Table of Contents

Structures and algorithms

Brilliant introduction by Robert Sedgewick:

- Часть первая
- Часть вторая

Кормэн Алгоритмы. Построение и анализ
Thomas Cormen. Introduction to algorithms - great reference

Java

- IDE: IntellejIDEA
- Build system: Gradle Build Tool

Introduction to language:

- Java 8. Руководство для начинающих
- Java: A Beginner's Guide by Herbert Schildt
- Хороший обзор парадигмы языка - Философия Жава Эккеля
- High level overview of java language paradigm, Thinking in java by Bruce Eckel

Parallel and distributed computing

Introductionary video-courses:

- https://www.coursera.org/learn/parprog1
- https://www.coursera.org/learn/scala-spark-big-data
- General overview of big data landscape and common approach: Designing_Data-Intensive_Applications.pdf
- Java Concurrency in Practice

CVS - control version systems git + gitlab

Interactive intro

- https://try.github.io/levels/1/challenges/1
- https://git-scm.com/docs/gittutorial

Reference card

- http://rogerdudler.github.io/git-guide/
- https://www.atlassian.com/git/tutorials/atlassian-git-cheatsheet

Linux

debian based

- Монументальный труд от таненбаума тынц
- Deep and comprehensive study from Andrew Tanenbaum - modern operating systems link
- Only in russian, as far as I know. Робачевский - дает хороше и краткое(!) представление об внутреннм устройстве ОС, что весьма полезно для системного программиста - тынц
- Useful commands reference card - тынц

DevOps

docker, docker-compose - virtualisation & containerisation

- https://www.digitalocean.com/community/tutorials/the-docker-ecosystem-an-overview-of-containerization
- https://docs.docker.com/engine/docker-overview/

vagrant - handy wrappers for work with VirtualBox or VmWare
the key difference between docker - provisioning, versioning and focus on interactive work with VM.
https://www.vagrantup.com/intro/index.html[effective-java-3rd.pdf](/uploads/ccd7ccaf8f612a620253590cf59ecdac/effective-java-3rd.pdf)

Java Welcome Page

Устройство Java

Java Runtime Environment (сокр. JRE) - минимальная реализация виртуальной машины, необходимая для исполнения Java-приложений, без компилятора и других средств разработки. Состоит из виртуальной машины - Java Virtual Machine и библиотеки Java-классов.

Java Development Kit (сокращенно JDK) - комплект разработчика приложений на языке Java, включающий в себя компилятор Java (javac), стандартные библиотеки классов Java, примеры, документацию, различные утилиты и исполнительную систему Java (JRE).

Java Virtual Machine (сокращенно Java VM, JVM) - виртуальная машина Java - основная часть исполняющей системы Java, так называемой Java Runtime Environment (JRE). Виртуальная машина Java интерпретирует Байт-код Java, предварительно созданный из исходного текста Java-программы компилятором.

Загрузчики классов

- Базовый загрузчик
- Загрузчик расширений
- Системный загрузчик

Любой клас, используемый в Java программе был загружен в контекст программы каким-либо загрузчиком. Все JVM включают хотя бы один загрузчик классов, так называемый базовый загрузчик. Он загружает все основные классы.(!Уточнить, что за основные классы) Этот загрузчик никак не связан с программой, то есть мы не можжем получить например у `java.lang.Object` имя загрузчика, а метод `getClassLoader()` вернет *null*

Следующий по иерархии загрузчик - загручик рашриений, он загружает классы из `$JAVA_HOME/lib/ext` Следующий - Системный загрузчик, он загружает классы, путь к которым указан а переменной `$CLASSPATH` Для примера предположим что у нас есть некий пользовательский класс MyClass и мы его используем. Как идет его загрузка:

- Сначала системный загрузчик пытается найти его в памяти, если найден - класс успешно загружается, иначе управление загрузкой передается загрузчику расширений
- Загручик расширений также проверяет на наличие класса в памяти и в случае неудачи передает задачу базовому загрузчику.
- Базовый загрузчик снова пытается найти класс в памяти и в случае неудачи пытается его загрузить, если загрузка прошла успешно - загрузка закончена. Если нет - передает управление загрузчику расширений
- Загрузчик расширений пытается загрузить класс и в случае неудачи передает это дело системному загрузчику
- Системный загрузчик пытается загрузить класс и в случае неудачи вызывается исключение `java.lang.ClassNotFoundException`.

Если в системе присутствуют пользовательские загрузчики, то они должны быть унаследованны от класса `java.lang.ClassLoader`

Что такое статическая и динамеческая загрузка классов

Статическая загрузка класса происходит при использовании оператора `new`

Динамическая загрузка происходит в ходе выполнения программы с помощью статического метода класса `Class.forName(className)`. 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 `forName()`.

Какие механизмы обеспечивают безопасность в технологии Java

В технологии Java безопасность обеспечивают следующие три механизма:

- Структурные функциональные возможности языка (например, проверка границ массивов, запрет на преобразования непроверенных типов, отсутствие указателей и прочее).
- Средства контроля доступа, определяющие действия, которые разрешается или запрещается выполнять в коде (например, может ли код получать доступ к файлам, передавать данные по сети и прочее).
- Механизм цифровой подписи, предоставляющий авторам возможность применять стандартные алгоритмы для аутентификации своих программ, а пользователям - точно определять, кто создал код и изменился ли он с момента его подписания.

Назовите несколько видов проверок которые выполняет верификатор байт кода Java

Ниже приведены некоторые виды проверок, выполняемых верификатором.

- Инициализация переменных перед их использованием.
- Согласование типов ссылок при вызове метода.
- Соблюдение правил доступа к закрытым данным и методам.
- Доступ к локальным переменным в стеке во время выполнения.
- Отсутствие переполнения стека.

При невыполнении какой-нибудь из этих проверок класс считается поврежденным и загружаться не будет.

Что вы знаете о "диспетчере защиты" в Java

В качестве диспетчера защиты служит класс, определяющий, разрешено ли коду выполнять ту или иную операцию. Ниже перечислены операции, подпадающие под контроль диспетчера защиты. Существует немало других проверок, выполняемых диспетчером защиты в библиотеке Java.

- Создание нового загрузчика классов.
- Выход из виртуальной машины.
- Получение доступа к члену другого класса с помощью рефлексии.
- Получение доступа к файлу.
- Установление соединения через сокет.
- Запуск задания на печать.
- Получение доступа к системному буферу обмена.
- Получение доступа к очереди событий в AWT.
- Обращение к окну верхнего уровня.

Потоки ввода-вывода в Java

Потоки ввода-вывода бывают двух типов:

- Байтовый поток(`InputStream` и `OutputStream`)
- Символьный поток(`Reader` и `Writer`)

Это все абстрактные классы екораторы. которым можно добавлять дополнительный функционал. `InputStream in = new FileInputStream(new File("file.json"));`

Для чего в Java статические блоки

Статичесике блоки в Java выполняются до выполнения конструктора, с их помощью инициализируются статические переменные.

```
static final int i;
```

```
static {
    i=10;
}
```

Писать реализацию статического блока инициализации, в целом, необязательно, он создается сам при компиляции программы. И при написании `public static int VAR = 404;` Будет генерироваться следующий блок кода:

```
public static int VAR;
static {
    VAR = 404;
}
```

Можно ли перегрузить @Override static метод

~~Статические методы могут перегружаться нестатическими и наоборот - без ограничений. А вот в переопределении статического метода смысла нет.~~(!Уточнить, почему нет смысла?)

Можно ли при переопределении метода @Override что либо изменить

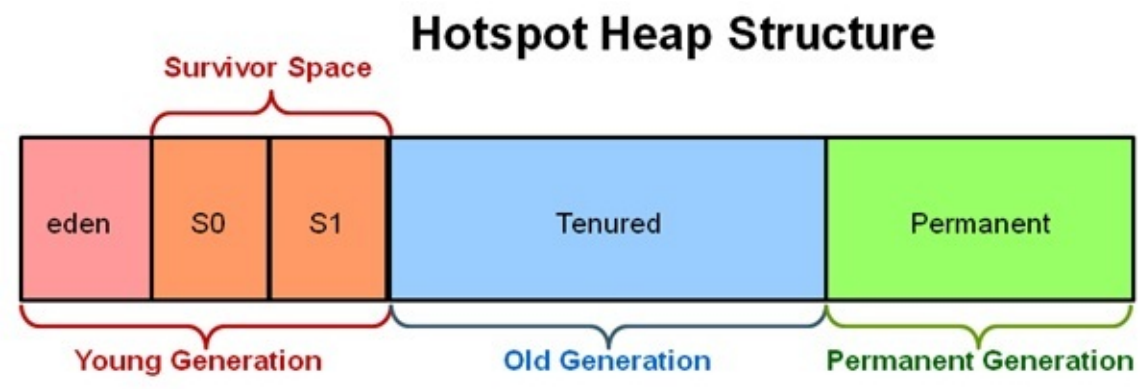
- Модификатор доступа возможно изменить только в сторону расширения
- Возвращемый тип, если выполняется Downcasting (понижающее преобразования, до более узкого типа(Наследника))
- Тип аргументов или их количество, нет это называется перегрузкой (Overload)
- Имя аргументов, можно менять без органичений
- Изменить порядок, количество или убрать секцию `throws`, возможно изменить порядок, так же возможно убрать секцию `throws`, добавлять новые исключения, наследники определенных.

Переопределение методов действует при наследовании классов, т.е. в классе наследнике объявлен метод с такой же сигнатурой что и в классе родителе. Значит этот метод переопределил метод своего суперкласса.

Несколько нюансов по этому поводу:

- Модификатор доступа в методе класса наследника должен быть НЕ приватнее чем в классе родителе, иначе будет ошибка компиляции.
- Описание исключения в переопределенном методе класса наследника должен быть НЕ шире чем в классе родителе, иначе ошибка компиляции.
- Метод обьявленный как "private" в классе родителе нельзя переопределить!

Устройство памяти в Java



Что такое Heap и Stack память в Java

Java Heap - динамически распредляемая область памяти, создаваемая при старте JVM. Используется Java Runtime для выделения памяти под объекты и JRE классы. Создание нового объекта также происходит в куче. Здесь работает сборщик мусора, освобождает память путем удаления объектов на которые нет каких-либо ссылок. Любой объект, созданный в куче, имеет глобальный доступ и на него могут ссылаться с любой части приложения.

- Все объекты обитают в куче и попадают туда при создании
- Объект состоит из полей класса и методов
- В куче выделяется место под сам объект, количество выделенной памяти зависит от полей. если полем класса. скажем служет переменная типа `int`, то не важно инициализируешь ты ее как 0 или как 1000000 объект займет в куче свои биты + столько сколько вмещает тип int + 32 бита.

Стековая память в Java работает по схеме LIFO (Последний-зашел-Первый-вышел). Всякий раз, когда вызывается метод, в памяти стека создается новый блок, который содержит примитивы и ссылки на другие объекты в методе расположенные в RAM. 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.(!Уточнить)

- Все методы находятся в стеке и попадают туда при вызове
- Переменные в методах так же имеют стековую память, поскольку они локальные
- Если в методе создается объект, то он помещается в кучу, но его ссылка все еще будет хранится в стеке и после того, как метод покинет стек, объект, который более не имеет ссылок будет уничтожен в куче.

Какая разница межу Stack и Heap паматью в Java

- Куча используется всеми частями приложения в то время как стек используется только одним потоком исполнения программы
- Всякий раз, когда создается объект, он всегда хранится в куче, а в памяти стека содержится ссылка на него. Память стека содержит только локальные переменные примитивных типов и ссылки на объекты в куче
- Объекты в куче доступны с любой точки программы, в то время как стековая память не может быть доступна для других потоков
- Управление паматью в стеке осуществляется по схеме LIFO
- Стековая память существует лишь какое-то время работы программы, а память в куче живет с самого начала до конца работы программы
- Мы можем использовать `-Xms` и `-Xmx` опции JVM, чтобы определить начальный и максимальный размер памяти в куче. Для стека определить размер памяти можно с помощью опции `-Xss`
- Если память стека полностью занята, то Java Runtime вызывает `java.lang.StackOverflowError`, а если память кучи заполнена, то вызывается исключение `java.lang.OutOfMemoryError`
- Размер памяти стека намного меньше памяти в куче. Из-за простоты распределения памяти (LIFO), стековая память работает намного быстрее кучи.

Main tesis

- Основные разделы куча (Heap) и стек
- В куче содержится статический контекст и непосредственно куча
- Куча состоит из двух частей Новая куча (Young generation Heap) и старой кучи (Tenured (Old Generation Heap))
- Новая куча тоде состоит из двух частей первая куча (Eden) и выжившая куча (Survivor Space)

Eden Space (Heap) - В этой области выделятся память под все создаваемые из программы объекты. Большая часть объектов живет недолго (итераторы, временные объекты, используемые внутри методов и т.п.), и удаляются при выполнении сборок мусора этой области памяти. Когда данная область заполняется (т.е. количество выделенной памяти в этой области превышает некоторый заданный процент), GC выполняет быструю (minor collection) сборку мусора. По сравнению с полной сборкой мусора она занимает мало времени, и затрагивает только эту область памяти - очищает от устаревших объектов Eden Space и перемещает выжившие объекты в следующую область.

Survivor Space (Heap) – Сюда перемещаются объекты из предыдущей, после того, как они пережили хотя бы одну сборку мусора. Время от времени долгоживущие объекты из этой области перемещаются в Tenured Space.

Tenured (Old Generation Heap) - Здесь скапливаются долгоживущие объекты (крупные высокоуровневые объекты, синглтоны, менеджеры ресурсов и проч.). Когда заполняется эта область, выполняется полная сборка мусора (full, major collection), которая обрабатывает все созданные JVM объекты.

Permanent Generation (Non-Heap) - Здесь хранится метаинформация, используемая JVM (используемые классы, методы и т.п.).

Garbage Collector

- Много алгоритмов работы
- Очистка происходит если память в первой куче (Eden) полностью заполнена
- Удаляет объекты с пустой ссылкой (Более не использующиеся)
- Объекты на которые остались ссылки (Survived) переносит в выжившую кучу (Survivor Space) а первую кучу просто полностью очищает
- Если в первой куче осталось мало памяти для хранения, но при этом мало объектов не имеет ссылок, помечает и удаляет мусор, остальное перекомпоновывает.
- При нехватке места в выжевшей куче (Survivor Space) объекты все еще имеющие ссылки перемещает в старую кучу (Tenured (Old Generation Heap))
- Переодически вызывается сам, без события переполнения памяти

Что произойдет со сборщиком мусора, если во время выполнения метода finalize() некоторого объекта произойдет исключение

Во время старта JVM запускается поток `finalizer`, который работает в фоне. Этот поток имеет метод `runFinalizer()`, который игнорирует все исключения методов `finalize` объектов перед сборкой мусора.

То есть если во время выполнения метода `finalize()` возникнет исключительная ситуация, его выполнение будет остановлено и это никак не скажется на работоспособности самого сборщика мусора

Каким образом передаются переменные в методы, по значению или по ссылке

В Java параметры в методы передаются по значению, фактически создаются копии параметров и с ними ведется работа в методе. В случае с примитивными типами, при передачи параметра сама переменная не будет меняться так как в метод просто копируется ее значение. А вот при передачи объекта копируется ссылка на объект, тоесть если в методе мы поменяем состояние объекта, то и за методом состояние объекта тоже поменяется. Но если мы этой копии ссылки попытаемся присвоить новую ссылку на обьект, то старая ссылка у нас не изменится. В случае передачи по значению параметр копируется. Изменение параметра не будет заметно на вызывающей стороне. В Java объекты всегда передаются по ссылке, а примитивы - по значению.

Опишите метод `Object.finalize()`

Метод `finalize()`. Java обеспечивает механизм, который может использоваться для того, чтобы произвести процесс очистки перед возвращением управления операционной системе.

Применяя метод `finalize()`, можно определять специальные действия, которые будут выполняться тогда, когда объект будет использоваться сборщиком мусора. Данный метод вызывается при уничтожении объекта автоматическим сборщиком мусора (garbage collector). В классе `Object` он ничего не делает, однако в классе наследнике позволяет описать все действия, необходимые для корректного удаления объекта, такие как закрытие соединений с БД, сетевых соединений, снятие блокировок на файлы и т.д. В обычном режиме напрямую этот метод вызывать не нужно, он отработает автоматически. Если необходимо, можно обратиться к нему явным образом.

Его синтаксис: `protected void finalize() throws Throwable`.

~~Ссылки не являются собранным мусором; только объекты - собранный мусор.~~

Чем отличаются слова `final`, `finally` и `finalize`

- `final` - Нельзя наследоваться от `final` класса. Нельзя переопределить `final` метод. Нельзя изменить значение `final` поля.
- `finally` - используется при обработке ошибок, вызывается всегда, даже если произошла ошибка(кроме `System.exit(0)`). Удобно использовать для освобождения ресурсов.

В чем разница между переменной экземпляра и статической переменной

Статические переменные инициализируются при загрузке класса ClassLoader'ом, и не зависят от объекта. Переменная экземпляра инициализируется при создании класса.

Когда используется статический метод

~~Статические методы могут быть использованы для инициализации статических переменных. Часто статические методы используются в классах утилитах, таких как Collections, Math, Arrays~~(!Уточнить)

Модификаторы доступа

В Java существуют следующие модификаторы доступа:

- private - Доступ разрешен только в текущем классе.
- default - Доступ на уровне пакета
- protected - Доступен в пакете и наследникам класса
- public - Доступен всем

Последовательность модификаторов указана по степени убывания уровня закрытости.

Что такое Package Level Access

Доступ из классов одного пакета в классы другого пакета.

Что такое статический класс и какие особенности его использования

- Статическим классом может быть только внутренний класс (Размещен внутри другого класса)
- В объекте обычного вложенного класса хранится ссылка на объект внешнего класса
- Внутри статического вложенного класса такой ссылки нет.(!Уточнить)
- Для создания объекта вложенного статического класса не требуется объект внешнего класса.
- Из объекта статического вложенного класса нельзя обратиться к нестатическим членам внешнего класса напрямую.
- Обычные вложенные классы не могут содержать статические методы и поля.

Что такое вложенный класс

Вложенный класс - это класс, который находится внутри класса или интерфейса. При этом он получает доступ ко всем полям и методам своего внешнего класса. Он применяется, чтобы обеспечить какую-то дополнительную логику класса. Хотя использование внутренних классов усложняет программу, рекомендуется избегать их использования.

Зачем нужны вложенные классы

Каждый вложенный класс может независимо наследовать определенную реализацию (То есть вложенный класс не ограничен в наследовании в ситуациях, когда внешний класс уже наследуется от какой-то реализации, то есть это решает проблему множественного наследования)

Каким образом из вложенного класса можно получить доступ к полям внешнего

Если это простой вложенный класс, то: *ИмяВнешнегоКласса.this.имяПоляВнешнегоКласса* Если это статический вложенный класс, то нужно создать объект внешнего класса, и через созданный объект получить доступ к его полям. Или же можно объявить это поле как static внутри внешнего класса

Какие существуют типы вложенных классов

Вложенные классы существуют внутри других классов. Нормальный класс - полноценный член пакета. Вложенные классы, которые стали доступны начиная с Java 1.1, могут быть четырех типов:

- Статические члены класса
- Члены класса
- Локальные классы
- Анонимные классы

Статические члены классов, как и любой другой статический метод, имеет доступ к любым статическим методам и полям своего внешнего класса, в том числе и приватым. Нестатические же могут использоваться только через ссылку на экземпляр внешнего класса.

Члены класса, локальные классы, которые были объявлены внутри блока кода. Эти классы видны только внутри блока.

Анонимные классы, классы не имеющие имени, видны только внутри блока.

Что такое JAAS

JAAS (Java Authentication and Authorization Service - служба аутентификации и авторизации Java) - служба JAAS, по существу, представляет собой встраиваемый прикладной интерфейс API, отделяющий прикладные программы на Java от конкретной технологии, применяемой для реализации средств аутентификации. Помимо прочего, эта служба поддерживает механизмы регистрации в UNIX и NT, механизм аутентификации Kerberos и механизмы аутентификации по сертификатам.

После аутентификации за пользователем может быть закреплен определенный набор полномочий. Входит в состав платформы Java начиная с версии Java SE 1.4.

В чем разница между String, StringBuffer, StringBuilder

String - неизменяемый класс, тоесть для для добавление данных в уже существующую строку, создается новый объект строки. StringBuffer и StringBuilder могут изменятся и добавление строки не такое дорогостоющее с точки зрения памяти. Первы - синхронизированный, второй - нет. Это их единственное различие. Правда если нам нужно сделать подстроку строки, то лучше использовать String, так как ее массив символов не меняется и не создается заново для новой строки. А вот в StringBuffer и StringBuilder для создания подстроки создается новый массив символов.

Какие есть оссобенности класса `String`? что делает метод `intern()`

Внутреннее состояние класса `String` нельзя изменить после его создания, т.е. этот класс неизменяемый (**immutable**) поэтому когда вы пишете `String` str = "One" + "Two"; создается три! объекта класса `String`.

От него нельзя унаследоваться, потому что класс `String` объявлен как `final`: `public final class String` Метод `hashCode()` класса `String` переписан и возвращает: `s[0]*31^(n-1) + s[1]*31^(n-2) + ... + s[n-1]` У класса `String` есть метод public `String` intern(), который возвращает строку в каноническом ее представлении из внутреннего пула строк, поддерживаемого JVM, он нужен чтобы вместо `String.equals()` использовать ==.

Понятно, что оператор сравнения ссылок выполняется гораздо быстрее, чем посимвольное сравнение строк. Используют в основном, где приходится сравнивать много строк, например в каких нибудь XML парсерах. А вообще по увеличению производительности ещё вопрос. Ибо метод `intern()` тогда должен выполняться быстрее чем `equals()`, каждый раз когда вы вызываете метод `intern()` просматривается пул строк на наличие такой строки и если такая уже есть в пуле, то возвращается ссылка на нее.Сравниваются они через `equal()`.

