnable.run()</code> не возвращает никакого значения,</p> <p><code>Callable</code> - это параметризованный функциональный интерфейс. <code>Callable.call()</code> возвращает <code>Object</code>, если он не параметризован, иначе указанный тип.</p> <p>Метод <code>run()</code> НЕ может выбрасывать проверяемые исключения, в то время как метод <code>call()</code> может.</p>
<p>Что такое <code>FutureTask</code>?</p>	<p><code>FutureTask</code> представляет собой отменяемое асинхронное вычисление в параллельном потоке. Этот класс предоставляет базовую реализацию <code>Future</code>, с методами для запуска и остановки вычисления, методами для запроса состояния вычисления и извлечения результатов. Результат может быть получен только когда вычисление завершено, метод получения будет заблокирован, если вычисление ещё не завершено. Объекты <code>FutureTask</code> могут быть использованы для обёртки объектов <code>Callable</code> и <code>Runnable</code>. Так как <code>FutureTask</code> помимо <code>Future</code> реализует <code>Runnable</code>, его можно передать в <code>Executor</code> на выполнение.</p>



<p><b>Что такое deadlock?</b></p>	<p>Взаимная блокировка (deadlock) - <b>явление</b> при котором <b>все потоки</b> находятся в <b>режиме ожидания</b> и своё состояние не меняют. Происходит, когда достигаются состояния:</p> <p><b>взаимного исключения:</b> по крайней мере <b>один</b> ресурс занят в режиме неделимости и следовательно только один поток может использовать ресурс в данный момент времени.</p> <p><b>удержания и ожидания:</b> поток удерживает как минимум один ресурс и запрашивает дополнительные ресурсы, которые удерживаются другими потоками.</p> <p><b>отсутствия предпочистки:</b> операционная система не переназначает ресурсы: если они уже заняты, они должны отдаваться удерживающим потокам сразу же.</p> <p><b>циклического ожидания:</b> поток ждет освобождения ресурса другим потоком, который в свою очередь ждёт освобождения ресурса заблокированного первым потоком.</p> <p>Простейший способ <b>избежать взаимной блокировки</b> – <b>не допускать циклического ожидания</b>. Этого можно достичь, получая <b>мониторы разделяемых ресурсов в определенном порядке и освобождая их в обратном порядке</b>.</p>
<p><b>Что такое livelock?</b></p>	<p>livelock – тип <b>взаимной блокировки</b>, при котором <b>несколько потоков выполняют бесполезную работу</b>, попадая в <b>зацикленность при попытке получения</b> каких-либо <b>ресурсов</b>. При этом <b>их состояния</b> постоянно <b>изменяются в зависимости друг от друга</b>. Фактической <b>ошибки не возникает</b>, но КПД системы падает до 0. Часто возникает в результате попыток предотвращения deadlock. Реальный <b>пример livelock</b>, – когда два человека встречаются в узком коридоре и каждый, пытаясь быть вежливым, отходит в сторону, и так они бесконечно двигаются из стороны в сторону, абсолютно не продвигаясь в нужном им направлении.</p>
<p><b>Что такое race condition?</b></p>	<p>Состояние гонки (race condition) - <b>ошибка проектирования многопоточной системы или приложения, при которой работа зависит от того, в каком порядке выполняются потоки</b>. Состояние гонки <b>возникает когда поток, который должен исполниться в начале, проиграл гонку и первым исполняется другой поток</b>: поведение кода изменяется, из-за чего возникают недетерминированные ошибки.</p> <p><b>DataRace</b> - это свойство выполнения программы. Согласно JMM, выполнение считается содержащим гонку данных, если оно содержит по крайней мере два конфликтующих доступа (чтение или запись в одну и ту же переменную), которые не упорядочены отношениями «happens before».</p> <p><b>Starvation</b> - потоки не заблокированы, но есть нехватка ресурсов из-за чего потоки ничего не делают.</p> <p>Самый простой способ решения — копирование переменной в локальную переменную. Или просто синхронизация потоков методами и sync-блоками.</p>

<p><b>Что такое Фреймворк fork/join? Для чего он нужен?</b></p>	<p>Фреймворк Fork/Join, представленный в JDK 7, - это набор классов и интерфейсов позволяющих использовать преимущества многопроцессорной архитектуры современных компьютеров. <b>Он разработан для выполнения задач, которые можно рекурсивно разбить на маленькие подзадачи, которые можно решать параллельно.</b>  <b>Этап Fork:</b> большая задача разделяется на несколько меньших подзадач, которые в свою очередь также разбиваются на меньшие. <b>И так до тех пор, пока задача не становится тривиальной и решаемой последовательным способом.</b>  <b>Этап Join:</b> далее (опционально) идёт процесс «свёртки» - <b>решения подзадач некоторым образом объединяются пока не получится решение всей задачи.</b>  Решение всех подзадач (в т.ч. и само разбиение на подзадачи) происходит параллельно.  Для <b>решения некоторых задач этап Join не требуется. Например</b>, для параллельного QuickSort — массив рекурсивно делится на всё меньшие и меньшие диапазоны, пока не вырождается в тривиальный случай из 1 элемента. Хотя в некотором смысле Join будет необходим и тут, т.к. всё равно остаётся необходимость дождаться пока не закончится выполнение всех подзадач.  Ещё одно <b>преимущество</b> этого фреймворка заключается в том, что он <b>использует work-stealing алгоритм: потоки</b>, которые <b>завершили</b> выполнение <b>собственных подзадач</b>, могут <b>«украсть» подзадачи у других потоков, которые всё ещё заняты.</b></p>
<p><b>Что означает ключевое слово synchronized? Где и для чего может использоваться?</b></p>	<p>Зарезервированное слово позволяет добиваться синхронизации в помеченных им методах или блоках кода.</p>
<p><b>Что является монитором у статического synchronized- метода?</b></p>	<p>Объект типа Class, соответствующий классу, в котором определен метод.</p>
<p><b>Что является монитором у нестатического synchronized- метода?</b></p>	<p>Объект this</p>
	<p><a href="http://java-online.ru/concurrent.shtml">http://java-online.ru/concurrent.shtml</a>  Классы и интерфейсы пакета java.util.concurrent объединены в несколько групп по функциональному признаку:</p> <p><b>collections</b> - Набор эффективно работающих в многопоточной среде коллекций. CopyOnWriteArrayList(Set), ConcurrentHashMap. Итераторы классов данного пакета представляют данные на определенный момент времени. Все операции по изменению коллекции (add, set, remove) приводят к созданию новой копии внутреннего массива. Этим гарантируется, что при проходе итератором по коллекции не будет ConcurrentModificationException.</p> <p>Отличие <b>ConcurrentHashMap</b> связано с внутренней структурой хранения пар key-value. ConcurrentHashMap использует несколько сегментов, и данный класс нужно рассматривать как группу HashMap'ов. Количество сегментов по умолчанию равно 16. Если пара key-value хранится в 10-ом сегменте, то ConcurrentHashMap блокирует, при необходимости, только 10-й сегмент, и не будет блокировать остальные 15.</p>

**util. Concurrent  
поверхностно.**

**CopyOnWriteArrayList:**

- volatile массив внутри
- lock только при модификации списка, поэтому операции чтения очень быстрые
- новая копия массива при модификации
- fail-fast итератор
- модификация через iterator невозможна - UnsupportedOperationException

**synchronizers** - Объекты синхронизации, позволяющие разработчику управлять и/или ограничивать работу нескольких потоков. Содержит пять объектов синхронизации: semaphore, countdownLatch, cyclicBarrier, exchanger, phaser.

**CountDownLatch** - объект синхронизации потоков, блокирующий один или несколько потоков до тех пор, пока не будут выполнены определенные условия. Количество условий задается счетчиком. При обнулении счетчика, т.е. при выполнении всех условий, блокировки выполняемых потоков будут сняты и они продолжат выполнение кода. Одноразовый.

**CyclicBarrier** — барьерная синхронизация останавливает поток в определенном месте в ожидании прихода остальных потоков группы. Как только все потоки достигнут барьера, барьер снимается и выполнение потоков продолжается. Как и CountDownLatch, использует счетчик и похож на него. Отличие связано с тем, барьер можно использовать повторно(в цикле).

**Exchanger** — объект синхронизации, используемый для двустороннего обмена данными между двумя потоками. При обмене данными допускается null значения, что позволяет использовать класс для односторонней передачи объекта или же просто, как синхронизатор двух потоков. Обмен данными выполняется вызовом метода exchange, сопровождаемый самоблокировкой потока. Как только второй поток вызовет метод exchange, то синхронизатор Exchanger выполнит обмен данными между потоками.

**Phaser** — объект синхронизации типа «Барьер», но, в отличие от CyclicBarrier, может иметь несколько барьеров (фаз), и количество участников на каждой фазе может быть разным.

**atomic** - Набор атомарных классов для выполнения атомарных операций. Операция является атомарной, если её можно безопасно выполнять при параллельных вычислениях в нескольких потоках, не используя при этом ни блокировок, ни синхронизацию synchronized.

**Queues** - содержит классы формирования неблокирующих и блокирующих очередей для многопоточных приложений. Неблокирующие очереди «заточены» на скорость выполнения, блокирующие очереди приостанавливают потоки при работе с очередью.

**Locks** - Механизмы синхронизации потоков, альтернативы базовым synchronized, wait, notify, notifyAll: Lock, Condition, ReadWriteLock.

Lock — базовый интерфейс, предоставляющий более гибкий подход при ограничении доступа к ресурсам/блокам по сравнению с использованием synchronized. Так, при использовании нескольких блокировок, порядок их освобождения может быть произвольный. Имеется возможность перехода к альтернативному сценарию, если блокировка уже захвачена.

Condition — интерфейсное условие в сочетании с блокировкой Lock позволяет заменить методы монитора/мьютекса (wait, notify и notifyAll) объектом, управляющим ожиданием событий. Объект с условием чаще всего получается из блокировок с использованием метода lock.newCondition(). Таким образом можно получить несколько комплектов wait/notify для одного объекта. Блокировка Lock заменяет использование synchronized, а Condition — объектные методы монитора.

ReadWriteLock — интерфейс создания read/write блокировок, который реализует один единственный класс ReentrantReadWriteLock. Блокировку чтение-запись следует использовать при длительных и частых операциях чтения и редких операциях записи. Тогда при доступе к защищенному ресурсу используются разные методы блокировки, как показано ниже :

```
ReadWriteLock rwl = new ReentrantReadWriteLock();  
Lock readLock = rwl.readLock();  
Lock writeLock = rwl.writeLock();
```

	<p><b>Executors</b> - включает средства, называемые сервисами исполнения, позволяющие управлять потоковыми задачами с возможностью получения результатов через интерфейсы Future и Callable.</p> <p><b>ExecutorService</b> служит альтернативой классу Thread, предназначенному для управления потоками. В основу сервиса исполнения положен интерфейс Executor, в котором определен один метод :</p> <pre>void execute(Runnable thread);</pre> <p>При вызове метода execute выполняется поток thread.</p>
<p><b>Stream API &amp; ForkJoinPool. Как связаны, что это такое.</b></p>	<p>В Stream API есть простой способ распараллеливания потока методом parallel() или parallelStream(), чтобы получить выигрыш в производительности на многоядерных машинах.</p> <p>По-умолчанию parallel stream используют ForkJoinPool.commonPool. Этот пул создается статически и живет пока не будет вызван System::exit. Если задачам не указывать конкретный пул, то они будут выполняться в рамках commonPool.</p> <p>По-умолчанию, размер пула равен на 1 меньше, чем количество доступных ядер.</p> <p>Когда некий тред отправляет задачу в common pool, то пул может использовать вызывающий тред (caller-thread) в качестве воркера. ForkJoinPool пытается загрузить своими задачами и вызывающий тред.</p>
<p><b>Java Memory Model</b></p>	<p><b>Описывает как потоки должны взаимодействовать через общую память.</b> Определяет набор действий межпоточного взаимодействия. В частности, чтение и запись переменной, захват и освобождения монитора, чтение и запись volatile переменной, запуск нового потока.</p> <p>JMM определяет отношение между этими действиями "happens-before" - абстракцией обозначающей, что если операция X связана отношением happens-before с операцией Y, то весь код следующий за операцией Y, выполняемый в одном потоке, видит все изменения, сделанные другим потоком, до операции X.</p> <p>Можно выделить несколько основных областей, имеющих отношение к модели памяти:</p> <p><b>Видимость</b> (visibility). Один поток может временно сохранить значения некоторых полей не в основную память, а в регистры или локальный кэш процессора, таким образом второй поток, читая из основной памяти, может не увидеть последних изменений поля. И наоборот, если поток на протяжении какого-то времени работает с регистрами и локальными кэшами, читая данные оттуда, он может сразу не увидеть изменений, сделанных другим потоком в основную память.</p> <p>К вопросу видимости имеют отношение следующие ключевые слова языка Java: synchronized, volatile, final.</p> <p>С точки зрения Java все переменные (за исключением локальных переменных, объявленных внутри метода) хранятся в heap памяти, которая доступна всем потокам. Кроме этого, каждый поток имеет локальную—рабочую—память, где он хранит копии переменных, с которыми он работает, и при выполнении программы поток работает только с этими копиями.</p> <p><b>synchronized</b> - При входе в synchronized метод или блок поток обновляет содержимое локальной памяти, а при выходе из synchronized метода или блока поток записывает изменения, сделанные в локальной памяти, в главную. Такое поведение synchronized методов и блоков следует из правил для отношения «происходит раньше»</p> <p><b>volatile</b> - запись volatile-переменных производится в основную память, минуя локальную. и чтение volatile переменной производится также из основной памяти, то есть значение переменной не может сохраняться в регистрах или локальной памяти потока и операция чтения этой переменной гарантированно вернёт последнее записанное в неё значение.</p> <p><b>final</b> - после того как объект был корректно создан, любой поток может видеть значения его final полей без дополнительной синхронизации. «Корректно создан» означает, что ссылка на создающийся объект не должна использоваться до тех пор, пока не завершится конструктор объекта.</p> <p>Рекомендуется изменять final поля объекта только внутри конструктора, в противном случае поведение не специфицировано.</p>

**Переупорядочивание** (Reordering). Для увеличения производительности процессор/компилятор могут переставлять местами некоторые инструкции/операции. Процессор может решить поменять порядок выполнения операций, если, например, сочтет что такая последовательность выполнится быстрее. Эффект может наблюдаться, когда один поток кладет результаты первой операции в регистр или локальный кэш, а результат второй операции попадает непосредственно в основную память. Тогда второй поток, обращаясь к основной памяти может сначала увидеть результат второй операции, и только потом первой, когда все регистры или кэши синхронизируются с основной памятью.

Также регулируется набором правил «happens-before»: операции чтения и записи volatile переменных не могут быть переупорядочены с операциями чтения и записи других volatile и не-volatile переменных.

<https://habr.com/ru/company/golovachcourses/blog/221133/>

## И

<p><b>Что такое DDL?</b>  <b>Какие операции в него входят?</b>  <b>Рассказать про них.</b></p>	<p><b>DDL</b> (Data Definition Language) - операторы определения данных (Data Definition Language, DDL):          CREATE создает объект БД (базу, таблицу, представление, пользователя и т. д.),          ALTER изменяет объект,          DROP удаляет объект;          TRUNCATE удаляет таблицу и создает её пустую заново, но если в таблице были foreign key, то создать таблицу не получится. rollback после TRUNCATE невозможен</p>
<p><b>Что такое DML?</b>  <b>Какие операции в него входят?</b>  <b>Рассказать про них.</b></p>	<p>операторы манипуляции данными (Data Manipulation Language, DML):          SELECT выбирает данные, удовлетворяющие заданным условиям,          INSERT добавляет новые данные,          UPDATE изменяет существующие данные,          DELETE удаляет данные при выполнении условия WHERE;</p>
<p><b>Что такое TCL? Какие операции в него входят? Рассказать про них.</b></p>	<p>операторы управления транзакциями (Transaction Control Language, TCL):          BEGIN служит для определения начала транзакции          COMMIT применяет транзакцию,          ROLLBACK откатывает все изменения, сделанные в контексте текущей транзакции,          SAVEPOINT разбивает транзакцию на более мелкие.</p>
<p><b>Что такое DCL?</b>  <b>Какие операции в него входят?</b>  <b>Рассказать про них.</b></p>	<p>операторы определения доступа к данным (Data Control Language, DCL):          GRANT предоставляет пользователю (группе) разрешения на определенные операции с объектом,          REVOKE отзывает ранее выданные разрешения,          DENY задает запрет, имеющий приоритет над разрешением;</p>
<p><b>Нюансы работы с NULL в SQL. Как проверить поле на NULL?</b></p>	<p>NULL - <b>специальное значение</b> (псевдозначение), которое может быть записано в поле таблицы базы данных. NULL соответствует понятию «пустое поле», то есть «поле, не содержащее никакого значения».          NULL <b>означает отсутствие</b>, неизвестность <b>информации</b>. Значение NULL не является значением в полном смысле слова: по определению оно означает отсутствие значения и не принадлежит ни одному типу данных. Поэтому NULL не равно ни логическому значению FALSE, ни пустой строке, ни 0. <b>При сравнении NULL с любым значением будет получен результат NULL</b>, а не FALSE и не 0. Более того, <b>NULL не равно NULL!</b>          команды: IS NULL, IS NOT NULL</p>
	<p>JOIN - оператор языка SQL, который является реализацией операции соединения реляционной алгебры.          Предназначен для обеспечения выборки данных из двух таблиц и включения этих данных в один результирующий набор.</p> <p>Особенностями операции соединения являются следующее:</p> <ul style="list-style-type: none"> <li>- в схему таблицы-результата входят столбцы обеих исходных таблиц (таблиц-операндов), то есть схема результата является «сцеплением» схем операндов;</li> <li>- каждая строка таблицы-результата является «сцеплением» строки из одной таблицы-операнда со строкой второй таблицы-операнда;</li> <li>- при необходимости соединения не двух, а нескольких таблиц, операция соединения применяется несколько раз (последовательно).</li> </ul>

<b>Виды Join'ов?</b>	<p><b>Какие существуют типы JOIN?</b></p> <p>(INNER) JOIN Результатом объединения таблиц являются записи, общие для левой и правой таблиц. Порядок таблиц для оператора не важен, поскольку оператор является симметричным.</p> <p>LEFT (OUTER) JOIN Кортежи из внутреннего соединения, и не вошедшие во внутреннее соединение кортежи из левого источника. Атрибуты в кортежах, которые не имеют совпадений по общим столбцам заполняются неопределенными значениями. Порядок таблиц для оператора важен, поскольку оператор не является симметричным.</p> <p>RIGHT (OUTER) JOIN</p> <p>FULL (OUTER) JOIN Результатом объединения таблиц являются все записи, которые присутствуют в таблицах. Порядок таблиц для оператора не важен, поскольку оператор является симметричным.</p> <p>CROSS JOIN (декартово произведение)</p> <pre>SELECT *     FROM actor NATURAL JOIN film_actor NATURAL JOIN film</pre> <p>Обратите внимание, в этом запросе нет необходимости указывать какие-либо критерии объединения, поскольку предложение NATURAL JOIN автоматически определяет столбцы, имеющие одинаковые имена в обеих объединяемых таблицах, и помещает их в «скрытое» предложение USING. Если первичные и внешние ключи имеют одинаковые имена, этот подход может показаться полезным, однако это не так.</p>
<b>Что лучше использовать join или подзапросы? Почему?</b>	<p>Обычно лучше использовать JOIN, поскольку в большинстве случаев он более понятен и лучше оптимизируется СУБД (но 100% этого гарантировать нельзя). Так же JOIN имеет заметное преимущество над подзапросами в случае, когда список выбора SELECT содержит столбцы более чем из одной таблицы.</p> <p>Подзапросы лучше использовать в случаях, когда нужно вычислять агрегатные значения и использовать их для сравнений во внешних запросах.</p>
<b>Что делает UNION?</b>	<p>В языке SQL ключевое слово UNION применяется для объединения результатов двух SQL-запросов в единую таблицу, состоящую из схожих записей. Оба запроса должны возвращать одинаковое число столбцов и совместимые типы данных в соответствующих столбцах. Необходимо отметить, что UNION сам по себе не гарантирует порядок записей. Записи из второго запроса могут оказаться в начале, в конце или вообще перемешаться с записями из первого запроса. В случаях, когда требуется определенный порядок, необходимо использовать ORDER BY.</p> <p>Разница между UNION и UNION ALL заключается в том, что UNION будет пропускать дубликаты записей, тогда как UNION ALL будет включать дубликаты записей.</p>



<b>Чем WHERE отличается от HAVING (ответа про то что используются в разных частях запроса - недостаточно)?</b>	<p>WHERE нельзя использовать с агрегатными функциями, HAVING можно (предикаты тоже).</p> <p>В HAVING можно использовать псевдонимы только если они используются для наименования результата агрегатной функции, в WHERE можно всегда. &lt;-----????????</p> <p>HAVING стоит после GROUP BY, но может использоваться и без него. При отсутствии предложения GROUP BY агрегатные функции применяются ко всему выходному набору строк запроса, т.е. в результате мы получим всего одну строку, если выходной набор не пуст.</p>
<b>Что такое ORDER BY?</b>	ORDER BY упорядочивает вывод запроса согласно значениям в том или ином количестве выбранных столбцов. Многочисленные столбцы упорядочиваются один внутри другого. Возможно определять возрастание ASC или убывание DESC для каждого столбца. По умолчанию установлено - возрастание.
<b>Что такое DISTINCT?</b>	DISTINCT указывает, что для вычислений используются только уникальные значения столбца.
<b>Что такое GROUP BY?</b>	<p>GROUP BY используется для агрегации записей результата по заданным атрибутам.</p> <p>Создает отдельную группу для всех возможных значений (включая значение NULL)</p> <p>При использовании GROUP BY все значения NULL считаются равными.</p>
<b>Что такое LIMIT?</b>	Ограничивает выборку заданным числом.
<b>Что такое EXISTS?</b>	EXISTS берет подзапрос, как аргумент, и оценивает его как TRUE, если подзапрос возвращает какие-либо записи и FALSE, если нет.
<b>Расскажите про операторы IN, BETWEEN, LIKE.</b>	<ul style="list-style-type: none"> <li>• IN - определяет набор значений. SELECT * FROM Persons WHERE name IN ('Ivan','Petr','Pavel');</li> <li>• BETWEEN определяет диапазон значений. В отличие от IN, BETWEEN чувствителен к порядку, и первое значение в предложении должно быть первым по алфавитному или числовому порядку. SELECT * FROM Persons WHERE age BETWEEN 20 AND 25;</li> <li>• LIKE применим только к полям типа CHAR или VARCHAR, с которыми он используется чтобы находить подстроки. В качестве условия используются символы шаблонизации (wildcards) - специальные символы, которые могут соответствовать чему-нибудь: _ замещает любой одиночный символ. Например, 'b_t' будет соответствовать словам 'bat' или 'bit', но не будет соответствовать 'brat'. % замещает последовательность любого числа символов. Например '%p%t' будет соответствовать словам 'put', 'posit', или 'opt', но не 'spite'. SELECT * FROM UNIVERSITY WHERE NAME LIKE '%o';</li> </ul>
<b>Что делает оператор MERGE? Какие у него есть ограничения?</b>	<p>MERGE позволяет осуществить слияние данных одной таблицы с данными другой таблицы. При слиянии таблиц проверяется условие, и если оно истинно, то выполняется UPDATE, а если нет - INSERT. При этом изменять поля таблицы в секции UPDATE, по которым идет связывание двух таблиц, нельзя.</p> <p><b>MERGE Ships AS t</b> -- таблица, которая будет меняться  <b>USING (SELECT запрос ) AS s ON (t.name = s.ship)</b> -- условие слияния  <b>THEN UPDATE SET t.launched = s.year</b> -- обновление  <b>WHEN NOT MATCHED</b> -- если условие не выполняется  <b>THEN INSERT VALUES(s.ship, s.year)</b> -- вставка</p>