Types

~~Можно ли наследовать строковый тип~~

Классы объявлены `final`, поэтому наследоваться не получится

Почему строка является популярным ключом в HashMap в Java

Поскольку строки неизменны, их хэшкод кэшируется в момент создания, и не требует повторного пересчета. Это делает строки отличным кандидатом для ключа в Map и они обрабатываются быстрее, чем другие объекты-ключи HashMap. Вот почему строки преимущественно используются в качестве ключей HashMap.

Что такое конкатенация строк

Конкатенация - операция объединения строк. Результатом является объединения второй строки с окончанием первой. Операция конкатенации могут быть выполнены так:

```
StringBuffer stringBuffer = new StringBuffer();
StringBuilder stringBuilder = new StringBuilder();
String str = "Hello";

str += "World!";
String strTwo = "Hello".concat("World").concat("!");
stringBuffer.append("Hello").append("World!");
stringBuilder.append("Hello").append("World!");

Sustem.out.println(str); // HelloWorld!
Sustem.out.println(strTwo); // HelloWorld!
Sustem.out.println(stringBuffer.toString()); // HelloWorld!
Sustem.out.println(stringBuilder.toString()); // HelloWorld!
```

Как перевернуть строку

Один из способов как это можно сделать:

```
String str = "Hello World!"
StringBuilder stringBuilder = new StringBuilder(str);
stringBuilder.reverse();
System.out.println(stringBuilder.toString()); // !dlroW olleH
```

Как сравнить значение двух строк

Строка в Java - это отдельный объект, который может не совпадать с другим объектом, хотя на экране результат выводимой строки может выглядеть одинаково. Оператор == (а также !=) работает с ссылками объекта `String`. Если две переменные `String` указывают на один и тот же объект в памяти,сравнение вернет результат `true`. В противном случае результат будет `false`, несмотря на то что текст может содержать в точности такие же символы. Для сравнения посимвольно на эквивалентность необходимо использовать метод `equals()`.

```
String s1 = new String("Hello World!");
String s2 = new String("Hello World!");
String s3 = "Hello World!";
String s4 = "Hello World!";

System.out.println(s1 == s2); // false
System.out.println(s3 == s4); // true, Потому что один набор литералов будет указывать на одну область в памяти
System.out.println(s1.equals(s2)); // true
```

```
s1 = s2;
System.out.println(s1 == s2); // true
```

Как обрезать пробелы в начале и конце строки

С помощью метода `trim()` класса `String`:

```
String s = "   Hello   "
System.out.println(s.trim() + "World!") // HelloWorld!
```

Дайте определение понятию "пул строк"

Пул строк - это набор строк, который хранится в памяти Java Heap. Мы знаем, что `String` это специальный класс в Java, и мы можем создавать объекты этого класса, используя оператор new точно так же, как и создавать объекты, предоставляя значение строки в двойных кавычках. Диаграмма ниже объясняет, как пул строк размещается в памяти Java Heap и что происходит, когда мы используем различные способы создания строк.

![StringPool](../../assets/JavaCore/stringpool.jpg

Пул строк возможен исключительно благодаря неизменяемости строк в Java и реализации идеи интернирования строк.

Пул строк помогает экономить большой объем памяти, но с другой стороны создание строки занимает больше времени.

Когда мы используем двойные кавычки для создания строки, сначала ищется строка в пуле с таким же значением, если находится, то просто возвращается ссылка, иначе создается новая строка в пуле, а затем возвращается ссылка.

Тем не менее, когда мы используем оператор new, мы принуждаем класс `String` создать новый объект строки, а затем мы можем использовать метод `intern()` для того, чтобы поместить строку в пул, или получить из пула ссылку на другой объект `String` с таким же значением.

Можно ли синхронизировать доступ к строке

`String` сам по себе потокобезопасный класс. Если мы мы работаем с изменяемыми строками, то нужно использовать `StringBuffer`.

Почему строка неизменная и финализированная в Java

Есть несколько преимуществ в неизменности строк:

- Строковый пул возможен только потому, что строка неизменна в Java, таким образом виртуальная машина сохраняет много места в памяти(Heap Space), поскольку разные строковые переменные указывают на одну переменную в пуле. Если бы строка не была неизмененяемой, тогда бы интернирование строк не было бы возможным, потому что если какая-либо переменная изменит значение, это отразится также и на остальных переменных, ссылающихся на эту строку.
- Если строка будет изменяемой, тогда это станет серьезной угрозой безопасности приложения. Например, имя пользователя базы данных и пароль передаются строкой для получения соединения с базой данных и в программировании сокетов реквизиты хоста и порта передаются строкой. Так как строка неизменяемая, её значение не может быть изменено, в противном случае любой хакер может изменить значение ссылки и вызвать проблемы в безопасности приложения.
- Строки используются в Java ClassLoader и неизменность обеспечивает правильность загрузки класса при помощи ClassLoader. К примеру, задумайтесь об экземпляре класса, когда вы пытаетесь загрузить `java.sql.Connection` класс, но значение ссылки изменено на `myhacked.Connection` класс, который может осуществить нежелательные вещи с вашей базой данных.
- Поскольку строка неизменная, её **HashCode** кэшируется в момент создания и нет необходимости рассчитывать его снова. Это делает строку отличным кандидатом для ключа в **Map** и его обработка будет быстрее, чем других ключей **HashMap**. Это причина, почему строка наиболее часто используемый объект, используемый в качестве ключа **HashMap**.

Напишите метод удаления данного символа из строки

Мы можем использовать метод `replaceAll()` для замены всех вхождений в строку другой строкой. Обратите внимание на то, что метод получает в качестве аргумента строку, поэтому мы используем класс `Character` для создания строки из символа, и используем её для замены всех символов на пустую строку.

```
public static String removeChar(String str, char c) {
    return str == null ? null : str.replaceAll(Character.toString(c), "");
}
```

Для чего используются обобщенные типы в Java

~~Обобщенные типы в Java были изобретены, в первую очередь, для реализации обобщенных коллекций.~~

Что такое autoboxing (Автоупаковка)

Autoboxing/Unboxing - автоматическое преобразование между скалярными типами Java и соответствующими типами-врапперами (например, между int - Integer). Наличие такой возможности сокращает код, поскольку исключает необходимость выполнения явных преобразований типов в очевидных случаях.

Приведение типов (Понижение повышение типа) ошибка ClassCastException

Приведение типов - это установка переменной типа отличного от текущего. В Java есть два типа приведения:

- Автоматическое
- Ручное

Автоматическое происходит следующим образом (!Уточнить) byte -> short -> int -> long -> float -> double То есть, если мы расширяем тип, то явное приведение не требуется, в случае когда точность снижается (Например float -> int) то требуется явно указать приведение к типу. Объекты можно автоматически привести от наследника к родителю, наоборот вызовется исключение `ClassCastException`

Что такое Аннотации в Java

Аннотации - это своего рода метатеги, которые добавляются к коду и применяются к объявлению пакетов, классов, конструкторов, методов, полей, параметров и локальных переменных. Аннотации всегда обладают некоторой информацией и связывают эти "дополнительные данные" и все перечисленные конструкции языка.

Фактически аннотации представляют собой их дополнительные модификаторы, применение которых не влечет за собой изменений ранее созданного кода.

Какие функции выполняет Аннотации

Аннотация выполняет следующие функции:

- Дает необходимую информацию для компилятора;
- Дает информацию различным инструментам для генерации другого кода, конфигураций;
- Может использоваться во время работы кода;

Самая часто встречаемая аннотация, которую встречал любой программист, даже начинающий это `@Override`.

~~Какие встроенные аннотации в Java вы знаете~~

В языке Java SE определено несколько встроенных аннотаций, большинство из их являются специализированными. Четыре типа `@Retention`, `@Documented`, `@Target` и `@Inherited` - из пакета `java.lang.annotation`.

Из оставшиеся выделяются - `@Override`, `@Deprecated`, `@SafeVarargs` и `@SuppressWarnings` - из пакета `java.lang.` Широкое использование аннотаций в различных технологиях и фреймворках обуславливается возможностью сокращения кода и снижения его связанности.

Что делают аннотации `@Retention`, `@Documented`, `@Target` и `@Inherited`

Эти аннотации, имеют следующее значение:

- `@Retention` - эта аннотация предназначена для применения только в качестве аннотации к другим аннотациям, позволяет указать жизненный цикл аннотации: будет она присутствовать только в исходном коде, в скомпилированном файле, или она будет также видна и в процессе выполнения. Выбор нужного типа зависит от того, как вы хотите использовать аннотацию.
- `@Documented` - это маркер-интерфейс, который сообщает инструменту, что аннотация должна быть документирована.
- `@Target` - эта аннотация задает тип объявления, к которым может быть применима аннотация. Принимает один аргумент, который должен быть константой из перечисления ElementType, это может быть поле, метод, тип и т.д. Например,чтобы указать, что аннотация применима только к полям и локальным переменным: @Targer({ ElementType.FIELD, ElementTyle.LOCAL_VARIABLE })
- `@Inherited` - это аннотация-маркер, которая может применяться в другом объявление аннотации, она касается только тех аннотаций, что будут использованы в объявлениях классов. Эта аннотация позволяет аннотации супер класса быть унаследованной в подклассе.

Что делают аннотации `@Override`, `@Deprecated`, `@SafeVarargs` и `@SuppressWarnings`

Эти аннотации предназначены для:

- `@Override` - аннотация-маркер, которая может применяться только к методам. Метод, аннотированный как `@Override`, должен переопределять метод супер класса.
- `@Deprecated` - указывает, что объявление устарело и должно быть заменено более новой формой.

- `@SafeVarargs` - аннотация-маркер, применяется к методам и конструкторам. Она указывает, что никакие небезопасные действия, связанные с параметром переменного количества аргументов, недопустимы. Применяется только к методам и конструкторам с переменным количеством аргументов, которые объявлены как `static` или `final`.
- `@SuppressWarnings` - эта аннотация указывает, что одно или более предупреждений, которые могут быть выданы компилятором следует подавить.

Какой жизненный цикл аннотации можно указать с помощью @Retention

Существует 3 возможные варианты чтобы указать где аннотация будет жить. Они инкапсулированы в перечисление `java.lang.annotation.RetentionPolicy`. Это **SOURSE**, **CLASS**, **RUNTIME**.

- **SOURCE** - содержаться только в исходном файле и отбрасываются при компиляции.
- **CLASS** - сохраняются в файле, однако они недоступны JVM во время выполнения.
- **RUNTIME** - сохраняются в файле во время компиляции и остаются доступными JVM во время выполнения.

К каким элементам можно применять аннотацию, как это указать

Для того чтобы ограничить использование аннотации её нужно проаннотировать. Для этого существует аннотация `@Target`.

- `@Target(ElementType.PACKAGE)` - только для пакетов
- `@Target(ElementType.TYPE)` - только для классов
- `@Target(ElementType.CONSTRUCTOR)` - только для конструкторов
- `@Target(ElementType.METHOD)` - только для методов
- `@Target(ElementType.FIELD)` - только для атрибутов(переменных) класса
- `@Target(ElementType.PARAMATER)` - только для параметров метода
- `@Target(ElementType.LOCAL_VARIABLE)` - только для локальных переменных

В случае если вы хотите, что бы ваша аннотация использовалась больше чем для одного типа параметров, то можно указать @Target следующим образом:

`@Target({ElementType.PARAMETER, ElementType.LOCAL_VARIABLE})`

Тут мы говорим, аннотацию можно использовать только для параметров метода и для локальных переменных.

Как создать свою Аннотацию

Написать свою аннотацию не так сложно, как могло бы казаться. В следующем коде приведено объявление аннотации.

```
public @interface About {
    String info() default "";
}
```

Как видно на месте где обычно пишут `class` или `interface` у нас написано `@interface`.

Структура практически та же, что и у интерфейсов, только пишется `@interface`.

`@interface` - указывает на то, что это аннотация `default` - говорит про то, что метод по умолчанию будет возвращать определённое значение.

~~Конфигурация аннотаций~~

Атрибуты каких типов допустимы в аннотациях

Атрибуты могут иметь только следующие типы:

- `String`
- `Class` или «any parameterized invocation of Class»

- `Enum`
- `Annotation`
- Одномерный массив любого из вышеперечисленных типов

Иерархия исключений

Все классы-исключения расширяют класс `Throwable` - непосредственное расширение класса object. У класса `Throwable` и у всех его расширений по традиции два конструктора:

- `Throwable e` - конструктор по умолчанию
- `Throwable (String message)` - создаваемый объект будет содержать произвольное сообщение `message`

Записанное в конструкторе сообщение можно получить затем методом `getMessage()`. Если объект создавался конструктором по умолчанию, то данный метод возвратит `null`.

Метод `toString()` возвращает краткое описание события, именно он работал в предыдущих листингах. Три метода выводят сообщения обо всех методах, встретившихся по пути "полета" исключения:

- `printstackTrace()` - выводит сообщения в стандартный вывод, как правило, это консоль
- `printStackTrace(PrintStream stream)` - выводит сообщения в байтовый поток stream
- `printStackTrace(PrintWriter stream)` - выводит сообщения в символьный поток stream

У класса `Throwable` два непосредственных наследника - классы `Error` и `Exception`. Они не добавляют новых методов, а служат для разделения классов-исключений на два больших семейства - семейство классов-ошибок (`Error`) и семейство собственно классов-исключений (`Exception`).

Классы ошибки, расширяющие класс `Error`, свидетельствуют о возникновении сложных ситуаций в виртуальной машине Java. Их обработка требует глубокого понимания всех тонкостей работы JVM. Ее не рекомендуется выполнять в обычной программе. Не советуют даже выбрасывать ошибки оператором throw. Не следует делать свои классы-исключения расширениями класса `Error` или какого-то его подкласса.

Имена классов-ошибок, по соглашению, заканчиваются словом `Error`.

Классы-исключения, расширяющие класс `Exception`, отмечают возникновение обычной нештатной ситуации, которую можно и даже нужно обработать. Такие исключения следует выбросить оператором `throw`.

Классов-исключений очень много, более двухсот. Они разбросаны буквально по всем пакетам J2SDK. В большинстве случаев вы способны подобрать готовый класс-исключение для обработки исключительных ситуаций в своей программе. При желании можно создать и свой класс-исключение, расширив класс `Exception` или любой его подкласс.

Среди классов-исключений выделяется класс `RuntimeException` - прямое расширение класса `Exception`. В нем и его подклассах отмечаются исключения, возникшие при работе JVM, но не столь серьезные, как ошибки. Их можно обрабатывать и выбрасывать, расширять своими классами, но лучше доверить это JVM, поскольку чаще всего это просто ошибка в программе, которую надо исправить. Особенность исключений данного класса в том, что их не надо отмечать в заголовке метода пометкой `throws`. Имена классов-исключений, по соглашению, заканчиваются словом `Exception`.

Какие есть виды исключений в Java, чем они отличаются

Все исключительные ситуации можно разделить на две категории: проверяемые(**checked**) и непроверяемые(**unchecked**).

Все исключения, порождаемые от `Throwable`, можно разбить на три группы. Они определяются тремя базовыми типами: наследниками `Throwable` - классами `Error` и `Exception`, а также наследником `Exception` - `RuntimeException`.

Ошибки, порожденные от `Exception` (и не являющиеся наследниками `RuntimeException`), являются проверяемыми, во время компиляции проверяется, предусмотрена ли обработка возможных исключительных ситуаций. Как правило, это ошибки, связанные с окружением программы (сетевым, файловым вводом-выводом и др.), которые могут возникнуть вне зависимости от того, корректно написан код или нет. Например, открытие сетевого соединения или файла может привести к возникновению ошибки и компилятор требует от программиста предусмотреть некие действия для обработки возможных проблем. Таким образом повышается надежность программы, ее устойчивость при возможных сбоях.

Исключения, порожденные от `RuntimeException`, являются непроверяемыми и компилятор не требует обязательной их обработки.

Как правило, это ошибки программы, которые при правильном кодировании возникать не должны (например, `IndexOutOfBoundsException` - выход за границы массива, `java.lang.ArithmeticException` - деление на ноль). Поэтому, чтобы не загромождать программу, компилятор оставляет на усмотрение программиста обработку таких исключений с помощью блоков `try-catch`.

Исключения, порожденные от `Error`, также не являются проверяемыми. Они предназначены для того, чтобы уведомить приложение о возникновении фатальной ситуации, которую программным способом устранить практически невозможно (хотя формально обработчик допускается). Они могут свидетельствовать об ошибках программы, но, как правило, это неустранимые проблемы на уровне JVM. В качестве примера можно привести `StackOverflowError` - (переполнение стека), `OutOfMemoryError` - (нехватка памяти).

Методы, код которых может порождать проверяемые исключения, должны либо сами их обрабатывать, либо в заголовке метода должно быть указано ключевое слово `throws` с перечислением необрабатываемых проверяемых исключений. На непроверяемые ошибки это правило не распространяется. Переопределенный (**overridden**) метод не может расширять список возможных исключений исходного метода.

Что такое Checked и Unchecked Exception

Checked исключения, это те, которые должны обрабатываться блоком `catch` или описываться в сигнатуре метода. **Unchecked** могут не обрабатываться и не быть описанными.

Unchecked исключения в Java - наследованные от `RuntimeException`, **Checked** - от `Exception`. Пример: **Unchecked** исключения - `NullPointerException`, **Checked** исключения - `IOException`.

Какие есть Unchecked Exeption

- `ArithmeticException` - Арифметическая ошибка: деление на ноль
- `ArrayIndexOutOfBoundsException` - Индекс выходит за пределы массива
- `ArrayStoreException` - Присвоение элементу массива значения несовместимого типа
- `ClassCastException` - Недопустимое приведение к типу
- `ConcurrentModificationException` - Некорректная модификация коллекции
- `IllegalArgumentException` - При вызове метода ипользован неправильный аргумент
- `IllegalMonitorStateException` - ~~Неверная операция на экземпляре~~
- `IllegalStateException` - ~~Среда или приложение находятся в некорректном состоянии~~
- `IllegalThreadStateException` - ~~Операция не совместима с текущим состоянием потока~~
- `IndexOutOfBoundsException` - ~~Индекс выходит за пределы~~
- `NegativeArraySizeException` - Массив был создан с отрицательным размером
- `NullPointerException` - Недопустимое использование нулевой ссылки
- `NumberFormatException` - Невозможно преобразовать строку в числовой формат
- `StringIndexOutOfBoundsException` - Индекс выходит за пределы строки
- `UnsupportedOperationException` - Неподдерживаемая операция

Что такое `Error`

Исключения, порожденные от `Error`, не являются проверяемыми. Они предназначены для того, чтобы уведомить приложение о возникновении фатальной ситуации, которую программным способом устранить практически невозможно (хотя формально обработчик допускается). Они могут свидетельствовать об ошибках программы, но, как правило, это неустранимые проблемы на уровне JVM. В качестве примера можно привести `StackOverflowError` (переполнение стека), `OutOfMemoryError` (нехватка памяти).

Методы, код которых может порождать проверяемые исключения, должны либо сами их обрабатывать, либо в заголовке метода должно быть указано ключевое слово throws с перечислением необрабатываемых проверяемых исключений. На непроверяемые ошибки это правило не распространяется.

~~Что такое `Exception`~~

Принцип работы блока try-catch-finally

Если срабатывает один из блоков `catch`, то остальные блоки в данной конструкции `try-catch` выполняться не будут.

Свойством транзакционности исключения не обладают - действия, произведенные в блоке `try` до возникновения исключения, не отменяются после его возникновения.

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`. Это может быть полезно, например, в следующей ситуации.

При выходе из метода вам надо произвести какое-либо действие. А `return` в этом методе стоит в нескольких местах. Писать одинаковый код перед каждым `return` нецелесообразно. Гораздо проще и эффективнее поместить основной код в `try`, а код, выполняемый при выходе - в `finally`.

Всегда ли исполняется блок finally

Не всегда например в следующих ситуациях:

- Существуют потоки-демоны - потоки предоставляющие некие сервисы, работая в фоновом режиме во время выполнения программы, но при этом не являются ее неотъемлеммой частью.Таким образом когда все потоки не демоны завершаются, программа завершает свою работу. В потоках демонах блок `finally` не выполняеться, они прерываются внезапно.
- `System.exit(0)`
- Если в блоке `finally` произошло исключение и нет обработчика, то оставшийся код в блоке `finally` может не выполнятся.

Что такое Дженерики (Generics)

Это технический термин, обозначающий набор свойств языка позволяющих определять и использовать обобщенные типы и методы. Обобщенные типы или методы отличаются от обычных тем, что имеют типизированные параметры. Примером дженериков или обобщенных типов может служить библитека с коллекциями в Java. Например, класс `LinkedList<E>` - типичный обобщенный тип. Он содержит параметр E, который представляет тип элементов, которые будут храниться в коллекции. Вместо того, чтобы просто использовать LinkedList, ничего не говоря о типе элемента в списке, мы можем использовать `LinkedList<String>` или `LinkedList<Integer>`. Создание объектов обобщенных типов происходит посредством замены параметризированных типов реальными типами данных. Класс типа `LinkedList<E>` - обобщенный тип, который содержит параметр E. Создание объектов, типа `LinkedList<String>` или `LinkedList<Integer>` называются параметризированными типами, а String и Integer - реальные типы аргументов.

Почему в некоторых интерфейсах не определяют методов

Это так называемые интерфейсы - маркеры. Они просто указывают что класс относится к определенной группе классов. Например интерфейс `Clonable` указывает на то, что класс поддерживает механизм клонирования.