<p><b>Какие агрегатные функции вы знаете?</b></p>	<p>Агрегатных функции - функции, которые берут группы значений и сводят их к одиночному значению.  Несколько агрегатных функций:  COUNT - производит подсчет записей, удовлетворяющих условию запроса;  CONCAT - соединяет строки;  SUM - вычисляет арифметическую сумму всех значений колонки;  AVG - вычисляет среднее арифметическое всех значений;  MAX - определяет наибольшее из всех выбранных значений;  MIN - определяет наименьшее из всех выбранных значений.</p>
<p><b>Что такое ограничения (constraints)? Какие вы знаете?</b></p>	<p>Ограничения - это ключевые слова, которые помогают установить правила размещения данных в базе. Используются при создании БД.</p> <p><b>NOT NULL</b> указывает, что значение не может быть пустым.  <b>UNIQUE</b> обеспечивает отсутствие дубликатов.  <b>PRIMARY KEY</b> - комбинация NOT NULL и UNIQUE. Помечает каждую запись в базе данных уникальным значением.  <b>CHECK</b> проверяет вписывается ли значение в заданный диапазон ( s_id int CHECK(s_id &gt; 0) )  <b>FOREIGN KEY</b> создает связь между двумя таблицами и защищает от действий, которые могут нарушить связи между таблицами. FOREIGN KEY в одной таблице указывает на PRIMARY KEY в другой.  <b>DEFAULT</b> устанавливает значение по умолчанию, если значения не предоставлено (name VARCHAR(20) DEFAULT 'noname').</p> <p>Какие отличия между PRIMARY и UNIQUE?  По умолчанию PRIMARY создает кластерный индекс на столбце, а UNIQUE - некластерный. PRIMARY не разрешает NULL записей, в то время как UNIQUE разрешает одну (а в некоторых СУБД несколько) NULL запись.  Таблица может иметь один PRIMARY KEY и много UNIQUE.</p> <p>Может ли значение в столбце, на который наложено ограничение FOREIGN KEY, равняться NULL?  Может, если на данный столбец не наложено ограничение NOT NULL.</p>
<p><b>Что такое суррогатные ключи?</b></p>	<p>Суррогатный ключ — это дополнительное служебное поле, автоматически добавленное к уже имеющимся информационным полям таблицы, предназначение которого — служить первичным ключом.</p>
<p><b>Что такое индексы?</b></p>	<p>Индексы относятся к методу настройки производительности, позволяющему быстрее извлекать записи из таблицы. Индекс создает структуру для индексируемого поля. Необходимо просто добавить указатель индекса в таблицу.</p> <p>Есть три типа индексов, а именно:</p> <p><b>Уникальный индекс (Unique Index):</b> этот индекс не позволяет полю иметь повторяющиеся значения. Если первичный ключ определен, уникальный индекс применен автоматически.</p> <p><b>Кластеризованный индекс (Clustered Index):</b> сортируют и хранят строки данных в таблицах или представлениях на основе их ключевых значений. Это ускоряет операции чтения из БД.</p> <p><b>Некластеризованный индекс (Non-Clustered Index):</b> внутри таблицы есть упорядоченный список, содержащий значения ключа некластеризованного индекса и указатель на строку данных, содержащую значение ключа. Каждый новый индекс увеличивает время, необходимое для создания новых записей из-за упорядочивания. Каждая таблица может иметь много некластеризованных индексов.</p>

Какие они бывают?	<p>Как создать индекс? b3</p> <p>Индекс можно создать либо с помощью выражения CREATE INDEX:  CREATE INDEX index_name ON table_name (column_name)  либо указав ограничение целостности в виде уникального UNIQUE или первичного PRIMARY ключа в операторе создания таблицы CREATE TABLE.</p> <p>Имеет ли смысл индексировать данные, имеющие небольшое количество возможных значений?</p> <p>Примерное правило, которым можно руководствоваться при создании индекса - если объем информации (в байтах) НЕ удовлетворяющей условию выборки меньше, чем размер индекса (в байтах) по данному условию выборки, то в общем случае оптимизация приведет к замедлению выборки.</p> <p>Когда полное сканирование набора данных выгоднее доступа по индексу?</p> <p>Полное сканирование производится многоблочным чтением. Сканирование по индексу - одноблочным. Также, при доступе по индексу сначала идет сканирование самого индекса, а затем чтение блоков из набора данных. Число блоков, которые надо при этом прочитать из набора зависит от фактора кластеризации. Если суммарная стоимость всех необходимых одноблочных чтений больше стоимости полного сканирования многоблочным чтением, то полное сканирование выгоднее и оно выбирается оптимизатором.</p> <p>Таким образом, полное сканирование выбирается при слабой селективности предикатов зароса и/или слабой кластеризации данных, либо в случае очень маленьких наборов данных.</p>
Чем TRUNCATE отличается от DELETE?	<p>DELETE - оператор DML, удаляет записи из таблицы, которые удовлетворяют условиям WHERE. Медленнее, чем TRUNCATE. Есть возможность восстановить данные.</p> <p>TRUNCATE - DDL оператор, удаляет все строки из таблицы. Нет возможность восстановить данные - сделать ROLLBACK.</p>
Что такое хранимые процедуры? Для чего они нужны?	<p><b>Хранимая процедура — объект базы данных, представляющий собой набор SQL-инструкций, который хранится на сервере.</b> Хранимые процедуры очень похожи на обыкновенные методы языков высокого уровня, у них могут быть входные и выходные параметры и локальные переменные, в них могут производиться числовые вычисления и операции над символьными данными, результаты которых могут присваиваться переменным и параметрам. В хранимых процедурах могут выполняться стандартные операции с базами данных (как DDL, так и DML). Кроме того, в хранимых процедурах возможны циклы и ветвления, то есть в них могут использоваться инструкции управления процессом исполнения. Хранимые процедуры позволяют повысить производительность, расширяют возможности программирования и поддерживают функции безопасности данных. В большинстве СУБД при первом запуске хранимой процедуры она компилируется (выполняется синтаксический анализ и генерируется план доступа к данным) и в дальнейшем её обработка осуществляется быстрее.</p>
Что такое представления (VIEW)? Для чего они нужны?	<p>View - виртуальная таблица, представляющая данные одной или более таблиц альтернативным образом.</p> <p>В действительности представление – всего лишь результат выполнения оператора SELECT, который хранится в структуре памяти, напоминающей SQL таблицу. Они работают в запросах и операторах DML точно также как и основные таблицы, но не содержат никаких собственных данных. Представления значительно расширяют возможности управления данными. Это способ дать публичный доступ к некоторой (но не всей) информации в таблице.</p> <p>Представления могут основываться как на таблицах, так и на других представлениях, т.е. могут быть вложенными (до 32 уровней вложенности).</p>
Что такое временные таблицы? Для чего они нужны?	<p>Подобные таблицы удобны для каких-то временных промежуточных выборок из нескольких таблиц.</p> <p>Создание временной таблицы начинается со знака решетки #. Если используется один знак #, то создается локальная таблица, которая доступна в течение текущей сессии. Если используются два знака ##, то создается глобальная временная таблица. В отличие от локальной глобальная временная таблица доступна всем открытым сессиям базы данных.</p> <p>CREATE TABLE #ProductSummary  (ProdId INT IDENTITY,  ProdName NVARCHAR(20),  Price MONEY)</p>

**Что такое транзакции?  
Расскажите про принципы ACID.**

**Транзакция** - это воздействие на базу данных, переводящее её из одного целостного состояния в другое и выражаемое в изменении данных, хранящихся в базе данных.

ACID-принципы транзакций:

- **Атомарность** (atomicity) гарантирует, что транзакция будет полностью выполнена или потерпит неудачу, где транзакция представляет одну логическую операцию данных. Это означает, что при сбое одной части любой транзакции происходит сбой всей транзакции и состояние базы данных остается неизменным.
- **Согласованность** (consistency). Транзакция, достигающая своего завершения и фиксирующая свои результаты, сохраняет согласованность базы данных
- **Изолированность** (isolation). Во время выполнения транзакции параллельные транзакции не должны оказывать влияние на ее результат.
- **Долговечность** (durability). Независимо от проблем (к примеру, потеря питания, сбой или ошибки любого рода) изменения, сделанные успешно завершённой транзакцией, должны остаться сохранёнными после возвращения системы в работу.

**Расскажите про уровни изолированности транзакций.**

В порядке увеличения изолированности транзакций и, соответственно, надежности работы с данными:

- **Чтение неподтверждённых данных (read uncommitted)** — чтение незафиксированных изменений как своей транзакции, так и параллельных транзакций. Нет гарантии, что данные, изменённые другими транзакциями, не будут в любой момент изменены в результате их отката, поэтому такое чтение является потенциальным источником ошибок. Возможны неповторяемое чтение, фантомы и грязное чтение.
  - **Чтение подтверждённых данных (read committed)** — чтение всех изменений своей транзакции и зафиксированных изменений параллельных транзакций, но процессы-писатели могут изменять уже прочитанные читателем данные. Возможны неповторяемое чтение и фантомы.
  - **Повторяемость чтения (repeatable read)** — Уровень, позволяющий предотвратить неповторяемое чтение. Т.е. мы не видим в исполняющейся транзакции изменённые и удалённые записи этой же или другой транзакцией. Но все еще видим вставленные записи из другой транзакции. В MySQL и PostgreSQL отсутствует эффект чтения фантомов для этого уровня.
  - **Упорядочиваемость (serializable)** — гарантирует неизменяемость данных другими процессами до завершения транзакции. Проблемы синхронизации не возникают.
- <https://habr.com/ru/post/469415/> [https://www.youtube.com/watch?v=5Z2iFX3OeTo&ab\\_channel=%D0%A3%D1%80%D0%BE%B8JavaD0%BA%D0%](https://www.youtube.com/watch?v=5Z2iFX3OeTo&ab_channel=%D0%A3%D1%80%D0%BE%B8JavaD0%BA%D0%)

При параллельном выполнении транзакций возможны следующие проблемы:

Потерянное обновление (lost update) — при одновременном изменении одного блока данных разными транзакциями одно из изменений теряется;  
«Грязное» чтение (dirty read) — чтение данных, добавленных или изменённых транзакцией, которая впоследствии не подтвердится (откатится);  
Неповторяющееся чтение (non-repeatable read) — при повторном чтении в рамках одной транзакции ранее прочитанные данные оказываются изменёнными;

Фантомное чтение (phantom reads) — одна транзакция в ходе своего выполнения несколько раз выбирает множество записей по одним и тем же критериям. Другая транзакция в интервалах между этими выборками добавляет или удаляет записи или изменяет столбцы некоторых записей, используемых в критериях выборки первой транзакции, и успешно заканчивается. В результате получится, что одни и те же выборки в первой транзакции дают разные множества записей.

Что такое  
нормализация и  
денормализация?  
Расскажите про 3  
нормальные  
формы?

Нормализация - это процесс преобразования отношений базы данных к виду, отвечающему нормальным формам (пошаговый, обратимый процесс приведения данных в более простую и логичную структуру).

Целью является уменьшение потенциальной противоречивости хранимой в базе данных информации.

Денормализация базы данных — это процесс обратный от нормализации. Эта техника добавляет избыточные данные в таблицу, учитывая частые запросы к базе данных, которые объединяют данные из разных таблиц в одну таблицу. Необходимо для повышения производительности и скорости извлечения данных, за счет увеличения избыточности данных.

Каждая нормальная форма включает в себя предыдущую. Типы форм:

- Первая нормальная форма (1NF) - значения всех полей атомарны (неделимы), нет множества значений в одном поле.
- Вторая нормальная форма (2NF) - все неключевые поля зависят только от ключа целиком, а не от какой-то его части.
- Третья нормальная форма (3NF) - все неключевые поля не зависят друг от друга.
- Нормальная форма Бойса-Кодда, усиленная 3 нормальная форма (BCNF) - когда каждая её нетривиальная и неприводимая слева функциональная зависимость имеет в качестве своего детерминанта некоторый потенциальный ключ.
- Четвёртая нормальная форма (4NF) - не содержатся независимые группы полей, между которыми существует отношение «многие-ко-многим».
- Пятая нормальная форма (5NF) - каждая нетривиальная зависимость соединения в ней определяется потенциальным ключом (ключами) этого отношения.
- Доменно-ключевая нормальная форма (DKNF) - каждое наложенное на нее ограничение является логическим следствием ограничений доменов и ограничений ключей, наложенных на данное отношение.
- Шестая нормальная форма (6NF) - удовлетворяет всем нетривиальным зависимостям соединения, то есть не может быть подвергнута дальнейшей декомпозиции без потерь. Введена как обобщение пятой нормальной формы для хронологической базы данных.

Что такое  
TIMESTAMP?

**DATETIME** предназначен для хранения целого числа: YYYYMMDDHHMMSS. И это время не зависит от временной зоны настроенной на сервере.

Размер: 8 байт

**TIMESTAMP** хранит значение равное количеству секунд, прошедших с полуночи 1 января 1970 года по усреднённому времени Гринвича. Тогда была создана Unix. При получении из базы отображается с учётом часового пояса. Размер: 4 байта

При большом количестве данных запросы начинают долго выполняться, и сервер начинает не справляться с нагрузкой. Одно из решений, что с этими данными делать — это масштабирование базы данных. Например, шардинг или репликация.

**Шардинг бывает вертикальным(партиционирование) и горизонтальным.**

У нас есть большая таблица, например, с пользователями. Партиционирование — это когда мы одну большую таблицу разделяем на много маленьких по какому-либо принципу.

Единственное отличие горизонтального масштабирования от вертикального в том, что горизонтальное будет разносить данные по разным инстансам в других базах.

## Шардирование БД

```
01. CREATE TABLE news (  
02.     id bigint not null,  
03.     category_id int not null,  
04.     author character varying not null,  
05.     rate int not null,  
06.     title character varying  
07. )
```

Есть таблица news, в которой есть идентификатор, есть категория, в которой эта новость расположена, есть автор новости...

**Нужно сделать 2 действия над табличкой** — это поставить у нашего шарда, например, news\_1, то, что она будет наследоваться от news.

Наследованная таблица будет иметь все колонки родителя, а также она может иметь свои колонки, которые мы дополнительно туда добавим. Там не будет ограничений, индексов и триггеров от родителя — это важно.

**2-ое действие** — это поставить ограничения. Это будет проверка, что в эту таблицу будут попадать данные только с нужным признаком.

```
01. CREATE TABLE news_1 (  
02.     CHECK ( category_id = 1 )  
03. ) INHERITS (news)
```

Т.е. только записи с category\_id=1 будут попадать в эту таблицу.

На базовую таблицу надо добавить правило. Когда мы будем работать с таблицей news, вставка на запись с category\_id = 1 должна попасть именно в партицию news\_1. Правило называем как хотим.

```
01. CREATE RULE news_insert_to_1 AS ON INSERT TO news  
02. WHERE ( category_id = 1 )  
03. DO INSTEAD INSERT INTO news_1 VALUES (NEW.*)
```

<b>EXPLAIN</b>	<p>Когда вы выполняете какой-нибудь запрос, оптимизатор запросов MySQL пытается придумать оптимальный план выполнения этого запроса. Можно посмотреть этот план используя запрос с ключевым словом EXPLAIN перед оператором SELECT.</p> <p><b>EXPLAIN SELECT * FROM categories</b></p> <p>После EXPLAIN в запросе вы можете использовать ключевое слово EXTENDED и MySQL покажет вам дополнительную информацию о том, как выполняется запрос. Чтобы увидеть эту информацию, вам нужно сразу после запроса с EXTENDED выполнить запрос SHOW WARNINGS.</p> <p>EXPLAIN <b>EXTENDED</b> SELECT City.Name FROM City</p> <p>Затем</p> <p><b>SHOW WARNINGS</b></p>
<b>Как сделать запрос из двух баз?</b>	<p>Если в запросе таблица указывается с именем базы данных database1.table1, то таблица выбирается из database1, если просто table1, то - из активной базы данных.</p> <p>Надо, чтобы базы были на одном сервере.</p> <p>SELECT t1.*, t2.*</p> <p>FROM database1.table1 AS t1</p> <p>INNER JOIN database2.table2 AS t2 ON t1.field1 = t2.field1</p>
<b>Что быстрее убирает дубликаты distinct или group by?</b>	<p>Если нужны уникальные значения - DISTINCT.</p> <p>Если нужно группировать значения - GROUP BY.</p> <p>Если задача заключается именно в поиске дубликатов - <b>GROUP BY будет лучше.</b></p>
<b>Механизмы оптимизации запросов в БД</b>	<p>Например, добавить индекс по нужной колонке.</p>
<b>Что такое «триггер»?</b>	<p>Триггер (trigger) — это хранимая процедура особого типа, исполнение которой обусловлено действием по модификации данных: добавлением, удалением или изменением данных в заданной таблице реляционной базы данных. Триггер запускается сервером автоматически и все производимые им модификации данных рассматриваются как выполняемые в транзакции, в которой выполнено действие, вызвавшее срабатывание триггера.</p> <p>Момент запуска триггера определяется с помощью ключевых слов BEFORE (триггер запускается до выполнения связанного с ним события) или AFTER (после события).</p>

индексы

<https://habr.com/ru/articles/247373/>

## Hibernate

### Что такое ORM? Что такое JPA? Что такое Hibernate?

**ORM**(Object Relational Mapping) - это концепция преобразования данных из объектно-ориентированного языка в реляционные БД и наоборот.  
**JPA**(Java Persistence API) - это стандартная для Java спецификация, описывающая принципы ORM. JPA не умеет работать с объектами, а только определяет правила как должен действовать каждый провайдер (Hibernate, EclipseLink), реализующий стандарт JPA

**Hibernate** - библиотека, являющаяся реализацией этой спецификации, в которой можно использовать стандартные API-интерфейсы JPA.

Важные интерфейсы Hibernate:

**Session** - обеспечивает физическое соединение между приложением и БД. Основная функция - предлагать DML-операции для экземпляров сущностей.

**SessionFactory** - это фабрика для объектов Session. Обычно создается во время запуска приложения и сохраняется для последующего использования. Является потокобезопасным объектом и используется всеми потоками приложения.

**Transaction** - однопоточный короткоживущий объект, используемый для атомарных операций. Это абстракция приложения от основных JDBC транзакций. Session может занимать несколько Transaction в определенных случаях, является необязательным API.

**Query** - интерфейс позволяет выполнять запросы к БД. Запросы написаны на HQL или на SQL.

### Что такое EntityManager?

EntityManager интерфейс JPA, который описывает API для всех основных операций над Entity, а также для получения данных и других сущностей JPA.

Основные операции:

1) Операции над Entity: persist (добавление Entity), merge (обновление), remove (удаления), refresh (обновление данных), detach (удаление из управление JPA), lock (блокирование Entity от изменений в других thread),

2) Получение данных: find (поиск и получение Entity), createQuery, createNamedQuery, createNativeQuery, contains, createNamedStoredProcedureQuery, createStoredProcedureQuery

3) Получение других сущностей JPA: getTransaction, getEntityManagerFactory, getCriteriaBuilder, getMetamodel, getDelegate

4) Работа с EntityGraph: createEntityGraph, getEntityGraph

5) Общие операции над EntityManager или всеми Entities: close, clear, isOpen, getProperties, setProperty.

Объекты EntityManager не являются потокобезопасными. Это означает, что каждый поток должен получить свой экземпляр EntityManager, поработать с ним и закрыть его в конце.

**Entity** это легковесный хранимый объект бизнес логики. Основная программная сущность это entity-класс, который так же может использовать дополнительные классы, которые могут использоваться как вспомогательные классы или для сохранения состояния entity.



<p><b>Каким условиям должен удовлетворять класс чтобы являться Entity?</b></p>	<ol style="list-style-type: none"> <li>1) Entity класс должен быть помечен аннотацией Entity или описан в XML файле</li> <li>2) Entity класс должен содержать public или protected конструктор без аргументов (он также может иметь конструкторы с аргументами) - при получении данных из БД и формировании из них объекта сущности, Hibernate должен создать этот самый объект сущности,</li> <li>3) Entity класс должен быть классом верхнего уровня (top-level class),</li> <li>4) Entity класс не может быть enum или интерфейсом,</li> <li>5) Entity класс не может быть финальным классом (final class),</li> <li>6) Entity класс не может содержать финальные поля или методы, если они участвуют в маппинге (persistent final methods or persistent final instance variables),</li> <li>7) Если объект Entity класса будет передаваться по значению как отдельный объект (detached object), например через удаленный интерфейс (through a remote interface), он так же должен реализовывать Serializable интерфейс</li> <li>8) Поля Entity класс должны быть напрямую доступны только методам самого Entity класса и не должны быть напрямую доступны другим классам, использующим этот Entity. Такие классы должны обращаться только к методам (getter/setter методам или другим методам бизнес-логики в Entity классе),</li> <li>9) Entity класс должен содержать первичный ключ, то есть атрибут или группу атрибутов которые уникально определяют запись этого Entity класса в базе данных</li> </ol>
<p><b>Может ли абстрактный класс быть Entity?</b></p>	<p>Может, при этом он сохраняет все свойства Entity, за исключением того что его нельзя непосредственно инициализировать.</p>
<p><b>Может ли Entity класс наследоваться от не Entity классов (non-entity classes)?</b></p>	<p>Может</p>
<p><b>Может ли Entity класс наследоваться от других Entity классов?</b></p>	<p>Может</p>
<p><b>Может ли не Entity класс наследоваться от Entity класса?</b></p>	<p>Может</p>
<p><b>Что такое встраиваемый (Embeddable) класс? Какие требования JPA устанавливает к встраиваемым (Embeddable) классам?</b></p>	<p><b>Embeddable класс - это класс, который не используется сам по себе, а является частью одного или нескольких Entity-классов.</b> Entity-класс может содержать как одиночные встраиваемые классы, так и коллекции таких классов. Также такие классы могут быть использованы как ключи или значения map. Во время выполнения каждый встраиваемый класс принадлежит только одному объекту Entity-класса и не может быть использован для передачи данных между объектами Entity-классов (то есть такой класс не является общей структурой данных для разных объектов). В целом, такой класс служит для того чтобы выносить определение общих атрибутов для нескольких Entity.</p> <ol style="list-style-type: none"> <li>1. Такие классы должны удовлетворять тем же правилам что Entity классы, за исключением того что они <b>не обязаны содержать первичный ключ и быть отмечены аннотацией Entity</b></li> <li>2. Embeddable класс должен быть помечен аннотацией @Embeddable или описан в XML файле конфигурации JPA. А поле этого класса в Entity аннотацией @Embedded</li> </ol> <p>Embeddable-класс может содержать другой встраиваемый класс.</p>



	Встраиваемый класс может содержать связи с другими Entity или коллекциями Entity, если такой класс не используется как первичный ключ или ключ map'ы.
Что такое Mapped Superclass?	<p>Mapped Superclass - это класс, от которого наследуются Entity, он может содержать аннотации JPA, однако сам такой класс не является Entity, ему не обязательно выполнять все требования установленные для Entity (например, он может не содержать первичного ключа). Такой класс не может использоваться в операциях EntityManager или Query. Такой класс должен быть отмечен аннотацией MappedSuperclass или описан в xml файле. Создание такого класса-предка оправдано тем, что мы заранее определяем ряд свойств и методов, которые должны быть определены в сущностях. Использование такого подхода позволило сократить количество кода</p>
Какие три типа стратегий наследования мапинга (Inheritance Mapping Strategies) описаны в JPA?	<p>Inheritance Mapping Strategies описывает как JPA будет работать с классами-наследниками Entity:</p> <p>1) Одна таблица на всю иерархию классов (SINGLE_TABLE) — все entity, со всеми наследниками записываются в одну таблицу, для идентификации типа entity определяется специальная колонка "discriminator column". Например, если есть entity Animals с классами-потомками Cats и Dogs, при такой стратегии все entity записываются в таблицу Animals, но при это имеют дополнительную колонку animalType в которую соответственно пишется значение «cat» или «dog». Минусом является то что в общей таблице, будут созданы все поля уникальные для каждого из классов-потомков, которые будут пусты для всех других классов-потомков. Например, в таблице animals окажется и скорость лазанья по дереву от cats и может ли пес приносить тапки от dogs, которые будут всегда иметь null для dog и cat соответственно. Нельзя делать констраинт notNull, но можно использовать триггеры.</p> <p>2) Стратегия «соединения» (JOINED) — в этой стратегии каждый класс entity сохраняет данные в свою таблицу, но только уникальные поля (не унаследованные от классов-предков) и первичный ключ, а все унаследованные колонки записываются в таблицы класса-предка, дополнительно устанавливается связь (relationships) между этими таблицами, например в случае классов Animals (см.выше), будут три таблицы animals, cats, dogs, причем в cats будет записана только ключ и скорость лазанья, в dogs — ключ и умеет ли пес приносить палку, а в animals все остальные данные cats и dogs с ссылкой на соответствующие таблицы. Минусом тут являются потери производительности от объединения таблиц (join) для любых операций.</p> <p>3) Таблица для каждого класса (TABLE_PER_CLASS) — каждый отдельный класс-наследник имеет свою таблицу, т.е. для cats и dogs (см. выше) все данные будут записываться просто в таблицы cats и dogs как если бы они вообще не имели общего суперкласса. Минусом является плохая поддержка полиморфизма (polymorphic relationships) и то что для выборки всех классов иерархии потребуются большое количество отдельных sql запросов или использование UNION запроса.</p> <p>Для задания стратегии наследования используется аннотация Inheritance (или соответствующие блоки)</p>
Как маются Enum'ы?	<p><b>@Enumerated(EnumType.STRING)</b> - означает, что в базе будут храниться имена Enum.</p> <p><b>@Enumerated(EnumType.ORDINAL)</b> - в базе будут храниться порядковые номера Enum.</p> <p>Другой вариант - мы можем смэпить наши enum в БД и обратно в методах с аннотациями <b>@PostLoad</b> и <b>@PrePersist</b>. <b>@EntityListener</b> над классом <b>Entity</b>, в которой указать класс, в котором создать два метода, помеченных этими аннотациями.</p> <p>Идея в том, чтобы в сущности иметь не только поле с Enum, но и вспомогательное поле. Поле с Enum аннотируем <b>@Transient</b>, а в БД будет храниться значение из вспомогательного поля.</p> <p>В JPA с версии 2.1 можно использовать Converter для конвертации Enum'a в некое его значение для сохранения в БД и получения из БД. Все, что нам нужно сделать, это создать новый класс, который реализует <code>javax.persistence.AttributeConverter</code> и аннотировать его с помощью <b>@Converter</b> и поле в сущности аннотацией <b>@Convert</b>.</p>
Как маются даты (до java 8 и после)?	<p>Аннотация <b>@Temporal</b> до Java 8, в которой надо было указать какой тип даты мы хотим использовать.</p> <p>В Java 8 и далее аннотацию ставить не нужно.</p>

<p>Как “смапить” коллекцию примитивов?</p>	<p><b>@ElementCollection</b>  <b>@OrderBy</b>          Если у нашей сущности есть поле с коллекцией, то мы привыкли ставить над ним аннотации @OneToMany либо @ManyToMany. Но данные аннотации применяются в случае, когда это коллекция других сущностей (entities). Если у нашей сущности коллекция не других сущностей, а базовых или встраиваемых (embeddable) типов для этих случаев в JPA имеется специальная аннотация <b>@ElementCollection</b>, которая указывается в классе сущности над полем коллекции. Все записи коллекции хранятся в отдельной таблице, то есть в итоге получаем две таблицы: одну для сущности, вторую для коллекции элементов.          При добавлении новой строки в коллекцию, она полностью очищается и заполняется заново, так как у элементов нет id. Можно решить с помощью <b>@OrderColumn</b>  <b>@CollectionTable</b> - позволяет редактировать таблицу с коллекцией, прочитать</p>
<p>Какие есть виды связей?</p>	<p>Существуют 4 типа связей:          1. OneToOne - когда один экземпляр Entity может быть связан не больше чем с одним экземпляром другого Entity.          2. OneToMany - когда один экземпляр Entity может быть связан с несколькими экземплярами других Entity.          3. ManyToOne - обратная связь для OneToMany. Несколько экземпляров Entity могут быть связаны с одним экземпляром другого Entity.          4. ManyToMany - экземпляры Entity могут быть связаны с несколькими экземплярами друг друга.</p> <p>Каждую из которых можно разделить ещё на два вида:          1. Bidirectional с использованием mappedBy на стороне, где указывается @OneToMany          2. Unidirectional</p> <p><b>Bidirectional</b> — ссылка на связь устанавливается у всех Entity, то есть в случае OneToOne A-B в Entity A есть ссылка на Entity B, в Entity B есть ссылка на Entity A. Entity A считается владельцем этой связи (это важно для случаев каскадного удаления данных, тогда при удалении A также будет удалено B, но не наоборот).  <b>Unidirectional</b>- ссылка на связь устанавливается только с одной стороны, то есть в случае OneToOne A-B только у Entity A будет ссылка на Entity B, у Entity B ссылки на A не будет.</p>
<p>Что такое владелец связи?</p>	<p>В отношениях между двумя сущностями всегда есть одна владеющая сторона, а владеемой может и не быть, если это однонаправленные отношения.          По сути, <b>у кого есть внешний ключ на другую сущность - тот и владелец связи</b>. То есть, если в таблице одной сущности есть колонка, содержащая внешние ключи от другой сущности, то первая сущность признаётся владельцем связи, вторая сущность - владеемой.          В однонаправленных отношениях сторона, которая имеет поле с типом другой сущности, является владельцем этой связи по умолчанию.</p>
<p>Что такое каскады?</p>	<p>Каскадирование - это когда мы выполняем какое-то действие с целевой Entity, то же самое действие будет применено к связанной Entity.          JPA CascadeType:          ALL - гарантируют, что все персистентные события, которые происходят на родительском объекте, будут переданы дочернему объекту.          PERSIST - означает, что операции save () или persist () каскадно передаются связанным объектам.          MERGE - означает, что связанные entity объединяются, когда объединяется entity-владелец.          REMOVE - удаляет все entity, связанные с удаляемой entity.          DETACH - отключает все связанные entity, если происходит «ручное отключение».          REFRESH - повторно считывают значение данного экземпляра и связанных сущностей из базы данных при вызове refresh()).</p>
<p>Разница между PERSIST и MERGE?</p>	<p>persist(entity) следует использовать с совершенно новыми объектами, чтобы добавить их в БД (если объект уже существует в БД, будет выброшено исключение EntityExistsException).          Но в случае merge(entity) сущность, которая уже управляется в контексте персистентности, будет заменена новой сущностью (обновленной), и копия этой обновленной сущности вернется обратно. Но рекомендуется использовать для уже сохраненных сущностей.</p>

Какие два типа fetch стратегии в JPA вы знаете?	1) <b>LAZY</b> — Hibernate может загружать данные не сразу, а при первом обращении к ним, но так как это необязательное требование, то Hibernate имеет право изменить это поведение и загружать их сразу. Это поведение по умолчанию для полей, аннотированных <code>@OneToMany</code> , <code>@ManyToMany</code> и <code>@ElementCollection</code> . <b>В объект загружаются прокси lazy-поля.</b> 2) <b>EAGER</b> — данные поля будут загружены немедленно. Это поведение по умолчанию для полей, аннотированных <code>@Basic</code> , <code>@ManyToOne</code> и <code>@OneToOne</code> .
Какие четыре статуса жизненного цикла Entity объекта (Entity Instance's Life Cycle) вы можете перечислить?	<b>Transient (New)</b> — свежесозданная оператором <code>new()</code> сущность не имеет связи с базой данных, не имеет данных в базе данных и не имеет сгенерированных первичных ключей. <b>managed</b> - объект создан, сохранён в бд, имеет <code>primary key</code> , управляется JPA <b>detached</b> - объект создан, не управляется JPA. В этом состоянии сущность не связана со своим контекстом (отделена от него) и нет экземпляра Session, который бы ей управлял. <b>removed</b> - объект создан, управляется JPA, будет удален при <code>commit</code> -е и статус станет опять <code>detached</code>
Как влияет операция <code>persist</code> на Entity объекты каждого из четырех статусов?	<b>new</b> → <code>managed</code> , и объект будет сохранен в базу при <code>commit</code> -е транзакции или в результате <code>flush</code> операций <b>managed</b> → операция игнорируется, однако зависимые Entity могут поменять статус на <code>managed</code> , если у них есть аннотации каскадных изменений <b>detached</b> → <code>exception</code> сразу или на этапе <code>commit</code> -а транзакции <b>removed</b> → <code>managed</code> , но только в рамках одной транзакции.
Как влияет операция <code>remove</code> на Entity объекты каждого из четырех статусов?	<b>new</b> → операция игнорируется, однако зависимые Entity могут поменять статус на <code>removed</code> , если у них есть аннотации каскадных изменений и они имели статус <code>managed</code> <b>managed</b> → <code>removed</code> и запись объект в базе данных будет удалена при <code>commit</code> -е транзакции (также произойдут операции <code>remove</code> для всех каскадно зависимых объектов) <b>detached</b> → <code>exception</code> сразу или на этапе <code>commit</code> -а транзакции <b>removed</b> → операция игнорируется
Как влияет операция <code>merge</code> на Entity объекты каждого из четырех статусов?	<b>new</b> → будет создан новый <code>managed entity</code> , в который будут скопированы данные прошлого объекта <b>managed</b> → операция игнорируется, однако операция <code>merge</code> сработает на каскадно зависимые Entity, если их статус не <code>managed</code> <b>detached</b> → либо данные будут скопированы в существующий <code>managed entity</code> с тем же первичным ключом, либо создан новый <code>managed</code> в который скопируются данные <b>removed</b> → <code>exception</code> сразу или на этапе <code>commit</code> -а транзакции
Как влияет операция <code>refresh</code> на Entity объекты каждого из четырех статусов?	<b>managed</b> → будут восстановлены все изменения из базы данных данного Entity, также произойдет <code>refresh</code> всех каскадно зависимых объектов <b>new, removed, detached</b> → <code>exception</code>
Как влияет операция <code>detach</code> на Entity объекты каждого из четырех статусов?	<b>managed, removed</b> → <code>detached</code> . <b>new, detached</b> → операция игнорируется

### Для чего нужна аннотация Basic?

**@Basic** - указывает на простейший тип маппинга данных на колонку таблицы базы данных. Также в параметрах аннотации можно указать fetch стратегию доступа к полю и является ли это поле обязательным или нет. Может быть применена к полю любого из следующих типов:

1. Прimitives и их обертки.
2. java.lang.String
3. java.math.BigInteger
4. java.math.BigDecimal
5. java.util.Date
6. java.util.Calendar
7. java.sql.Date
8. java.sql.Time
9. java.sql.Timestamp
10. byte[] or Byte[]
11. char[] or Character[]
12. enums
13. любые другие типы, которые реализуют Serializable.

Вообще, аннотацию @Basic можно не ставить, как это и происходит по умолчанию.

Аннотация @Basic определяет 2 атрибута:

1. optional - boolean (по умолчанию true) - определяет, может ли значение поля или свойства быть null. Игнорируется для примитивных типов. Но если тип поля не примитивного типа, то при попытке сохранения сущности будет выброшено исключение.
2. fetch - FetchType (по умолчанию EAGER) - определяет, должен ли этот атрибут извлекаться незамедлительно (EAGER) или лениво (LAZY). Однако, это необязательное требование JPA, и провайдером разрешено незамедлительно загружать данные, даже для которых установлена ленивая загрузка.

Без аннотации @Basic при получении сущности из БД по умолчанию её поля базового типа загружаются принудительно (EAGER) и значения этих полей могут быть null

### Для чего нужна аннотация Column?

**@Column** сопоставляет поле класса столбцу таблицы, а её атрибуты определяют поведение в этом столбце, используется для генерации схемы базы данных

@Basic vs @Column:

1. Атрибуты @Basic применяются к сущностям JPA, тогда как атрибуты @Column применяются к столбцам базы данных.
2. @Basic имеет атрибут optional, который говорит о том, может ли поле объекта быть null или нет; с другой стороны атрибут nullable аннотации @Column указывает, может ли соответствующий столбец в таблице быть null.
3. Мы можем использовать @Basic, чтобы указать, что поле должно быть загружено лениво.
4. Аннотация @Column позволяет нам указать имя столбца в таблице и ряд других свойств:
  - a. insertable/updatable - можно ли добавлять/изменять данные в колонке, по умолчанию true;
  - b. length - длина, для строковых типов данных, по умолчанию 255.

Коротко, в Column (колум) мы задаем constraints (констрейнтс), а в Basic (бейсик) - ФЕТЧ ТАЙП

Для чего нужна  
аннотация Access?

Она определяет тип доступа (access type) для класса entity, суперкласса, embeddable или отдельных атрибутов, то есть как JPA будет обращаться к атрибутам entity, как к полям класса (FIELD) или как к свойствам класса (PROPERTY), имеющие геттеры (getter) и сеттеры (setter).

Определяет тип доступа к полям сущности. Для чтения и записи этих полей есть два подхода:

1. Field access (доступ по полям). При таком способе аннотации маппинга (Id, Column,...) размещаются над полями, и Hibernate напрямую работает с полями сущности, читая и записывая их.
2. Property access (доступ по свойствам). При таком способе аннотации размещаются над методами-геттерами, но никак не над сеттерами.

По умолчанию тип доступа определяется местом, в котором находится аннотация @Id. Если она будет над полем - это будет AccessType.FIELD, если над геттером - это AccessType.PROPERTY.

Чтобы явно определить тип доступа у сущности, нужно использовать аннотацию @Access, которая может быть указана у сущности, Mapped Superclass и Embeddable class, а также над полями или методами.

Поля, унаследованные от суперкласса, имеют тип доступа этого суперкласса.

Когда у одной сущности определены разные типы доступа, то нужно использовать аннотацию @Transient для избежания дублирования маппинга.

Для чего нужна  
аннотация  
@Cacheable?

@Cacheable - необязательная аннотация JPA, используется для указания того, должна ли сущность храниться в кэше второго уровня. JPA говорит о пяти значениях shared-cache-mode из persistence.xml, который определяет как будет использоваться second-level cache:

- ❖ **ENABLE\_SELECTIVE**: только сущности с аннотацией @Cacheable (равносильно значению по умолчанию @Cacheable(value=true)) будут сохраняться в кэше второго уровня.
- ❖ **DISABLE\_SELECTIVE**: все сущности будут сохраняться в кэше второго уровня, за исключением сущностей, помеченных @Cacheable(value=false) как некешируемые.
- ❖ **ALL**: сущности всегда кешируются, даже если они помечены как некешируемые.
- ❖ **NONE**: ни одна сущность не кешируется, даже если помечена как кешируемая. При данной опции имеет смысл вообще отключить кэш второго уровня.
- ❖ **UNSPECIFIED**: применяются значения по умолчанию для кэша второго уровня, определенные Hibernate. Это эквивалентно тому, что вообще не используется shared-cache-mode, так как Hibernate не включает кэш второго уровня, если используется режим UNSPECIFIED.

Аннотация @Cacheable размещается над классом сущности. Её действие распространяется на эту сущность и её наследников, если они не определили другое поведение.

### Для чего нужна аннотация @Cache?

Это аннотация Hibernate, настраивающая тонкости кэширования объекта в кэше второго уровня Hibernate. @Cache принимает три параметра:

- ❖ **include** - имеет по умолчанию значение **all** и означающий кэширование всего объекта. Второе возможное значение - **non-lazy**, запрещает кэширование лениво загружаемых объектов. Кэш первого уровня не обращает внимания на эту директиву и всегда кэширует лениво загружаемые объекты.

- ❖ **region** - позволяет задать имя региона кэша для хранения сущности. Регион можно представить как разные области кэша, имеющие разные настройки на уровне реализации кэша. Например, можно было бы создать в конфигурации ehcache два региона, один с краткосрочным хранением объектов, другой с долгосрочным и отправлять часто изменяющиеся объекты в первый регион, а все остальные - во второй. Ehcache по умолчанию создает регион для каждой сущности с именем класса этой сущности, соответственно в этом регионе хранятся только эти сущности. К примеру, экземпляры Foo хранятся в Ehcache в кэше с именем "com.baeldung.hibernate.cache.model.Foo".

- ❖ **usage** - задаёт стратегию одновременного доступа к объектам.

transactional

read-write

nonstrict-read-write

read-only

@Embeddable - аннотация JPA, размещается над классом для указания того, что класс является встраиваемым в другие классы.

@Embedded - аннотация JPA, используется для размещения над полем в классе-сущности для указания того, что мы внедряем встраиваемый класс.

Составной первичный ключ, также называемый составным ключом, представляет собой комбинацию из двух или более столбцов для формирования первичного ключа таблицы.

@IdClass

Допустим, у нас есть таблица с именем Account, и она имеет два столбца - accountNumber и accountType, которые формируют составной ключ.

Чтобы обозначить оба этих поля как части составного ключа мы должны создать класс, например, ComplexKey с этими полями.

Затем нам нужно аннотировать сущность Account аннотацией @IdClass(ComplexKey.class). Мы также должны объявить поля из класса ComplexKey в сущности Account с такими же именами и аннотировать их с помощью @Id.

## Как сделать составной ключ?

```
public class ComplexKey implements Serializable {
    private String accountNumber;
    private String accountType;
    // default constructor
    public AccountId(String accountNumber, String accountType) {
        this.accountNumber = accountNumber;
        this.accountType = accountType;
    }
    // equals() and hashCode()
}
```

```
@Entity
@IdClass(ComplexKey.class)
public class Account {
    @Id
    private String accountNumber;
    @Id
    private String accountType;
    // other fields, getters and setters
}
```

@EmbeddedId

Рассмотрим пример, в котором мы должны сохранить некоторую информацию о книге с заголовком и языком в качестве полей первичного ключа. В этом случае класс первичного ключа, BookId, должен быть аннотирован @Embeddable. Затем нам нужно встроить этот класс в сущность Book, используя @EmbeddedId.

@Embeddable

```
public class BookId implements Serializable {
    private String title;
    private String language;
    // default constructor
    public BookId(String title, String language) {
        this.title = title;
        this.language = language;
    }
    // getters, equals() and hashCode() methods
}
```

@Entity

```
public class Book {
    @EmbeddedId
    private BookId bookId;
    // constructors, other fields, getters and setters
}
```



<p>Для чего нужна аннотация ID? Какие @GeneratedValue вы знаете?</p>	<p>Аннотация @Id определяет простой (не составной) первичный ключ, состоящий из одного поля. В соответствии с JPA, допустимые типы атрибутов для первичного ключа:</p> <ol style="list-style-type: none"> <li>1. примитивные типы и их обертки;</li> <li>2. строки;</li> <li>3. BigDecimal и BigInteger;</li> <li>4. java.util.Date и java.sql.Date.</li> </ol> <p>Если мы хотим, чтобы значение первичного ключа генерировалось для нас автоматически, мы можем добавить первичному ключу, отмеченному аннотацией @Id, аннотацию @GeneratedValue.</p> <p>Возможно 4 варианта:</p> <p><b>AUTO(default)</b> - Указывает, что Hibernate должен выбрать подходящую стратегию для конкретной базы данных, учитывая её диалект, так как у разных БД разные способы по умолчанию. Поведение по умолчанию - исходить из типа поля идентификатора.</p> <p><b>IDENTITY</b> - для генерации значения первичного ключа будет использоваться столбец IDENTITY, имеющийся в базе данных. Значения в столбце автоматически увеличиваются вне текущей выполняемой транзакции(на стороне базы, так что этого столбца мы не увидим), что позволяет базе данных генерировать новое значение при каждой операции вставки. В промежутках транзакций сущность будет сохранена.</p> <p><b>SEQUENCE</b> - тип генерации, рекомендуемый документацией Hibernate. Для получения значений первичного ключа Hibernate должен использовать имеющиеся в базе данных механизмы генерации последовательных значений (Sequence). В бд можно будет увидеть дополнительную таблицу. Но если наша БД не поддерживает тип SEQUENCE, то Hibernate автоматически переключится на тип TABLE. В промежутках транзакций сущность не будет сохранена, так как хибер возьмет из таблицы id hibernate-sequence и вернётся обратно в приложение.</p> <p><b>SEQUENCE</b> - это объект базы данных, который генерирует инкрементные целые числа при каждом последующем запросе.</p> <p><b>TABLE</b> - Hibernate должен получать первичные ключи для сущностей из создаваемой для этих целей таблицы, способной содержать именованные сегменты значений для любого количества сущностей. Требует использования пессимистических блокировок, которые помещают все транзакции в последовательный порядок и замедляет работу приложения.</p>
<p>Расскажите про аннотации @JoinColumn и @JoinTable? Где и для чего они используются?</p>	<p><b>@JoinColumn</b> используется для указания столбца FOREIGN KEY, используемого при установлении связей между сущностями или коллекциями. Мы помним, что только сущность-владелец связи может иметь внешние ключи от другой сущности (владеемой). Однако, мы можем указать @JoinColumn как во владеемой таблице, так и во владеющей, но столбец с внешними ключами всё равно появится во владеющей таблице.</p> <p>Особенности использования:</p> <ul style="list-style-type: none"> <li>❖ @OneToOne: означает, что появится столбец в таблице сущности-владельца связи, который будет содержать внешний ключ, ссылающийся на первичный ключ владеемой сущности.</li> <li>❖ @OneToMany/@ManyToOne: если не указать на владеемой стороне связи атрибут mappedBy, создается joinTable с ключами обеих таблиц. Но при этом же у владельца создается столбец с внешними ключами.</li> </ul> <p><b>@JoinColumns</b> используется для группировки нескольких аннотаций @JoinColumn, которые используются при установлении связей между сущностями или коллекциями, у которых составной первичный ключ и требуется несколько колонок для указания внешнего ключа. В каждой аннотации @JoinColumn должны быть указаны элементы name и referencedColumnName.</p> <p><b>@JoinTable</b> используется для указания связывающей (сводной, третьей) таблицы между двумя другими таблицами.</p>
	<p><b>@OrderBy</b> указывает порядок, в соответствии с которым должны располагаться элементы коллекций сущностей, базовых или встраиваемых типов при их извлечении из БД. Если в кэше есть нужные данные, то сортировки не будет. Так как @OrderBy просто добавляет к sql-запросу Order By, а при получении данных из кэша, обращения к бд нет. Эта аннотация может использоваться с аннотациями @ElementCollection, @OneToMany, @ManyToMany.</p> <p>При использовании с коллекциями базовых типов, которые имеют аннотацию @ElementCollection, элементы этой коллекции будут отсортированы в натуральном порядке, по значению базовых типов.</p>



<p>Для чего нужны аннотации <code>@OrderBy</code> и <code>@OrderColumn</code>, чем они отличаются?</p>	<p>Если это коллекция встраиваемых типов (<code>@Embeddable</code>), то используя точку (".") мы можем сослаться на атрибут внутри встроеного атрибута. Если это коллекция сущностей, то у аннотации <code>@OrderBy</code> можно указать имя поля сущности, по которому сортировать эти самые сущности: Если мы не укажем у <code>@OrderBy</code> параметр, то сущности будут упорядочены по первичному ключу. В случае с сущностями доступ к полю по точке не работает. Попытка использовать вложенное свойство, например <code>@OrderBy ("supervisor.name")</code> повлечет <code>Runtime Exception</code>.</p> <p><b>@OrderColumn</b> создает в таблице столбец с индексами порядка элементов, который используется для поддержания постоянного порядка в списке, но этот столбец не считается частью состояния сущности или встраиваемого класса. Hibernate отвечает за поддержание порядка как в базе данных при помощи столбца, так и при получении сущностей и элементов из БД. Hibernate отвечает за обновление порядка при записи в базу данных, чтобы отразить любое добавление, удаление или иное изменение порядка, влияющее на список в таблице.</p> <p><b>@OrderBy vs @OrderColumn</b> Порядок, указанный в <code>@OrderBy</code>, применяется только в рантайме при выполнении запроса к БД, То есть в контексте персистентности, в то время как при использовании <code>@OrderColumn</code>, порядок сохраняется в отдельном столбце таблицы и поддерживается при каждой вставке/обновлении/удалении элементов.</p>
<p>Для чего нужна аннотация <code>Transient</code>?</p>	<p><b>@Transient</b> используется для объявления того, какие поля у сущности, встраиваемого класса или <code>Mapped SuperClass</code> <b>не будут сохранены в базе данных</b>. Persistent fields (постоянные поля) - это поля, значения которых будут по умолчанию сохранены в БД. Ими являются любые не static и не final поля. Transient fields (временные поля):</p> <ul style="list-style-type: none"> <li>❖ static и final поля сущностей;</li> <li>❖ иные поля, объявленные явно с использованием Java-модификатора <code>transient</code>, либо JPA-аннотации <code>@Transient</code>.</li> </ul>
<p>Какие шесть видов блокировок (lock)</p>	<p>В порядке от самого ненадежного и быстрого, до самого надежного и медленного:</p> <ol style="list-style-type: none"> <li>1. <b>NONE</b> — без блокировки.</li> <li>2. <b>OPTIMISTIC</b> (синоним <code>READ</code> в JPA 1) — оптимистическая блокировка, которая работает, как описано ниже: если при завершении транзакции кто-то извне изменит поле <code>@Version</code>, то будет сделан <code>RollBack</code> транзакции и будет выброшено <code>OptimisticLockException</code>.</li> <li>3. <b>OPTIMISTIC_FORCE_INCREMENT</b> (синоним <code>WRITE</code> в JPA 1) — работает по тому же алгоритму, что и <code>LockModeType.OPTIMISTIC</code> за тем исключением, что после <code>commit</code> значение поле <code>Version</code> принудительно увеличивается на 1. В итоге окончательно после каждого коммита поле увеличится на 2(увеличение, которое можно увидеть в <code>Post-Update</code> + принудительное увеличение).</li> <li>4. <b>PESSIMISTIC_READ</b> — данные блокируются в момент чтения и это гарантирует, что никто в ходе выполнения транзакции не сможет их изменить. Остальные транзакции, тем не менее, смогут параллельно читать эти данные. Использование этой блокировки может вызывать долгое ожидание блокировки или даже выкидывание <code>PessimisticLockException</code>.</li> <li>5. <b>PESSIMISTIC_WRITE</b> — данные блокируются в момент записи и никто с момента захвата блокировки <b>не может в них писать и не может их читать</b> до окончания транзакции, владеющей блокировкой. Использование этой блокировки может вызывать долгое ожидание блокировки.</li> <li>6. <b>PESSIMISTIC_FORCE_INCREMENT</b> — ведёт себя как <code>PESSIMISTIC_WRITE</code>, но в конце транзакции увеличивает значение поля <code>@Version</code>, даже если фактически сущность не изменилась.</li> </ol>

Блокировки (lock) описаны в спецификации JPA (или какие есть значения у enum LockModeType в JPA)?