Интерфейсы маркеры в Java:

- `Serializable`
- `Cloneable`
- `Remote`
- `ThreadSafe`

Чем абстрактный класс отличается от интерфейса

~~- Экземпляр абстрактного класса нельзя создать Абстрактный класс это класс, который помечен как "abstract", он может содержать абстрактные методы, а может их и не содержать. Класс, который наследуется от абстрактного класса может реализовывать абстрактные методы, а может и не реализовывать, тогда класс наследник должен быть тоже абстрактным. Также если класс наследник переопределяет реализованный в абстрактном классе родители метод, его можно переопределить с модификатором абстракт! Т.е отказаться от реализации. Соответственно данный класс должен быть также абстрактным также. Что касается интерфейса, то в нем находятся только абстрактные методы и константы, так было до выхода Java 8. Начиная с Java 8 кроме абстрактных методов мы также можем использовать в интерфейсах стандартные методы (default methods) и статические методы (static methods)

- Default метод в интерфейсе - это метод в интерфейсе с по умолчанию реализованной логикой, который не требуется обязательно определять в реализации этого интерфейса.
- Static методы в интерфейсе - это по существу то же самое, что static-методы в абстрактном классе. При реализации интерфейса, класс обязан реализовать все методы интерфейса. Иначе класс должен быть помечен как абстрактный. Интерфейс также может содержать внутренние классы. И не абстрактные методы в них. При реализации интерфейса, класс обязан реализовать все методы интерфейса. Иначе класс должен быть помечен как абстрактный. Интерфейс также может содержать внутренние классы. И не абстрактные методы в них. ~~

JavaIO

Что такое JMX

Управленческие расширения Java (Java Management Extensions, JMX) - API при помощи которого можно контролировать работу приложений и управлять различными параметрами удаленно в реальном времени. Причем управлять можно фактически чем угодно - лишь бы это было написано на Java. Это может быть микро-устройство типа считывателя отпечатка или система, включающая тысячи машин, каждая из которых предоставляет определенные сервисы. Данные ресурсы представляются MBean-объектами (управляемый Java Bean). JMX вошла в поставку Java начиная с версии 5.

Какие выгоды предлагает JMX

Вот как эти выгоды описывает Sun

- Простота реализации. Архитектура JMX основана на понятии "сервера управляемых объектов" который выступает как управляющий агент и может быть запущен на многих устройствах/компьютерах, которые поддерживают Java.
- Масштабируемость. Службы агентов JXM являются независимыми и могут быть встроены наподобие плагинов в агенте JMX. Компонентно-основанаая система позволяет создавать масштабируемые решения от крохотных устройств до очень крупных систем.
- Возможность расширять концепцию в будущем. JMX позволяет создавать гибкие решения. Например, JMX позволяет создавать удобные решения, которые могут находить различные сервисы.
- Концентрация на управлении. JMX предоставляет сервися, разработанные для работы в распределенных средах и его API спроектировано для решений, которые управляют приложениями, сетями, сервисами

Что еще умеет JMX кроме дистанционного управления

JMX делает гораздо больше, чем просто предоставляет рабочую оболочку для дистанционного управления. Она обеспечивает дополнительные услуги (services),способные занять ключевое место в процессе разработки. Приведу лишь краткое описание:

- Event notification: Интерфейсы оповещают исполнителей и слушателей о событиях типа изменения атрибута, что позволяет MBean-компонентам общаться с другими MBean-компонентами или удалённым "командным пунктом" и докладывать об изменениях своего состояния
- Monitor service: Monitor MBeans может посылать уведомления о событиях зарегистрированным слушателям. Слушателем может выступать другой MBean или управляющее приложение. В качестве основных атрибутов, для которых используется данное свойство, являются counter, gauge или string.
- Timer service: Timer MBean будет посылать уведомления зарегистрированным слушателям, с учётом определённого числа или временного промежутка.
- M-let service: M-let service может создавать и регистрировать экземпляры MBean-серверов. Список MBean-компонентов и имён из классов определяются в m-let файле с помощью MLET меток. URL указывает на месторасположения m-let файла.

Что такое MBean

MBeans - это Java-объекты, которые реализуют определенный интерфейс. Интерфейс включает:

- Некие величины, которые могут быть доступны
- Операции, которые могут быть вызваны
- Извещения, которые могут быть посланы
- Конструкторы

Какие типы MBeans существуют

Существует 4 типа MBeans:

- Standard MBeans. Самые простые бины. Их управляющий интерфейс определяется набором методов
- Dynamic MBeans. Они реализуют специализированный интерфейс, который делают доступным во время исполнения.
- Open MBeans. Это Dynamic MBeans, которые используют только основные типы данных для универсального управления.
- Model MBeans. Это Dynamic MBeans, которые полностью конфигурируемы и могут показать свое описание во время исполнения (нечто вроде `Reflection`)

Что такое MBean Server

MBean Server - это реестр объектов, которые используются для управления. Любой объект зарегистрированный на сервере становится доступным для приложений. Надо отметить, что сервер публикует только интерфейсы и не дает прямых ссылок на объекты. Любые ресурсы, которыми вы хотите управлять должны быть зарегистрированы на сервере как MBean. Сервер предоставляет стандартный интерфейс для доступа к MBean.

Интересно, что регистрировать MBean может любой другой MBean, сам агент или удаленное приложение через распределенные сервисы. Когда вы регистрируете MBean вы должны дать ему уникальное имя, которое будет использовано для обращения к даному объекту.

Objects

Какие методы есть у класса Object

~~Object это базовый класс для всех остальных объектов в Java. Каждый класс наследуется от Object. Соответственно все классы наследуют методы класса Object.~~(!Уточнить) Методы класса Object:

- public final native Class getClass()
- public native int hashCode()
- public boolean equals(Object obj)
- protected native Object clone() throws CloneNotSupportedException
- public String toString()
- public final native void notify()
- public final native void notifyAll()
- public final native void wait(long timeout) throws InterruptedException
- public final void wait(long timeout, int nanos) throws InterruptedException
- public final void wait() throws InterruptedException
- protected void finalize() throws Throwable

Правила переопределения метода Object.equals()

- Используйте оператор == что бы проверить ссылку на объект, переданную в метод equals. Если ссылки совпадают - вернуть true. Это не обязательно, нужно скорее для оптимизации, но может сэкономить время в случае "тяжёлых" сравнений.
- Используйте оператор instanceof для проверки типа аргумента. Если типы не совпадают, вернуть false.
- Преобразуйте аргумент к корректному типу. Так как на предыдущем шаге мы выполнили проверку, преобразование корректно.
- Пройтись по всем значимым полям объектов и сравнить их друг с другом. Если все поля равны - вернуть true. Для сравнения простых типов использовать ==. Для полей со ссылкой на объекты использовать equals, float преобразовывать в int с помощью Float.floatToIntBits и сравнить с помощью ==, double преобразовывать в long с помощью Double.doubleToLongBits и сравнить с помощью ==. Для коллекций вышеперечисленные правила применяются к каждому элементу коллекции. Нужно учитывать возможность null полей/объектов. Очерёдность сравнения полей может существенно влиять на производительность.
- Закончив реализацию equals задайте себе вопрос, является ли метод симметричным, транзитивным и не противоречивым.

И ещё несколько дополнительных правил:

- Переопределив equals, всегда переопределять hashCode
- Не использовать сложную семантику в equals (типа определения синонимов), equals должен сравнивать поля объектов, не более.

Какие условия должны удовлетворяться для переопределенного метода equals()

Метод equals() обозначает отношение эквивалентности объектов. Эквивалентным называется отношение, которое является симметричным, транзитивным и рефлексивным.

- Рефлексивность: для любого ненулевого x, x.equals(x) вернет true
- Транзитивность: для любого ненулевого x, y и z, если x.equals(y) и y.eqals(z) вернет true, тогда и x.equals(z) вернет true
- Симметричность: для любого ненулевого x и y, x.equals(y) должно вернуть true, тогда и только тогда, когда y.equals(x) вернет true

Также для любого ненулевого x, x.equals(null) должно вернуть false.

Какая связь между hashCode и equals

Объекты равны, когда `a.equals(b)=true и a.hashCode==b.hashcode -> true` Но необязательно, чтобы два различных объекта возвращали различные хэш-коды(такая ситуация называется коллизией).

Каким образом реализованы методы hashCode и equals в классе Object

Реализация метода equals в классе Object сводится к проверке на равенство двух ссылок:

```
public boolean equals(Object obj) {
    return (this == obj);
}
```

Реализация же метода hashCode класса Object сделана нативной, т.е. определенной не с помощью Java кода:

```
public native int hashCode();
```

Он обычно возвращает адрес объекта в памяти.

Что будет, если переопределить equals не переопределяя hashCode? Какие могут возникнуть проблемы

Они будут неправильно хранится в контейнерах, использующих хэш коды, таких как HashMap, HashSet. Например HashSet хранит элементы в случайном (на первый взгляд) порядке. Дело в том, что для быстрого поиска HashSet рассчитывает для каждого элемента hashCode и именно по этому ключу ищет и упорядочивает элементы внутри себя

Есть ли какие-либо рекомендации о том, какие поля следует использовать при подсчете hashCode

Необходимо использовать уникальные, лучше примитивные поля, такие как id, uuid, например. Причем если эти поля задействованы при вычислении hashCode, то нужно их задействовать при выполнении equals. Общий совет: выбирать поля, которые с большой долью вероятности будут различаться.

Для чего нужен метод `hashCode()`

Существуют коллекции(HashMap, HashSet), которые используют хэш код, как основу при работе с объектами. А если хэш для равных объектов будет разным, то в HashMap будут два равных значения, что является ошибкой. Поэтому необходимо соответствующим образом переопределить метод hashCode().

- Хеширование - преобразование входного массива данных произвольной длины в выходную битовую строку фиксированной длины. Такие преобразования также называются хеш-функциями или функциями свёртки, а их результаты называют хешем или хеш-кодом.
- Хеш-таблицы - это структура данных, реализующая интерфейс ассоциативного массива, а именно, она позволяет хранить пары (ключ, значение) и выполнять три операции: операцию добавления новой пары, операцию поиска и операцию удаления пары по ключу.

Выполнение операции в хеш-таблице начинается с вычисления хеш-функции от ключа. Получающееся хеш-значение `i = hash(key)` играет роль индекса в массиве H. Затем выполняемая операция (добавление, удаление или поиск) перенаправляется объекту, который хранится в соответствующей ячейке массива `H[i]`. Одним из методов построения хеш-функции есть метод деления с остатком (division method) состоит в том, что ключу k ставится в соответствие остаток от деления k на m, где m - число возможных хеш-значений.

Правила переопределения метода Object.hashCode()

При реализации hashCode используется несколько простых правил. Прежде всего, при вычислении хеш-кода следует использовать те же поля, которые сравниваются в equals. Это, во-первых, даст равенство хеш-кодов для равных обектов, во-вторых, распределено полученное значение будет точно так же, как и исходные данные. Теоретически, можно сделать так, чтобы хеш-код всегда был равен 0, и это будет абсолютно легальная реализация. Другое дело, что ее ценность будет равна тому же самому нулю. Несмотря на то, что хеш-коды равных объектов должны быть равны, обратное неверно! Два неравных объекта могут иметь равные хеш-коды. Решающее значение имеет не уникальность, а скорость вычисления, потому как это приходится делать очень часто. Потому, в некоторых случаях имеет смысл посчитать хеш-код заранее и просто выдавать его по запросу. Прежде всего это стоит делать тогда, когда вычисление трудоемко, а объект неизменен.

Клонирование объектов. В чем отличие между поверхностным и глубоким клонированием

Чтобы объект можно было клонировать, он должен реализовать интерфейс ~~Cloneable(маркер)~~. Использование этого интерфейса влияет на поведение метода `clone()` класса Object. Таким образом `myObj.clone()` создаст нам клон нашего объекта, но этот клон будет поверхностный. Что значит поверхностным? Это значит что клонируется только примитивные поля класса, ссылочные поля клонироваться не будут!

Для того, чтоб произвести глубокое клонирование, необходимо в клонируемом классе переопределить метод `clone()`, и в нем произвести клонирование изменяемых полей объекта.

Правила переопределения метода `Object.clone()`

Метод `clone()` в Java используется для клонирования объектов. Так как Java работает с объектами с помощью ссылок, то простым присваиванием тут не обойдешься, ибо в таком случае копируется лишь адрес, и мы получим две ссылки на один и тот же объект, а это не то что нам нужно. Механизм копирования обеспечивает метод `clone()` класса `Object`.

Метод `clone()` действует как конструктор копирования. Обычно он вызывает метод `clone()` суперкласса и т.д. пока не дойдет до `Object`.

Метод `clone()` класса `Object` создает и возвращает копию объекта с такими же значениями полей. `Object.clone()` кидает исключение `CloneNotSupportedException` если вы пытаетесь клонировать объект не реализующий интерфейс `Cloneable`. Реализация по умолчанию метода `Object.clone()` выполняет неполное/поверхностное (shallow) копирование. Если вам нужно полное/глубокое (deep) копирование класса, то в методе `clone()` этого класса, после получения клона суперкласса, необходимо скопировать нужные поля.

Синтаксис вызова `clone()` следующий:

```
Object copy = obj.clone();

// Или

MyClass copy = (MyClass) obj.clone();
```

Один из недостатков метода `clone()`, это тот факт, что возвращается тип `Object`, поэтому требуется нисходящее преобразование типа. Однако начиная с версии Java 1.5 при переопределении метода вы можете сузить возвращаемый тип.

Пару слов о `clone()` и `final` полях. Метод `clone()` несовместим с `final` полями. Если вы попробуете клонировать `final` поле компилятор остановит вас. Единственное решение - отказаться от `final`.

Где и как можно использовать закрытый конструктор

Например в качестве паттерна `Singleton`. В том же классе создается статический метод. Где и создается экземпляр класса, конечно если он уже не создан, тогда он просто возвращается методом.

Что такое конструктор по умолчанию

В Java если нет явным образом определённых конструкторов в классе, то компилятор использует конструктор по умолчанию, определённый неявным способом, который аналогичен "чистому", конструктору по умолчанию. Конструктор по умолчанию - это довольно простая конструкция, которая сводится к созданию для типа конструктора без параметров. Так, например, если при объявлении нестатического класса не объявить пользовательский конструктор, (не важно, с параметрами или без них), то компилятор самостоятельно сгенерирует конструктор без параметров. Некоторые программисты явным образом задают конструктор по умолчанию по привычке, чтобы не забыть в дальнейшем, но это не обязательно.

В Java если производный класс не вызывает явным образом конструктор базового класса (в Java используя `super()` в первой строчке), то конструктор по умолчанию вызывается неявно. Если базовый класс не имеет конструктора по умолчанию, то это считается ошибкой.

Может ли объект получить доступ к private переменной класса

Доступ у приватной переменной класса можно получить только внутри класса, в котором она объявлена. Также доступ к приватным переменным можно осуществить через механизм Java Reflection API.

Reflection

Что такое рефлексия

Рефлексия используется для получения или модификации информации о типах во время выполнения программы. Этот механизм позволяет получить сведения о классах, интерфейсах, полях, методах, конструкторах во время исполнения программы. При этом не нужно знать имена классов, методов или интерфейсов. Также этот механизм позволяет создавать новые объекты, выполнять методы и получать и устанавливать значения полей.

Object Relational Mapping (ORM) Hibernate

Что такое Hibernate

Это фреймворк для объектно-реляционного отображения сущностей в традиционные реляционные базу данных. Основные возможности фреймворка: Автоматическая генерация и обновление таблиц в базах данных; Поскольку основные запросы к базе данных (сохранение, обновление, удаление и поиск) представлены как методы фрейморка, то значительно сокращается код, который пишется разработчиком; Обеспечивает использование SQL подобного языка (HQL - Hibernate Query Language). Запросы HQL могут быть записаны рядом объектами данных (`POJO` классы подготовленные для работы с базой данных).

Что такое ORM

ORM (англ. Object Relational Mapping) - технология программирования, которая связывает базы данных с концепциями объектно-ориентированных языков программирования, создавая "виртуальную объектную базу данных".

Какие преимущства от использования Hibernate

Некоторые из них:

- Устраняет множество повторяющегося кода, который постоянно преследует разработчика при работе с JDBC. Скрывает от разработчика множество кода, необходимого для управления ресурсами и позволяет сосредоточиться на бизнес логике.
- Поддерживает XML так же как и JPA аннотации, что позволяет сделать реализацию кода независимой.
- Предоставляет собственный мощный язык запросов (HQL), который похож на SQL. Стоит отметить, что HQL полностью объектно-ориентирован и понимает такие принципы, как наследование, полиморфизм и ассоциации (связи).
- Hibernate легко интегрируется с другими Java EE фреймворками, например, Spring Framework поддерживает встроенную интеграцию с Hibernate.
- Поддерживает ленивую инициализацию используя proxy объекты и выполняет запросы к базе данных только по необходимости.
- Поддерживает разные уровни cache, а следовательно может повысить производительность.
- Важно, что Hibernate может использовать чистый SQL, а значит поддерживает возможность оптимизации запросов и работы с любым сторонним вендором БД.
- Hibernate - open source проект. Благодаря этому доступны тысячи открытых статей, примеров, а так же документации по использованию фреймворка.

Как Hibernate помогает в программировании

Hibernate реализует ряд фичь которые значительно упрощают работу разработчика. Одной из таких фич является то, что Hibernate позволяет разработчику избежать написания большинства SQL запросов (они уже реализованы , вам надо просто использовать методы которые предоставляет фреймворк).

Под бортом у Hibernate есть куча полезных инструментов которые значительно ускоряют работу приложения, самыми примечательными из них являются двухуровневое кэширования и тонкие настройки lazy и fetch изъятия. А так же, сам генерирует таблицы в базу данных.

Какие преимущества Hibernate над JDBC

Hibernate имеет ряд преимуществ перед JDBC API:

- Hibernate удаляет множество повторяющегося кода из JDBC API, а следовательно его легче читать, писать и поддерживать.
- Hibernate поддерживает наследование, ассоциации и коллекции, что не доступно в JDBC API.
- Hibernate неявно использует управление транзакциями. Большинство запросов нельзя выполнить вне транзакции. При

использовании JDBC API для управления транзакциями нужно явно использовать commit и rollback.
- JDBC API throws SQLException, которое относится к проверяемым исключениям, а значит необходимо постоянно писать множество блоков try-catch. В большинстве случаев это не нужно для каждого вызова JDBC и используется для управления транзакциями. Hibernate оборачивает исключения JDBC через непроверяемые JDBCException или HibernateException, а значит нет необходимости проверять их в коде каждый раз. Встроенная поддержка управления транзакциями в Hibernate убирает блоки try-catch.
- Hibernate Query Language (HQL) более объектно ориентированный и близкий к Java язык программирования, чем SQL в JDBC.
- Hibernate поддерживает кэширование, а запросы JDBC - нет, что может понизить производительность.
- Hibernate предоставляет возможность управления БД (например создания таблиц), а в JDBC можно работать только с существующими таблицами в базе данных.
- Конфигурация Hibernate позволяет использовать JDBC вроде соединения по типу JNDI DataSource для пула соединений. Это важная фича для энтерпрайз приложений, которая полностью отсутствует в JDBC API.
- Hibernate поддерживает аннотации JPA, а значит код является переносимым на другие ORM фреймворки, реализующие стандарт, в то время как код JDBC сильно привязан к приложению.

Что такое конфигурационный файл Hibernate

Файл конфигурации Hibernate содержит в себе данные о базе данных и необходим для инициализации SessionFactory. В .xml файле необходимо указать вендора базы данных или JNDI ресурсы, а так же информацию об используемом диалекте, что поможет hibernate выбрать режим работы с конкретной базой данных.

Способы конфигурации работы с Hibernate

Существует четыре способа конфигурации работы с Hibernate :

- Annotation;
- Hibernate.cfg.xml
- Hibernate.properties
- Persistence.xml

Самый частый способ конфигурации: через аннотации и файл persistence.xml, что касается файлов hibernate.properties и hibernate.cfg.xml, то hibernate.cfg.xml главнее (если в приложение есть оба файла, то принимаются настройки из файла hibernate.cfg.xml). Конфигурация аннотациями, хоть и удобна, но не всегда возможна, к примеру, если для разных баз данных или для разных ситуаций вы хотите иметь разные конфигурацию сущностей, то следует использовать xml файлы конфигураций.

Что такое Hibernate mapping file

Mapping file используется для связи entity бинов и колонок в таблице базы данных. В случаях, когда не используются аннотации JPA, файл отображения .xml может быть полезен (например при использовании сторонних библиотек).

Что такое Переходные объекты (Transient Objects)

Экземпляры долгоживущих классов, которые в настоящее время не связаны с Сессией. Они, возможно, были инициализированы в приложении и еще не сохранены, или же они были инициализированы закрытой Сессией.

Что такое постоянные объекты (Persistent objects)

Короткоживущие, однопоточные объекты, содержащие постоянное состояние и бизнес-функции. Это могут быть простые Java Beans/POJOs. Они связаны только с одной Сессией. После того, как Сессия закрыта, они будут отделены и свободны для использования в любом протоколе прикладного уровня (например, в качестве объектов передачи данных в и из представления).

Что такое TransactionFactory

Фабрика для экземпляров Transaction. Интерфейс не открыт для приложения, но может быть расширен или реализован разработчиком.

Что такое ConnectionProvider

Фабрика и пул JDBC соединений. Интерфейс абстрагирует приложение от основного источника данных или диспетчера драйверов. Он не открыт для приложения, но может быть расширен или реализован разработчиком.

Что такое Трансакция (Transaction)

Однопоточный, короткоживущий объект, используемый приложением для указания `atomic` переменных работы. Он абстрагирует приложение от основных JDBC, JTA или CORBA трансакций. Сессия может охватывать несколько Трансакций в некоторых случаях. Тем не менее, разграничение транзакций, также используемое в основах API или Transaction, всегда обязательно.