**Оптимистичное блокирование** - подход предполагает, что параллельно выполняющиеся транзакции редко обращаются к одним и тем же данным и позволяет им свободно выполнять любые чтения и обновления данных. Но при окончании транзакции производится проверка, изменились ли данные в ходе выполнения данной транзакции и, если да, транзакция обрывается и выбрасывается OptimisticLockException. Оптимистичное блокирование в JPA реализовано путём внедрения в сущность специального поля версии:

```
@Version
private long version;
```

Поле, аннотирование @Version, может быть целочисленным или временным. При завершении транзакции, если сущность была заблокирована оптимистично, будет проверено, не изменилось ли значение @Version кем-либо ещё, после того как данные были прочитаны, и, если изменилось, будет выкинуто OptimisticLockException. Использование этого поля позволяет отказаться от блокировок на уровне базы данных и сделать всё на уровне JPA, улучшая уровень конкурентности.

Позволяет отказаться от блокировок на уровне БД и делать всё с JPA.

**Пессимистичное блокирование** - подход напротив, ориентирован на транзакции, которые часто конкурируют за одни и те же данные и поэтому блокирует доступ к данным в тот момент когда читает их. Другие транзакции останавливаются, когда пытаются обратиться к заблокированным данным и ждут снятия блокировки (или кидают исключение). Пессимистичное блокирование выполняется на уровне базы и поэтому не требует вмешательства в код сущности.

Блокировки ставятся путём вызова метода lock() у EntityManager, в который передаётся сущность, требующая блокировки и уровень блокировки:

```
EntityManager em = entityManagerFactory.createEntityManager();
em.lock(company1, LockModeType.OPTIMISTIC);
```

1. first-level cache (кэш первого уровня) — кэширует данные одной транзакции;
2. second-level cache (кэш второго уровня) — кэширует данные транзакций от одной фабрики сессий. Провайдер JPA может, но не обязан реализовывать работу с кэшем второго уровня.

**Кэш первого уровня** — это кэш сессии (Session), который является обязательным, это и есть PersistenceContext. Через него проходят все запросы. В том случае, если мы выполняем несколько обновлений объекта, Hibernate старается отсрочить (насколько это возможно) обновление этого объекта для того, чтобы сократить количество выполненных запросов в БД. Например, при пяти истребованиях одного и того же объекта из БД в рамках одного persistence context, запрос в БД будет выполнен один раз, а остальные четыре загрузки будут выполнены из кэша. Если мы закроем сессию, то все объекты, находящиеся в кэше, теряются, а далее — либо сохраняются в БД, либо обновляются.

Особенности кэша первого уровня:

- ❖ включен по умолчанию, его нельзя отключить;
- ❖ связан с сессией (контекстом персистентности), то есть разные сессии видят только объекты из своего кэша, и не видят объекты, находящиеся в кэшах других сессий;
- ❖ при закрытии сессии PersistenceContext очищается - кэшированные объекты, находившиеся в нем, удаляются;
- ❖ при первом запросе сущности из БД, она загружается в кэш, связанный с этой сессией;
- ❖ если в рамках этой же сессии мы снова запросим эту же сущность из БД, то она будет загружена из кэша, и никакого второго SQL-запроса в БД сделано не будет;
- ❖ сущность можно удалить из кэша сессии методом evict(), после чего следующая попытка получить эту же сущность повлечет обращение к базе данных;
- ❖ метод clear() очищает весь кэш сессии.

Какие два вида кэшей (cache) вы знаете в JPA и для чего они нужны?

	<p>Если кэш первого уровня привязан к объекту сессии, то <b>кэш второго уровня</b> привязан к объекту-фабрике сессий (Session Factory object) и, следовательно, кэш второго уровня доступен одновременно в нескольких сессиях или контекстах персистентности. Кэш второго уровня требует некоторой настройки и поэтому не включен по умолчанию. Настройка кэша заключается в конфигурировании реализации кэша и разрешения сущностям быть закэшированными.</p> <p>Hibernate не реализует сам никакого in-memory cache, а использует существующие реализации кэшей.</p>
<p><b>Как работать с кешем 2 уровня?</b></p>	<p>Чтение из кэша второго уровня происходит только в том случае, если нужный объект не был найден в кэше первого уровня. Hibernate поставляется со встроенной поддержкой стандарта кэширования Java JCache, а также двух популярных библиотек кэширования: Ehcache и Infinispan.</p> <p>В Hibernate кэширование второго уровня реализовано в виде абстракции, то есть мы должны предоставить любую её реализацию, вот несколько провайдеров: Ehcache, OSCache, SwarmCache, JBoss TreeCache. Для Hibernate требуется только реализация интерфейса org.hibernate.cache.spi.RegionFactory, который инкапсулирует все детали, относящиеся к конкретным провайдерам. По сути, RegionFactory действует как мост между Hibernate и поставщиками кэша. В примерах будем использовать Ehcache. Что нужно сделать: ♦ добавить мавен-зависимость кэш-провайдера нужной версии ♦ включить кэш второго уровня и определить конкретного провайдера hibernate.cache.use_second_level_cache=true hibernate.cache.region.factory_class=org.hibernate.cache.ehcache.EhCacheRegionFactory ♦ установить у нужных сущностей JPA-аннотацию @Cacheable, обозначающую, что сущность нужно кэшировать, и Hibernate-аннотацию @Cache, настраивающую детали кэширования, у которой в качестве параметра указать стратегию параллельного доступа" на "Чтение из кэша второго уровня происходит только в том случае, если нужный объект не был найден в кэше первого уровня. Hibernate поставляется со встроенной поддержкой стандарта кэширования Java JCache, а также двух популярных библиотек кэширования: Ehcache и Infinispan.</p> <p>Стратегии параллельного доступа к объектам Проблема заключается в том, что кэш второго уровня доступен из нескольких сессий сразу и несколько потоков программы могут одновременно в разных транзакциях работать с одним и тем же объектом. Следовательно надо как-то обеспечивать их одинаковым представлением этого объекта. ♦ READ_ONLY: Используется только для сущностей, которые никогда не изменяются (будет выброшено исключение, если попытаться обновить такую сущность). Очень просто и производительно. Подходит для некоторых статических данных, которые не меняются. ♦ NONSTRICT_READ_WRITE: Кэш обновляется после совершения транзакции, которая изменила данные в БД и закоммитила их. Таким образом, строгая согласованность не гарантируется, и существует небольшое временное окно между обновлением данных в БД и обновлением тех же данных в кэше, во время которого параллельная транзакция может получить из кэша устаревшие данные. ♦ READ_WRITE: Эта стратегия гарантирует строгую согласованность, которую она достигает, используя «мягкие» блокировки: когда обновляется кэшированная сущность, на нее накладывается мягкая блокировка, которая снимается после коммита транзакции. Все параллельные транзакции, которые пытаются получить доступ к записям в кэше с наложенной мягкой блокировкой, не смогут их прочитать или записать и отправят запрос в БД. Ehcache использует эту стратегию по умолчанию. ♦ TRANSACTIONAL: полноценное разделение транзакций. Каждая сессия и каждая транзакция видят объекты, словно они работали с ними последовательно одна транзакция за другой. Плата за это — блокировки и потеря производительности</p>
<p><b>Что такое JPQL/HQL и чем он отличается от SQL?</b></p>	<p>Hibernate Query Language (HQL) и Java Persistence Query Language (JPQL) - оба являются объектно-ориентированными языками запросов, схожими по природе с SQL. JPQL - это подмножество HQL.</p> <p>JPQL - это язык запросов, практически такой же как SQL, однако, вместо имен и колонок таблиц базы данных, он использует имена классов Entity и их атрибуты. В качестве параметров запросов также используются типы данных атрибутов Entity, а не полей баз данных. В отличие от SQL в JPQL есть автоматический полиморфизм, то есть каждый запрос к Entity возвращает не только объекты этого Entity, но также объекты всех его классов-потомков, независимо от стратегии наследования. В JPA запрос представлен в виде javax.persistence.Query или javax.persistence.TypedQuery, полученных из EntityManager.</p> <p>В Hibernate HQL-запрос представлен org.hibernate.query.Query, полученный из Session. Если HQL является именованным запросом, то будет использоваться Session#getNamedQuery, в противном случае требуется Session#createQuery.</p>

**Что такое Criteria API  
и для чего он  
используется?**

Начиная с версии 5.2 Hibernate Criteria API объявлен deprecated. Вместо него рекомендуется использовать JPA Criteria API.  
JPA Criteria API - это актуальный API, используемый только для выборки(select) сущностей из БД в более объектно-ориентированном стиле.  
Основные преимущества JPA Criteria API:

- ❖ ошибки могут быть обнаружены во время компиляции;
- ❖ позволяет динамически формировать запросы на этапе выполнения приложения.

Основные недостатки:

- ❖ нет контроля над запросом, сложно отловить ошибку
- ❖ влияет на производительность, множество классов

Для динамических запросов - фрагменты кода создаются во время выполнения - JPA Criteria API является предпочтительней.

Вот некоторые области применения Criteria API:

Criteria API поддерживает проекцию, которую мы можем использовать для агрегатных функций вроде sum(), min(), max() и т.д.

Criteria API может использовать ProjectionList для извлечения данных только из выбранных колонок.

Criteria API может быть использована для join запросов с помощью соединения нескольких таблиц, используя методы createAlias(), setFetchMode() и setProjection().

Criteria API поддерживает выборку результатов согласно условиям (ограничениям). Для этого используется метод add() с помощью которого добавляются ограничения (Restrictions).

Criteria API позволяет добавлять порядок (сортировку) к результату с помощью метода addOrder().

Проблема N+1 запросов возникает, когда получение данных из БД выполняется за N дополнительных SQL-запросов для извлечения тех же данных, которые могли быть получены при выполнении основного SQL-запроса.

**1. JOIN FETCH**

И при FetchType.EAGER и при FetchType.LAZY нам поможет JPQL-запрос с JOIN FETCH. Опцию «FETCH» можно использовать в JOIN (INNER JOIN или LEFT JOIN) для выборки связанных объектов в одном запросе вместо дополнительных запросов для каждого доступа к ленивым полям объекта. Лучший вариант решения для простых запросов (1-3 уровня вложенности связанных объектов).

```
select pc
  from PostComment pc
 join fetch pc.post p
```

**2. EntityGraph**

В случаях, когда нам нужно получить по-настоящему много данных, и у нас jpql запрос - лучше всего использовать EntityGraph.

**3. @Fetch(FetchMode.SUBSELECT)**

Аннотация Hibernate. **Можно использовать только с коллекциями.** Будет сделан один sql-запрос для получения корневых сущностей и, если в контексте персистентности будет обращение к ленивым полям-коллекциям, то выполнится еще один запрос для получения связанных коллекций:

```
@Fetch(value = FetchMode.SUBSELECT)
private Set<Order> orders = new HashSet<>();
```

**Расскажите про  
проблему N+1 Select  
и путях ее решения.**

#### 4. **Batch fetching**

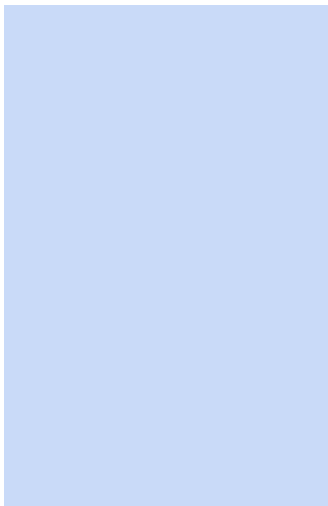
Это Аннотация Hibernate, в JPA её нет. Указывается над классом сущности или над полем коллекции с ленивой загрузкой. Будет сделан один sql-запрос для получения корневых сущностей и, если в контексте персистентности будет обращение к ленивым полям-коллекциям, то выполнится еще один запрос для получения связанных коллекций. Количество загружаемых сущностей указывается в аннотации.

```
@BatchSize(size=5)
```

```
private Set<Order> orders = new HashSet<>();
```

#### 5. **HibernateSpecificMapping, SqlResultSetMapping**

Для нативных запросов рекомендуется использовать именно их.



## Spring

Что такое инверсия контроля (IoC) и внедрение зависимостей (DI)? Как эти принципы реализованы в Spring?	<p><b>Inversion of Control</b> - подход, который позволяет конфигурировать и управлять объектами Java с помощью рефлексии. Вместо ручного внедрения зависимостей, фреймворк забирает ответственность за это посредством IoC-контейнера. Контейнер отвечает за управление жизненным циклом объекта: создание объектов, вызов методов инициализации и конфигурирование объектов путём связывания их между собой. Объекты, создаваемые контейнером, называются beans. Конфигурирование контейнера осуществляется путём внедрения аннотаций, но также, есть возможность, по старинке, загрузить XML-файлы, содержащие определение bean'ов и предоставляющие информацию, необходимую для создания bean'ов.</p> <p><b>Dependency Injection</b> — является одним из способов реализации принципа IoC в Spring. Это шаблон проектирования, в котором контейнер передает экземпляры объектов по их типу другим объектам с помощью конструктора или метода класса(setter), что позволяет писать слабосвязный код.</p>		
Что такое IoC контейнер?	<p><b>Inversion of Control</b> - подход, который позволяет конфигурировать и управлять объектами Java с помощью рефлексии. Вместо ручного внедрения зависимостей, фреймворк забирает ответственность за это посредством IoC-контейнера. Контейнер отвечает за управление жизненным циклом объекта: создание объектов, вызов методов инициализации и конфигурирование объектов путём связывания их между собой. Объекты, создаваемые контейнером, называются beans. Конфигурирование контейнера осуществляется путём внедрения аннотаций, но также, есть возможность, по старинке, загрузить XML-файлы, содержащие определение bean'ов и предоставляющие информацию, необходимую для создания bean'ов.</p> <p><b>Dependency Injection</b> — является одним из способов реализации принципа IoC в Spring. Это шаблон проектирования, в котором контейнер передает экземпляры объектов по их типу другим объектам с помощью конструктора или метода класса(setter), что позволяет писать слабосвязный код.</p>		
Расскажите про ApplicationContext и BeanFactory, чем отличаются? В каких случаях что стоит использовать?	Функционал	BeanFactory	ApplicationContext
	Инициализация/автоматическое связывание бина	Да	Да
	Автоматическая регистрация BeanPostProcessor	Нет	Да
	Автоматическая регистрация BeanFactoryPostProcessor	Нет	Да
	Удобный гоступ к MessageSource (для i18n)	Нет	Да
	ApplicationEvent публикация	Нет	Да
<p><b>ApplicationContext</b> является наследником BeanFactory и полностью реализует его функционал, добавляя больше специфических enterprise-функций. Может работать с бинами всех скоупов.</p> <p><b>BeanFactory</b> - это фактический контейнер, который создает, настраивает и управляет рядом bean-компонентов. Эти бины обычно взаимодействуют друг с другом и, таким образом, имеют зависимости между собой. Эти зависимости отражены в данных конфигурации, используемых BeanFactory. Может работать с бинами singleton и prototype.</p> <p>BeanFactory обычно используется тогда, когда ресурсы ограничены (мобильные устройства), так как он легче по сравнению с ApplicationContext. Поэтому, если ресурсы не сильно ограничены, то лучше использовать ApplicationContext.</p> <p><b>ApplicationContext загружает все бины при запуске, а BeanFactory по требованию.</b></p>			
Расскажите про аннотацию @Bean?	<p><b>Аннотация @Bean используется для указания того, что метод создает, настраивает и инициализирует новый объект, управляемый IoC-контейнером.</b> Такие методы можно использовать как в классах с аннотацией @Configuration, так и в классах с аннотацией @Component(или её наследниках).</p> <p>Имеет следующие свойства:</p> <p><b>destroyMethod, initMethod</b> — варианты переопределения методов инициализации и удаления бина, указав их имена в аннотации.</p> <p><b>name</b> — имя бина. По умолчанию именем бина является имя метода.</p> <p><b>value</b> — алиас для name()</p>		
Расскажите про аннотацию @Component?	@Component - используется для указания класса в качестве компонента spring. Такой класс будет сконфигурирован как spring Bean.		
Чем отличаются аннотации @Bean и @Component?	@Bean - ставится над методом и позволяет добавить bean, уже реализованного сторонней библиотекой класса, в контейнер, а @Component используется для указания класса, написанного программистом.		
Расскажите про аннотации @Service и @Repository. Чем они отличаются?	<p>@Repository - указывает, что класс используется для работы с поиском, получением и хранением данных. Аннотация может использоваться для реализации шаблона DAO.</p> <p>@Service - указывает, что класс является сервисом для реализации бизнес-логики.</p> <p>@Repository, @Service, @Controller и @Configuration являются алиасами @Component, их также называют стереотипными аннотациями.</p> <p>Задача @Repository заключается в том, чтобы отлавливать определенные исключения персистентности и пробрасывать их как одно непроверенное исключение Spring Framework. Для этого в контекст должен быть добавлен класс PersistenceExceptionTranslationPostProcessor.</p>		
Расскажите про аннотацию @Autowired	<p><b>@Autowired</b> — автоматическое внедрение подходящего бина:</p> <ol style="list-style-type: none"><li>1) Контейнер определяет тип объекта для внедрения</li><li>2) Контейнер ищет соответствующий тип бина в контексте(он же контейнер)</li><li>3) Если есть несколько кандидатов, и один из них помечен как @Primary, то внедряется он</li><li>4) Если используется @Qualifier, то контейнер будет использовать информацию из @Qualifier, чтобы понять, какой компонент внедрять</li><li>5) В противном случае контейнер внедрит бин, основываясь на его имени или ID</li><li>6) Если ни один из способов не сработал, то будет выброшено исключение</li></ol> <p>Контейнер обрабатывает DI с помощью AutowiredAnnotationBeanPostProcessor. В связи с этим, аннотация не может быть использована ни в одном BeanFactoryPP или BeanPP.</p> <p>В аннотации есть один параметр <b>required = true/false</b> - указывает, обязательно ли делать DI. По умолчанию true. Либо можно не выбрасывать исключение, а оставить поле с null, если нужный бин не был найден - false.</p> <p>При циклической зависимости, когда объекты ссылаются друг на друга, нельзя ставить над конструктором.</p>		



Расскажите про аннотацию @Resource	<p>@Resource(аннотация java) пытается получить зависимость: по имени, по типу, затем по описанию (Qualifier). Имя извлекается из имени аннотируемого сеттера или поля, либо берется из параметра name.</p> <p><b>@Resource //По умолчанию поиск бина с именем "context"</b></p> <pre>private ApplicationContext context;</pre> <p><b>@Resource(name="greetingService") //Поиск бина с именем "greetingService"</b></p> <pre>public void setGreetingService(GreetingService service) {     this.greetingService = service; }</pre> <p>Разница с @Autowired:</p> <ul style="list-style-type: none"><li>❖ ищет бин сначала по имени, а потом по типу;</li><li>❖ не нужна дополнительная аннотация для указания имени конкретного бина;</li><li>❖ @Autowired позволяет отметить место вставки бина как необязательное @Autowired(required = false);</li><li>❖ при замене Spring Framework на другой фреймворк, менять аннотацию @Resource не нужно.</li></ul>
Расскажите про аннотацию @Inject	<p>@Inject входит в пакет javax.inject и, чтобы её использовать, нужно добавить зависимость:</p> <pre>&lt;dependency&gt;   &lt;groupId&gt;javax.inject&lt;/groupId&gt;   &lt;artifactId&gt;javax.inject&lt;/artifactId&gt;   &lt;version&gt;1&lt;/version&gt; &lt;/dependency&gt;</pre> <p>@Inject (аннотация java) аналог @Autowired (аннотация spring) в первую очередь пытается подключить зависимость по типу, затем по описанию и только потом по имени. В ней нет параметров. Поэтому при использовании конкретного имени (Id) бина используем @Named:</p> <pre>@Inject @Named("yetAnotherFieldInjectDependency") private ArbitraryDependency yetAnotherFieldInjectDependency;</pre>
Расскажите про аннотацию @Lookup	<p>Обычно бины в приложении Spring являются синглтонами, и для внедрения зависимостей мы используем конструктор или сеттер. Но бывает и другая ситуация: имеется бин Car – синглтон (singleton bean), и ему требуется каждый раз новый экземпляр бина Passenger. То есть Car – синглтон, а Passenger – так называемый прототипный бин (prototype bean). Жизненные циклы бинов разные. Бин Car создается контейнером только раз, а бин Passenger создается каждый раз новый – допустим, это происходит каждый раз при вызове какого-то метода бина Car. Вот здесь то и пригодится внедрение бина с помощью Lookup метода. Оно происходит не при инициализации контейнера, а позднее: каждый раз, когда вызывается метод. Суть в том, что вы создаете метод-заглушку в бине Car и помечаете его специальным образом – аннотацией @Lookup. Этот метод должен возвращать бин Passenger, каждый раз новый. Контейнер Spring под капотом создаст подкласс и переопределит этот метод и будет вам выдавать новый экземпляр бина Passenger при каждом вызове аннотированного метода. Даже если в вашей заглушке он возвращает null (а так и надо делать, все равно этот метод будет переопределен).</p>
Можно ли вставить бин в статическое поле? Почему?	<p>Spring не позволяет внедрять бины напрямую в статические поля. Это связано с тем, что когда загрузчик классов загружает статические значения, контекст Spring ещё не загружен. Чтобы исправить это, создайте нестатический сеттер-метод с @Autowired:</p> <pre>private static OrderItemService orderItemService;  @Autowired public void setOrderItemService(OrderItemService orderItemService) {     TestDataInit.orderItemService = orderItemService; }</pre>
	<p><b>@Qualifier</b> применяется если кандидатов для автоматического связывания несколько, она позволяет указать в качестве аргумента имя конкретного бина, который следует внедрить. Она может быть применена к отдельному полю класса, к отдельному аргументу метода или конструктора:</p>

Расскажите про аннотации @Primary и @Qualifier	<pre>public class AutowiredClass {      @Autowired //к полям класса     @Qualifier("main")     private GreetingService greetingService;      @Autowired //к отдельному аргументу конструктора или метода     public void prepare(@Qualifier("main") GreetingService greetingService){         /* что-то делаем... */     }; }</pre> <p>Соответственно, у одной из реализации GreetingService должна быть установлена соответствующая аннотация @Qualifier:</p> <pre>@Component @Qualifier("main") public class GreetingServiceImpl implements GreetingService {     //... }</pre> <p><b>@Primary</b> тоже используется, чтобы отдавать предпочтение бину, когда есть несколько бинов одного типа, но в ней нельзя задать имя бина, она определяет значение по умолчанию, в то время как @Qualifier более специфичен. Если присутствуют аннотации @Qualifier и @Primary, то аннотация <b>@Qualifier будет иметь приоритет</b>.</p>
Как заинжектировать примитив?	<p>Для этого можно использовать аннотацию <b>@Value</b>. Можно ставить над полем, конструктором, методом. Такие значения можно получать из property файлов, из бинов, и т.п.</p> <pre>@Value("\${some.key}") public String stringWithDefaultValue;</pre> <p>В эту переменную будет внедрена строка, например из property или из view. Кроме того, для внедрения значений мы можем использовать язык SpEL (Spring Expression Language)</p>
Как заинжектировать коллекцию?	<p>Если внедряемый объект массив, коллекция, или map с дженериком, то используя аннотацию @Autowired, Spring внедрит все бины подходящие по типу в этот массив(или другую структуру данных). В случае с map ключом будет имя бина.</p> <p>Используя аннотацию @Qualifier можно настроить тип искомого бина.</p> <p>Бины могут быть упорядочены, когда они вставляются в списки (не Set или Map) или массивы. Поддерживаются как аннотация @Order, так и интерфейс Ordered.</p>
Расскажите про аннотацию @Conditional	<p>Spring предоставляет возможность на основе вашего алгоритма включить или выключить определение бина или всей конфигурации через <b>@Conditional</b>, в качестве параметра которой указывается класс, реализующий интерфейс <b>Condition</b>, с единственным методом <b>matches(ConditionContext var1, AnnotatedTypeMetadata var2)</b>, возвращающий boolean.</p> <p>Для создания более сложных условий можно использовать классы <b>AnyNestedCondition</b>, <b>AllNestedConditions</b> и <b>NoneNestedConditions</b>.</p> <p>Аннотация @Conditional указывает, что компонент имеет право на регистрацию в контексте только тогда, когда все условия соответствуют.</p> <p>Условия проверяются непосредственно перед тем, как должен быть зарегистрирован BeanDefinition компонента, и они могут помешать регистрации данного BeanDefinition.</p> <p>Поэтому нельзя допускать, чтобы при проверке условий мы взаимодействовали с бинами, которых еще не существует, с их BeanDefinition-ами можно.</p> <p>Для того, чтобы проверить несколько условий, можно передать в @Conditional несколько классов с условиями:</p> <pre>@Conditional(HibernateCondition.class, OurConditionClass.class)</pre> <p><b>Если класс @Configuration помечен как @Conditional, то на все методы @Bean, аннотации @Import и аннотации @ComponentScan, связанные с этим классом, также будут распространяться указанные условия.</b></p>
Расскажите про аннотацию @Profile	<p>Профили - это ключевая особенность Spring Framework, позволяющая нам относить наши бины к разным профилям (логическим группам), например, dev, test, prod.</p> <p>Мы можем активировать разные профили в разных средах, чтобы загрузить только те бины, которые нам нужны.</p> <p>Используя аннотацию @Profile, мы относим бин к конкретному профилю. Её можно применять на уровне класса или метода. Аннотация @Profile принимает в качестве аргумента имя одного или нескольких профилей.</p> <p>Она фактически реализована с помощью гораздо более гибкой аннотации @Conditional.</p> <p>Ее можно ставить на @Configuration и Component классы.</p>
	<b>1) Парсирование конфигурации и создание BeanDefinition</b>

Расскажите про  
жизненный цикл  
бина, аннотации  
@PostConstruct и  
@PreDestroy()

После выхода четвертой версии спринга, у нас появилось четыре способа конфигурирования контекста:

- Xml конфигурация — `ClassPathXmlApplicationContext("context.xml")`
- Конфигурация через аннотации с указанием пакета для сканирования — `AnnotationConfigApplicationContext("package.name")`
- Конфигурация через аннотации с указанием класса (или массива классов) помеченного аннотацией `@Configuration` -`AnnotationConfigApplicationContext(JavaConfig.class)`. Этот способ конфигурации называется — `JavaConfig`.
- Groovy конфигурация — `GenericGroovyApplicationContext("context.groovy")`

Цель первого этапа — это создание всех `BeanDefinition`.

`BeanDefinition` — это специальный интерфейс, через который можно получить доступ к метаданным будущего бина. В зависимости от того, какая у вас конфигурация, будет использоваться тот или иной механизм парсирования конфигурации.

Допустим, что наша конфигурация основана на аннотациях. Если заглянуть внутрь `AnnotationConfigApplicationContext`, то можно увидеть два поля.

```
private final AnnotatedBeanDefinitionReader reader; private final ClassPathBeanDefinitionScanner scanner;
```

`ClassPathBeanDefinitionScanner` сканирует указанный пакет на наличие классов помеченных аннотацией `@Component` (или её алиаса).

Найденные классы парсируются и для них создаются `BeanDefinition`. Чтобы было запущено сканирование, в конфигурации должен быть указан пакет для сканирования `@ComponentScan({"package.name"})`.

`AnnotatedBeanDefinitionReader` работает в несколько этапов.

1. Первый этап — это регистрация всех `@Configuration` для дальнейшего парсирования. Если в конфигурации используются `Conditional`, то будут зарегистрированы только те конфигурации, для которых `Condition` вернет `true`.
2. Второй этап — это регистрация `BeanDefinitionRegistryPostProcessor`, который при помощи класса `ConfigurationClassPostProcessor` парсирует `JavaConfig` и создает `BeanDefinition`. **BeanDefinition** — это объект, который хранит в себе информацию о бине.

Сюда входит: из какого класса бин надо создать, scope, установлена ли ленивая инициализация, нужно ли перед данным бином инициализировать другой, `init` и `destroy` методы, зависимости.

Все полученные `BeanDefinition`'ы складываются в `HashMap`, в которой ключом является имя бина, а объект - сам `BeanDefinition`.

При старте приложения, в `IoC` контейнер попадут бины, которые имеют scope `Singleton` (устанавливается по-умолчанию), остальные же создаются, тогда когда они нужны.

## 2) Настройка созданных BeanDefinition

Есть возможность повлиять на бины до их создания, иначе говоря мы имеем доступ к метаданным класса.

Для этого существует специальный интерфейс **BeanFactoryPostProcessor**, реализовав который, мы получаем доступ к созданным **BeanDefinition** и можем их изменять. В нем один метод.

Метод `postProcessBeanFactory` принимает параметром `ConfigurableListableBeanFactory`. Данная фабрика содержит много полезных методов, в том числе `getBeanDefinitionNames`, через который мы можем получить все `BeanDefinitionNames`,

а уже потом по конкретному имени получить `BeanDefinition` для дальнейшей обработки метаданных.

Разберем одну из родных реализаций интерфейса `BeanFactoryPostProcessor`. Обычно, настройки подключения к базе данных выносятся в отдельный `property` файл, потом при помощи `PropertySourcesPlaceholderConfigurer` они загружаются и делается `inject` этих значений в нужное поле. Так как `inject` делается по ключу, то до создания экземпляра бина нужно заменить этот ключ на само значение из `property` файла.

Эта замена происходит в классе, который реализует интерфейс `BeanFactoryPostProcessor`. Название этого класса — `PropertySourcesPlaceholderConfigurer`. Он должен быть объявлен как `static`

```
@Bean
public static PropertySourcesPlaceholderConfigurer configurer() {
    return new PropertySourcesPlaceholderConfigurer();
}
```

## 3) Создание кастомных FactoryBean

`FactoryBean` — это `generic` интерфейс, которому можно делегировать процесс создания бинов типа .

В те времена, когда конфигурация была исключительно в `xml`, разработчикам был необходим механизм с помощью которого они бы могли управлять процессом создания бинов. Именно для этого и был сделан этот интерфейс.

Создадим фабрику которая будет отвечать за создание всех бинов типа — `Color`.

```
public class ColorFactory implements FactoryBean<Color> {
    @Override
    public Color getObject() throws Exception {
        Random random = new Random();
        Color color = new Color(random.nextInt(255), random.nextInt(255), random.nextInt(255));
        return color;
    }

    @Override
    public Class<?> getObjectType() {
        return Color.class;
    }

    @Override
    public boolean isSingleton() {
        return false;
    }
}
```

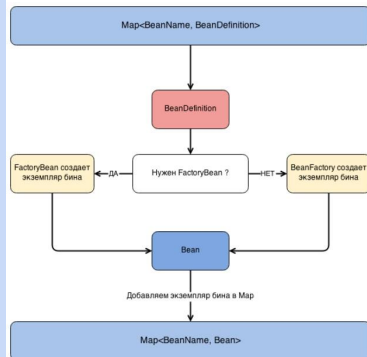
Теперь создание бина типа `Color.class` будет делегироваться `ColorFactory`, у которого при каждом создании нового бина будет вызываться метод `getObject`.

Для тех кто пользуется `JavaConfig`, этот интерфейс будет абсолютно бесполезен.

## 4) Создание экземпляров бинов

Созданием экземпляров бинов занимается `BeanFactory` на основе ранее созданных `BeanDefinition`. Из `Map<BeanName, BeanDefinition>` получаем `Map<BeanName, Bean>`

Создание бинов может делегироваться кастомным `FactoryBean`. О их создании читай выше.



Класс, имплементирующий `BeanPostProcessor`, обязательно должен быть бин, поэтому мы его помечаем аннотацией `@Component`.  
`SCOPE_SINGLETON` — инициализация произойдет один раз на этапе поднятия контекста. `SCOPE_PROTOTYPE` — инициализация будет выполняться каждый раз по запросу. Причем во втором случае ваш бин будет проходить через все `BeanPostProcessor`-ы что может значительно ударить по производительности.

Расскажите про скоупы бинов? Какой скоуп используется по умолчанию? Что изменилось в Spring 5?

Существует 2 области видимости по умолчанию.

**Singleton** - область видимости по умолчанию. В контейнере будет создан только один бин, и все запросы на него будут возвращать один и тот же бин.

**Prototype** - приводит к созданию нового бина каждый раз, когда он запрашивается.

Для бинов со scope "prototype" Spring не вызывает метод `destroy()`, так как не берет на себя контроль полного жизненного цикла этого бина. Spring не хранит такие бины в своём контексте ( контейнере), а отдаёт их клиенту и больше о них не заботится (в отличие от синглтон-бинов).

И 4 области видимости в веб-приложении.

**Request** - Область видимости — 1 HTTP запрос. На каждый запрос создается новый бин

**Session** - Область видимости — 1 сессия. На каждую сессию создается новый бин

**Application** - Область видимости — жизненный цикл `ServletContext`

**WebSocket** - Область видимости — жизненный цикл `WebSocket`

Жизненный цикл web scope полный.

В пятой версии Spring Framework не стало **Global session** scope. И появились `Application` и `WebSocket`

Первый шаг для описания конфигурации Spring это добавление аннотаций — `@Component` или наследников.

Однако, Spring должен знать где искать их. В `@ComponentScan` вы указываете пакеты, которые должны сканироваться. Можно указать массив строк.

Spring будет искать бины и в их подпакетах.

Мы можем расширить это поведение с помощью `includeFilters` и `excludeFilters` параметров в аннотации.

Для `ComponentScan.Filter` доступно пять типов фильтров:

`ANNOTATION`

`ASSIGNABLE_TYPE`

`ASPECTJ`

`REGEX`

`CUSTOM`

Нужно для того, что например, имея какой-то ненужный класс в не нашей библиотеке, мы можем создать для него фильтр, чтобы его бин не инициализировался.

[marcobeher.com/guides/spring-transaction-management-transactional-in-depth](https://marcobeher.com/guides/spring-transaction-management-transactional-in-depth)

Расскажите про аннотацию @ComponentScan

Как спринг работает с транзакциями? Расскажите про аннотацию @Transactional.

**Коротко:** Spring создает прокси для всех классов, помеченных @Transactional (либо если любой из методов класса помечен этой аннотацией), что позволяет вводить транзакционную логику до и после вызываемого метода. При вызове такого метода происходит следующее:

- проху, который создал Spring, создаёт persistence context (или соединение с базой),
- открывает в нём транзакцию и сохраняет всё это в контексте нити исполнения (натурально, в ThreadLocal).
- По мере надобности всё сохранённое достаётся и внедряется в бины.

Таким образом, если в вашем коде есть несколько параллельных нитей, у вас будет и несколько параллельных транзакций, которые будут взаимодействовать друг с другом согласно уровням изоляции.

Что произойдёт, если один метод с @Transactional вызовет другой метод с @Transactional?

Если это происходит в рамках одного сервиса, то второй транзакционный метод будет считаться частью первого, так как вызван у него изнутри, а так как спринг не знает о внутреннем вызове, то не создаст прокси для второго метода.

Что произойдёт, если один метод БЕЗ @Transactional вызовет другой метод с @Transactional?

Так как spring не знает о внутреннем вызове, то не создаст прокси для второго метода.

Будет ли транзакция откатена, если будет брошено исключение, которое указано в контракте метода?

Если в контракте описано это исключение, то она не откатится. Unchecked исключения в транзакционном методе можно ловить, а можно и не ловить. Значения атрибута propagation у аннотации:

**MANDATORY** — всегда используется существующая транзакция и кидается исключение, если текущей транзакции нет.  
**SUPPORTS** — метод с этим правилом будет использовать текущую транзакцию, если она есть, либо будет исполняться без транзакции, если её нет.  
**NOT\_SUPPORTED** — при входе в метод текущая транзакция, если она есть, будет приостановлена и метод будет выполняться без транзакции.  
**NEVER** — явно запрещает исполнение в контексте транзакции. Если при входе в метод будет существовать транзакция, будет выброшено исключение.  
Остальные атрибуты:  
**rollbackFor** = Exception.class - если какой-либо метод выбрасывает указанное исключение, контейнер всегда откатывает текущую транзакцию. По умолчанию отлавливает RuntimeException  
**noRollbackFor** = Exception.class - указание того, что любое исключение, кроме заданных, должно приводить к откату транзакции.  
**rollbackForClassName** и **noRollbackForClassName** - для задания имен исключений в строковом виде.  
**readOnly** - разрешает только операции чтения.  
В свойстве **transactionManager** хранится ссылка на менеджер транзакций, определенный в конфигурации Spring.  
**timeOut** - По умолчанию используется таймаут, установленный по умолчанию для базовой транзакционной системы.  
Сообщает менеджеру tx о продолжительности времени, чтобы дождаться простоя tx, прежде чем принять решение об откате не отвечающих транзакций.  
**isolation** - уровень изолированности транзакций

Подробнее:

Для работы с транзакциями Spring Framework использует AOP-прокси:  
Для включения возможности управления транзакциями нужно разместить аннотацию @EnableTransactionManagement у класса конфигурации @Configuration.  
Она означает, что классы, помеченные @Transactional, должны быть обернуты аспектом транзакций. Отвечает за регистрацию необходимых компонентов Spring, таких как TransactionInterceptor и советы прокси.  
Регистрируемые компоненты помещают перехватчик в стек вызовов при вызове методов @Transactional. Если мы используем Spring Boot и имеем зависимости spring-data-\* или spring-tx, то управление транзакциями будет включено по умолчанию.  
Пропейшн работает только если метод вызывает другой метод в другом сервисе. Если метод вызывает другой метод в этом же сервисе, то используется this и вызов проходит мимо прокси. Это ограничение можно обойти при помощи self-injection.  
Слой логики(Service) - лучшее место для @Transactional.  
Помечая @Transactional класс @Service, то все его методы станут транзакционными. Так, при вызове, например, метода save() произойдет примерно следующее:

1. Вначале мы имеем:
  - ❖ класс **TransactionInterceptor**, у которого вызывается метод **invoke(...)**, внутри которого вызывается метод класса-родителя **TransactionAspectSupport: invokeWithinTransaction(...)**, в рамках которого происходит магия транзакций.
  - ❖ **TransactionManager: решает, создавать ли новый EntityManager и/или транзакцию.**
  - ❖ **EntityManager проху: EntityManager** - это интерфейс, и то, что внедряется в бин в слое DAO на самом деле не является реализацией EntityManager.

В это поле внедряется EntityManager проху, который будет перехватывать обращение к полю EntityManager и делегировать выполнение конкретному EntityManager в рантайме. Обычно EntityManager проху представлен классом SharedEntityManagerInvocationHandler.

2. Transaction Interceptor  
В **TransactionInterceptor** отработает код до работы метода **save()**, в котором будет определено, выполнить ли метод **save()** в пределах уже существующей транзакции БД или должна стартовать новая отдельная транзакция.  
**TransactionInterceptor** сам не содержит логики по принятию решения, решение начать новую транзакцию, если это нужно, делегируется **TransactionManager**.  
Грубо говоря, на данном этапе наш метод будет обернут в try-catch и будет добавлена логика до его вызова и после:

```
try {
    transaction.begin();    // логика до
    service.save();
    transaction.commit();   // логика после
} catch(Exception ex) {
    transaction.rollback();
    throw ex;
}
```

3. TransactionManager  
Менеджер транзакций должен предоставить ответ на два вопроса:

- ❖ Должен ли создаться новый EntityManager?
- ❖ Должна ли стартовать новая транзакция БД?

Решение принимается, основываясь на следующих фактах:

- ❖ выполняется ли хоть одна транзакция в текущий момент или нет;
- ❖ атрибута «propagation» в **@Transactional**.

Если **TransactionManager** решил создать новую транзакцию, тогда:

- ❖ Создается новый EntityManager;
- ❖ EntityManager «привязывается» к текущему потоку (Thread);
- ❖ «Получается» соединение из пула соединений БД;
- ❖ Соединение «привязывается» к текущему потоку.

И EntityManager и это соединение привязываются к текущему потоку, используя переменные **ThreadLocal**.

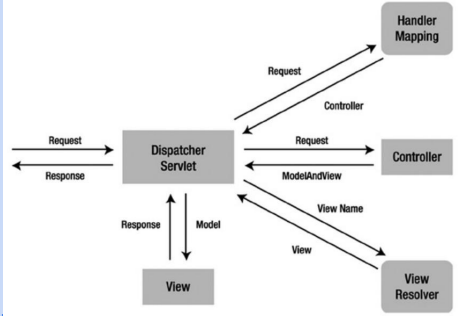
4. EntityManager proxy  
Когда метод **save()** слоя Service делает вызов метода **save()** слоя DAO, внутри которого вызывается, например, **entityManager.persist()**, то не происходит вызов метода **persist()** напрямую у EntityManager, записанного в поле класса DAO.  
Вместо этого метод вызывает EntityManager proxy, который достает текущий EntityManager для нашего потока, и у него вызывается метод **persist()**.

5. Отрабатывает DAO-метод **save()**.

6. TransactionInterceptor  
Отработает код после работы метода **save()**, а именно будет принято решение по коммиту/откату транзакции.

Кроме того, если мы в рамках одного метода сервиса обращаемся не только к методу **save()**, а к разным методам Service и DAO, то все они будут работать в рамках одной транзакции, которая оборачивает этот метод сервиса. Вся работа происходит через прокси-объекты разных классов. Представим, что у нас в классе сервиса только один метод с аннотацией **@Transactional**, а остальные нет. Если мы вызовем метод с **@Transactional**, из которого вызовем метод без **@Transactional**, то оба будут отработаны в рамках прокси и будут обернуты в нашу транзакционную логику. Однако, если мы вызовем метод без **@Transactional**, из которого вызовем метод с **@Transactional**, то они уже не будут работать в рамках прокси и не будут обернуты в нашу транзакционную логику.

Расскажите про аннотации @Controller и @RestController. Чем они отличаются? Как вернуть ответ со своим статусом (например 213)?	<p><b>@Controller</b> - специальный тип класса, обрабатывает HTTP-запросы и часто используется с аннотацией <b>@RequestMapping</b>.</p> <p><b>@RestController</b> ставится на класс-контроллер вместо <b>@Controller</b>. Она указывает, что этот класс оперирует не моделями, а данными. Она состоит из аннотаций <b>@Controller</b> и <b>@ResponseBody</b>. Была введена в Spring 4.0 для упрощения создания RESTful веб-сервисов.</p> <p><b>@ResponseBody</b> сообщает контроллеру, что возвращаемый объект автоматически сериализуется (используя Jackson message converter) в json или xml и передается обратно в объект <b>HttpResponse</b>.</p> <p><b>ResponseEntity</b> используется для формирования кастомизированного HTTP-ответа с пользовательскими параметрами (заголовки, код статуса и тело ответа). Во всех остальных случаях достаточно использовать <b>@ResponseBody</b>. Если мы хотим использовать <b>ResponseEntity</b>, то просто должны вернуть его из метода, Spring позаботится обо всем остальном.</p> <p><b>return ResponseEntity.status(213);</b></p>
Что такое ViewResolver?	<p><b>ViewResolver</b> - распознаватель представлений - это способ работы с представлениями(html-файлы), который поддерживает их распознавание на основе имени, возвращаемого контроллером.</p> <p>Spring Framework поставляется с большим количеством реализаций <b>ViewResolver</b>. Например, класс <b>UriBasedViewResolver</b> поддерживает прямое преобразование логических имен в URL.</p> <p><b>InternalResourceViewResolver</b> — реализация <b>ViewResolver</b> по умолчанию, которая позволяет находить представления, которые возвращает контроллер для последующего перехода к ним. Ищет по заданному пути, префиксу, суффиксу и имени.</p> <p>Любым реализациям <b>ViewResolver</b> желательно поддерживать интернационализацию, то есть множество языков.</p> <p>Существует также несколько реализаций для интеграции с различными технологиями представлений, такими как <b>FreeMarker</b> (<b>FreeMarkerViewResolver</b>), <b>Velocity</b> (<b>VelocityViewResolver</b>) и <b>JasperReports</b> (<b>JasperReportsViewResolver</b>).</p>