Какие существуют стратегии загрузки объектов в Hibernate

Существуют следующие типа fetch'a:

- Join fetching: hibernate получает ассоциированные объекты и коллекции одним SELECT используя OUTER JOIN
- Select fetching: использует уточняющий SELECT чтобы получить ассоциированные объекты и коллекции. Если вы не установите lazy fetching определив lazy="false", уточняющий SELECT будет выполнен только когда вы запрашиваете доступ к ассоциированным объектам
- Subselect fetching: поведение такое же, как у предыдущего типа, за тем исключением, что будут загружены ассоциации для все других коллекций, "родительским" для которых является сущность, которую вы загрузили первым SELECT'ом.
- Batch fetching: оптимизированная стратегия вида select fetching. Получает группу сущностей или коллекций в одном SELECT

Какие бывают id generator классы в Hibernate

- increment - генерирует идентификатор типа long, short или int, которые будет уникальным только в том случае, если другой процесс не добавляет запись в эту же таблицу в это же время.
- identity - генерирует идентификатор типа long, short или int. Поддерживается в DB2, MySQL, MS SQL Server, Sybase и HypersonicSQL.
- sequence - использует последовательности в DB2, PostgreSQL, Oracle, SAP DB, McKoi или генератор Interbase. Возвращает идентификатор типа long, short или int.
- hilo - использует алгоритм hi/lo для генерации идентификаторов типа long, short или int. Алгоритм гарантирует генерацию идентификаторов, которые уникальны только в данной базе данных.
- seqhilo - использует алгоритм hi/lo для генерации идентификаторов типа long, short или int учитывая последовательность базы данных.
- uuid - использует для генерации идентификатора алгоритм 128-bit UUID. Идентификатор будет уникальным в пределах сети. UUID представляется строкой из 32 чисел.
- guid - использует сгенерированую БД строку GUID в MS SQL Server и MySQL.
- native - использует identity, sequence или hilo в завимисимости от типа БД, с которой работает приложение
- assigned - позволяет приложению устанавливать идентификатор объекту, до вызова метода save(). Используется по умолчанию, если тег `<generator>` не указан.
- select - получает первичный ключ, присвоенный триггером БД
- foreign - использует идентификатор другого, связанного с данным объекта. Используется в `<one-to-one>` ассоциации

первичных ключей.
- sequence-identity - специализированный генератор идентификатора.

Какие ключевые интерфейсы использует Hibernate

Существует пять ключевых интерфейсов которые используются в каждом приложении связанном с Hibernate:

- Session interface
- SessionFactory interface
- Configuration interface
- Transaction interface
- Query and Criteria interfaces

Назовите некоторые важные аннотации, используемые для отображения в Hibernate

Hibernate поддерживает как аннотации из JPA, так и свои собственные, которые находятся в пакете org.hibernate.annotations. Наиболее важные аннотации JPA и Hibernate:

- `javax.persistence.Entity` : используется для указания класса как entity bean.
- `javax.persistence.Table` : используется для определения имени таблицы из БД, которая будет отображаться на entity bean.
- `javax.persistence.Access` : определяет тип доступа, поле или свойство. Поле — является значением по умолчанию и если нужно, чтобы hibernate использовать методы getter/setter, то их необходимо задать для нужного свойства.
- `javax.persistence.Id` : определяет primary key в entity bean.
- `javax.persistence.EmbeddedId` : используется для определения составного ключа в бине.
- `javax.persistence.Column` : определяет имя колонки из таблицы в базе данных.
- `javax.persistence.GeneratedValue` : задает стратегию создания основных ключей. Используется в сочетании с - `javax.persistence.GenerationType` `enum` .
- `javax.persistence.OneToOne` : задает связь один-к-одному между двумя сущностными бинами. Соответственно есть другие аннотации `OneToMany` , `ManyToOne` и `ManyToMany` .
- `org.hibernate.annotations.Cascade` : определяет каскадную связь между двумя `entity` бинами. Используется в связке с org.hibernate.annotations.CascadeType.
- `javax.persistence.PrimaryKeyJoinColumn` : определяет внешний ключ для свойства. Используется вместе с `org.hibernate.annotations.GenericGenerator` и `org.hibernate.annotations.Parameter` .

Какая роль интерфейса `Session` в Hibernate

`Session` - это основной интерфейс, который отвечает за связь с базой данных. Так же, он помогает создавать объекты запросов для получение персистентных объектов. (персистентный объект - объект который уже находится в базе данных; объект запроса - объект который получается когда мы получаем результат запроса в базу данных, именно с ним работает приложение). Обьект `Session` можно получить из `SessionFactory` :

```
Session session = sessionFactory.openSession();
```

Роль интерфейса `Session` : является оберткой для jdbc подключения к базе данных; является фабрикой для транзакций (согласно официальной документации transaction - allows the application to define units of work, что , по сути, означает что транзакция определяет границы операций связанных с базой данных). является хранителем обязательного кэша первого уровня.

Какая роль интерфейса `SessionFactory` в Hibernate

Именно из объекта `SessionFactory` мы получаем объекты типа Session. На все приложение существует только одна `SessionFactory` и она инициализируеться вместе со стартом приложения. `SessionFactory` кэширует мета-дату и SQL запросы которые часто используются приложением во время работы. Так же оно кэширует информацию которая была получена в одной из транзакций и может быть использована и в других транзакциях. Обьект `SessionFactory` можно получить следующим обращением:

```
SessionFactory sessionFactory = configuration.buildSessionFactory();
```

Является ли Hibernate `SessionFactory` потокобезоспансым

Т.к. объект `SessionFactory` `immutable` (неизменяемый), то да, он потокобезопасный. Множество потоков может обращаться к одному объекту одновременно.

В чем разница между `openSession()` и `getCurrentSession()`

Hibernate SessionFactory getCurrentSession() возвращает сессию, связанную с контекстом. Но для того, чтобы это работало, нам нужно настроить его в конфигурационном файле hibernate. Так как этот объект session связан с контекстом hibernate, то отпадает необходимость к его закрытию. Объект session закрывается вместе с закрытием `SessionFactory` .

```
<property name="hibernate.current_session_context_class">thread</property>
```

Метод `openSession()` всегда создает новую сессию. Мы должны обязательно контролировать закрытие объекта сеанса по завершению всех операций с базой данных. Для многопоточной среды необходимо создавать новый объект session для каждого запроса.

Существует еще один метод `openStatelessSession()` , который возвращает `Session` без поддержки состояния. Такой объект не реализует первый уровень кэширования и не взаимодействует с вторым уровнем. Сюда же можно отнести игнорирование коллекций и некоторых обработчиков событий. Такие объекты могут быть полезны при загрузке больших объемов данных без удержания большого кол-ва информации в кэше.

Какие типы коллекций представлены в Hibernate

- `Bag`
- `Set`
- `List`
- `Map`
- `Array`

Какие типы менеджмента транзакций поддерживаются в Hibernate

Hibernate взаимодействует с БД через JDBC-соединение. Таким образом он поддерживает управляемые и не управляемые транзакции. Неуправляемые транзакции в web-контейнере:

```
<bean id="transactionManager" class="org.springframework.orm.hibernate.HibernateTransactionManager">
    <property name="sessionFactory">
        <ref local="sessionFactory" />
    </property>
</bean>
```

Управляемые транзакции на сервере приложений, использующий JTA:

```
<bean id="transactionManager" class="org.springframework.transaction.jtaTransactionManager">
    <property name="sessionFactory">
        <ref local="sessionFactory" />
    </property>
</bean>
```

Что собой являет коллекция типа `Bag` и зачем она используется

Своей реализации тип коллекции `Bag` очень напоминает `Set`, разница состоит в том, что `Bag` может хранить повторяющиеся значения. `Bag` хранит непроиндексированный список элементов. Большинство таблиц в базе данных имеют индексы отображающие положение элемента данных один относительно другого, данные индексы имеют представление в таблице в виде отдельной колонки. При объектно-реляционном маппинге, значения колонки индексов мапится на индекс в `Array`, на индекс в `List` или на `key` в `Map`. Если вам надо получить коллекцию объектов не содержащих данные индексы, то вы можете воспользоваться коллекциями типа `Bag` или `Set` (коллекции содержат данные в неотсортированном виде, но могут быть отсортированы согласно запросу).

Какие типы кэша используются в Hibernate

Hibernate использует 2 типа кэша: кэш первого уровня и кэш второго уровня. Кэш первого уровня ассоциирован с объектом сесии, в то время, как кэш второго уровня ассоциирован с объектом фабрики сессий. По-умолчанию Hibernate использует кэш первого уровня для каждой операции в транзакции. В первую очередь кэш используется чтобы уменьшить количество SQL-запросов. Например если объект модифицировался несколько раз в одной и той же транзакции, то Hibernate сгенерирует только один UPDATE. Чтобы уменьшить трафик с БД, Hibernate использует кэш второго уровня, который является общим для всего приложения, а не только для данного конкретного пользователя. Таким образом если результат запроса находится в кэше, мы потенциально уменьшаем количество транзакций к БД. EHCache - это быстрый и простой кэш. Он поддерживает read-only и read/write кэширование, а так же кэширование в память и на диск. Но не поддерживает кластеризацию. OSCache - это другая opensource реализация кэша. Помимо всего, что поддерживает EHCache, эта реализация так же поддерживает кластеризацию через JavaGroups или JMS. SwarmCache - это просто cluster-based решение, базирующееся на JavaGroups. Поддерживает read-only и нестрогое read/write кэширование. Этот тип кэширование полезен, когда количество операций чтения из БД превышает количество операций записи. JBoss TreeCache - предоставляет полноценный кэш транзакции.

Какие существуют типы стратегий кэша

Read-only: эта стратегия используется когда данные вычитываются, но никогда не обновляется. Самая простая и производительная стратегия Read/write: может быть использована, когда данные должны обновляться. Нестрогий read/write: эта стратегия не гарантирует, что две транзакции не модифицируют одни и те же данные синхронно. Transactional: полноценное кэширование транзакций. Доступно только в JTA окружении.

Что вы знаете о кэширование в Hibernate? Объясните понятие кэш первого уровня в Hibernate

Hibernate использует кэширование, чтобы сделать приложение быстрее. идея кэширования заключается в сокращении количества запросов к БД.

Кэщ первого уровня Hibernate, связан с объектом Session. Кэш первого уровня у Hibernate включен по умолчаню и не существует никакого способа, чтобы его отключить. Однако Hibernate предоставляет методы, с помощью которых мы можем удалить выбранные объекты из кэша или полностью очистить кэш.

Любой объект закэшированный в Session не будет виден другим объектам Session, после закрытия сессии, все кэшированные объекты будут потеряны.

Как настраивается кэш второго уровня в Hibernate

Чтобы указать кэш второго уровня нужно определить hibernate.cache.provider_class в hibernate.cfg.xml:

```
<hibernate-configuration>
    <session-factory>
        <property name="hibernate.cache.provider_class">org.hibernate.cache.EHCacheProvider</property>
    </session-factory>
</hibernate-configuration>
```

По-умолчанию используется EHCache. Чтобы использвать кэш запросов нужно его включить установив свойство hibernate.cache.use_query_cache в true в hibernate.properties.

Какая разница в работе методов load() и get()

Hibernate session обладает различными методами для загрузки данных из базы данных. Наиболее часто используемые методы для этого - get() и load(). Метод load(); обычно используется когда в не уверен что запрашиваемый объект уже находится в базе данных. Если объект не найден, то метод кидает исключение Если объект найден — метод возвращает прокси объект, который является ссылкой на объект находящийся в базе данных (запрос в базу данных еще не был осуществлен, своего рода lazy изъятие), непосредственный запрос к базе данных когда мы непосредственно обращаемся к необходимому объекту через прокси объект. Метод get(); используется тогда, вы на 100 процентов не уверены есть ли запрашиваемый объект в базе данных. В случае обращение к несуществующему объекту, метод get(); вернет null. В случае нахождения объект, метод get(); вернет сам объект и запрос в базу данных будет произведен немедленно.

Каковы существуют различные состояния у entity bean

Transient: состояние, при котором объект никогда не был связан с какой-либо сессией и не является персистентностью. Этот объект находится во временном состоянии. Объект в этом состоянии может стать персистентным при вызове метода save(), persist() или saveOrUpdate(). Объект персистентности может перейти в transient состоянии после вызова метода delete(). Persistent: когда объект связан с уникальной сессией он находится в состоянии persistent (персистентности). Любой экземпляр, возвращаемый методами get() или load() находится в состоянии persistent. Detached: если объект был персистентным, но сейчас не связан с какой-либо сессией, то он находится в отвязанном (detached) состоянии. Такой объект можно сделать персистентным используя методы update(), saveOrUpdate(), lock() или replicate(). Состояния transient или detached так же могут перейти в состояние persistent как новый объект персистентности после вызова метода merge().

Что произойдет, если будет отсутствовать конструктор без аргументов у Entity Bean

Hibernate использует рефлексию для создания экземпляров Entity бинов при вызове методов get() или load(). Для этого используется метод Class.newInstance(), который требует наличия конструктора без параметров. Поэтому, в случае его отсутствия, вы получите ошибку HibernateException.

Как используется вызов метода Hibernate Session merge()

Hibernate merge() может быть использован для обновления существующих значений, однако этот метод создает копию из переданного объекта сущности и возвращает его. Возвращаемый объект является частью контекста персистентности и отслеживает любые изменения, а переданный объект не отслеживается.

В чем разница между Hibernate save(), saveOrUpdate() и persist()

Hibernate save() используется для сохранения сущности в базу данных. Проблема с использованием метода save() заключается в том, что он может быть вызван без транзакции. А следовательно если у нас имеется отображение нескольких объектов, то только первичный объект будет сохранен и мы получим несогласованные данные. Также save() немедленно возвращает сгенерированный идентификатор. Hibernate persist() аналогичен save() с транзакцией. persist() не возвращает сгенерированный идентификатор сразу. Hibernate saveOrUpdate() использует запрос для вставки или обновления, основываясь на предоставленных данных. Если данные уже присутствуют в базе данных, то будет выполнен запрос обновления. Метод saveOrUpdate() можно применять без транзакции, но это может привести к аналогичным проблемам, как и в случае с методом save().

Что такое Lazy fetching(изъятие) в Hibernate

Тип изьятия Lazy, в Hibernate, связан с листовыми(дочерними) сущностями и определяют политику совместного изъятия, если идет запрос на изъятие сущности родителя. Простой пример: Есть сущность Дом. Он хранит информацию о своем номере, улице, количество квартир и информацию о семьях которые живут в квартирах, эти семьи формируют дочернюю сущность относительно сущности Дом. Когда мы запрашиваем информацию о Доме, нам может быть совершенно ненужным знать информацию семьях которые в нем проживают, тут нам на помощь приходит lazy(ленивое) изъятие(fetching) которая позволяет сконфигурировать сущность Дом, чтобы информацию о семьях подавалась только по востребованию, это значительно облегчает запрос и ускоряет работу приложения.

В чем разница между sorted collection и ordered collection? Какая из них лучше

При использовании алгоритмов сортировки из Collection API для сортировки коллекции, то он вызывает отсортированный список (sorted list). Для маленьких коллекций это не приводит к излишнему расходу ресурсов, но на больших коллекциях это может привести к потери производительности и ошибкам OutOfMemory. Так же entity бины должны реализовывать интерфейс Comparable или Comparator для работы с сортированными коллекциями. При использовании фреймворка Hibernate для загрузки данных из базы данных мы можем применить Criteria API и команду order by для получения отсортированного списка (ordered list). Ordered list является лучшим выбором к sorted list, т.к. он использует сортировку на уровне базы данных. Она быстрее и не может привести к утечке памяти.

Как реализованы Join’ы Hibernate

Существует несколько способов реализовать связи в Hibernate. Использовать ассоциации, такие как one-to-one, one-to-many, many-to-many. Использовать в HQL запросе команду JOIN. Существует другая форма «join fetch«, позволяющая загружать данные немедленно (не lazy). Использовать чистый SQL запрос с командой join.

Почему мы не должны делать Entity class как final

Хибернейт использует прокси классы для ленивой загрузки данных (т.е. по необходимости, а не сразу). Это достигается с помощью расширения entity bean и, следовательно, если бы он был final, то это было бы невозможно. Ленивая загрузка данных во многих случаях повышает производительность, а следовательно важна.

Что вы знаете о HQL и каковы его преимущества

Hibernate Framework поставляется с мощным объектно-ориентированным языком запросов - Hibernate Query Language (HQL). Он очень похож на SQL, за исключением, что в нем используются объекты вместо имен таблиц, что делает язык ближе к объектно-ориентированному программированию. HQL является регистронезависимым, кроме использования в запросах имен

java переменных и классов, где он подчиняется правилам Java. Например, SelECt то же самое, что и select, но com.blogspot.jsehelper.MyClass отличен от com.blogspot.jsehelper.MyCLASS. Запросы HQL кэшируются (это как плюс так и минус).

Что такое Query Cache в Hibernate

Hibernate реализует область кэша для запросов resultset, который тесно взаимодействует с кэшем второго уровня Hibernate. Для подключения этой дополнительной функции требуется несколько дополнительных шагов в коде. Query Cache полезны только для часто выполняющихся запросов с повторяющимися параметрами. Для начала необходимо добавить эту запись в файле конфигурации Hibernate:

```
<property name="hibernate.cache.use_query_cache">true</property>
```

Уже внутри кода приложения для запроса применяется метод setCacheable(true), как показано ниже:

```
Query query = session.createQuery("from Employee");
query.setCacheable(true);
query.setCacheRegion("ALL_EMP");
```

Можем ли мы выполнить SQL запрос в Hibernate

С помощью использования SQLQuery можно выполнять чистый запрос SQL. В общем случае это не рекомендуется, т.к. вы потеряете все преимущества HQL (ассоциации, кэширование). Выполнить можно примерно так:

```
Transaction tx = session.beginTransaction();
SQLQuery query = session.createSQLQuery("SELECT * FROM Employee");
List<Object[]> rows = query.list();
for(Object[] row : rows) {
    Employee emp = new Employee();
    emp.setId(Long.parseLong(row[0].toString()));
    emp.setName(row[1].toString());
    emp.setSalary(Double.parseDouble(row[2].toString()));
    System.out.println(emp);
}
```

Назовите преимущества поддержки нативного sql в Hibernate

Использование нативного SQL может быть необходимо при выполнении запросов к некоторым базам данных, которые могут не поддерживаться в Hibernate. Примером может служить некоторые специфичные запросы и «фишки» при работе с БД от Oracle.

Что такое Named SQL Query

Hibernate поддерживает именованный запрос, который мы можем задать в каком-либо центральном месте и потом использовать его в любом месте в коде. Именованные запросы поддерживают как HQL, так и Native SQL. Создать именованный запрос можно с помощью JPA аннотаций @NamedQuery, @NamedNativeQuery или в конфигурационном файле отображения (mapping files).

Каковы преимущества Named SQL Query

- Именованный запрос Hibernate позволяет собрать множество запросов в одном месте, а затем вызывать их в любом классе.
- Синтаксис Named Query проверяется при создании session factory, что позволяет заметить ошибку на раннем этапе, а не при запущенном приложении и выполнении запроса.
- Named Query глобальные, т.е. заданные однажды, могут быть использованы в любом месте. Однако одним из основных недостатков именованного запроса является то, что его очень трудно отлаживать (могут быть сложности с поиском места

определения запроса).

Как добавить логирование log4j в Hibernate приложение

Добавить зависимость log4j в проект. Создать log4j.xml или log4j.properties файл и добавить его в classpath. Для веб приложений используйте ServletContextListener, а для автономных приложений DOMConfigurator или PropertyConfigurator для настройки логирования. Создайте экземпляр org.apache.log4j.Logger и используйте его согласно задачи.

Как логировать созданные Hibernate SQL запросы в лог-файлы

Для логирования запросов SQL добавьте в файл конфигурации Hibernate строчку

```
<property name="hibernate.show_sql">true</property>
```

Что вы знаете о Hibernate прокси и как это помогает в ленивой загрузке (lazy load)

Hibernate использует прокси объект для поддержки отложенной загрузки. Обычно при загрузке данных из таблицы Hibernate не загружает все отображенные (замаппинные) объекты. Как только вы ссылаетесь на дочерний объект или ищите объект с помощью геттера, если связанная сущность не находиться в кэше сессии, то прокси код перейдет к базе данных для загрузки связанной сущности. Для этого используется javassist, чтобы эффективно и динамически создавать реализации подклассов ваших entity объектов.

Как управлять транзакциями с помощью Hibernate

Hibernate вообще не допускает большинство операций без использования транзакций. Поэтому после получения экземпляра session от SessionFactory необходимо выполнить beginTransaction() для начала транзакции. Метод вернет ссылку, которую мы можем использовать для подтверждения или отката транзакции. В целом, управление транзакциями в фреймворке выполнено гораздо лучше, чем в JDBC, т.к. мы не должны полагаться на возникновение исключения для отката транзакции. Любое исключение автоматически вызовет rollback.

Что такое каскадные связи (обновления) в Hibernate

Если у нас имеются зависимости между сущностями (entities), то нам необходимо определить как различные операции будут влиять на другую сущность. Это реализуется с помощью каскадных связей (или обновлений). Вот пример кода с использованием аннотации @Cascade:

```
import org.hibernate.annotations.Cascade;

@Entity
@Table(name="Employee")
public class Employee {

    @OneToOne(mappedBy = "employee")
    @Cascade(value = org.hibernate.annotations.CascadeType.ALL)
    private Address address;
}
```

Есть некоторые различия между enum CascadeType в Hibernate и в JPA. Поэтому обращайте внимание какой пакет вы импортируете при использовании аннотации и константы типа.

Какие каскадные типы есть в Hibernate

Наиболее часто используемые CascadeType перечисления описаны ниже:

- None: без Cascading. Формально это не тип, но если мы не указали каскадной связи, то никакая операция для родителя не будет иметь эффекта для ребенка.
- ALL: Cascades save, delete, update, evict, lock, replicate, merge, persist. В общем — всё.
- SAVE_UPDATE: Cascades save и update. Доступно только для hibernate.
- DELETE: передает в Hibernate native DELETE действие. Только для hibernate.
- DETATCH, MERGE, PERSIST, REFRESH и REMOVE – для простых операций.
- LOCK: передает в Hibernate native LOCK действие.
- REPLICATE: передает в Hibernate native REPLICATE действие.