<b>Чем отличаются Model, ModelMap и ModelAndView?</b>	<p><b>Model - интерфейс</b>, представляет коллекцию пар ключ-значение Map&lt;String, Object&gt;. Содержимое модели используется для отображения данных во View. Например, если View выводит информацию об объекте Customer, то она может ссылаться к ключам модели, например customerName, customerPhone, и получать значения для этих ключей. Объекты-значения из модели также могут содержать бизнес-логику.</p> <p><b>ModelMap - класс, наследуется от LinkedHashMap</b>, тоже используется для передачи значений для визуализации представления. Преимущество ModelMap заключается в том, что он дает нам возможность передавать коллекцию значений и обрабатывать эти значения, как если бы они были внутри Map.</p> <p><b>ModelAndView</b> - это просто контейнер для ModelMap, объект View и HttpStatus. Это позволяет контроллеру возвращать все значения как одно.</p> <p><b>View</b> используется для отображения данных приложения пользователю.</p> <p>Spring MVC поддерживает несколько поставщиков View(они называются шаблонизаторы) — JSP, JSF, Thymeleaf, и т.п.</p> <p>Интерфейс View преобразует объекты в обычные сервлеты.</p>
<b>Расскажите про паттерн Front Controller, как он реализован в Spring?</b>	<p>Front controller - обеспечивает единую точку входа для всех входящих запросов. Все запросы обрабатываются одним обработчиком – DispatcherServlet с маппингом “/”. Этот обработчик может выполнить аутентификацию, авторизацию, регистрацию или отслеживание запроса, а затем распределяет их между контроллерами, обрабатывающими разные URL. Это и есть реализация паттерна Front Controller.</p> <p>Веб-приложение может определять любое количество DispatcherServlet-ов. Каждый из них будет работать в своем собственном пространстве имен, загружая свой собственный дочерний WebApplicationContext с вьюшками, контроллерами и т.д.</p> <ul style="list-style-type: none"><li>❖ Один из контекстов будет корневым, а все остальные контексты будут дочерними.</li><li>❖ Все дочерние контексты могут получить доступ к бинам, определенным в корневом контексте, но не наоборот.</li><li>❖ Каждый дочерний контекст внутри себя может переопределить бины из корневого контекста.</li></ul> <p><b>WebApplicationContext</b> расширяет ApplicationContext (создаёт и управляет бинами и т.д.), но помимо этого он имеет дополнительный метод getServletContext(), через который у него есть возможность получать доступ к ServletContext-у.</p> <p><b>ContextLoaderListener</b> создает корневой контекст приложения и будет использоваться всеми дочерними контекстами, созданными всеми DispatcherServlet.</p>
<b>Расскажите про паттерн MVC, как он реализован в Spring?</b>	<p>MVC — это шаблон проектирования, делящий программу на 3 вида компонентов:</p> <p><b>Model</b> — модель отвечает за хранение данных.</p> <p><b>View</b> — отвечает за вывод данных на фронтенде.</p> <p><b>Controller</b> — оперирует моделями и отвечает за обмен данными model с view.</p> <p>Основная цель следования принципам MVC — отделить реализацию бизнес-логики приложения (модели) от ее визуализации (view).</p> <p><b>Spring MVC</b> - это веб-фреймворк, основанный на Servlet API, с использованием двух шаблонов проектирования - Front controller и MVC.</p> <p>Spring MVC реализует четкое разделение задач, что позволяет нам легко разрабатывать и тестировать наши приложения. Данные задачи разбиты между разными компонентами: Dispatcher Servlet, Controllers, View Resolvers, Views, Models, ModelAndView, Model and Session Attributes, которые полностью независимы друг от друга, и отвечают только за одно направление. Поэтому MVC дает нам довольно большую гибкость. Он основан на интерфейсах (с предоставленными классами реализации), и мы можем настраивать каждую часть фреймворка с помощью пользовательских интерфейсов.</p> <p><b>Основные интерфейсы для обработки запросов:</b></p> <p>\</p> <p><b>View.</b> Отвечает за возвращение ответа клиенту в виде текстов и изображений. Используются встраиваемые шаблонизаторы (Thymeleaf, FreeMarker и т.д.), так как у Spring нет родных. Некоторые запросы могут идти прямо во View, не заходя в Model, другие проходят через все слои.</p> <p><b>HandlerAdapter.</b> Помогает DispatcherServlet вызвать и выполнить метод для обработки входящего запроса.</p> <p><b>ContextLoaderListener</b> - слушатель при старте и завершении корневого класса Spring WebApplicationContext. Основным назначением является связывание жизненного цикла ApplicationContext и ServletContext, а также автоматического создания ApplicationContext. Можно использовать этот класс для доступа к бинам из различных контекстов спринг.</p> <p><b>Ниже приведена последовательность событий, соответствующая входящему HTTP-запросу:</b></p>  <pre>graph TD     Request --&gt; DS[DispatcherServlet]     DS --&gt; HM[Handler Mapping]     HM --&gt; C[Controller]     C --&gt; DS     DS --&gt; VR[View Resolver]     VR --&gt; V[View]     V --&gt; DS     DS --&gt; Response</pre> <p>❖ После получения HTTP-запроса DispatcherServlet обращается к интерфейсу HandlerMapping, который определяет, какой Контроллер (Controller) должен быть вызван, после чего HandlerAdapter, отправляет запрос в нужный метод Контроллера.</p> <p>❖ Контроллер принимает запрос и вызывает соответствующий метод. Вызванный метод формирует данные Model и возвращает их в DispatcherServlet вместе с именем View (как правило имя html-файла).</p> <p>❖ При помощи интерфейса ViewResolver DispatcherServlet определяет, какое View нужно использовать на основании имени, полученного от контроллера.</p> <p>&gt; если это REST-запрос на сырые данные (JSON/XML), то DispatcherServlet сам его отправляет, минуя ViewResolver;</p> <p>&gt; если обычный запрос, то DispatcherServlet отправляет данные Model в виде атрибутов во View - шаблонизаторы Thymeleaf, FreeMarker и т.д., которые сами отправляют ответ.</p> <p>Как видим, все действия происходят через один DispatcherServlet.</p>



<p><b>Что такое АОП? Как реализовано в спринге?</b></p>	<p>Аспектно-ориентированное программирование (АОП) — это парадигма программирования, целью которой является повышение модульности за счет разделения междисциплинарных задач. Это достигается путем добавления дополнительного поведения к существующему коду без изменения самого кода.</p> <p>АОП предоставляет возможность реализации сквозной логики в одном месте - т.е. логики, которая применяется к множеству частей приложения - и обеспечения автоматического применения этой логики по всему приложению.</p> <p><b>Аспект</b> в АОП - это модуль или класс, реализующий сквозную функциональность. Аспект изменяет поведение остального кода, применяя совет в точках соединения, определенных некоторым срезом.</p> <p><b>Совет</b> (advice) – дополнительная логика — код, который должен быть вызван из точки соединения.</p> <p><b>Точка соединения</b> (join point) — место в выполняемой программе (вызов метода, создание объекта, обращение к переменной), где следует применить совет;</p> <p><b>Срез</b> (pointcut) — набор точек соединения.</p> <p><b>Подход Spring к АОП заключается в создании "динамических прокси" для целевых объектов и "привязывании" объектов к конфигурированному совету для выполнения сквозной логики.</b></p> <p>Есть два варианта создания прокси-класса:</p> <ol style="list-style-type: none"> <li>либо он должен наследоваться от оригинального класса (<b>CGLIB</b>) и переопределять его методы, добавляя нужную логику;</li> <li>либо он должен имплементировать те же самые интерфейсы, что и первый класс (<b>Dynamic Proxy</b>).</li> </ol>
<p><b>В чем разница между Filters, Listeners and Interceptors?</b></p>	<p><b>Filter</b> выполняет задачи фильтрации либо по пути запроса к ресурсу, либо по пути ответа от ресурса, либо в обоих направлениях. Фильтры выполняют фильтрацию в методе doFilter. Каждый фильтр имеет доступ к объекту FilterConfig, из которого он может получить параметры инициализации, и ссылку на ServletContext. Фильтры настраиваются в дескрипторе развертывания веб-приложения.</p> <p>При создании цепочки фильтров, веб-сервер решает, какой фильтр вызывать первым, в соответствии с порядком регистрации фильтров.</p> <p>Когда вызывается метод doFilter(...) первого фильтра, веб-сервер создает объект FilterChain, представляющий цепочку фильтров, и передаёт её в метод.</p> <p>Зависят от контейнера сервлетов. Могут работать с js, css</p> <p><b>preHandle</b> — метод используется для обработки запросов, которые еще не были переданы в метод контроллера. Должен вернуть true для передачи следующему перехватчику или в handler method. False укажет на обработку запроса самим обработчиком и отсутствию необходимости передавать его дальше. Метод имеет возможность выкидывать исключения и пересылать ошибки к представлению.</p> <p><b>postHandle</b> — вызывается после handler method, но до обработки DispatcherServlet для передачи представлению. Может использоваться для добавления параметров в объект ModelAndView.</p> <p><b>afterCompletion</b> — вызывается после отрисовки представления.</p> <p><b>Listener</b> - это класс, имплементирующий интерфейс ServletContextListener с аннотацией @WebListener. Listener ждет когда произойдет указанное событие, затем «перехватывает» событие и запускает собственное событие.</p> <p>Он инициализируется только один раз при запуске веб-приложения и уничтожается при остановке веб-приложения. Все ServletContextListeners уведомляются об инициализации контекста до инициализации любых фильтров или сервлетов в вебприложении и об уничтожении контекста после того, как все сервлеты и фильтры уничтожены.</p>
<p><b>Можно ли передать в запросе один и тот же параметр несколько раз? Как?</b></p>	<p>Да, можно принять все значения, используя массив в методе контроллера:</p> <pre>http://localhost:8080/login?name=Ranga&amp;name=Ravi&amp;name=Sathish public String method(@RequestParam(value="name") String[] names){...}</pre> <pre>http://localhost:8080/api/foos?id=1,2,3 public String getFoos(@RequestParam List&lt;String&gt; id){...}</pre>
<p><b>Как работает Spring Security? Как сконфигурировать? Какие интерфейсы используются?</b></p>	<p><b>В кратце</b>, основными блоками Spring Security являются:</p> <p>SecurityContextHolder, чтобы обеспечить доступ к SecurityContext.</p> <p>SecurityContext, содержит объект Authentication и в случае необходимости информацию системы безопасности, связанную с запросом.</p> <p>Authentication представляет принципа с точки зрения Spring Security.</p> <p>GrantedAuthority отражает разрешения выданные доверителю в масштабе всего приложения.</p> <p>UserDetails предоставляет необходимую информацию для построения объекта Authentication из DAO объектов приложения или других источника данных системы безопасности.</p> <p>UserDetailsService, чтобы создать UserDetails, когда передано имя пользователя в виде String (или идентификатор сертификата или что-то подобное).</p> <p><b>Подробнее:</b></p> <p>Самым фундаментальным является <b>SecurityContextHolder</b>. В нем мы храним информацию о текущем контексте безопасности приложения, который включает в себя подробную информацию о пользователе, работающем с приложением. По умолчанию SecurityContextHolder использует <b>MODE_THREADLOCAL</b> для хранения такой информации, что означает, что контекст безопасности всегда доступен для методов исполняющихся в том же самом потоке, даже если контекст безопасности явно не передается в качестве аргумента этих методов:</p> <p>SecurityContextHolder.getContext().getAuthentication().getPrincipal();</p> <p>UserDetails выступает в качестве принципа.</p> <p><b>MODE_GLOBAL</b> - все потоки Java-машины используют один контекст безопасности.</p> <p><b>MODE_INHERITABLETHREADLOCAL</b> - потоки порожденные от одного защищенного потока, наличие аналогичной безопасности.</p> <p>Интерфейс <b>UserDetailsService</b> - подход к загрузке информации о пользователе в Spring Security. Единственный метод этого интерфейса принимает имя пользователя в виде String и возвращает <b>UserDetails</b>. Он представляет собой принципа, но в расширенном виде и с учетом специфики приложения.</p> <p>В случае успешной аутентификации, UserDetails используется для создания Authentication объекта, который хранится в SecurityContextHolder.</p> <p>Ещё одним важным методом Authentication является getAuthorities() - предоставляет массив объектов GrantedAuthority(поли).</p> <p>Credentials - под ними понимаются пароль пользователя, но им может быть и отпечаток пальца, фото сетчатки и т.п.</p> <p>Процесс аутентификации:</p> <ol style="list-style-type: none"> <li>UsernamePasswordAuthenticationFilter получают имя пользователя и пароль и создает экземпляр класса UsernamePasswordAuthenticationToken (экземпляр интерфейса Authentication).</li> <li>Токен передается экземпляру AuthenticationManager для проверки.</li> <li>AuthenticationManager возвращает полностью заполненный экземпляр Authentication в случае успешной аутентификации.</li> <li>Устанавливается контекст безопасности путем вызова SecurityContextHolder.getContext().setAuthentication(...), куда передается вернувшийся экземпляр Authentication.</li> <li>При успешной аутентификации можно использовать successHandler</li> </ol>

Что такое SpringBoot?  
Какие у него  
преимущества? Как  
конфигурируется?  
Подробно.

Spring Boot - это модуль Spring-a, который предоставляет функцию RAD для среды Spring (Rapid Application Development - Быстрая разработка приложений). Он обеспечивает более простой и быстрый способ настройки и запуска как обычных, так и веб-приложений. Он просматривает наши пути к классам и настроенные нами бины, делает разумные предположения о том, чего нам не хватает, и добавляет эти элементы.

Ключевые особенности и преимущества Spring Boot:

1. Простота управления зависимостями (spring-boot-starter-\* в pom.xml).

Чтобы ускорить процесс управления зависимостями Spring Boot неявно упаковывает необходимые сторонние зависимости для каждого типа приложения на основе Spring и предоставляет их разработчику в виде так называемых starter-пакетов.

Starter-пакеты представляют собой набор удобных дескрипторов зависимостей, которые можно включить в свое приложение. Это позволяет получить универсальное решение для всех технологий, связанных со Spring, избавляя программиста от лишнего поиска необходимых зависимостей, библиотек и решения вопросов, связанных с конфликтом версий различных библиотек.

Например, если вы хотите начать использовать Spring Data JPA для доступа к базе данных, просто включите в свой проект зависимость spring-boot-starter-data-jpa.

Starter-пакеты можно создавать и свои.

2. Автоматическая конфигурация.

Автоматическая конфигурация включается аннотацией @EnableAutoConfiguration. (входит в состав аннотации

@SpringBootApplication)

После выбора необходимых для приложения starter-пакетов Spring Boot попытается автоматически настроить Spring-приложение на основе выбранных jar-зависимостей, доступных в classpath классов, свойств в application.properties и т.п. Например, если добавим springboot-starter-web, то Spring boot автоматически сконфигурирует такие бины как DispatcherServlet, ResourceHandlers, MessageSource итд. Автоматическая конфигурация работает в последнюю очередь, после регистрации пользовательских бинов и всегда отдает им приоритет. Если ваш код уже зарегистрировал бин DataSource — автоконфигурация не будет его переопределять.

3. Встроенная поддержка сервера приложений/контейнера сервлетов (Tomcat, Jetty).

Каждое Spring Boot web-приложение включает встроенный web-сервер. Не нужно беспокоиться о настройке контейнера сервлетов и развертывания приложения в нем. Теперь приложение может запускаться само как исполняемый .jar-файл с использованием встроенного сервера.

4. Готовые к работе функции, такие как метрики, проверки работоспособности, security и внешняя конфигурация.

5. Инструмент CLI (command-line interface) для разработки и тестирования приложения Spring Boot.

6. Минимизация boilerplate кода (код, который должен быть включен во многих местах практически без изменений), конфигурации XML и аннотаций.

Как происходит автоконфигурация в Spring Boot:

1. Отмечаем main класс аннотацией @SpringBootApplication (аннотация инкапсулирует в себе: @SpringBootConfiguration, @ComponentScan, @EnableAutoConfiguration), таким образом наличие @SpringBootApplication включает сканирование компонентов, автоконфигурацию и показывает разным компонентам Spring (например, интеграционным тестам), что это Spring Boot приложение.
2. @EnableAutoConfiguration импортирует класс EnableAutoConfigurationImportSelector. Этот класс не объявляет бины сам, а использует фабрики.
3. Класс EnableAutoConfigurationImportSelector импортирует BCE (более 150) перечисленные в META-INF/spring.factories конфигурации, чтобы предоставить нужные бины в контекст приложения.
4. Каждая из этих конфигураций пытается сконфигурировать различные аспекты приложения (web, JPA, AMQP и т.д.), регистрируя нужные бины. Логика при регистрации бинов управляется набором @ConditionalOn\* аннотаций. Можно указать, чтобы бин создавался при наличии класса в classpath (@ConditionalOnClass), наличии существующего бина (@ConditionalOnBean), отсутствии бина (@ConditionalOnMissingBean) и т.п. Таким образом наличие конфигурации не значит, что бин будет создан и зачастую конфигурация ничего делать и создавать не будет.
5. Созданный в итоге AnnotationConfigEmbeddedWebApplicationContext ищет в том же DI контейнере фабрику для запуска embedded servlet container.
6. Servlet container запускается, приложение готово к работе

Расскажите про  
нововведения Spring  
5.

- Используется JDK 8+ (Optional, CompletableFuture, Time API, java.util.function, default methods)
- Поддержка Java 9 (Automatic-Module-Name in 5.0, module-info in 6.0+, ASM 6)
- Поддержка HTTP/2 (TLS, Push), NIO/NIO.2
- Поддержка Kotlin
- Реактивность (веб-инфраструктура с реактивным стеком, «Spring WebFlux»)
- Null-safety аннотации(@Nullable), новая документация
- Совместимость с Java EE 8 (Servlet 4.0, Bean Validation 2.0, JPA 2.2, JSON Binding API 1.0)
- Поддержка JUnit 5 + Testing Improvements (conditional and concurrent)
- Удалена поддержка: Portlet, Velocity, JasperReports, XMLBeans, JDO, Guava

<p><b>Назовите основные характеристики шаблонов.</b></p>	<p>Имя - все шаблоны имеют уникальное имя, служащее для их идентификации;          Назначение данного шаблона;          Задача, которую шаблон позволяет решить;          Способ решения, предлагаемый в шаблоне для решения задачи в том контексте, где этот шаблон был найден;          Участники - сущности, принимающие участие в решении задачи;          Следствия от использования шаблона как результат действий, выполняемых в шаблоне;          Реализация - возможный вариант реализации шаблона.</p>
<p><b>Назовите три основные группы паттернов.</b></p>	<p><b>Порождающие</b> - отвечают за удобное и безопасное создание новых объектов или даже целых семейств объектов без внесения в программу лишних зависимостей.  <b>Структурные</b> - отвечают за построение удобных в поддержке иерархий классов  <b>Поведенческие</b> - заботятся об эффективной коммуникации между объектами.  <b>Основные</b> - основные строительные блоки, используемые для построения других шаблонов. Например, интерфейс.</p>
<p><b>Расскажите про паттерн Одиночка (Singleton).</b></p>	<p><b>Порождающий паттерн проектирования, который гарантирует, что у класса есть только один экземпляр, и предоставляет к нему глобальную точку доступа.</b>          Конструктор помечается как private, а для создания нового объекта Singleton использует специальный метод getInstance(). Он либо создаёт объект, либо отдаёт существующий объект, если он уже был создан.</p> <pre>private static Singleton instance;</pre> <pre>public static Singleton getInstance() {     if (instance == null) {         instance = new Singleton();     }     return instance; }</pre> <p>+ : можно не создавать множество объектов для ресурсоемких задач, а пользоваться одним          - : нарушает принцип единой ответственности, так как его могут использовать множество объектов</p> <p><b>Почему считается антипаттерном?</b>          -Нельзя тестировать с помощью mock, но можно использовать powerMock.          -Нарушает принцип единой ответственности          -Нарушает Open/Close принцип, его нельзя расширить</p> <p><b>Можно ли его синхронизировать без synchronized у метода?</b>          -Можно сделать его Enum (eager). Это статический final класс с константами. JVM загружает final и static классы на этапе компиляции, а значит несколько потоков не могут создать несколько инстансов.          -С помощью double checked locking (lazy). Synchronized внутри метода:</p>

```

private static volatile Singleton instance;

public static Singleton getInstance() {
    Singleton localInstance = instance;
    if (localInstance == null) {
        // first check
        synchronized (Singleton.class) {
            localInstance = instance;
            if (localInstance == null) {
                // second check
                instance = localInstance = new Singleton();
            }
        }
    }
    return localInstance;
}

```

### Расскажите про паттерн Строитель (Builder).

**Порождающий паттерн, который позволяет создавать сложные объекты пошагово. Строитель даёт возможность использовать один и тот же код строительства для получения разных представлений одного объекта.**

Паттерн предлагает вынести конструирование объекта за пределы его собственного класса, поручив это дело отдельным объектам, называемым строителями.

Процесс конструирования объекта разбить на отдельные шаги (например, построить Стены, вставить Двери). Чтобы создать объект, вам нужно поочерёдно вызывать методы строителя. Причём не нужно запускать все шаги, а только те, что нужны для производства объекта определённой конфигурации.

Можно пойти дальше и выделить вызовы методов строителя в отдельный класс, называемый **директором**. В этом случае директор будет задавать порядок шагов строительства, а строитель — выполнять их.

+: Позволяет использовать один и тот же код для создания различных объектов. Изолирует сложный код сборки объектов от его основной бизнес-логики.

- : Усложняет код программы из-за введения дополнительных классов.

### Расскажите про паттерн Фабричный метод (Factory Method).

**Порождающий шаблон проектирования, в котором подклассы имплементируют общий интерфейс с методом для создания объектов. Переопределенный метод в каждом наследнике возвращает нужный вариант объекта.**

Объекты всё равно будут создаваться при помощи new, но делать это будет фабричный метод. Таким образом можно 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.

+: Выделяет код производства объектов в одно место, упрощая поддержку кода. Реализует принцип открытости/закрытости.

- : Может привести к созданию больших параллельных иерархий классов, так как для каждого класса продукта надо создать свой подкласс создателя.

<p><b>Расскажите про паттерн Абстрактная фабрика (Abstract Factory).</b></p>	<p><b>Порождающий паттерн проектирования, который представляет собой интерфейс для создания других классов, не привязываясь к конкретным классам создаваемых объектов.</b></p> <p>Абстрактная фабрика предлагает выделить общие интерфейсы для отдельных продуктов, составляющих семейства. Так, все вариации кресел получают общий интерфейс Кресло, все диваны реализуют интерфейс Диван и так далее.</p> <p>Далее вы создаёте абстрактную фабрику — общий интерфейс, который содержит <b>фабричные методы</b> создания всех продуктов семейства (например, создатьКресло, создатьДиван и создатьСтолик). Эти операции должны возвращать абстрактные типы продуктов, представленные интерфейсами, которые мы выделили ранее — Кресла, Диваны и Столики.</p> <p>+ : гарантированно будет создаваться тип одного семейства</p> <p>- : Усложняет код программы из-за введения множества дополнительных классов.</p>
<p><b>Расскажите про паттерн Прототип (Prototype).</b></p>	<p><b>Порождающий паттерн проектирования, который позволяет копировать объекты, не вдаваясь в подробности их реализации.</b></p> <p>Паттерн поручает создание копий самим копируемым объектам. Он вводит общий интерфейс с методом clone для всех объектов, поддерживающих клонирование. Реализация этого метода в разных классах очень схожа. Метод создаёт новый объект текущего класса и копирует в него значения всех полей собственного объекта.</p> <p>+ : Позволяет клонировать объекты, не привязываясь к их конкретным классам.</p> <p>- : Сложно клонировать составные объекты, имеющие ссылки на другие объекты.</p>
<p><b>Расскажите про паттерн Адаптер (Adapter).</b></p>	<p><b>Структурный паттерн проектирования, который позволяет объектам с несовместимыми интерфейсами работать вместе.</b></p> <p>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.</p> <p>+ : Отделяет и скрывает от клиента подробности преобразования различных интерфейсов.</p> <p>- : Усложняет код программы из-за введения дополнительных классов.</p>
<p><b>Расскажите про паттерн Декоратор (Decorator).</b></p>	<p><b>Структурный паттерн проектирования, который позволяет добавлять объектам новую функциональность, оборачивая их в полезные «обёртки».</b></p> <p>Целевой объект помещается в другой объект-обёртку, который запускает базовое поведение обёрнутого объекта, а затем добавляет к результату что-то своё.</p> <p>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.</p> <p>Адаптер не меняет состояния объекта, а декоратор может менять.</p> <p>+ : Большая гибкость, чем у наследования.</p> <p>- : Труднее конфигурировать многократно обёрнутые объекты.</p>
<p><b>Расскажите про паттерн Заместитель (Proxy).</b></p>	<p><b>Структурный паттерн проектирования, который позволяет подставлять вместо реальных объектов специальные объекты-заместители, которые перехватывают вызовы к оригинальному объекту, позволяя сделать что-то до или после передачи вызова оригиналу.</b></p> <p>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, выполняя промежуточную логику, которая выполнялась бы до (или после) вызовов этих же методов в настоящем объекте.</p> <p>+ : Позволяет контролировать сервисный объект незаметно для клиента.</p> <p>- : Увеличивает время отклика от сервиса.</p>

<p>Расскажите про паттерн Итератор (Iterator).</p>	<p><b>Поведенческий паттерн проектирования, который даёт возможность последовательно обходить элементы составных объектов, не раскрывая их внутреннего представления.</b> Идея состоит в том, чтобы вынести поведение обхода коллекции из самой коллекции в отдельный класс.</p> <p>Детали: Создается итератор и интерфейс, который возвращает итератор. В классе, в котором надо будет вызывать итератор, имплементируем интерфейс, возвращающий итератор, а сам итератор делаем там нестатическим вложенным классом, так как он нигде использоваться больше не будет.</p>
<p>Расскажите про паттерн Шаблонный метод (Template Method).</p>	<p><b>Поведенческий паттерн проектирования, который пошагово определяет алгоритм и позволяет наследникам переопределять некоторые шаги алгоритма, не изменяя его структуру в целом.</b> Паттерн предлагает разбить алгоритм на последовательность шагов, описать эти шаги в отдельных методах и вызывать их в одном шаблонном методе друг за другом. Для описания шагов используется абстрактный класс. Общие шаги можно будет описать прямо в абстрактном классе. Это позволит подклассам переопределять некоторые шаги алгоритма, оставляя без изменений его структуру и остальные шаги, которые для этого подкласса не так важны.</p>
<p>Расскажите про паттерн Цепочка обязанностей (Chain of Responsibility).</p>	<p>Поведенческий паттерн проектирования, который позволяет передавать запросы последовательно по цепочке обработчиков. Каждый обработчик решает, может ли он обработать запрос сам и стоит ли передавать запрос дальше по цепи. Базируется на том, чтобы превратить каждую проверку в отдельный класс с единственным методом выполнения. Данные запроса, над которым происходит проверка, будут передаваться в метод как аргументы. Каждый из методов будет иметь ссылку на следующий метод-обработчик, что образует цепь. Таким образом, при получении запроса обработчик сможет не только сам что-то с ним сделать, но и передать обработку следующему объекту в цепочке. Может и не передавать, если проверка в одном из методов не прошла, например.</p>
<p>Какие паттерны используются в Spring Framework?</p>	<p><b>Singleton</b> - Bean scopes <b>Factory</b> - Bean Factory classes <b>Prototype</b> - Bean scopes <b>Adapter</b> - Spring Web and Spring MVC <b>Proxy</b> - Spring Aspect Oriented Programming support <b>Template Method</b> - JdbcTemplate, HibernateTemplate etc <b>Front Controller</b> - Spring MVC DispatcherServlet <b>DAO</b> - Spring Data Access Object support <b>Dependency Injection</b></p>
<p>Какие паттерны используются в Hibernate?</p>	<p><b>Domain Model</b> — объектная модель предметной области, включающая в себя как поведение так и данные. <b>Data Mapper</b> — слой мапперов (Mappers), который передает данные между объектами и базой данных, сохраняя их независимыми друг от друга и себя. <b>Proxy</b> — применяется для ленивой загрузки. <b>Factory</b> — используется в SessionFactory</p>
<p>Шаблоны GRASP: Low Coupling (низкая связанность) и High Cohesion (высокая сплоченность)</p>	<p><b>Low Coupling</b> - части системы, которые изменяются вместе, должны находиться близко друг к другу. <b>High Cohesion</b> - если возвести Low Coupling в абсолюте, то можно прийти к тому, чтобы разместить всю функциональность в одном единственном классе. В таком случае связей не будет вообще, но что-то тут явно не так, ведь в этот класс попадет совершенно несвязанная между собой бизнес-логика. Принцип High Cohesion говорит следующее: части системы, которые изменяются параллельно, должны иметь как можно меньше зависимостей друг на друга.</p> <p>Low Coupling и High Cohesion представляют из себя два связанных между собой паттерна, рассматривать которые имеет смысл только вместе. Их суть: система должна состоять из слабо связанных классов, которые содержат связанную бизнес-логику. Соблюдение этих принципов позволяет удобно переиспользовать созданные классы, не теряя понимания о их зоне ответственности.</p>

## Расскажите про паттерн Saga

Saga — это механизм, обеспечивающий согласованность данных в микросервисах без применения распределенных транзакций.

Для каждой системной команды, которой надо обновлять данные в нескольких сервисах, создается некоторая сага. Сага представляет из себя некоторый «чек-лист», состоящий из последовательных локальных ACID-транзакций, каждая из которых обновляет данные в одном сервисе. Для обработки сбоев применяется компенсирующая транзакция. Такие транзакции выполняются в случае сбоя на всех сервисах, на которых локальные транзакции выполнились успешно.

Типов транзакций в саге четыре:

**Компенсирующая** — отменяет изменение, сделанное локальной транзакцией.

**Компенсируемая** — это транзакция, которую необходимо компенсировать (отменить) в случае, если последующие транзакции завершаются неудачей.

**Поворотная** — транзакция, определяющая успешность всей саги. Если она выполняется успешно, то сага гарантированно дойдет до конца.

**Повторяемая** — идет после поворотной и гарантированно завершается успехом.



## Алгоритмы

Big O (О большое / символ Ландау) - математическое обозначение порядка функции для сравнения асимптотического поведения функций.

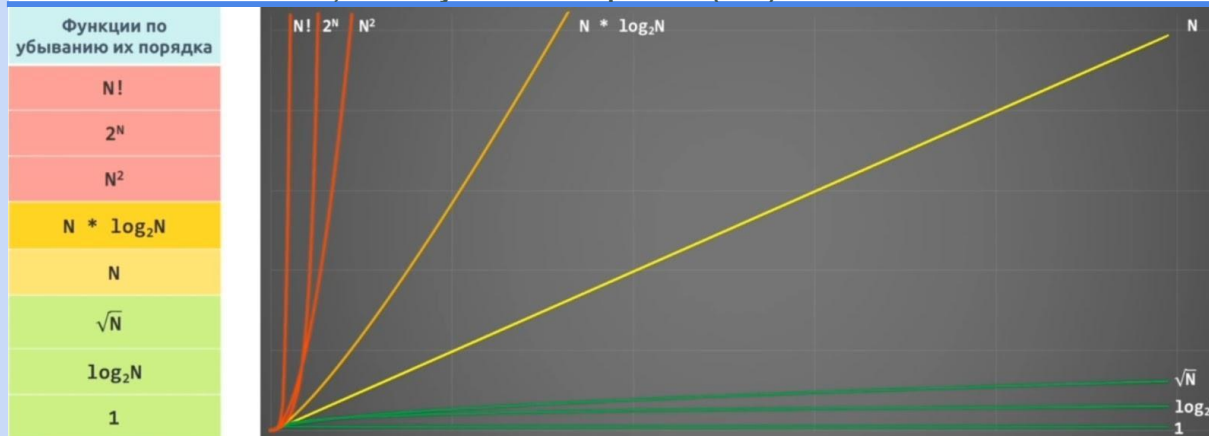
Асимптотика - характер изменения функции при стремлении ее аргумента к определённой точке.

Любой алгоритм состоит из неделимых операций процессора(шагов), поэтому нужно измерять время в операциях процессора, вместо секунд.

DTIME - количество шагов(операций процессора), необходимых, чтобы алгоритм завершился.

Временная сложность обычно оценивается путём подсчёта числа элементарных операций, осуществляемых алгоритмом. Время исполнения одной такой операции при этом берётся константой, то есть асимптотически оценивается как  $O(1)$ . Сложность алгоритма состоит из двух факторов: временная сложность и сложность по памяти. **Временная сложность** - функция, представляющая зависимость количество операций процессора, необходимых, чтобы алгоритм завершился, от размера входных данных. Все неделимые операции языка(операции сравнения, арифметические, логические, инициализации и возврата) считаются выполняемыми за 1 операцию процессора, эта погрешность считается приемлемой. При росте  $N$ , слагаемые с меньшей скоростью роста всё меньше влияют на значение функции. Поэтому, вне зависимости от констант при слагаемых, **слагаемое с большей скоростью роста определяет значение функции. Данное слагаемое называют порядком функции.** Пример:  $T(N) = 5 * N^2 + 999 * N...$  Где  $(5 * N^2)$  и  $(9999 * N)$  являются слагаемыми функции. **Константы(5 и 999) не указываются в рамках нотации Big O, так как не показывают абсолютную сложность алгоритма, так как могут изменяться в зависимости от машины, поэтому сложность равна  $O(N^2)$**

Что такое Big O? Как происходит оценка асимптотической сложности алгоритмов?



	<p>В порядке возрастания сложности:</p> <ol style="list-style-type: none"> <li>1. <math>O(1)</math> - константная, чтение по индексу из массива</li> <li>2. <math>O(\log(n))</math> - логарифмическая, бинарный поиск в отсортированном массиве</li> <li>3. <math>O(\sqrt{n})</math> - сублинейная</li> <li>4. <math>O(n)</math> - линейная, перебор массива в цикле, два цикла подряд, линейный поиск наименьшего или наибольшего элемента в неотсортированном массиве</li> <li>5. <math>O(n \cdot \log(n))</math> - квазилинейная, сортировка слиянием, сортировка кучей</li> <li>6. <math>O(n^2)</math> - полиномиальная(квадратичная), вложенный цикл, перебор двумерного массива, сортировка пузырьком, сортировка вставками</li> <li>7. <math>O(2^n)</math> - экспоненциальная, алгоритмы разложения на множители целых чисел</li> <li>8. <math>O(n!)</math> - факториальная, решение задачи коммивояжёра полным перебором</li> </ol> <p><b>Алгоритм считается приемлемым, если сложность не превышает <math>O(n \cdot \log(n))</math>, иначе говнокод.</b></p>
<p><b>Что такое рекурсия?</b>  <b>Сравните преимущества и недостатки итеративных и рекурсивных алгоритмов. С примерами.</b></p>	<p>Рекурсия - способ отображения какого-либо процесса внутри самого этого процесса, то есть ситуация, когда процесс является частью самого себя.</p> <p>Рекурсия состоит из базового случая и шага рекурсии. Базовый случай представляет собой самую простую задачу, которая решается за одну итерацию, например, <code>if(n == 0) return 1</code>.</p> <p>В базовом случае обязательно присутствует условие выхода из рекурсии;</p> <p>Смысл рекурсии в движении от исходной задачи к базовому случаю, пошагово уменьшая размер исходной задачи на каждом шаге рекурсии.</p> <p>После того, как будет найден базовый случай, срабатывает условие выхода из рекурсии, и стек рекурсивных вызовов разворачивается в обратном порядке, пересчитывая результат исходной задачи, который основан на результате, найденном в базовом случае.</p> <p>Так работает рекурсивное вычисление факториала:</p> <pre>int factorial(int n) {     if(n == 0) return 1;           // базовый случай с условием выхода     else return n * factorial(n - 1); // шаг рекурсии (рекурсивный вызов) }</pre> <p>Или даже так:</p> <pre>return (n==0) ? 1 : n * factorial(n-1);</pre> <p><b>Рекурсия имеет линейную сложность <math>O(n)</math>;</b></p> <hr/> <p>Циклы дают лучшую производительность, чем рекурсивные вызовы, поскольку вызовы методов потребляют больше ресурсов, чем исполнение обычных операторов.</p> <p>Циклы гарантируют отсутствие переполнения стека, т.к. не требуется выделения доп. памяти.</p> <p>В случае рекурсии стек вызовов разрастается, и его необходимо просматривать для получения конечного ответа.</p> <p>При использовании головной рекурсии также необходимо принимать во внимание размер стека.</p> <p>Если уровней вложенности много или изменяются, то предпочтительна рекурсия. Если их несколько, то лучше цикл.</p>

<p><b>Что такое жадные алгоритмы? Приведите пример.</b></p>	<p>Жадные алгоритмы являются одной из 3х техник создания алгоритмов, вместе с принципом "Разделяй и властвуй" и динамическим программированием.</p> <p><b>Жадный алгоритм - это алгоритм, который на каждом шагу совершает локально оптимальные решения, т.е. максимально возможное из допустимых, не учитывая предыдущие или следующие шаги. Последовательность этих локально оптимальных решений приводит (не всегда) к глобально оптимальному решению.</b></p> <p><b>Т.е. задача разбивается на подзадачи, в каждой подзадаче делается оптимальное решение и, в итоге, вся задача решается оптимально. При этом важно является ли каждое локальное решение безопасным шагом. Безопасный шаг - приводящий к оптимальному решению.</b></p> <p>К примеру, алгоритм Дейкстры нахождения кратчайшего пути в графе вполне себе жадный, потому что мы на каждом шагу ищем вершину с наименьшим весом, в которой мы еще не бывали, после чего обновляем значения других вершин. При этом можно доказать, что кратчайшие пути, найденные в вершинах, являются оптимальными.</p>
<p><b>Расскажите про пузырьковую сортировку.</b></p>	<p>Будем идти по массиву слева направо. Если текущий элемент больше следующего, меняем их местами. Делаем так, пока массив не будет отсортирован.</p> <p>Асимптотика в худшем и среднем случае – <math>O(n^2)</math>, в лучшем случае – <math>O(n)</math> - массив уже отсортирован.</p>
<p><b>Расскажите про быструю сортировку.</b></p>	<p>Выберем некоторый опорный элемент(pivot). После этого перекинем все элементы, меньшие его, налево, а большие – направо. Для этого используются дополнительные переменные - значения слева и справа, которые сравниваются с pivot. Рекурсивно вызовемся от каждой из частей, где будет выбран новый pivot. В итоге получим отсортированный массив, так как каждый элемент меньше опорного стоял раньше каждого большего опорного.</p> <p>Асимптотика: <math>O(n \cdot \log(n))</math> в среднем и лучшем случае. Наихудшая оценка <math>O(n^2)</math> достигается при неудачном выборе опорного элемента.</p>
<p><b>Расскажите про сортировку слиянием.</b></p>	<p>Основана на парадигме «разделяй и властвуй». Будем делить массив пополам, пока не получим множество массивов из одного элемента. После чего выполним процедуру слияния: поддерживаем два указателя, один на текущий элемент первой части, второй – на текущий элемент второй части. Из этих двух элементов выбираем минимальный, вставляем в ответ и сдвигаем указатель, соответствующий минимуму. Так сделаем слияния массивов из 1го элемента в массивы по 2 элемента, затем из 2х в 4 и т.д. Слияние работает за <math>O(n)</math>, уровней всего <math>\log(n)</math>, поэтому <b>асимптотика <math>O(n \cdot \log(n))</math>.</b></p>
<p><b>Расскажите про бинарное дерево.</b></p>	<p>Бинарное дерево - иерархическая структура данных, в которой каждый узел может иметь двух потомков. Как правило, первый называется родительским узлом, а наследники называются левым и правым нодами/узлами. Каждый узел в дереве задаёт поддереву, корнем которого он является. Оба поддерева — левое и правое — тоже являются бинарными деревьями. Ноды, которые не имеют потомков, называются листьями дерева. У всех узлов левого поддерева произвольного узла X значения ключей данных меньше, нежели значение ключа данных самого узла X. У всех узлов правого поддерева произвольного узла X значения ключей данных больше либо равны, нежели значение ключа данных самого узла X. Этим достигается упорядоченная структура данных, то есть всегда отсортированная.</p> <p>Поиск в лучшем случае - <math>O(\log(n))</math>, худшем - <math>O(n)</math> - при вырождении в связанный список.</p>

**Расскажите про красно-  
черное дерево.**

Усовершенствованная версия бинарного дерева. Каждый узел в к/ч дереве имеет дополнительное поле - цвет. К/ч дерево отвечает следующим требованиям:

- 1) Узел либо красный, либо черный.
- 2) Корень - черный.
- 3) Все листья - черные и не хранят данных.
- 4) Оба потомка каждого красного узла - черные.
- 5) Любой простой путь от узла-предка до листового узла-потомка содержит одинаковое число черных узлов. Если не одинаковое, то происходит переворот.

При добавлении постоянно увеличивающихся/уменьшающихся чисел в бинарное дерево, оно вырождается в связанный список и теряет свои преимущества. Тогда как к/ч дерево может потребовать до двух поворотов для поддержки сбалансированности, чтобы избежать вырождения.

При операциях удаления в бинарном дереве для удаляемого узла надо найти замену. К/ч дерево делает тоже самое, но потребует до трёх поворотов для поддержки сбалансированности.

В этом и состоит преимущество.

Сложность поиска, вставки и удаления -  $O(\log(n))$

**Расскажите про линейный и  
бинарный поиск.**

Линейный поиск - сложность  $O(n)$ , так как все элементы проверяются по очереди.

Бинарный поиск -  $O(\log(n))$ . Массив должен быть отсортирован. Происходит поиск индекса в массиве, содержащего искомое значение.

- 1) Берем значение из середины массива и сравниваем с искомым. Индекс середины считается по формуле  $mid = (high + low) / 2$

low - индекс начала левого подмассива, high - индекс конца правого подмассива.

- 2) Если значение в середине больше искомого, то рассматриваем левый подмассив и  $high = middle - 1$

- 3) Если меньше, то правый и  $low = middle + 1$

- 4) Повторяем, пока mid не станет равен искомому элементу или подмассив не станет пустым.

```
public static int binarySearch(int[] a, int key) {
    int low = 0;
    int high = a.length - 1;

    while (low <= high) {
        int mid = (low + high)/2;

        if (key > a[mid]) {
            low = mid + 1;
        } else if (key < a[mid]) {
            high = mid - 1;
        } else return mid;
    }
    return -1;
}
```

Расскажите про очередь и стек.

**Stack** это область хранения данных, находящееся в общей оперативной памяти (RAM). Всякий раз, когда вызывается метод, в памяти стека создается новый блок-фрейм, который содержит локальные переменные метода и ссылки на другие объекты в методе. Как только метод заканчивает работу, блок также перестает использоваться, тем самым предоставляя доступ для следующего метода. Размер стековой памяти намного меньше объема памяти в куче. Стек в Java работает по схеме LIFO

**Queue** - это очередь, которая обычно (но необязательно) строится по принципу FIFO (First-In-First-Out) - соответственно извлечение элемента осуществляется с начала очереди, вставка элемента - в конец очереди.

Хотя этот принцип нарушает, к примеру PriorityQueue, использующая «natural ordering» или переданный Comparator при вставке нового элемента.

**Deque** (Double Ended Queue) расширяет Queue и согласно документации это линейная коллекция, поддерживающая вставку/извлечение элементов с обоих концов. Помимо этого реализации интерфейса Deque могут строиться по принципу FIFO, либо LIFO.

Реализации и Deque, и Queue обычно не переопределяют методы equals() и hashCode(), вместо этого используются унаследованные методы класса Object, основанные на сравнении ссылок.

Сравните сложность вставки, удаления, поиска и доступа по индексу в ArrayList и LinkedList.

	Временная сложность							
	Среднее				Худшее			
	Индекс	Поиск	Вставка	Удаление	Индекс	Поиск	Вставка	Удаление
ArrayList	O(1)	O(n)	O(n)	O(n)	O(1)	O(n)	O(n)	O(n)
Vector	O(1)	O(n)	O(n)	O(n)	O(1)	O(n)	O(n)	O(n)
LinkedList	O(n)	O(n)	O(1)	O(1)	O(n)	O(n)	O(1)	O(1)
Hashtable	n/a	O(1)	O(1)	O(1)	n/a	O(n)	O(n)	O(n)
HashMap	n/a	O(1)	O(1)	O(1)	n/a	O(n)	O(n)	O(n)
LinkedHashMap	n/a	O(1)	O(1)	O(1)	n/a	O(n)	O(n)	O(n)
TreeMap	n/a	O(log(n))	O(log(n))	O(log(n))	n/a	O(log(n))	O(log(n))	O(log(n))
HashSet	n/a	O(1)	O(1)	O(1)	n/a	O(n)	O(n)	O(n)
LinkedHashSet	n/a	O(1)	O(1)	O(1)	n/a	O(n)	O(n)	O(n)
TreeSet	n/a	O(log(n))	O(log(n))	O(log(n))	n/a	O(log(n))	O(log(n))	O(log(n))

## ДопВопросы

### Вопросы на проработку легенды

Как происходил процесс автоматизации версий и выхода в продакшн? Как заменяли версии?  
Как происходил деплой и где хранили проект?  
Обязательно возьмите две интересные такси из вашей легенды и придумайте их реализацию.  
Почему ушли с прошлого места?  
...

### Жизненный цикл проекта в Maven

**Проверка — validate.** Фреймворк проверяет, корректен ли проект и предоставлена ли вся необходимая для сборки информация.  
**Компиляция — compile.** Maven компилирует исходники проекта.  
**Тест — test.** Проверка скомпилированных файлов. В нашем случае будет использована библиотека JUnit.  
**Сборка проекта — package.** По умолчанию осуществляется в формате JAR. Этот параметр можно изменить, добавив в project tag packaging.  
**Интеграционное тестирование — integration-test.** Maven обрабатывает и при необходимости распаковывает пакет в среду, где будут выполняться интеграционные тесты.  
**Верификация — verify.** Артефакт проверяется на соответствие критериям качества.  
**Инсталляция — install.** Артефакт попадает в локальный репозиторий. Теперь его можно использовать в качестве зависимости.  
**Размещение проекта в удалённом репозитории — deploy,** — финальная стадия работы.  
  
Помимо этого есть две фазы, выполняющиеся отдельно, только прямой командой:  
**Clean** — очистка, удаляющая предыдущие сборки.  
**Site** — создание документации для сайта.

### Случаи утечки памяти в Java

**ObjectInputStream и ObjectOutputStream** хранят ссылки на все объекты, с которыми они работали, чтобы передавать их вместо копий. Для решения этой проблемы необходимо периодически вызывать метод reset().  
**Каждый экземпляр класса Thread в Java выделяет память для своего стека** (по умолчанию, это 512 Кб; изменяется с помощью параметра -Xss). Неоптимизированные приложения, использующие множество потоков, могут привести к необоснованно высокому потреблению памяти.  
**Нестатичный внутренний класс**, который вы используете, хранит ссылку на внешний класс. Это приводит к хранению большого графа объектов.  
Как только экземпляр-синглтон был инициализирован, он остаётся в памяти на всё время жизни приложения.  
**Статичная переменная** хранится своим классом, а как следствие, его загрузчиком (classloader). По причине внешнего использования увеличивается шанс, что сборщик мусора не соберёт данный экземпляр.