Что такое сесиия и фаблика сессий в Hibernate? Как настроить session factory в конфигурационном файле Spring

Hibernate сессия - это главный интерфейс взаимодействия Java-приложения и Hibernate. SessionFactory позволяет создавать сессии согласно конфигурации hibernate.cfg.xml. Например:

```
// Initalize the Hibernate environment
Configuration cfg = new Configuration().configure();
// Create the session factory
SessionFactory factory = cfg.buildSessionFactory();
// Obtain the new session object
Session session = factory.openSession();
```

При вызове Configuration().configure() загружается файл hibernate.cfg.xml и происходит настройка среды Hibernate. После того, как конфигурация загружена, вы можете сделать дополнительную модификацию настроек уже на программном уровне. Данные корректировки возможны до создания экземпляра фабрики сессий. Экземпляр SessionFactory как правило создается один раз и используется во всем приложении. Главная задача сессии - обеспечить механизмы создания, чтения и удаления для экземпляров примапленых к БД классов. Экземпляры могут находиться в трёх состояниях: transient - никогда не сохранялись, не ассоциированы ни с одной сессией; persistent - ассоциированы с уникальной сессией; detached - ранее сохраненные, не ассоциированы с сессией. Объект Hibernate Session представляет одну операцию с БД. Сессию открывает фабрика сессий. Сессия должна быть закрыта, когда все операции с БД совершены. Пример:

```
Session session = null;
UserInfo user = null;
Transaction tx = null;

try {
    session = factory.openSession();
    tx = session.beginTransaction();
    user = (UserInfo) session.load(UserInfo.class, id);
    tx.commit();
} catch(Exception e) {
    if (tx != null) {
        try {
            tx.rollback();
        } catch (HibernateException he) {
            throw new DAOException(he.toString());
        }
    }
    throw new DAOException(e.toString());
} finally {
    if (session != null) {
        try {
            session.close();
        } catch (HibernateException e) {

        }
    }
```

```
}
```

Как использовать JNDI DataSource сервера приложений с Hibernate Framework

В веб приложении лучше всего использовать контейнер сервлетов для управления пулом соединений. Поэтому лучше определить JNDI ресурс для DataSource и использовать его в веб приложении. Для этого в Hibernate нужно удалить все специфичные для базы данных свойства из и использовать указания свойства JNDI DataSource:

```
<property name="hibernate.connection.datasource">java:comp/env/jdbc/localdb</property>
```

Как интегрировать Hibernate и Spring

Лучше всего прочитать о настройках на сайтах фреймворков для текущей версии. Оба фреймворка поддерживают интеграцию из коробки и в общем настройка их взаимодействия не составляет труда. Общие шаги выглядят следующим образом.

- Добавить зависимости для hibernate-entitymanager, hibernate-core и spring-orm.
- Создать классы модели и передать реализации DAO операции над базой данных. Важно, что DAO классы используют SessionFactory, который внедряется в конфигурации бинов Spring.
- Настроить конфигурационный файл Spring (смотрите в офф. документации или из примера на этом сайте).
- Дополнительно появляется возможность использовать аннотацию @Transactional и перестать беспокоиться об управлением транзакцией Hibernate.

Какие паттерны применяются в Hibernate

Domain Model Pattern - объектная модель предметной области, включающая в себя как поведение так и данные. Data Mapper - слой мапперов (Mappers), который передает данные между объектами и базой данных, сохраняя их независимыми друг от друга и себя. Proxy Pattern - применяется для ленивой загрузки. Factory pattern - используется в SessionFactory

Расскажите о Hibernate Validator Framework

Проверка данных является неотъемлемой частью любого приложения. Hibernate Validator обеспечивает эталонную реализацию двух спецификаций JSR-303 и JSR-349 применяемых в Java. Для настройки валидации в Hibernate необходимо сделать следующие шаги.

- Добавить hibernate validation зависимости в проект

```
<dependency>
    <groupid>javax.validation</groupid>
    <artifactid>validation-api</artifactid>
    <version>${validation-api-version}</version>
</dependency>
<dependency>
    <groupid>org.hibernate</groupid>
    <artifactid>hibernate-validator</artifactid>
    <version>${hibernate-vlidator-version}</version>
</dependency>
```

- Так же требуются зависимости из JSR 341, реализующие Unified Expression Language для обработки динамических выражений и сообщений о нарушении ограничений

```
<dependency>
    <groupid>javax.el</groupid>
```

```
    <artifactid>javax.el-api</artifactid>
    <version>${el-api-version}</version>
</dependency>
<dependency>
    <groupid>org.glassfish.web</groupid>
    <artifactid>javax-el</artifactid>
    <version>${javax-el-version}</version>
</dependency>
```

- Использовать необходимые аннотации в бинах

```
import javax.validation.constraints.Min;
import javax.validation.constraints.NotNull;
import javax.validation.constraints.Size;

import org.hibernate.validator.constraints.CreditCardNumber;
import org.hibernate.validator.constraints.Email;

public class Employee {

    @Min(value=1, groups=EmpIdCheck.class)
    private int id;

    @NotNull(message="Name cannot be null")
    @Size(min=5, max=30)
    private String name;

    @Email
    private String email;

    @CreditCardNumber
    private String creditCardNumber;
}
```

Spring Framework

Что такое Spring? Из каких частей состоит Spring Framework

Spring - фреймворк с открытым исходным кодом, предназначеный для упрощения разработки enterprise-приложений. Одним из главным преимуществом **Spring** является его слоистая архитектура, позволяющая вам самим определять какие компоненты будут использованы в вашем приложении. Модули **Spring** построены на базе основного контейнера, который определяет создание, конфигурация и менеджмент бинов. Основные модули:

- **Spring Core** - предоставляет основной функционал Spring. Главным компонентом контейнера является BeanFactory - реализация паттерна Фабрика. BeanFactory позволяет разделить конфигурацию приложения и информацию о зависимостях от кода.
- **Spring context** - конфигурационный файл, который предоставляет информация об окружающей среде для Spring. Сюда входят такие enterprise-сервисы, как JNDI, EJB, интернационализация, валиадция и т.п.
- **Spring AOP** - отвечает за интеграцию аспектно-ориентированного программирования во фреймворк. Spring AOP обеспечивает сервис управления транзакциями для Spring-приложения.
- **Spring DAO** - абстрактный уровень Spring JDBC DAO предоставляет иерархию исключений и множество сообщений об ошибках для разных БД. Эта иерархия упрощает обработку исключений и значительно уменьшает количество кода, которое вам нужно было бы написать для таких операций, как, например, открытие и закрытие соединения.
- **Spring ORM** - отвечает за интеграцию Spring и таких популярных ORM-фреймворков, как Hibernate, iBatis и JDO.
- **Spring Web module** - классы, которые помогают упростить разработку Web (авторизация, доступ к бинам Spring-а из web).
- **Spring MVC framework** - реализация паттерна MVC для построения Web-приложений.

Каковы некоторые из важных особенностей и преимуществ Spring Framework

Spring Framework обеспечивает решения многих задач, с которыми сталкиваются Java-разработчики и организации, которые хотят создать информационную систему, основанную на платформе Java. Из-за широкой функциональности трудно определить наиболее значимые структурные элементы, из которых он состоит. Spring Framework не всецело связан с платформой Java Enterprise, несмотря на его масштабную интеграцию с ней, что является важной причиной его популярности.

- Относительная легкость в изучении и применении фреймворка в разработке и поддержке приложения.
- Внедрение зависимостей (DI) и инверсия управления (IoC) позволяют писать независимые друг от друга компоненты, что дает преимущества в командной разработке, переносимости модулей и т.д..
- Spring IoC контейнер управляет жизненным циклом Spring Bean и настраивается наподобие JNDI lookup (поиска).
- Проект Spring содержит в себе множество подпроектов, которые затрагивают важные части создания софта, такие как вебсервисы, веб программирование, работа с базами данных, загрузка файлов, обработка ошибок и многое другое. Всё это настраивается в едином формате и упрощает поддержку приложения.

Каковы преимущества использования Spring Tool Suite

Для упрощения процесса разработки основанных на Spring приложений в Eclipse (наиболее часто используемая IDE-среда для разработки Java-приложений), в рамках Spring создан проект Spring IDE. Проект бесплатный. Он интегрирован в Eclipse IDE, Spring IDE, Mylyn (среда разработки в Eclipse, основанная на задачах), Maven for Eclipse, AspectJ Development Tool.

Что такое AOP? Как это относится к IoC

Аспектно-ориентированное программирование (АОП) - парадигма программирования, основанная на идее разделения функциональности для улучшения разбиения программы на модули. AOP и Spring - взаимодополняющие технологии, которые позволяют решать сложные проблемы путем разделения функционала на отдельные модули. АОП предоставляет возможность

реализации сквозной логики - т.е. логики, которая применяется к множеству частей приложения - в одном месте и обеспечения автоматического применения этой логики по всему приложению. Подход Spring к АОП заключается в создании "динамических прокси" для целевых объектов и "привязывании" объектов к конфигурированному совету для выполнения сквозной логики.

Что такое Aspect, Advice, Pointcut, JointPoint и Advice Arguments в АОП

Основные понятия АОП:

- Аспект (англ. aspect) - модуль или класс, реализующий сквозную функциональность. Аспект изменяет поведение остального кода, применяя совет в точках соединения, определённых некоторым срезом.
- Совет (англ. advice) - фрагмент кода, который должен выполняться в отдельной точке соединения. Существует несколько типов советов, совет может быть выполнен до, после или вместо точки соединения.
- Точка соединения (англ. joinpoint) - это четко определенная точка в выполняемой программе, где следует применить совет. Типовые примеры точек соединения включают обращение к методу, собственно Method Invocation, инициализацию класса и создание экземпляра объекта. Многие реализации АОП позволяют использовать вызовы методов и обращения к полям объекта в качестве точек соединения.
- Срез (англ. pointcut) - набор точек соединения. Срез определяет, подходит ли данная точка соединения к данному совету. Самые удобные реализации АОП используют для определения срезов синтаксис основного языка (например, в AspectJ применяются Java-сигнатуры) и позволяют их повторное использование с помощью переименования и комбинирования.
- Связывание(англ. weaving) представляет собой процесс действительной вставки аспектов в определенную точку кода приложения. Для решений АОП времени компиляции это делается на этапе компиляции, обычно в виде дополнительного шага процесса сборки. Аналогично, для решений АОП времени выполнения связывание происходит динамически во время выполнения. В AspectJ поддерживается еще один механизм связывания под названием связывание во время загрузки (load-time weaving - LTW), который перехватывает лежащий в основе загрузчик классов JVM и обеспечивает связывание с байт-кодом, когда он загружается загрузчиком классов.
- Цель(англ. target) - это объект, поток выполнения которого изменяется каким-то процессом АОП. На целевой объект часто ссылаются как на объект, снабженный советом.
- Внедрение (англ. introduction, введение) - представляет собой процесс, посредством которого можно изменить структуру объекта за счет введения в него дополнительных методов или полей, изменение иерархии наследования для добавления функциональности аспекта в инородный код. Обычно реализуется с помощью некоторого метаобъектного протокола (англ. metaobject protocol, MOP).

В чем разница между Spring AOP и AspectJ АОП

AspectJ де-факто является стандартом реализации АОП. Реализация АОП от Spring имеет некоторые отличия:

- Spring AOP немного проще, т.к. нет необходимости следить за процессом связывания.
- Spring AOP поддерживает аннотации AspectJ, таким образом мы можем работать в спринг проекте похожим образом с AspectJ проектом.
- Spring AOP поддерживает только proxy-based АОП и может использовать только один тип точек соединения - Method Invocation. AspectJ поддерживает все виды точек соединения.

Недостатком Spring AOP является работа только со своими бинами, которые существуют в Spring Context.

Что такое IoC контейнер Spring

По своей сути IoC, а, следовательно, и DI, направлены на то, чтобы предложить простой механизм для предоставления зависимостей компонента (часто называемых коллаборатораторами объекта) и управления этими зависимостями на протяжении всего их жизненного цикла. Компонент, который требует определенных зависимостей, зачастую называют зависимым объектом или, в случае IoC, целевым объектом. IoC предоставляет службы, через которые компоненты могут получать доступ к своим зависимостям, и службы для взаимодействия с зависимостями в течение их времени жизни. В общем случае IoC может быть расщеплена на два подтипа: инверсия управления (Dependency Injection) и инверсия поиска (Dependency Lookup). Инверсия управления — это крупная часть того, делает Spring, и ядро реализации Spring основано на инверсии управления, хотя также предоставляются и средства Dependency Lookup. Когда платформа Spring предоставляет коллабораторы зависимому объекту

автоматически, она делает это с использованием инверсии управления (Dependency Injection). В приложении, основанном на Spring, всегда предпочтительнее применять Dependency Injection для передачи коллабораторов зависимым объектам вместо того, чтобы заставлять зависимые объекты получать коллабораторы через поиск.

Что такое Spring bean

Термин бин (англ. Bean) - в Spring используется для ссылки на любой компонент, управляемый контейнером. Обычно бины на определенном уровне придерживаются спецификации JavaBean, но это не обязательно особенно если для связывания бинов друг с другом планируется применять Constructor Injection. Для получения экземпляра бина используется ApplicationContext. IoC контейнер управляет жизненным циклом спринг бина, областью видимости и внедрением.

Какое значение имеет конфигурационный файл Spring Bean

Конфигурационный файл спринг определяет все бины, которые будут инициализированы в Spring Context. При создании экземпляра Spring ApplicationContext будет прочитан конфигурационный xml файл и выполнены указанные в нем необходимые инициализации. Отдельно от базовой конфигурации, в файле могут содержаться описание перехватчиков (interceptors), view resolvers, настройки локализации и др...

Каковы различные способы настроить класс как Spring Bean

Существует несколько способов работы с классами в Spring: XML конфигурация:

```
<bean name="name" class="com.swchck.spring.beans.BeanName"/>
```

Java based конфигурация. Все настройки и указания бинов прописываются в java коде:

```
@Configuration
@ComponentScan(value = "com.swchck.sping.main")
public class Configuration {
    @Bean
    public Service getService() {
        return new Service();
    }
}
```

Для извлечения бина из контекста используется следующий подход:

```
AnnotationConfigApplicationContext context = new AnnotationConfigApplicationContext(Configuration.class);
Service service = context.getBean(Service.class);
```

Annotation based конфигурация. Можно использовать внутри кода аннотации @Component, @Service, @Repository, @Controller для указания классов в качестве спринг бинов. Для их поиска и управления контейнером прописывается настройка в xml файле:

```
<context:component-scan base-package="com.swchck.spring">
```

Какие вы знаете различные scope у Spring Bean

В Spring предусмотрены различные области времени действия бинов: singleton - может быть создан только один экземпляр бина. Этот тип используется спрингом по умолчанию, если не указано другое. Следует осторожно использовать публичные свойства класса, т.к. они не будут потокобезопасными. prototype - создается новый экземпляр при каждом запросе. request - аналогичен prototype, но название служит пояснением к использованию бина в веб приложении. Создается новый экземпляр при каждом HTTP request. session - новый бин создается в контейнере при каждой новой HTTP сессии. global-session: используется для создания глобальных бинов на уровне сессии для Portlet приложений.

Что такое жизненный цикл Spring Bean

Жизненный цикл Spring бина - время существования класса. Spring бины инициализируются при инициализации Spring контейнера и происходит внедрение всех зависимостей. Когда контейнер уничтожается, то уничтожается и всё содержимое. Если нам необходимо задать какое-либо действие при инициализации и уничтожении бина, то нужно воспользоваться методами `init()` и `destroy()`. Для этого можно использовать аннотации `@PostConstruct` и `@PreDestroy`.

```
@PostConstruct
public void init() {
    System.out.println("Bean init")
}

@PreDestroy
public void destroy() {
    System.out.println("Bean destroyed")
}
```

Или через xml конфигурацию:

```
<bean name="Bean" class="com.swchck.spring.beans.Bean" init-method="init" destroy-method="destroy">
    <property name="propertyName" ref="propertyName"/>
</bean>
```

Объясните работу BeanFactory в Spring

BeanFactory - это реализация паттерна Фабрика, его функциональность покрывает создание бинов. Так как эта фабрика знает многие об объектах приложения, то она может создавать связи между объектами на этапе создания экземпляра. Существует несколько реализаций BeanFactory, самая используемся - "org.springframework.beans.factory.xml.XmlBeanFactory". Она загружает бины на основе конфигурационного XML-файла. Чтобы создать XmlBeanFactory передайте конструктору InputStream, например:

```
BeanFactory factory = new XmlBeanFactory(new FileInputStream("Bean.xml"));
```

После этой строки фабрика знает о бинах, но их экземпляры еще не созданы. Чтобы инстанцировать бин нужно вызвать метод getBean(). Например:

```
Bean bean_one = (Bean) factory.getBean("Bean")
```

Как получить объекты ServletContext и ServletConfig внутри Spring Bean

Доступны два способа для получения основных объектов контейнера внутри бина: Реализовать один из Spring*Aware (ApplicationContextAware, ServletContextAware, ServletConfigAware и др.) интерфейсов. Использовать автоматическое связывание @Autowired в спринг. Способ работает внутри контейнера спринг.

```
@Autowired
ServletContext servletContext;
```

В чем роль ApplicationContext в Spring

В то время, как BeanFactory используется в простых приложениях, Application Context - это более сложный контейнер. Как и BeanFactory он может быть использован для загрузки и связывания бинов, но еще он предоставляет:

- возможность получения текстовых сообщений, в том числе поддержку интернационализации
- общий механизм работы с ресурсами;
- события для бинов, которые зарегестрированы как слушатели.

Из-за большей функциональности рекомендуется использование Application Context вместо BeanFactory. Последний используется только в случаях нехватки ресурсов, например при разработке для мобильных устройств

Как выглядит типичная реализция метода используя Spring

Для типичного Spring-приложения нам необходимы следующие файлы: Интерфейс, описывающий функционал приложения Реализация интерфейса, содержащая свойства, сэттеры-гэттеры, функции и т.п. Конфигурационный XML-файл Spring'а. Клиентское приложение, которое использует функцию.

Что такое связывание в Spring и расскажите об аннотации `@Autowired`

Процесс внедрения зависимостей в бины при инициализации называется Spring Bean Wiring. Считается хорошей практикой задавать явные связи между зависимостями, но в Spring предусмотрен дополнительный механизм связывания @Autowired. Аннотация может использоваться над полем или методом для связывания по типу. Чтобы аннотация заработала, необходимо указать небольшие настройки в конфигурационном файле спринг с помощью элемента context:annotation-config.

Каковы различные типы автоматического связывания в Spring

Существует четыре вида связывания в спринг:

- autowire byName,
- autowire byType,
- autowire by constructor,
- autowiring by @Autowired and @Qualifier annotations

Приведите пример часто используемых аннотаций Spring

- `@Controller` - класс фронт контроллера в проекте Spring MVC.
- `@RequestMapping` - позволяет задать шаблон маппинга URI в методе обработчике контроллера.
- `@ResponseBody` - позволяет отправлять Object в ответе. Обычно используется для отправки данных формата XML или JSON.
- `@PathVariable` - задает динамический маппинг значений из URI внутри аргументов метода обработчика.
- `@Autowired` - используется для автоматического связывания зависимостей в spring beans.
- `@Qualifier` - используется совместно с @Autowired для уточнения данных связывания, когда возможны коллизии (например одинаковых имен\типов).
- `@Service` - указывает что класс осуществляет сервисные функции.
- `@Scope` - указывает scope у spring bean.
- `@Configuration` , `@ComponentScan` и `@Bean` - для java based configurations.
- AspectJ аннотации для настройки aspects и advices, `@Aspect` , `@Before` , `@After` , `@Around` , `@Pointcut` и др.

Можем ли мы послать объект как ответ метода обработчика контроллера

Да, это возможно. Для этого используется аннотация @ResponseBody. Так можно отправлять ответы в виде JSON, XML в restful веб сервисах.

Является ли Spring bean потокобезопасным

По умолчанию бин задается как синглтон в Spring. Таким образом все публичные переменные класса могут быть изменены одновременно из разных мест. Так что - нет, не является. Однако поменяв область действия бина на request, prototype, session он станет потокобезопасным, но это скажется на производительности.

Как создать ApplicationContext в программе Java

В независимой Java программе ApplicationContext можно создать следующим образом: AnnotationConfigApplicationContext - при использовании Spring в качестве автономного приложения можно создать инициализировать контейнер с помощью аннотаций. Пример:

```
ApplicationContext context = new AnnotationConfigApplicationContext("bean.xml");
```

ClassPathXmlApplicationContext - получает информацию из xml-файла, находящегося в classpath. Пример:

```
ApplicationContext context = new ClassPathXmlApplicationContext("bean.xml");
```

FileSystemXmlApplicationContext - получает информацию из xml-файла, но с возможностью загрузки файла конфигурации из любого места файловой системы. Пример:

```
ApplicationContext context = new FileSystemXmlApplicationContext("bean.xml");
```

XmlWebApplicationContext - получает информацию из xml-файла за пределами web-приложения.

Можем ли мы иметь несколько файлов конфигурации Spring

С помощью указания contextConfigLocation можно задать несколько файлов конфигурации Spring. Параметры указываются через запятую или пробел:

```
<servlet>
    <servlet-name>app</servlet-name>
    <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
    <init-param>
        <param-name>contextConfigLocation</paramName>
        <param-value>/../../servlet-context-1.xml, /../../servlet-context-2.xml</param-value>
    </init-param>
    <load-on-startup>1</load-on-startup>
</servlet>
```

Поддерживается возможность указания нескольких корневых файлов конфигурации Spring:

```
<context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>../root-context.xml ../root-security.xml</param-value>
</context-param>
```

Файл конфигурации можно импортировать:

```
<bean:import resource="root-security.xml"/>
```

Как внедрить java.util.Properties в Spring Bean

Для возможности использования Spring EL для внедрения свойств (properties) в различные бины необходимо определить propertyConfigure bean, который будет загружать файл свойств.

```
<bean id="propertyConfigurer" class="org.springframework.context.support.PropertySourcesPlaceholderConfigurer">
    <property name="location" value="../application.properties"/>
</bean>

<bean class="com.swchck.spring.dao.impl.TestDaoImpl">
    <property name="maxReadResults" value="${results.read.max}"/>
</bean>
```

Или через аннотации:

```
@Value("${maxReadResults}")
private int maxReadResults;
```

Как настраивается соединение с БД в Spring

Используя datasource `org.springframework.jdbc.datasource.DriverManagerDataSource`. Пример:

```
<bean id="dataSource" class="org.springframework.jdbc.datasource.DriverManagerDataSource">
    <property name="driverClassName">
        <value>org.hsqldb.jdbcDriver</value>
    </property>
    <property name="url">
        <value>jdbc:hsqldb:db/appfuse</value>
    </property>
    <property name="username">
        <value>user</value>
    </property>
    <property name="password">
        <value>password</value>
    </property>
</bean>
```

Как сконфигурировать JNDI не через datasource в applicationContext.xml

Используя `org.springframework.jndi.JndiObjectFactoryBean`. Пример

```
<bean id="dataSource" class="org.springframework.jndi.JndiObjectFactoryBean">
    <property name="jndiName">
        <value>java:../jdbc/</value>
    </property>
</bean>
```

Каким образом можно управлять транзакциями в Spring

Транзакциями в Spring управляют с помощью Declarative Transaction Management (программное управление). Используется аннотация @Transactional для описания необходимости управления транзакцией. В файле конфигурации нужно добавить настройку transactionManager для DataSource.

```
<bean id="transactionsManager" class="org.springframework.jdbc.datasource.DataSourceTransactionsManager">
    <property name="dataSource" ref="dataSource"/>
</bean>
```

Каким образом Spring поддерживает DAO

Spring DAO предоставляет возможность работы с доступом к данным с помощью технологий вроде JDBC, Hibernate в удобном виде. Существуют специальные классы: JdbcDaoSupport, HibernateDaoSupport, JdoDaoSupport, JpaDaoSupport. Класс HibernateDaoSupport является подходящим суперклассом для Hibernate DAO. Он содержит методы для получения сессии или фабрики сессий. Самый популярный метод - getHibernateTemplate(), который возвращает HibernateTemplate. Этот темплейт оборачивает checked-исключения Hibernate в runtime-исключения, позволяя вашим DAO оставаться независимыми от исключений Hibernate. Пример:

```
public class UserDAOHibernate extends HibernateDaoSupport {
    public User getUser(Long id) {
        return (User) getHibernateTemplate().get(User.class, id);
    }
    public void saveUser(User user) {
        getHibernateTemplate().saveOrUpdate(user);
        if (log.isDebugEnabled()) {
            log.debug("userId set to: " + user.getId())
        }
    }
    public void removeUser(Long id) {
        Object user = getHibernateTemplate().load(User.class, id);
        getHibernateTemplate().delete(user);
    }
}
```

Как интегрировать Spring и Hibernate

Для интеграции Hibernate в Spring необходимо подключить зависимости, а так же настроить файл конфигурации Spring. Т.к. настройки несколько отличаются между проектами и версиями, то смотрите официальную документацию Spring и Hibernate для уточнения настроек для конкретных технологий.

Как задаются файлы маппинга Hibernate в Spring

Через applicationContext.xml в web/WEB-INF. Например:

```
<property name="mappingResources">
    <list>
        <value>com/swchck/model/User.hbm.xml</value>
    </list>
</property>
```

Как добавить поддержку Spring в web-приложение

Достаточно просто указать ContextLoaderListener в web.xml файле приложения:

```
<litener>
    <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>
</listener>
```

Можно ли использовать xyz.xml вместо applicationContext.xml

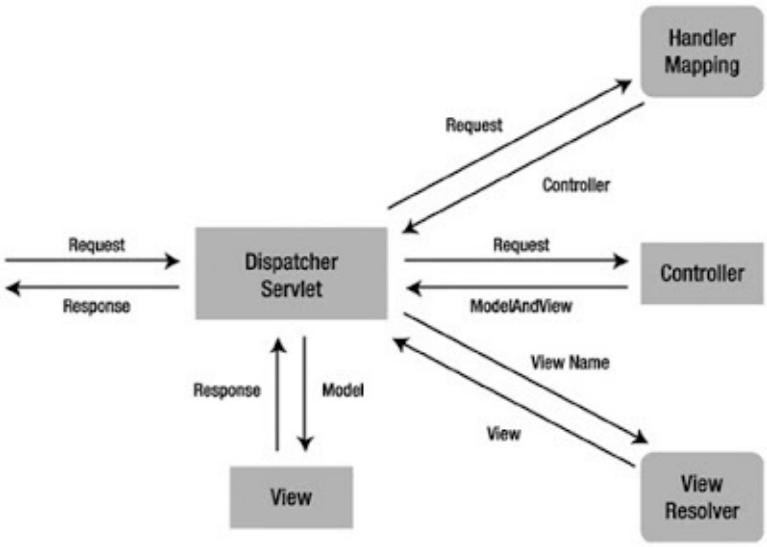
ContextLoaderListener - это ServletContextListener, который инициализируется когда ваше web-приложение стартует. По-умолчанию оно загружает файл WEB-INF/applicationContext.xml. Вы можете изменить значение по-умолчанию, указав параметр contextConfigLocation. Пример:

```
<listener>
    <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>
        <context-param>
```

```
            <param-name>contextConfigLocation</param-name>
            <param-value>/WEB-INF/notapplicationcontext.xml</param-value>
        </context-param>
    </listener-class>
</listener>
```

Что такое контроллер в Spring MVC

Ключевым интерфейсом в Spring MVC является Controller. Контроллер обрабатывает запросы к действиям, осуществляемые пользователями в пользовательском интерфейсе, взаимодействуя с уровнем обслуживания, обновляя модель и направляя пользователей на соответствующие представления в зависимости от результатов выполнения. Controller - управление, связь между моделью и видом.



Основным контроллером в Spring MVC является org.springframework.web.servlet.DispatcherServlet. Задается аннотацией @Controller и часто используется с аннотацией @RequestMapping, которая указывает какие запросы будут обрабатываться этим контроллером.

Какая разница между аннотациями @Component, @Repository и @Service в Spring

- `@Component` - используется для указания класса в качестве компонента spring. При использовании поиска аннотаций, такой класс будет сконфигурирован как spring bean.
- `@Controller` - специальный тип класса, применяемый в MVC приложениях. Обрабатывает запросы и часто используется с аннотацией @RequestMapping.
- `@Repository` - указывает, что класс используется для работы с поиском, получением и хранением данных. Аннотация может использоваться для реализации шаблона DAO.
- `@Service` - указывает, что класс является сервисом для реализации бизнес логики (на самом деле не отличается от Component, но просто помогает разработчику указать смысловую нагрузку класса). Для указания контейнеру на класс-бин можно использовать любую из этих аннотаций. Но различные имена позволяют различать назначение того или иного класса.

Расскажите, что вы знаете о DispatcherServlet и ContextLoaderListener

DispatcherServlet - сервлет диспатчер. Этот сервлет анализирует запросы и направляет их соответствующему контроллеру для обработки. В Spring MVC класс DispatcherServlet является центральным сервлетом, который получает запросы и направляет их соответствующим контроллерам. В приложении Spring MVC может существовать произвольное количество экземпляров DispatcherServlet, предназначенных для разных целей (например, для обработки запросов пользовательского интерфейса, запросов веб-служб REST и т.д.). Каждый экземпляр DispatcherServlet имеет собственную конфигурацию WebApplicationContext, которая определяет характеристики уровня сервлета, такие как контроллеры, поддерживающие сервлет, отображение обработчиков, распознавание представлений, интернационализация, оформление темами, проверка достоверности, преобразование типов и форматирование и т.п. ContextLoaderListener - слушатель при старте и завершении корневого класса Spring WebApplicationContext. Основным назначением является связывание жизненного цикла ApplicationContext и ServletContext, а так же автоматического создания ApplicationContext. Можно использовать этот класс для доступа к бинам из различных контекстов спринг. Настраивается в web.xml:

```
<context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>/WEB-INF/spring/root-context.xml</param-value>
</context-param>

<listener>
    <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>
</listener>
```

Что такое ViewResolver в Spring

ViewResolver - распознаватель представлений. Интерфейс ViewResolver в Spring MVC (из пакета org.springframework.web.servlet) поддерживает распознавание представлений на основе логического имени, возвращаемого контроллером. Для поддержки различных механизмов распознавания представлений предусмотрено множество классов реализации. Например, класс UrlBasedViewResolver поддерживает прямое преобразование логических имен в URL. Класс ContentNegotiatingViewResolver поддерживает динамическое распознавание представлений в зависимости от типа медиа, поддерживаемого клиентом (XML, PDF, JSON и т.д.). Существует также несколько реализаций для интеграции с различными технологиями представлений, такими как FreeMarker (FreeMarkerViewResolver), Velocity (VelocityViewResolver) и JasperReports (JasperReportsViewResolver).

```
<!-- Resolves views selected for rendering by @Controllers to .jsp resources
          in the /WEB-INF/views directory -->
<bean class="org.springframework.web.servlet.view.InternalResource.ViewController">
    <property name="prefix" value="/WEB-INF/views"/>
    <property name+"suffix" value=".jsp"/>
</bean>
```

InternalResourceViewResolver - реализация ViewResolver, которая позволяет находить представления, которые возвращает контроллер для последующего перехода к нему. Ищет по заданному пути, префиксу, суффиксу и имени. Что такое MultipartResolver и когда его использовать? Интерфейс MultipartResolver используется для загрузки файлов. Существуют две реализации: CommonsMultipartResolver и StandardServletMultipartResolver, которые позволяют фреймворку загружать файлы. По умолчанию этот интерфейс не включается в приложении и необходимо указывать его в файле конфигурации. После настройки любой запрос о загрузке будет отправляться этому интерфейсу.

```
<beans:bean id="multipartResolver" class="org.springframework.web.multipart.commons.CommonsMultipartResolver">
    <!-- setting maximum upload size -->
    <beans:property name="maxUploadSize" value="100000"/>
</beans:bean>
```

Как загрузить файл в Spring MVC

Внутри спринг предусмотрен интерфейс MultipartResolver для обеспечения загрузки файлов. Фактически нужно настроить файл конфигурации для указания обработчика загрузки файлов, а затем задать необходимый метод в контроллере spring.

Как обрабатывать исключения в Spring MVC Framework

В Spring MVC интерфейс HandlerExceptionResolver (из пакета org.springframework.web.servlet) предназначен для работы с непредвиденными исключениями, возникающими во время выполнения обработчиков. По умолчанию DispatcherServlet регистрирует класс DefaultHandlerExceptionResolver (из пакета org.springframework.web.servlet.mvc.support). Этот распознаватель обрабатывает определенные стандартные исключения Spring MVC, устанавливая специальный код состояния ответа. Можно также реализовать собственный обработчик исключений, аннотировав метод контроллера с помощью аннотации @ExceptionHandler и передав ей в качестве атрибута тип исключения. В общем случае обработку исключений можно описать таким образом: Controller Based - указать методы для обработки исключения в классе контроллере. Для этого нужно пометить такие методы аннотацией @ExceptionHandler. Global Exception Handler - для обработки глобальных исключений spring предоставляет аннотацию @ControllerAdvice. HandlerExceptionResolver implementation – общие исключений большая часть времени обслуживают статические страницы. Spring Framework предоставляет интерфейс HandlerExceptionResolver, который позволяет задать глобального обработчика исключений. Реализацию этого интерфейса можно использовать для создания собственных глобальных обработчиков исключений в приложении.

Каковы минимальные настройки, чтобы создать приложение Spring MVC

Для создания простого Spring MVC приложения необходимо пройти следующие шаги:

- Добавить зависимости spring-context и spring-webmvc в проект.
- Указать DispatcherServlet в web.xml для обработки запросов внутри приложения.
- Задать определение spring bean (аннотацией или в xml).
- Добавить определение view resolver для представлений.
- Настроить класс контроллер для обработки клиентских запросов.

Как бы вы связали Spring MVC Framework и архитектуру MVC

Модель (Model) - выступает любой Java bean в Spring. Внутри класса могут быть заданы различные атрибуты и свойства для использования в представлении. Преставление (View) - JSP страница, HTML файл и т.п. служат для отображения необходимой информации пользователю. Представление передает обработку запросов к диспетчеру сервлетов (контроллеру). DispatcherServlet (Controller) - это главный контроллер в приложении Spring MVC, который обрабатывает все входящие запросы и передает их для обработки в различные методы в контроллеры.

Как добиться локализации в приложениях Spring MVC

Spring MVC предоставляет очень простую и удобную возможность локализации приложения. Для этого необходимо сделать следующее:

- Создать файл resource bundle, в котором будут заданы различные варианты локализированной информации.
- Определить messageSource в конфигурации Spring используя классы ResourceBundleMessageSource или ResourceBundleMessageSource.
- Определить localceResolver класса CookieLocaleResolver для включения возможности переключения локали.
- С помощью элемента spring:message DispatcherServlet будет определять в каком месте необходимо подставлять локализированное сообщение в ответе.

```
<beans:bean id="messageSource" class="org.springframework.context.support.ReloadableResourceCundleMessageSource">
    <beans:property name="basename" value="classpath:messages"/>
    <beans:property name="defaultEncoding" value="UTF-8"/>
</bean>

<beans:bean id="localeResolver" class="org.springframework.web.servlet.i18n.CookieLocaleResolver">
    <beans:property name="defaultLocale" value="en"/>
```

```
    <beans:property name="cookieName" value="appLocaleCookie"/>
    <beans:property name="cookieMaxAge" value="3600"/>
</beans:bean>

<interceptors>
    <beans:bean class="org.springframework.web.servlet.i18n.LocaleChangeInterceptor">
        <beans:property name="paramName" value="locale"/>
    </beans:bean>
</interceptors>
```

Как мы можем использовать Spring для создания веб-службы RESTful, возвращающей JSON

Spring Framework позволяет создавать Resful веб сервисы и возвращать данные в формате JSON. Spring обеспечивает интеграцию с Jackson JSON API для возможности отправки JSON ответов в resful web сервисе. Для отправки ответа в формате JSON из Spring MVC приложения необходимо произвести следующие настройки: Добавить зависимости Jackson JSON. С помощью maven это делается так:

```
<!-- Jackson -->
<dependency>
    <groupid>com.fasterxml.jackson.core</groupid>
    <artifactid>jackson-databind</artifactid>
    <version>${jackson.databind.version}</version>
</dependency>
```

Настроить бин RequestMappingHandlerAdapter в файле конфигурации Spring и задать свойство messageConverters на использование бина MappingJackson2HttpMessageConverter.

```
<!-- Configure to plugin JSON as request and responce in method handler -->
<beans:bean class="org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerAdapter">
    <beans:property name="messageConverters">
        <beans:list>
            <beans:ref bean="jsonMessageConverter"/>
        </beans:list>
    </beans:property>
</bean>

<!-- Configure bean to convert JSON to POJO and vice versa -->
<beans:bean id="jsonMessageConverter" class="org.springframework.http.converter.json.MappingJackson2HttpMessageConverter"/>
```

В контроллере указать с помощью аннотации @ResponseBody возвращение Object:

```
@RequestMapping(value = EmpRestURIConstants.GET_EMP, method = RequestMethod.GET)
public @ResponseBody Employee getEmployee(@PathVariable("id") int empId) {
    logger.info("getEmployee ID =" + empId);

    return empData.get(empId);
}
```

Как проверить (валидировать) данные формы в Spring Web MVC Framework

Spring поддерживает аннотации валидации из JSR-303, а так же возможность создания своих реализаций классов валидаторов. Пример использования аннотаций:

```
@Size(min = 2, max = 30)
private String name;
```

```
@NotEmpty @Email
private String email;

@NotNull @Min(18) @Max(100)
private Integer age;

@NotNull
private Gender gender;

@DateTimeFormat(pattern = MM/dd/yyyy)
@NotNull @Past
private Date birthday;
```

Что вы знаете Spring MVC Interceptor и как он используется

Перехватчики в Spring (Spring Interceptor) являются аналогом Servlet Filter и позволяют перехватывать запросы клиента и обрабатывать их. Перехватить запрос клиента можно в трех местах: preHandle, postHandle и afterCompletion.

- preHandle - метод используется для обработки запросов, которые еще не были переданы в метода обработчик 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.
- postHandle - вызывается после handler method, но до обработки DispatcherServlet для передачи представлению. Может использоваться для добавления параметров в объект ModelAndView.
- afterCompletion - вызывается после отрисовки представления.

Для создания обработчика необходимо расширить абстрактный класс HandlerInterceptorAdapter или реализовать интерфейс HandlerInterceptor. Так же нужно указать перехватчики в конфигурационном файле Spring.

Расскажите о Spring Security

Проект Spring Security предоставляет широкие возможности для защиты приложения. Кроме стандартных настроек для аутентификации, авторизации и распределения ролей и маппинга доступных страниц, ссылок и т.п., предоставляет защиту от различных вариантов атак (например CSRF). Имеет множество различных настроек, но остается легким в использовании.

Назовите некоторые из шаблонов проектирования, используемых в Spring Framework

Spring Framework использует множество шаблонов проектирования, например:

- Singleton Pattern: Creating beans with default scope.
- Factory Pattern: Bean Factory classes
- Prototype Pattern: Bean scopes
- Adapter Pattern: Spring Web and Spring MVC
- Proxy Pattern: Spring Aspect Oriented Programming support
- Template Method Pattern: JdbcTemplate, HibernateTemplate etc
- Front Controller: Spring MVC DispatcherServlet
- Data Access Object: Spring DAO support
- Dependency Injection and Aspect Oriented Programming

Объясните суть паттерна DI или IoC

Dependency injection (DI) - паттерн проектирования и архитектурная модель, так же известная как Inversion of Control (IoC). DI описывает ситуацию, когда один объект реализует свой функционал через другой объект. Например, соединение с базой данных передается конструктору объекта через аргумент, вместо того чтобы конструктор сам устанавливал соединение. Существуют три формы внедрения (но не типа) зависимостей: сэттер, конструктор и внедрение путем интерфейса. DI - это способ достижения слабой связанности. IoC предоставляет возможность объекту получать ссылки на свои зависимости. Обычно это реализуется через lookup-метод. Преимущество IoC в том, что эта модель позволяет отделить объекты от реализации механизмов, которые он использует. В результате мы получаем большую гибкость как при разработке приложений, так и при их тестировании.

Какие преимущества применения Dependency Injection (DI)

К преимуществам DI можно отнести:

- Сокращение объема связующего кода. Одним из самых больших плюсов DI является возможность значительного сокращения объема кода, который должен быть написан для связывания вместе различных компонентов приложения. Зачастую этот код очень прост - при создании зависимости должен создаваться новый экземпляр соответствующего объекта.
- Упрощенная конфигурация приложения. За счет применения DI процесс конфигурирования приложения значительно упрощается. Для конфигурирования классов, которые могут быть внедрены в другие классы, можно использовать аннотации или XML-файлы.
- Возможность управления общими зависимостями в единственном репозитории. При традиционном подходе к управлению зависимостями в общих службах, к которым относятся, например, подключение к источнику данных, транзакция, удаленные службы и т.п., вы создаете экземпляры (или получаете их из определенных фабричных классов) зависимостей там, где они нужны - внутри зависимого класса. Это приводит к распространению зависимостей по множеству классов в приложении, что может затруднить их изменение. В случае использования DI вся информация об общих зависимостях содержится в единственном репозитории (в Spring есть возможность хранить эту информацию в XML-файлах или Java классах).
- Улучшенная возможность тестирования. Когда классы проектируются для DI, становится возможной простая замена зависимостей. Это особенно полезно при тестировании приложения.
- Стимулирование качественных проектных решений для приложений. Вообще говоря, проектирование для DI означает проектирование с использованием интерфейсов. Используя Spring, вы получаете в свое распоряжение целый ряд средств DI и можете сосредоточиться на построении логики приложения, а не на поддерживающей DI платформе.

Какие IoC контейнеры вы знаете

Spring является **IoC** контейнером. Помимо него существуют **HiveMind**, **Avalon**, **PicoContainer** и т.д.

Как реализуется DI в Spring Framework

Реализация DI в Spring основана на двух ключевых концепциях Java - компонентах JavaBean и интерфейсах. При использовании Spring в качестве поставщика DI вы получаете гибкость определения конфигурации зависимостей внутри своих приложений разнообразными путями (т.е. внешне в XML-файлах, с помощью конфигурационных Java классов Spring или посредством аннотаций Java в коде). Компоненты JavaBean (также называемые POJO (Plain Old Java Object — простой старый объект Java)) предоставляют стандартный механизм для создания ресурсов Java, которые являются конфигурируемыми множеством способов. За счет применения DI объем кода, который необходим при проектировании приложения на основе интерфейсов, снижается почти до нуля. Кроме того, с помощью интерфейсов можно получить максимальную отдачу от DI, потому что бины могут использовать любую реализацию интерфейса для удовлетворения их зависимости.

Какие существуют виды DI? Приведите примеры

Существует два типа DI: через сэттер и через конструктор. Через сэттер: обычно во всех java beans используются гэттеры и сэттеры для их свойств:

```
public class BeanName {
    String name;
```

```
    public void setName(String a) {
      name = a;
    }
    public String getName() {
        return name;
    }
}
```

Мы создаем экземпляр бина NameBean (например, bean1) и устанавливаем нужное свойство, например:

```
bean.setName("Bean");
```

Используя Spring реализация была бы такой:

```
<bean id="bean" class="BeanName">
    <property name="name">
        <value>Bean</value>
    </property>
</bean>
```

Это и называет DI через сэттер. Пример внедрения зависимости между объектами:

```
<bean id="bean" class="Bean_One_Impl">
    <property name="test">
        <ref bean="bean_two"/>
    </property>
</bean>

<bean id="bean_two" class="Bean_Two_Impl"/>
```

Через конструктор: используется конструктор с параметрами. Например:

```
public class BeanName {
    String name;
    public BeanName(String name) {
        this.name = name;
    }
}
```

Теперь мы внедряем объект на этапе создания экземпляра класса, т.е.

```
bean = new BeanName("Bean")
```

Используя Spring это выглядело бы так:

```
<bean id="bean" class="BeanName">
    <constructor-arg>
        <value>Test</value>
    </constructor-arg>
</bean>
```

Java Collections

Что такое Коллекция

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

Назовите основные интерфейсы коллекций и их имплементации

`Collection` расширяет три интерфейса: `List`, `Set`, `Queue`.