<p><b>Обмен сообщениями в микросервисах</b></p>	<p>По существу, есть два варианта: <b>синхронное взаимодействие или асинхронное взаимодействие</b>.</p> <p><b>Синхронное</b> взаимодействие микросервисов обычно осуществляется через HTTP и REST-подобные сервисы, которые возвращают XML или JSON — хотя это не является обязательным (посмотрите, например, на Google Protocol Buffers). Используйте REST-коммуникацию, когда вам нужен немедленный ответ.</p> <p><b>Асинхронная</b> микросервисная связь обычно осуществляется посредством обмена сообщениями с помощью реализации JMS и / или с помощью протокола, такого как AMQP - это облегченный слой перед сервисами - брокерасообщений, используемый, например, в качестве балансировщика нагрузки или для реализации какого-либо вида аутентификации и/или авторизации. Например, Kafka или RabbitMQ. Обычно, на практике не следует недооценивать интеграцию по электронной почте / SMTP. Используйте асинхронное взаимодействие, когда вам не нужен немедленный ответ.</p> <p>Брокеры сообщений (очереди сообщений) - это проверенный способ реализации коммуникации между субсервисами, что позволяет:</p> <p><b>Асинхронная связь</b> (без ожидания)</p> <p><b>Может легко масштабировать различные службы</b> (нет зависимостей JVM, может развертывать различные службы на разных серверах)</p> <p><b>Может управлять событиями централизованно</b> (просто еще один паб sub)</p> <p><b>Масштабируемость</b> (легко добавить другой сервис, который должен взаимодействовать с некоторыми существующими сервисами)</p>
<p><b>Преимущества очередей</b></p>	<p>Основные преимущества использования очередей сообщений:</p> <ul style="list-style-type: none"> <li>– Асинхронное общение между компонентами с гарантированной доставкой сообщений даже в случае недоступности обработчика сообщений.</li> <li>– Позволяют горизонтально масштабировать приложения; распределяют процессы обработки информации; позволяют балансировать нагрузку;</li> <li>– Дают возможность выдерживать пиковые нагрузки;</li> <li>– Отказоустойчивость, хранит сообщения пока не истечет таймаут, или пока сообщение не будет обработано потребителем.</li> </ul>



Преимущества и недостатки микросервисов	<p><b>Преимущества:</b></p> <ul style="list-style-type: none"> <li>-маленькие и простые сервисы с понятными API и понятными связями между сервисами VS огромный монолит, в котором непонятно как всё взаимодействует.</li> <li>-проще тестировать</li> <li>-независимые релизы</li> <li>-независимая деградация, т.е. если какой-то сервис лежит, но он не взаимодействует с пользователями, то это не влияет на работу.</li> <li>-независимое масштабирование</li> <li>-легче пробовать новые технологии, переписав один сервис, а не весь монолит.</li> <li>-у сервиса есть команда-владелец, к которой можно обратиться за помощью.</li> </ul> <p><b>Недостатки:</b></p> <ul style="list-style-type: none"> <li>-сложно отслеживать запросы, так как они проходят через множество сервисов, а в монолите всё в одном месте. Решаемо с помощью установки идентификатора каждому запросу и поиску по нему с помощью graylog.</li> <li>-RPC сложнее вызова метода</li> <li>-сложности распределенных систем: отказы, таймауты, ретраи, дубли...</li> <li>-сложнее запустить: выбор языка, упаковка, настройка процедуры выкладки, ротации и заливки логов, мониторинга и триггеров и другие.</li> </ul>
Remote procedure call (RPC)	Вызов удалённых процедур — класс технологий, позволяющих программам вызывать процедуры в другом адресном пространстве.
git. merge vs rebase	<p><b>merge</b> принимает содержимое ветки источника и объединяет их с целевой веткой. В этом процессе изменяется только целевая ветка. История исходных веток остается неизменной.</p> <p><b>rebase</b> сжимает все изменения в один «патч». Затем он интегрирует патч в целевую ветку. В отличие от слияния, перемещение перезаписывает историю, потому что она передает завершённую работу из одной ветки в другую. В процессе устраняется нежелательная история. Можно ребейсить только свои ветки.</p>
git cherry-pick	Команда git cherry-pick используется для перенесения отдельных коммитов из одного места репозитория в другое, обычно между ветками разработки и обслуживания. Этот механизм отличается от привычных команд git merge и git rebase, которые переносят коммиты целыми цепочками.
Команда для просмотра всех процессов в Unix	<p>ps (показывает только запущенные контейнеры)</p> <p>ps -a (показывает все контейнеры, включая отработавшие)</p> <p>По крайней мере в докере)</p>

<p><b>TDD , DDD, BDD</b></p>	<p><b>TDD</b> — Test Driven Development - методология разработки ПО, которая основывается на повторении коротких циклов разработки: <b>RED-GREEN-REFACTOR</b>. На этапе <b>RED</b> пишется изначально пишется тест, который намеренно будет не проходить проверку. Это нужно для того, чтобы убедиться, что он вообще работает. На этапе <b>GREEN</b> пишется минимальный код-заглушка, который позволит пройти тест. И на этапе <b>REFACTORING</b> пишется код, который реализует желаемое поведение системы и позволит пройти написанный тест без "заглушек".</p> <p><b>TDD</b> — Type Driven Development - при разработке на основе типов ваши типы данных и сигнатуры типов являются спецификацией программы</p> <p><b>BDD</b> — Behaviour Driven Development - предполагает описание тестировщиком или аналитиком пользовательских сценариев на естественном языке вида "я как пользователь хочу когда нажали кнопку пуск тогда показывалось меню как на картинке"</p> <p><b>DDD</b> — Domain Driven Design - это набор принципов и схем, направленных на создание оптимальных систем объектов. Процесс разработки сводится к созданию программных абстракций, которые называются моделями предметных областей</p>
<p><b>Репликация данных</b></p>	<p>Репликация — механизм синхронизации содержимого нескольких копий объекта (например, содержимого БД). Репликация — это процесс, под которым понимается копирование данных из одного источника на другой (или на множество других) и наоборот.</p>
<p><b>Dirty checking механизм Hibernate</b></p>	<p>По умолчанию Hibernate проверяет все поля управляемых сущностей. Каждый раз, когда объект загружается, Hibernate делает дополнительные копии всех полей сущностей. Во время flush, каждое поле управляемой сущности сравнивается с копией, сделанной во время загрузки. Даже если только одно поле одной сущности изменилось, Hibernate сверит все сущности.</p> <p>С целью ускорить этот процесс пользуйтесь следующими фишками:</p> <p>em.detach / em.clear — открепляют сущности от EntityManager-a</p> <p>FlushMode=MANUAL- полезен в операциях чтения</p> <p>@Immutable — также позволяет избежать операций dirty checking</p>
<p><b>6 принципов REST API</b></p>	<p><b>Что такое REST?</b></p> <p>Representational State Transfer — передача состояния представления. Это архитектурный стиль взаимодействия компонентов распределенной системы в компьютерной сети. Проще говоря, REST определяет стиль взаимодействия (обмена данными) между разными компонентами системы, каждая из которых может физически располагаться в разных местах.</p> <p>Данный архитектурный стиль представляет собой согласованный набор ограничений, учитываемых при проектировании распределенной системы:</p> <p><b>Client-Server.</b> Отделяя пользовательский интерфейс от хранилища данных, мы улучшаем переносимость пользовательского интерфейса на другие платформы и улучшаем масштабируемость серверных компонент за счёт их упрощения.</p> <p><b>Stateless</b> (без состояния). Каждый запрос от клиента к серверу должен содержать в себе всю необходимую информацию и не может полагаться на какое-либо состояние, хранящееся на стороне сервера. Таким образом, информация о текущей сессии должна целиком храниться у клиента.</p>

<b>REST vs SOAP</b>	<p><b>SOAP</b> — это формат протокола обмена сообщениями, основанный на XML поверх HTTP. Нет ограничений на тип транспортного протокола. Можно и HTTP и MQ.</p> <p>SOAP использует WSDL (Web Services Description Language) — язык описания веб-сервисов и доступа к ним, основанный на языке XML.</p> <p>В SOAP необходимо определить свой сервис с использованием WSDL, и при обработке и анализе сообщений SOAP-XML возникают большие накладные расходы.</p> <p><b>REST</b> — это архитектурный подход. Можно обмениваться сообщениями на основе XML, JSON или любого другого формата. Использование транспортного протокола HTTP.</p> <p>REST не имеет стандартного языка определения сервиса (популярны Swagger и Open API).</p> <p>RESTful веб-сервисы проще реализовать, так как используется JSON, который легче анализировать и обрабатывать. И REST не требует наличия определения службы для предоставления веб-службы.</p> <p><b>Минус Rest при работе в highLoad:</b> формат JSON довольно тяжелый и содержит в себе схему объекта, из-за чего увеличивается нагрузка на сеть + время на сериализацию и десериализацию. Но есть бинарные протоколы-аналоги. Например, Protocol Buffers, который отправляет чисто данные и не содержит в себе схему. Они могут быть в 20-100 раз быстрее.</p>
<b>Различие между методами GET и POST</b>	<p><b>GET</b> передает данные в URL в виде пар "имя-значение", данные видны всем в адресной строке браузера. Длина запроса не более 2048 символов. Следует использовать для получения данных от сервера и не желательно в запросах, предполагающих внесений изменений в ресурс.</p> <p><b>POST</b> передает данные в теле запроса. Данные можно увидеть только с помощью инструментов разработчика, расширений браузера. Следует использовать в случаях, когда нужно вносить изменение в ресурс. HTTP метод POST поддерживает тип кодирования данных multipart/form-data, что позволяет <u>передавать файлы</u>.</p>
<b>Чем отличаются Mock и Spy?</b>	<p><b>В случае Mock</b>-объекта, единственное, что будут делать все методы такого объекта, если не определять их поведение, – возвращать значения по-умолчанию: void, default-ы для примитивов, пустые коллекции и null для всех остальных объектов.</p> <p><b>В случае Spy</b>-объекта, по-умолчанию будет исполняться оригинальное поведение методов объекта. Но как и в случае с Mock-объектами, их поведение можно переопределить.</p>
<b>Code first vs Design first</b>	<p><b>Code First</b> — сначала код, потом всё остальное. Сначала у нас считается логика, запускаются функции и нажимаются кнопки, а уже потом придумывается интерфейс.</p> <p><b>Design First</b> — сначала проектирование, потом всё остальное. Проектировщики сначала придумывают, как продукт должен выглядеть со стороны и работать, а потом уже программисты придумывают, как это оживить.</p>
<b>Java 8 vs 11</b>	<p>Новые методы для Map: PutIfAbsent(), ComputefIfAbsent()\ComputefIfPresent(), Remove(), GetOrDefault(), Merge()</p> <p>Новые методы для String: repeat(int), isBlank(), lines(), strip() - это не все.</p> <p>Динамическая типизация с var</p>
<b>Защита от рефлексии</b>	
<b>CI/CD</b>	<p><b>Continuous Integration/Continuous Delivery</b></p> <p>Continuous Integration - методология разработки и набор практик, при которых в код вносятся небольшие изменения с частыми коммитами в течение суток.</p> <p>Continuous Delivery - постоянная поддержка кода в состоянии готовом для выхода в продакшн.</p>

<b>nosql</b>	<p>Базы данных NoSQL предоставляют разнообразные модели данных, такие как пары «ключ-значение», документы и графы.</p> <p>Смягчают жесткие требования свойств ACID ради более гибкой модели данных, которая допускает горизонтальное масштабирование.</p>
<b>Компиляция и запуск программы из командной строки</b>	<p><b>javac className.class</b> - чтобы скомпилировать из класса файл .java с байт-кодом</p> <p><b>java className</b> - чтобы запустить файл</p> <p>Чтобы передать аргументы с main-метод, нужно просто перечислить их через пробел:</p> <p><b>java className arg0 arg1 arg2</b></p>
<b>HTTP протокол</b>	<p>HTTP является протоколом прикладного уровня, который чаще всего использует возможности другого протокола - TCP (или TLS - защищённый TCP) - для пересылки и получения сообщений. Протокол HTTP лежит в основе обмена данными в Интернете. HTTP является протоколом клиент-серверного взаимодействия, что означает инициирование запросов к серверу самим получателем, обычно веб-браузером (web-browser).</p>
<b>Что такое идиempотентный HTTP-метод?</b>	<p>В контексте API <b>идемпотентность</b> означает, что <b>многократные запросы</b> обрабатываются так же, как <b>однократные</b>. Это значит, что получив повторный запрос с теми же параметрами, в ответе должен быть результат исходного запроса.</p> <p>Такое поведение помогает избежать нежелательного повторения транзакций.</p> <p>Например, если при проведении платежа возникли проблемы с сетью, и соединение прервалось, вы сможете безопасно повторить нужный запрос неограниченное количество раз.</p> <p><b>GET</b>-запросы являются по умолчанию идиempотентными, так как <b>не имеют нежелательных последствий</b>. Для идиempотентности <b>POST</b>-запросов используется заголовок <b>"Idempotence-Key"</b> (ключ идиempотентности). Если вы повторяете запрос с теми же данными и тем же ключом, API обрабатывает его как повторный. Если данные в запросе те же, а ключ идиempотентности отличается, запрос выполняется как новый. В заголовке Idempotence-Key можно передавать любое значение, уникальное для этой операции на вашей стороне.</p> <p>Можно использовать ограниченные по времени ключи идиempотентности как дополнительное средство обеспечения безопасности использования вашего REST API.</p>
<b>Виды Statement</b>	<p><b>Statement</b> предназначен для выполнения простых SQL-запросов без параметров;</p> <p><b>PreparedStatement</b> используется для выполнения SQL-запросов с/без входных параметров; добавляет методы управления входными параметрами. Защищает от sql-инъекций.</p> <p><b>CallableStatement</b> используется для вызовов хранимых процедур; добавляет методы для манипуляции выходными параметрами.</p>
<b>CAP теорема</b>	<p>В CAP говорится, что в распределенной системе возможно выбрать только 2 из 3-х свойств:</p> <p>C (consistency) — согласованность. Каждое чтение даст вам самую последнюю запись.</p> <p>A (availability) — доступность. Каждый узел (не упавший) всегда успешно выполняет запросы (на чтение и запись).</p> <p>P (partition tolerance) — устойчивость к распределению. Даже если между узлами нет связи, они продолжают работать независимо друг от друга.</p>

<b>JMS и MOM</b>	<p><b>JMS, Java Message Service</b> - это Java API для работы с Message-Oriented Middleware(MOM), предоставляющий разработчикам возможность создавать гибкие и слабосвязанные приложения с использованием асинхронного обмена данными между приложениями (клиентами/серверами) через посредника. Асинхронность - это главная причина создания и использования JMS.</p> <p><b>MOM, Message-Oriented Middleware (промежуточное программное обеспечение)</b> - подпрограммное обеспечение промежуточного слоя, ориентированное на обмен сообщениями в распределённом окружении. Прежде всего предназначено для реализации отложенного обмена сообщениями, на основе которого и строится Messaging System.</p>
<b>Мемоизация</b>	<p>Memoization - вариант кеширования, заключающийся в том, что для функции создаётся таблица результатов. Результат функции, вычисленной при определённых значениях параметров, заносится в эту таблицу. В дальнейшем результат берётся из данной таблицы.</p> <p>Эта техника позволяет засчёт использования дополнительной памяти ускорить работу программы.</p> <p>Можно применить только к функциям, которые являются:</p> <ul style="list-style-type: none"> <li>а) детерминированными (т.е. при одном и том же наборе параметров функции должны возвращать одинаковое значение).</li> <li>б) без побочных эффектов (т.е. не должны влиять на состояние системы).</li> </ul> <p>В Java наиболее подходящей кандидатурой на роль хранилища является интерфейс Map. Сложность операций get, put, contains равна <math>O(1)</math>. Что позволяет гарантировать ограничение задержки при выполнении мемоизации.</p> <p>Мемоизация реализована в библиотеке ehcache.</p>
<b>различия между Repository и DAO:</b>	<p>краткий список тезисов, характеризующих различия между Repository и DAO:</p> <ul style="list-style-type: none"> <li>DAO инкапсулирует доступ к источнику данных</li> <li>DAO является реализацией слоя объектно-реляционного отображения</li> <li>DAO более ориентирован на источник данных</li> <li>Repository представляет более высокий уровень абстракции</li> <li>Repository выполняет роль коллекции объектов домена в памяти</li> <li>Repository ориентирован на модель предметной области</li> </ul>
<b>Статусы ответа HTTP</b>	<p>1xx: Informational (информационные):</p> <p>2xx: Success (успешно):</p> <p>3xx: Redirection (перенаправление):</p> <p>4xx: Client Error (ошибка клиента):</p> <p>5xx: Server Error (ошибка сервера):</p>

## Часто встречающиеся коды ошибок HTTP

### Обработка ошибок

400 Bad Request Универсальный код ошибки, если серверу непонятен запрос от клиента.

403 Forbidden Возвращается, если операция запрещена для текущего пользователя. Если у оператора есть учётка с более высокими правами, он должен перелогиниться самостоятельно. См. также 419

404 Not Found Возвращается, если в запросе был указан неизвестный entity или id несуществующего объекта. Списочные методы get не должны возвращать этот код при верном entity (см. выше). Если запрос вообще не удалось разобрать, следует возвращать 418.

415 Unsupported Media Type Возвращается при загрузке файлов на сервер, если фактический формат переданного файла не поддерживается. Также может возвращаться, если не удалось распарсить JSON запроса, или сам запрос пришёл не в формате JSON.

418 I'm a Teapot Возвращается для неизвестных серверу запросов, которые не удалось даже разобрать. Обычно это указывает на ошибку в клиенте, типа ошибки при формировании URI, либо что версии протокола клиента и сервера не совпадают. Этот ответ удобно использовать, чтобы отличать запросы на неизвестные URI (т.е. явные баги клиента) от ответов 404, у которых просто нет данных (элемент не найден). В отличие от 404, код 418 не бросается никаким промежуточным софтом. Альтернатива — использовать для обозначения ситуаций "элемент не найден" 410 Gone, но это не совсем корректно, т.к. предполагает, что ресурс когда-то существовал. Да и выделить баги клиента из потока 404 будет сложнее.

419 Authentication Timeout Отправляется, если клиенту нужно пройти повторную авторизацию (например, протухли куки или CSRF токены). При этом на клиенте могут быть несохранённые данные, которые будут потеряны, если просто выкинуть клиента на страницу авторизации.

422 Unprocessable Entity Запрос корректно разобрал, но содержание запроса не прошло серверную валидацию. Например, в теле запроса были указаны неизвестные серверу поля, или не были указаны обязательные, или с содержимым полей что-то не так. Обычно это означает ошибку в введённых пользователем данных, но может также быть вызвано ошибкой на клиенте или несовпадением версий.

500 Internal Server Error Возвращается, если на сервере вылетело необработанное исключение или произошла другая необработанная ошибка времени исполнения. Всё, что может сделать клиент в этом случае — это уведомить пользователя и сделать console.error(err) для более продвинутых товарищей (админов, разработчиков и тестировщиков).

501 Not Implemented Возвращается, если текущий метод неприменим (не реализован) к объекту запроса.

Алгоритм для высчитывания логарифма заданного числа по заданному основанию	<pre>static int isBitCount(double target, double osnovanie) {     return (int) (Math.log(target) / Math.log(osnovanie)) }</pre>
Рекурсивное вычисление факториала	<pre>public static int factorial(int n) {     return (n == 0) ? 1 : n * factorial(n - 1); }</pre>
Реверс числа без использования строк	<pre>public static int reverse(int numb) {     int local = 0;     while (numb &gt; 0) {         local *= 10;         local += numb % 10;         numb /= 10;     }     return local; }</pre>
Реализация бинарного дерева	<pre>class TreeNode {     int val;     TreeNode left;     TreeNode right;     TreeNode() {}     TreeNode(int val) { this.val = val; }     TreeNode(int val, TreeNode left, TreeNode right) {         this.val = val;         this.left = left;         this.right = right;     } }</pre>



Проверка симметричности  
бинарного дерева

```
public boolean isSymmetric(TreeNode root) {  
    return isSymmetricInternal(root, root);  
}  
  
private boolean isSymmetricInternal (TreeNode tn1, TreeNode tn2) {  
    if(tn1 == null || tn2 == null) {  
        return false;  
    }  
  
    if(tn1 == null && tn2 == null) {  
        return true;  
    }  
  
    return (tn1.val == tn2.val)  
        && isSymmetricInternal(tn1.left, tn2.right)  
        && isSymmetricInternal(tn1.right, tn2.left);  
}
```

Сложность  $O(n)$

Вычисление глубины дерева

```
public static int maxTreeDepth(TreeNode root) {  
    if(root == null) {  
        return 0;  
    }  
    int leftMaxDepth = maxTreeDepth(root.left);  
    int rightMaxDepth = maxTreeDepth(root.right);  
    return leftMaxDepth > rightMaxDepth ? leftMaxDepth + 1 : rightMaxDepth + 1;  
}
```

Дан массив с числами, одно  
число удалили и перемешали  
массив. Найти удаленное число.

```
int before = Arrays.stream(arrayBefore).sum();  
int after = Arrays.stream(arrayAfter).sum();  
  
System.out.println(before - after);
```

Найти общее число в трёх  
коллекциях, не используя  
дополнительные структуры

```
public static int findCommon(List<Integer> first, List<Integer> second,  
                             List<Integer> third) {  
    int wallSecond = 0;  
    int wallThird = 0;  
    for (Integer comparedEl : first) {  
        int commonCount = 0;  
  
        for (; wallSecond < second.size(); wallSecond++)  
        {  
            if (comparedEl < second.get(wallSecond)) {  
                break;  
            }  
            if (comparedEl.equals(second.get(wallSecond))) {  
                commonCount++;  
                break;  
            }  
        }  
        for (; wallThird < third.size(); wallThird++) {  
            if (comparedEl < third.get(wallThird)) {  
                break;  
            }  
            if (comparedEl.equals(third.get(wallThird))) {  
                commonCount++;  
                break;  
            }  
        }  
        if (commonCount == 2) {  
            return comparedEl;  
        }  
    }  
    return -1;  
}
```

<p>Является ли число простым</p>	<pre> public static boolean checkNumberIsPrime(int number) {     int factors = 0;     int counter = 1;      while(counter &lt;= number) {         if(number % counter == 0) {             factors++;         }         counter++;     }     return (factors == 2); } </pre>
<p>Найти нужное число в последовательности Фибоначчи</p>	<pre> //через формулу Бине public static long fibonacci(int index) {     double fi = (1 + Math.sqrt(5)) / 2;     return Math.round(Math.pow(fi, index) / Math.sqrt(5)); } </pre>
<p>Вычислить квадратный корень числа, возвращая только целую часть корня.</p>	<pre> public static int sqrt(int numb) {     for (int i = 1; i &lt; numb; i++) {         if (i*i == numb) return i;         if (i*i &gt; numb) return i-1;     }     return 0; } </pre>
<p>Последовательность Фибоначчи</p>	<pre> public static void fibonacci(int beforePreviousValue, int previousValue, int maxValue) {     if (previousValue ≤ maxValue) {         System.out.println(previousValue);         int currentValue = beforePreviousValue + previousValue;         fibonacci(previousValue, currentValue, maxValue);     } } </pre>

### пузырьковая сортировка

```
// метод пузырьковой сортировки
public static void bubbleSort(int[] num) {
    int j;
    boolean flag = true; // устанавливаем наш флаг в true для первого прохода по массиву
    int temp; // вспомогательная переменная

    while (flag) {
        flag = false; // устанавливаем флаг в false в ожидании возможного свопа (замены местами)
        for (j = 0; j < num.length - 1; j++) {
            if (num[j] < num[j + 1]) { // измените на > для сортировки по возрастанию
                temp = num[j]; // меняем элементы местами
                num[j] = num[j + 1];
                num[j + 1] = temp;
                flag = true; // true означает, что замена местами была проведена
            }
        }
    }
}
```

fizzbuzz \ «Напишите программу, которая выводит на экран числа от 1 до 100. При этом вместо чисел, кратных трем, программа должна выводить слово Fizz, а вместо чисел, кратных пяти — слово Buzz. Если число кратно пятнадцати, то программа должна выводить слово FizzBuzz. Задача может показаться очевидной, но нужно получить наиболее простое и красивое решение.»

```
public static void fizzBuzz3() {
    for (int i = 1; i <= 100; i++) {
        if (i % 15 == 0)
            System.out.println("FizzBuzz");
        else if (i % 5 == 0)
            System.out.println("Buzz");
        else if (i % 3 == 0)
            System.out.println("Fizz");
        else {
            System.out.println(i);
        }
    }
}
```

<b>Должность</b>	
<b>Проект</b>	Спросить про то, что используют; Кто проектировал, какой срок ставят и тд. Надо ли будет писать фронтенд? Чем больше вопросов уточняющих и ставящих их в неудобное положение, тем лучше
<b>ЗП</b>	Вопрос должен звучать примерно так: Сколько бюджета выделяется человеку на этой должности? До этого спросить про должность!
<b>Бэкграунд лида</b>	Вот тут надо начать узнавать про него, как можно подробнее НАПРИМЕР, опыт, экспертность в джаве, сколько лет в джаве, как силен в архитектуре
<b>Мотивации помимо зп</b>	Спросить про ДМС, компенсацию спорт зала, английский и тд.
<b>Идеальный кандидат</b>	Примерно вопрос такой: Какого человека ищите на эту должность, что он должен УМЕТЬ, что критично, а что нет)
<b>Общение</b>	Как происходит общение, как передаются таски и ТД
<b>Испытательный срок</b>	Есть ли он, возможно ли его закончить пораньше, Как они поймут, человек прошел испытательный срок или нет?
<b>Порог вхождения в адаптации</b>	Сколько месяцев занимает процесс полного понимания проекта?
<b>Конкуренты</b>	Вопрос примерно такой: Кого вы считаете своим основным конкурентом, кто вам мешает работать, хантит ваших разработчиков?
<b>Возможность перехода в другой проект</b>	Возможно ли перейти в другой проект и тд
<b>Переработки</b>	Есть ли переработки, как часто, оплачиваются ли?
<b>График</b>	По какому графику работают?
<b>Место работы</b>	Где работают, как часто в офисе, удаленка постоянная или планируете в офис вернуться...
<b>Оборудование</b>	Выдают ли ноутбук, а монитор если захочется и тд
<b>Этапы</b>	Сколько этапов, сколько из них технических, какой срок фидбека между ними и тд
<b>Команда</b>	Сколько человек в команде, сколько разработчиков, есть ли тех.лид и тд
<b>Количество человек в компании</b>	Спросить сколько всего человек в компании, сколько из них в айти отделе?
	Например, если сомневаюсь в адекватности компании, спрашиваю, какой подчиненный лучше: тот, кто делает без вопросов любые задачи и именно так, как написано, или тот, кто задаст кучу вопросов, может даже выпьет немного крови руководителя, но в итоге глубоко разберется и будет решать задачу от себя?