`List` - хранит упорядоченные елементы(могут быть одинаковые). Имеет такие реализации как `LinkedList`, `ArrayList` и `Vector`.

- `Vector` синхронизирован, и по этому в одном потоке, он работает медленней остальных реализаций.
- `ArrayList` - его преимущество в навигации по коллекции.
- `LinkedList` - Его преимущество в во вставке и удалении элементов в коллекции.

`Set` - коллекции, которые не содержат повторяющихся элементов. Основные реализации: `HashSet`, `TreeSet`, `LinkedHashSet`

- `TreeSet` - упорядочивает элементы по их значениям;
- `HashSet` - упорядочивает элементы по их хэш ключах, хотя на первый взляд может показаться что элементы хранятся в случайном порядке.
- `LinkedHashSet` - хранит элементы в порядке их добавления

`Queue` - интерфейс для реализации очереди в джава. Основные реализации: `LinkedList`, `PriorityQueue`. Очереди работают по принципу **FIFO – first in first out**.

`Map` - интерфейс для реализации так называемой карты, где элементы хранятся с их ключами. Основные реализации: `HashTable`, `HashMap`, `TreeMap`, `LinkedHashMap`

`HashTable` - синхронизированна, объявлена уставревшей. `HashMap` - порядок елементов рассчитывается по хэш ключу; `TreeMap` - элементы хранятся в отсортированном порядке `LinkedHashMap` - элементы хранятся в порядке вставки

Ключи в `Map` не могут быть одинаковыми! Синхронизировать не синхронизированные коллекции и карты можно посредством класса `Collections.synchronizedMap(MyMap)`, `synchronizedList(MyList)`.

Чем отличается `ArrayList` от `LinkedList`? В каких случаях лучше использовать первый, а в каких второй

Отличие заключается в способе хранения данных. `ArrayList` хранит в виде массива, а `LinkedList` - в виде списка (двунаправленного).

В `ArrayList` быстрее происходит сортировка, т.к. для ее выполнения данные списка копируются в массив (а копировать из массива `ArrayList` в массив для сортировки быстрее).

При большом числе операций добавления и удаления `LinkedList` должен быть более удачным выбором, т.к. при этих операциях не приходится перемещать части массива. Если при добавлении в `ArrayList` превышается его объем, размер массива увеличивается, новая емкость рассчитывается по формуле `(oldCapacity * 3) / 2 + 1`, поэтому лучше указывать размер при создании или, если он не известен, использовать LinkedList (но это может быть существенно при слишком уж больших объемах данных).

Чем отличается `HashMap` от `Hashtable`

Класс `HashMap` по функционалу очень похож на `Hashtable`. Главное отличие в том, что методы класса `Hashtable` синхронизированы, а `HashMap` - нет. Кроме этого класс `HashMap` в отличии от `Hashtable` разрешает использование `null` в качестве ключей и значений.

Наличие синхронизации в `Hashtable` уменьшает производительность кода, использующегоданный класс. Поэтому классы **JCF** (Java Collections Framework, появившийся в Java 2), втом числе и `HashMap`, несинхронизированы. Если синхронизация все же нужна, можноиспользовать методы класса `Collections`: `Collections.synchronizedMap(map)`, `Collections.synchronizedList(list)` или `Collections.synchronizedSet(set)`. Данные методы возвращают синхронизированный декоратор переданной коллекции. При этом всеравно в случае итерирования по коллекции требуется ручная синхронизация.

Начиная с Java 6 **JCF** был расширен специальными коллекциями, поддерживающимимногопоточный доступ, такими как `CopyOnWriteArrayList` и `ConcurrentHashMap`.

Чем отличается `ArrayList` от `Vector`

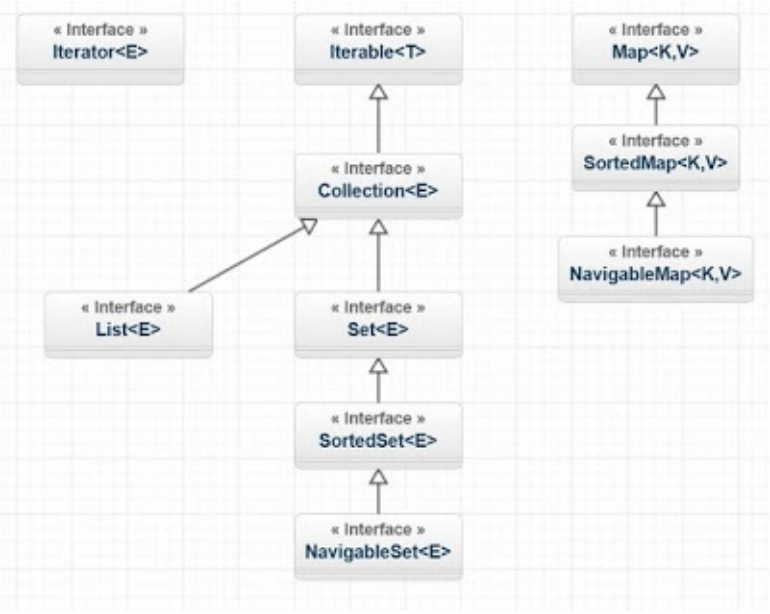
Методы класса `Vector` синхронизированы, в то время как `ArrayList` - нет.

Как сравниваются елементы коллекций

Для сравнения элементов коллекций используется метод `equals()` и `hashcode()`. Эти методы унаследованы от класса `Object`.

- Если наш пользовательский класс переопределяет `equals()`, то он должен и переопределить `hashcode()`
- Если два объекта эквивалентны, то и хэш коды этих объектов тоже должны быть равны
- Если поле не используется в `equals()`, то оно и не должно использоваться в `hashcode()`

Расположите в виде иерархии следующие интерфейсы: `List`, `Set`, `Map`, `SortedSet`, `SortedMap`, `Collection`, `Iterable`, `Iterator`, `NavigableSet`, `NavigableMap`



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". Соответственно некоторые методы интерфейса `Collection` нельзя использовать в `Map` . Например, метод `remove(Object o)` в интерфейсе `Collection` предназначен для удаления элемента, тогда как такой же метод `remove(Object key)` в интерфейсе `Map` - удаляет элемент по заданному ключу.

Дайте определение понятию `Iterator`

Итератор - объект, позволяющий перебирать элементы коллекции. Например foreach реализован с использованием итератора. Одним из ключевых методов интерфейса Collection является метод `Iterator<E>` iterator(). Он возвращает итератор - то есть объект, реализующий интерфейс Iterator. Интерфейс Iterator имеет следующее определение:

```
public interface Iterator<E> {
    E next();
    boolean hasNext();
    void remove();
}
```

Что вы знаете об интерфейсе `Iterable`

Все коллекции из java.util реализуют интерфейс Collection, который, в свою очередь, расширяет интерфейс Iterable. В интерфейсе Iterable описан только один метод: Iterator iterator(); Он возвращает Iterator, т.е. объект, который поочерёдно возвращает все элементы коллекции.

Как одной строчкой преобразовать `HashSet` в `ArrayList`

```
public static void main(String... args) {
    Set<String> set = new HashSet<>();
    set.add("A");
    set.add("B");
    List<String> list = new ArrayList<>(set);
}
```

Как одной строчкой преобразовать `ArrayList` в `HashSet`

```
public static void main(String... args) {
    List<String> list = new ArrayList<>();
    list.add("A");
    list.add("B");
    Set<String> list = new HashSet<>(list);
}
```

Как перебрать все ключи и/или значения `Map` учитывая, что `Map` - это не `Iterable`

Использовать метод `keySet()` , который возвращает множество `Set<K>` ключей для перебора.

Использовать метод `values()` , который возвращает коллекцию `Collection<V>` значений для перебора.

Использовать метод `entrySet()` , который возвращает множество `Set<Map.Entry<K, V>` пар *ключ-значение*.

В чем проявляется “сортированность” `SortedMap`, кроме того, что `toString()` выводит все по порядку

Естественное упорядочивание (natural ordering) отражается при итерации по коллекции ключей или значений хэш-таблицы (возвращаемых методами `keySet()`, `values()` и `entrySet()`).

Как одним вызовом копировать элементы из любой Collection в массив

```
public static void main(String... args) {
    List<String> list = new ArrayList<>();
    list.add("A");
    list.add("B");

    String[] strArr = list.toArray(new String[list.size()]);
    // OR
    Object[] objArr = list.toArray();
}
```

Реализуйте симметрическую разность двух коллекций используя методы `Collection` `addAll()`, `removeAll()`, `retainAll()`

Симметрическая разность двух коллекций - это множество элементов, одновременно не принадлежащих обоим исходным коллекциям.

```
public static <T> Collection<T> symmetricDifference(Collection<T> a, Collection<T> b) {
    // Создаем новую коллекцию, чтобы не изменять исходные
    Collection<T> intersection = new ArrayList<>(a)
    // Получаем пересечение коллекций
    intersection.retainAll(b);
    Collection<T> result = new ArrayList<>(a);
    // Объеденяем коллекции
    result.addAll(b);
    // Удаляем элементы, расположеннные в обоих коллекциях
    result.removeAll(intersection);

    return result;
}

public static void main(String... args) {
    List<String>a = new ArrayList<>(Arrays.asList("1", "2", "3", "4", "5"))
    List<String>a = new ArrayList<>(Arrays.asList("3", "4", "5", "6", "7"))
    Collection<String> c symmetricDifference(a, b);
     System.out.println("Collection a" + Arrays.toString(a.toArray()));
     System.out.println("Collection b" + Arrays.toString(b.toArray()));
     System.out.println("Collection c" + Arrays.toString(c.toArray()));
```

Сравните `Enumeration` и `Iterator`

Оба интерфейса предназначены для обхода коллекций. Интерфейс `Iterator` был введен несколько позднее в Java Collections Framework и его использование предпочтительнее. Основные различия `Iterator` по сравнению с `Enumeration`:

- Наличие метода `remove()` для удаления элемента из коллекции при обходе;
- Исправлены имена методов для повышения читаемости кода.

Как между собой связаны Iterable и Iterator

Интерфейс `Iterable` имеет только один метод - `iterator()`, который возвращает итератор коллекции для её обхода.

Как между собой связаны Iterable, Iterator и «for-each» введенный в Java 5

Экземпляры классов, реализующих интерфейс `Iterable`, могут использоваться в конструкции `foreach`.

Сравните `Iterator` и `ListIterator`

`ListIterator` расширяет интерфейс `Iterator`, позволяя клиенту осуществлять обход коллекции в обоих направлениях, изменять коллекцию и получать текущую позицию итератора. При этом важно помнить, что `ListIterator` не указывает на конкретный элемент, а его текущая позиция располагается между элементами, которые возвращают методы `previous()` и `next()`. Таким образом, модификация коллекции осуществляется для последнего элемента, который был возвращен методами `previous()` и `next()`.

Что произойдет, если я вызову `Iterator.next()` не «спросив» `Iterator.hasNext()`

Если итератор указывает на последний элемент коллекции, то возникнет исключение `NoSuchElementException`, иначе будет возвращен следующий элемент.

Что произойдет, если я вызову `Iterator.next()` перед этим 10 раз вызвав `Iterator.hasNext()`? Я пропущу 9 элементов

Нет, `hasNext()` осуществляет только проверку наличия следующего элемента.

Если у меня есть коллекция и порожденный итератор, изменится ли коллекция, если я вызову `iterator.remove()`

Вызов метода `iterator.remove()` возможен только после вызова метода `iterator.next()` хотя бы раз, иначе появится исключение `IllegalStateException`. Если `iterator.next()` был вызван прежде, то `iterator.remove()` удалит элемент, на который указывает итератор.

Если у меня есть коллекция и порожденный итератор, изменится ли итератор, если я вызову `collection.remove()`

Итератор не изменится, но при следующем вызове его методов возникнет исключение `ConcurrentModificationException`

Зачем добавили `ArrayList`, если уже был `Vector`

Обе структуры данных предназначены для хранения коллекции элементов, в том числе дупликатов и `null`. Они основаны на использовании массивов, динамически расширяющихся при необходимости. Класс `Vector` был введен в JDK 1.0 и не является частью Java Collection Framework. Методы класса `Vector` синхронизированы, что обеспечивает потокобезопасность, но это приводит к снижению производительности, поэтому и был введен класс `ArrayList`, методы которого не синхронизированы.

В реализации класса `ArrayList` есть следующие поля: `Object[] elementData` , `int size` . Объясните, зачем хранить отдельно `size` , если всегда можновзять `elementData.length`

Размер массива `elementData` представляет собой вместимость (capacity) `ArrayList` , которая всегда больше переменной `size` - реального количества хранимых элементов. С добавлением новых элементов вместимость автоматически возрастает при необходимости.

`LinkedList` - это односвязный, двусвязный или четырехсвязный список

Двухсвязный список: каждый элемент `LinkedList` хранит ссылку на предыдущий и следующий элементы.

Какое худшее время работы метода `contain()` для элемента, который есть в `LinkedList` , `ArrayList` (`O(1)` , `O(log(N))` , `O(N)` , `O(N*log(N))` , `O(N*N)`)

`O(N)` . Время поиска элемента линейно пропорционально количеству элементов с списке.

Какое худшее время работы метода `add()` для `LinkedList` (`O(1)` , `O(log(N))` , `O(N)` , `O(N*log(N))` , `O(N*N)`)

`O(N`). Здесь стоит заметить, что добавление элемента в конец списка с помощью методом `add(value)` , `addLast(value)` и добавление в начало списка с помощью `addFirst(value)` выполняется за время `O(1)` . `O(N)` - будет при добавление элемента в отсортированный список, а также при добавлении элемента с помощью метода `add(index, value)` .

Какое худшее время работы метода `add()` для `ArrayList` (`O(1)` , `O(log(N))` , `O(N)` , `O(N*log(N))` , `O(N*N)`)

`O(N)` . Вставка элемента в конец списка осуществляется за время `O(1)` , но если вместимость массива недостаточна, то происходит создание нового массива с увеличенным размером и копирование всех элементов из старого массива в новый.

Сколько выделяется элементов в памяти при вызове `ArrayList.add()`

Если в массиве достаточно места для размещения нового элемента, то дополнительное место в памяти не выделяется. Иначе происходит создание нового массива с размером:

```
int oldCapacity = elementData.length;
int newCapacity = oldCapacity + (oldCapacity >> 1);
```

Другими словами, создается новый массив, размер которого вычисляется как умножение старого размера на 1.5 (это верно для JDK 1.7, в более ранних версиях вычисления отличаются).

Сколько выделяется элементов в памяти при вызове `LinkedList.add()`

Создается один новый экземпляр вложенного класса `Node`

Оцените количество памяти на хранение одного примитива типа `byte` в `LinkedList`

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. Примитив типа `byte` занимает 1 байт памяти, но в списке примитивы упаковываются, соответственно получаем еще 8 байт. Таким образом, в x32 JVM около 32 байтоввыделяется для хранения одного значения типа `byte` в `LinkedList`. Для 64-битной JVM каждая ссылка занимает 64 бита (8 байт). Вычисления аналогичны.

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.

Я добавляю элемент в середину `List`: `list.add(list.size()/2, newElem)`. Для кого эта операция медленнее — для `ArrayList` или для `LinkedList`

Для `ArrayList`:

- проверка массива на вместимость. Если вместимости недостаточно, то - увеличение размера массива и копирование всех элементов в новый массив `O(N)`;
- копирование всех элементов, расположенных правее от позиции вставки, на одну позицию вправо (`O(N/2)`);
- вставка элемента `O(1)`.

Для `LinkedList`:

- поиск позиции вставки `O(N/2)`;
- вставка элемента `O(1)`.

В худшем случае вставка в середину списка эффективнее для `LinkedList`. В остальных - скорее всего, для `ArrayList`, поскольку копирование элементов осуществляется за счет системного метода `System.arraycopy()`.

Как перебрать элементы `LinkedList` в обратном порядке, не используя медленный `get(index)`

Использовать обратный итератор. Для этого в `LinkedList` есть метод `descendingIterator()`

Как одним вызовом из `List` получить `List` со всеми элементами, кроме первых и последних 3-х

```
List<Integer> sourceList = new ArrayList<>(Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9));
List<Integer> subList = sourceList.subList(3, sourceList.size( - 3));
```

Могут ли у разных объектов в памяти `ref0 != ref1` быть `ref0.hashCode() == ref1.hashCode()`

Да, могут. Метод `hashCode()` не гарантирует уникальность возвращаемого значения

Могут ли у разных объектов в памяти `ref0 != ref1` быть `ref0.equals(ref1) == true`

Да, могут. Для этого в классе этих объектов должен быть переопределен метод `equals()`. Если используется метод `Object.equals()`, то для двух ссылок x и y метод вернет `true` тогда и только тогда, когда обе ссылки указывают на один и тот же объект (т.е. `x == y` возвращает `true`).

Могут ли у разных ссылок на один объект в памяти `ref0 == ref1` быть `ref0.equals(ref1) == false`

Нет, не может. Метод `equals()` должен гарантировать свойство рефлексивности: для любых ненулевых ссылок метод `x.equals(x)` должен возвращать `true`.

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.

Если у класса `Point{int x, y;}` «правильно» реализовать метод `equals` (`return ref0.x == ref1.x && ref0.y == ref1.y`), но сделать хэш-код в виде `int hashCode() {return x;}`, то будут ли корректно такие точки помещаться и извлекаться из `HashSet`

`HashSet` использует `HashMap` для хранения элементов (в качестве ключа используется сам объект). При добавлении элемента в `HashMap` вычисляется хэшкод и позиция в массиве, куда будет вставлен новый элемент. У всех экземпляров класса `Point` одинаковый хэшкод, что приводит в вырождению хэш-таблицы в список. При возникновении коллизии осуществляется проверка на наличие уже такого элемента в текущем списке:

```
e.hash == hash && ((k = e.kay) == key || key.equals(k));
```

Если элемент найден, то его значение перезаписывается. В нашем случае для разных объектов метод `equals()` будет возвращать `false`. Соответственно новый элемент будет добавлен в `HashSet`. Извлечение элемента также будет осуществляться успешно. Но производительность такого кода будет низкой и преимущества хэш-таблиц использоваться не будут.

`equals()` порождает отношение эквивалентности. Какими из свойств обладает такое отношение: коммутативность, симметричность, рефлексивность, дистрибутивность, ассоциативность, транзитивность

Метод `equals()` должен обеспечивать:

- симметричность (для любых ненулевых ссылок x и y метод `x.equals(y)` должен возвращать `true` тогда и только тогда, когда `y.equals(x)` возвращает `true`)
- рефлексивность (для любых ненулевых ссылок x метод `x.equals(x)` должен возвращать `true`.)
- транзитивность (для любых ненулевых ссылок x, y и z, если `x.equals(y)` возвращает `true` и `y.equals(z)` возвращает `true`, тогда и `x.equals(z)` должен возвращать `true`).

Также есть ещё два свойства: постоянство и неравенство `null`.

Можно ли так реализовать `equals(Object that) {return this.hashCode() == that.hashCode()}`

Строго говоря нельзя, поскольку метод `hashCode()` не гарантирует уникальность значения для каждого объекта. Однако для сравнения экземпляров класса `Object` такой код допустим, т.к. метод `hashCode()` в классе `Object` возвращает уникальные значения для разных объектов (вычисления основаны на использовании адреса объекта в памяти).

В `equals` требуется проверять, что аргумент `equals(Object that)` такого же типа как и сам объект. В чем разница между `this.getClass() == that.getClass()` и `that instanceof MyClass`

Оператор `instanceof` сравнивает объект и указанный тип. Его можно использовать для проверки является ли данный объект экземпляром некоторого класса, либо экземпляром его дочернего класса, либо экземпляром класса, который реализует указанный интерфейс. `getClass() = ...` проверяет два типа на идентичность.

Для корректной реализации контракта метода `equals()` необходимо использовать точное сравнение с помощью `getClass()`.

Можно ли реализовать метод `equals` класса `MyClass` вот так: `class MyClass {public boolean equals(MyClass that) {return this == that;}}`

Реализовать можно, но данный метод не переопределяет метод `equals()` класса `Object`, а перегружает его.

Будет ли работать `HashMap`, если все ключи будут возвращать `int hashCode() {return 42;}`

Да, будет. Но тогда хэш-таблица вырождается в связный список и теряет свои преимущества.

Зачем добавили `HashMap`, если уже был `Hashtable`

Класс `Hashtable` был введен в JDK 1.0 и не является частью Java Collection Framework. Методы класса `Hashtable` синхронизированы, что обеспечивает потокобезопасность, но это приводит к снижению производительности, поэтому и был введен класс `HashMap`, методы которого не синхронизированы. Помимо этого класс `HashMap` обладает некоторыми другими отличиями: например, позволяет хранить один `null` ключ и множество `null` значений.

Согласно Кнуту и Кормену существует две основных реализации хэш-таблицы: на основе открытой адресацией и на основе метода цепочек. Как реализована `HashMap`? Почему так сделали (по вашему мнению)? В чем минусы и плюсы каждого подхода

Класс `HashMap` реализован с использованием метода цепочек, т.е. каждой ячейке массива соответствует свой связный список и при возникновении коллизии осуществляется добавление нового элемента в этот список.

Для метода цепочек коэффициент заполнения может быть больше 1, с увеличением числа элементов производительность убывает линейно. Такие таблицы удобно использовать, если заранее неизвестно количество хранимых элементов, либо их может быть достаточно много, что приводит к большим значениям коэффициента заполнения.

Среди методов открытой реализации различают:

- линейное пробирование
- квадратичное пробирование
- двойное хеширование

Основные недостатки структур с методом открытой адресации:

- Количество элементов в таблице не может превышать размера массива. По мере увеличения числа элементов в таблице и повышения коэффициента заполнения (load factor) производительность структуры резко падает, поэтому необходимо проводить перехеширование.
- Сложно организовать удаление элемента.
- Также первые два метода открытой адресации приводят к проблеме первичной и вторичной группировок.

Основное преимущество хэш-таблицы с открытой адресацией - это отсутствие затрат на создание и хранение объектов списка. Также проще организовать сериализацию/десериализацию объекта.

Сколько переходов по ссылкам происходит, когда вы делаете `HashMap.get(key)` по ключу, который есть в таблице

Возможно, я неправильно понял этот вопрос. За переходы по ссылке в данном ответе я считаю вызовы методов.

```
public V get(Object key) {
    if (key == null) {
        return getForNullKey();
    }
    Entry<K, V> entry = getEntryKey(key);

    return null == entry ? null : entry.getValue();
}
```

Рассмотрим первый случай, когда ключ равен null: выполняем метод getForNullKey().

```
public V get(Object key) {
    if (size == 0) {
        return null;
    }
    for (Entry<K, V> e = tabke[0]; e != null; e = e.next) {
        if (e.key == null)
            return e.value;
    }
    return null;
}
```

В цикле foreach проходимся по списку значений для ключа и возвращаем нужное значение. Таким образом, получаем 1 переход. Второй случай: ключ не равен null. Выполняем метод getEntry(key).

```
final Entry<K, V> getEntry(Oject key) {
    if (size == 0) {
        return null;
    }

    int hash = (key == null) ? 0 : hash(key);
    for (Entry<K, V> e table[indexFor(hash, table.length)]; e != null; e = e.next) {
        Object k;
        if (e.hash == hash && ((k = e.key) == key || (key != null && key.equals(k)))) {
            return e;
        }
    }
    return null;
}
```

Вычисляется хэш-код ключа (метод `hash(key)`), затем определяется индекс ячейки массива, в которой будем искать значение метод `indexFor(hash, table.length)`. После того, как нашли нужную пару "ключ-значение" возвращаем значение метод `entry.getValue()`. Таким образом, получаем 4 перехода.

Сколько создается новых объектов, когда вы добавляете новый элемент в `HashMap`

Один новый объект статического вложенного класса `Entry<K,V>`

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

По значению `hashCode` вычисляется индекс ячейки массива, в список которой будет происходить добавление элемента. Перед добавлением осуществляется проверка на наличие уже элементов в этой ячейке. Если элементов нет, то происходит добавление. Если возникает коллизия, то итеративно осуществляется обход списка в поисках элемента с таким же ключом и хэш-кодом. Если такой элемент найден, то его значение перезаписывается, а старое - возвращается. Поскольку в условии сказано, что добавляемые ключи - разные, то второй элемент будет добавлен в начало списка.

`HashMap` может выродиться в список даже для ключей с разным `hashCode`. Как это возможно

Это возможно в случае, если метод, определяющий номер ячейки массива по `hashCode` будет возвращать одинаковое значение

Какое худшее время работы метода `get(key)` для ключа, которого нет в таблице `O(1)`, `O(log(N))`, `O(N)`, `O(N*log(N))`, `O(N*N)`

`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 таблице `O(1)`, `O(log(N))`, `O(N)`, `O(N*log(N))`, `O(N*N)`

`O(N)`. Аналогичные рассуждения, что и для предыдущего вопроса.

Объясните смысл параметров в конструкторе `HashMap(int initialCapacity, float loadFactor)`

`int initialCapacity` - исходный размер `HashMap` (количество корзин в хэш-таблице в момент её создания), по умолчанию имеет значение 16. `float loadFactor` - коэффициент заполнения `HashMap`. Равен отношению числа хранимыхэлементов в таблице к её размеру. `loadFactor` - является мерой заполнения таблицыэлементами, при превышении количества хранимых таблицей значений, происходитавтоматическое перехеширование. Значение по умолчанию 0.75 является хорошим компромиссоммежду временем доступа и объемом хранимых данных.

В чем разница между `HashMap` и `IdentityHashMap`? Для чего нужна `IdentityHashMap`? Как может быть полезна для реализации сериализации или клонирования

`IdentityHashMap` - это структура данных, реализующая интерфейс `Map`, но использующая сравнение ссылок вместо метода `equals()` при сравнении ключей (значений). Другими словами, в `IdentityHashMap` два ключа k1 и k2 будут рассматриваться равными, если выполняется условие k1 == k2. `IdentityHashMap` не использует метод `hashCode()`, вместо которого применяется метод `System.identityHashCode(Object)`. Другое отличие (как следствие) заключается в более высокой производительности `IdentityHashMap` по сравнению с `HashMap`, если последний хранит объекты с дорогостоящими методами `equals()` и `hashCode()`. Одним из основных требований к использованию `HashMap` является неизменяемость ключа, однако это требование не распространяется на `IdentityHashMap`, который не использует методы `equals()` и `hashCode()`. Согласно документации, такая структура данных может применяться для реализации сериализации/клонирования. Для выполнения подобных алгоритмов программе необходимо обслуживать таблицу со всеми ссылками на объекты, которые уже были обработаны. Такая таблица не должна рассматривать уникальные объекты как равные, даже если метод `equals()` возвращает `true`.

В чем разница между `HashMap` и `WeakHashMap`? Для чего нужна `WeakHashMap`

Перед рассмотрением `WeakHashMap` кратко напомню, что такое *WeakReference*. В Java существует 4 типа ссылок: сильные (strong reference), мягкие (SoftReference), слабые (WeakReference) и фантомные (PhantomReference). Особенности каждого типа ссылок связаны с работой Garbage `Collector`. Если объект можно достичь только с помощью цепочки WeakReference (то есть на него не ссылаются сильные и мягкие ссылки), то данный объект будет отмечен для удаления.

`WeakHashMap` - это структура данных, реализующая интерфейс `Map` и основанная на использовании WeakReference для хранения ключей. Таким образом, пара "ключ-значение" будет удалена из WeakHashMap, если на объект-ключ более не имеется сильных ссылок.

В качестве примера использования такой структуры данных можно привести следующую ситуацию: допустим имеются объекты, которые необходимо расширить дополнительной информацией, при этом изменение класса этих объектов нежелательно либо невозможно. В этом случае добавляем каждый объект в `WeakHashMap` в качестве ключа, а в качестве значения - нужную информацию. Таким образом, пока на объект имеется сильная ссылка (либо мягкая), можно проверять хэш-таблицу и извлекать информацию. Как только объект будет удален, то WeakReference для этого ключа будет помещен в `ReferenceQueue` и затем соответствующая запись для этой слабой ссылки будет удалена из `WeakHashMap`.

В `WeakHashMap` используются WeakReferences. А почему бы не создать `SoftHashMap` на SoftReferences

`SoftHashMap` представлена в стронних библиотеках, например, в Apache Commons.

В `WeakHashMap` используются WeakReferences. А почему бы не создать `PhantomHashMap` на PhantomReferences

PhantomReference при вызове метода `get()` возвращает всегда `null`, поэтому, я думаю, создание `PhantomHashMap` просто невозможно. Плюс назначение такой структуры данных тяжело представить.

Сделайте `HashSet` из `HashMap` (используйте только множество ключей, но не множество значений)

```
Set<Objct> keySet = new HashSet<>(map.keySet());
```

Сделайте `HashMap` из `HashSet` `HashSet<Map.Entry<K, V>>`

```
Map<K, V> map = new HashMap<>(set.size());
for (Map.Entry<K, V> entry : set) {
    map.put(etry.getKey(), entry.getValue());
}
```

Сравните интерфейсы `java.util.Queue` и `java.util.Deque`

Согласно документации `Deque` ("дек", Double Ended Queue) - это линейная коллекция, поддерживающая вставку/извлечение элементов с обоих концов. Помимо этого реализации интерфейса `Deque` могут строится по принципу FIFO, либо LIFO.

`Queue` - это очередь, обычно (но необязательно) строится по принципу FIFO (First-In-First-Out) - соответственно извлечение элемента осуществляется с начала очереди, вставка элемента - в конец очереди. Этот принцип нарушает, к примеру, приоритетная очередь `PriorityQueue`, использующая переданный comparator при вставке нового элемента, либо расстановка элементов осуществляется согласно естественному упорядочиванию (natural ordering). `Deque` расширяет `Queue`. Реализации и `Deque`, и `Queue` обычно не переопределяют методы `equals()` и `hashCode()`, основанные на сравнении хранящихся элементов. Вместо этого используются унаследованные методы класса `Object`, основанные на сравнении ссылок.

Кто кого расширяет: `Queue` расширяет `Deque`, или `Deque` расширяет `Queue`

`Deque` расширяет `Queue`.

Почему `LinkedList` реализует и `List`, и `Deque`

`LinkedList` позволяет добавлять элементы в начало и конец списка за константное время, что хорошо подходит для реализации интерфейса `Deque` (в отличие, например, от `ArrayList`).

В чем разница между классами `java.util.Arrays` и `java.lang.reflect.Array`

- `java.util.Arrays` - класс, содержащий статические методы для работы с массивами, таких как, например, поиск по массиву и его сортировка.
- `java.lang.reflect.Array` - класс для работы с массивами при использовании рефлексии. Рефлексия - это механизм, позволяющий исследовать данные о программе во время её выполнения.

В чем разница между классами `java.util.Collection` и `java.util.Collections`

Класс `java.util.Collections` 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.

`java.util.Collection` - это корневой интерфейс Java Collections Framework. Этот интерфейс в основном применяется там, где требуется высокий уровень абстракции, например, в классе `java.util.Collections`.

Напишите не многопоточную программу, которая заставляет коллекцию выбросить `ConcurrentModificationException`

```
List<String> stringList = new ArrayList<>();
stringList.add("1");
stringList.add("2");
stringList.add("3");

Iterator<String> listIterator = stringList.iterator();
stringList.remove("2");
while (listIterator.hasNext()) {
    System.out.println(listIterator.next());
}
```

Что такое “fail-fast поведение”

Fail-fast поведение означает, что при возникновении ошибки или состояния, которое может привести к ошибке,система немедленно прекращает дальнейшую работу и уведомляет об этом.

В Java Collections API итераторы могут использовать либо fail-fast, либо fail-safe поведение, либо быть weakly consistent. Итератор с fail-fast поведением выбросит исключение `ConcurrentModificationException`, если после его создания была произведена модификация коллекции, т.е. добавлен или удален элемент (без использования метода `remove()` итератора). Реализация такого поведения осуществляется за счет подсчета количества модификаций коллекции (modification count): при изменении коллекции (удаление/добавление элемента) счетчик увеличивается; при создании итератора ему передается текущее значение счетчика; при каждом обращении к итератору сохраненное значение счетчика сравнивается с текущим, и, если они не совпадают, возникает исключение. Использование fail-fast подхода позволяет избежать недетерминированного поведения программы в течение времени. Также стоит отметить, что fail-fast поведение не может быть абсолютно гарантировано.

Для множеств enum-ов есть специальный класс `java.util.EnumSet`? Зачем? Чем авторов не устраивал `HashSet` или `TreeSet`

`EnumSet` - это одна из разновидностей реализации интерфейса `Set` для использования с перечислениями (`Enum`). `EnumSet` использует массив битов для хранения значений (bit vector), что позволяет получить высокую компактность и эффективность.

В структуре данных хранятся объекты только одного типа `Enum`, который указывается при создании экземпляра `EnumSet`. Все основные операции выполняются за константное время `O(1)` и в основном несколько быстрее (хотя и негарантированно), чем их аналоги в реализации `HashSet`. Пакетные операции bulk operations, например, `containsAll()` и `retainAll()` выполняются очень быстро, если их аргументом является экземпляр типаEnum.

Помимо этого класс `EnumSet` предоставляет множество статических методов инициализации для упрощенного и удобного создания экземпляров. Итерация по `EnumSet` осуществляется согласно порядку объявления элементов перечисления.

`java.util.Stack` - считается «устаревшим». Чем его рекомендуют заменять? Почему

Рекомендуется использовать интерфейс `Deque` ("дек", Double Ended Queue) и его реализации. Например:

```
Deque<Integer> stack = new ArrayDeque<Integer>();
```

Стек - это структура данных, построенная на принципе LIFO (Last-In-First-Out, либо по-другому FILO). Каждое новое значение добавляется на "вершину" стека, а извлекается последний добавленный элемент (с "вершины" стека). При извлечении элемента он удаляется из структуры данных. Класс Stack появился в JDK 1.0 и расширяет класс `Vector`, наследуя его функционал, что несколько нарушает понятие стека (например, класс `Vector` предоставляет возможность обращаться к любому элементу по

индексу). Также использование `Deque` позволяет следовать принципу программирования на уровне интерфейсов, а не конкретных реализаций, что облегчает дальнейшую поддержку разрабатываемого класса и повышает его гибкость, позволяя при необходимости менять реализацию дека на нужную.

Какая коллекция реализует дисциплину обслуживания FIFO

FIFO - First-In-First-Out (первый пришел, первым ушел). По этому принципу обычно построена такая структура данных как очередь (`java.util.Queue`).

Какая коллекция реализует дисциплину обслуживания FILO

FILO - First-In-Last-Out (первый пришел, последним ушел). По этому принципу построена такая структура данных как стек (`java.util.Stack`).

Приведите пример, когда какая-либо коллекция выбрасывает `UnsupportedOperationException`

```
List fixedList = Arrays.asList("1", "2", "3");
Iterator<String> listIterator = fixedList.iterator();
while (iterator.hasNext()) {
    String currentElement = iterator.next();
    if ("2".equals(currentElement)) {
        iterator.remove();
    }
}
```

В данном примере возникнет исключение `UnsupportedOperationException` , поскольку метод `asList()` возвращает список фиксированной длины, т.е. удаление/добавление элементов в такой список не поддерживается.

Почему нельзя написать `ArrayList<List> numbers = new ArrayList<ArrayList>();` но можно `List<ArrayList> numbers = new ArrayList<ArrayList>();`

Это связано с ограничениями использования generic types (обобщенных типов). `ArrayList<ArrayList>` не является подтипом `ArrayList<List>` , соответственно использование такой записи запрещено.

`LinkedHashMap` - что это еще за «зверь»? Что в нем от `LinkedList` , а что от `HashMap`

Реализация `LinkedHashMap` отличается от `HashMap` поддержкой двухсвязанного списка, определяющего порядок итерации по элементам структуры данных. По умолчанию элементы списка упорядочены согласно их порядку добавления в `LinkedHashMap` (insertion-order). Однако порядок итерации можно изменить, установив параметр конструктора accessOrder в значение `true` . В этом случае доступ осуществляется по порядку последнего обращения к элементу (access-order). Это означает, что при вызове методов `get()` или `put()` элемент, к которому обращаемся, перемещается в конец списка. При добавлении элемента, который уже присутствует в `LinkedHashMap` (т.е. с одинаковым ключом), порядок итерации по элементам не изменяется.

`LinkedHashSet` - что это еще за «зверь»? Что в нем от `LinkedList` , а что от `HashSet`

Реализация `LinkedHashSet` отличается от `HashSet` поддержкой двухсвязанного списка, определяющего порядок итерации по элементам структуры данных. Элементы списка упорядочены согласно их порядку добавления в `LinkedHashSet` (insertion-order). При добавлении элемента, который уже присутствует в `LinkedHashSet` (т.е. с одинаковым ключом), порядок итерации по элементам не изменяется.

Говорят, на `LinkedHashMap` легко сделать простенький кэш с *invalidation policy*, знаете как

Необходимо использовать LRU-алгоритм (Least Recently Used algorithm) и `LinkedHashMap` с access-order. В этом случае при обращении к элементу он будет перемещаться в конец списка, а наименее используемые элементы будут постепенно группироваться в начале списка. Для этого в стандартной реализации `LinkedHashMap` (source) есть метод `removeEldestEntries()`, который возвращает true, если текущий объект `LinkedHashMap` должен удалить наименее используемый элемент из коллекции. Метод вызывается при использовании методов `put()` и `putAll()`:

```
void addEntry(int hash, K key, V value, int bucketIndex) {
    createEntry(hash, key, value, bucketIndex);

    Entry<K, V> eldest = header.after;
    if (removeEldestEntry(eldest)) {
        removeEntryForKey(eldest.key);
    } else {
        if (size >= treshold) {
            resize(2 * table.length);
        }
    }
}
```

Простой пример реализации кэша с очисткой старых значений при превышении указанного порога:

```
public class LRUChache<K, V> extends LinkedHashMap<K, V> {
    private static final int MAX_ENTIRES = 10;

    public LRUChache(int initialCapacity) {
        super(initialCapacity, 0,85f, true);
    }

    @Override
    protected boolean removeEldestEntry(Map.Entry<K, V> eldest) {
        return size() > MAX_ENRIES;
    }
}
```

Стоит заметить, что `LinkedHashMap` не позволяет полностью реализовать LRU-алгоритм, поскольку при вставке уже имеющегося в коллекции элемента порядок итерации не меняется.

Что позволяет сделать PriorityQueue

`PriorityQueue` - это структура данных, располагающая элементы в порядке натурального упорядочивания, либо используя переданный конструктору `Comparator`. Используя `PriorityQueue`, можно, например, реализовать алгоритм Дейкстры для поиска кратчайшего пути от одной вершины графа к другой. Либо применять для хранения объектов согласно их приоритету: например, сортировка пациентов врача - экстренные пациенты перемещаются в начало очереди, менее срочные пациенты - ближе к концу очереди.

В чем заключаются отличия java.util.Comparator от java.lang.Comparable

Interface `Comparable` задает свойство сравнения объекту реализующему его. То есть делает объект сравнимым (по правилам разработчика). Interface `Comparator` позволяет создавать объекты, которые будут управлять процессом сравнения (например при сортировках).

JDBC

Что такое JDBC

API JDBC (Java DataBase Connectivity) - стандартный прикладной интерфейс языка Java для организации взаимодействия между приложением и СУБД. Взаимодействие осуществляется с помощью драйверов JDBC, обеспечиваю щих реализацию общих интерфейсов для конкретных СУБД и конкретних протоколов. В JDBC определяются четыре типа драйверов.

Этапы работы с базой данных с использованием JDBC

Этапы работы с базой данных с использованием JDBC:

- Подключение библиотеки с классом-драйвером базы данных
- Установка соединения с БД
- Создание объекта для передачи запросов
- Выполнение запроса
- Обработка результатов выполнения запроса
- Закрытие соединения

Как создать Connection

Для установки соединения с БД вызывается статический метод `getConnection()` класса `java.sql.DriverManager`. В качестве параметров методу передаются URL базы данных, логин пользователя БД и пароль доступа. Загрузка класса драйвера базы данных при отсутствии ссылки на экземпляр этого класса в JDBC происходит автоматически при установке соединения екземпляром `DriverManager`. Метод возвращает объект `Connection`. URL базы данных, состоящий из типа и адреса физического расположения БД, может создаваться в виде отдельной строки или извлекаться из файла ресурсов. Соответственно:

```
Connection cn = DriverManager.getConnection("jdbc:sql://localhost:port/dbName", "user", "pass")
```

В результате будет возвращен объект `Connection` и будет одно установленное соединение с БД с именем testphones. Класс `DriverManager` предоставляет средства для управления набором драйверов баз данных.

Чем отличается `Statement` от `PreparedStatement`

Объект `Statement` используется для выполнения SQL-запросов к БД. Существует три типа объектов `Statement`. Все три служат как бы контейнерами для выполнения SQL-выражений через данное соединение: `Statement`, `PreparedStatement`, наследующий от `Statement`, и `CallableStatement`, наследующий от `PreparedStatement`.

Они специализируются на различных типах запросов:

- `Statement` используется для выполнения простых SQL-запросов без параметров
- `PreparedStatement` используется для выполнения прекомпилированных SQL-запросов с или без входных (IN) параметров
- `CallableStatement` используется для вызовов хранимых процедур

Интерфейс `Statement` предоставляет базовые методы для выполнения запросов и извлечения результатов. Интерфейс `PreparedStatement` добавляет методы управления входными (IN) параметрами; `CallableStatement` добавляет методы для манипуляции выходними (OUT) параметрами.

Интерфейс `PreparedStatement` наследует от `Statement` и отличается от последнего следующим:

- Экземпляры `PreparedStatement` "помнят" скомпилированные SQL-выражения, именно поэтому они называются "prepared"

("подготовленные")
- SQL-выражения в `PreparedStatement` могут иметь один или более входной (IN)параметр. **Входной параметр** - это параметр, чье значение не указывается при создании SQL-выражения. Вместо него в выражении на месте каждого входного параметра ставится знак ("?"). Значение каждого вопросительного знака устанавливается методами setXXX перед выполнением запроса.

Как вызвать хранимую процедуру

Хранимые процедуры – это именованный набор операторов Transact-SQL хранящийся на сервере. Такую процедуру можно легко вызвать из Java-класса с помощью специального синтаксиса.

При вызове такой процедуры необходимо указать ее имя и определить список параметров. Имя и список параметров посылаются по JDBC-соединению в СУБД, которая выполняет вызываемую процедуру и возвращает результат (если таковой имеется) обратно, используя это же соединение.

JDBC Java-код для выполнения хранимой процедуры, использующий объект `Statement` без параметров:

```
public static void executeStoredProcedureNoParams(Connection c) {
    try {
        Statement stmt = c.createStatement();
        ResultSet rs = stmt.executrQuery("{CALL STPROCNAME}");
        while (rs.next()) {
            SystemOutPrintln(rs.getString(name));
        }
        rs.close();
        stmt.close();
    }
    catch (Exception e) {
        e.printStackTrace()
    }
}
```

Как правильно закрыть `Connection`

После того, как база больше не нужна, соединение закрывается. Для того, чтобы правильно пользоваться приведенными методами, программисту требуется знать типы полей БД. В распределенных системах это знание предполагается изначально. В Java 7 для объектов-ресурсов, требующих закрытия, реализована технология try with resources.

Какие есть уровни изоляции транзакций

Уровни изоляции транзакций определены в виде констант интерфейса `Connection` (по возрастанию уровня ограничения):

- **TRANSACTION_NONE** - информирует о том, что драйвер не поддерживает транзакции
- **TRANSACTION_READ_UNCOMMITTED** - позволяет транзакциям видеть несохраненные изменения данных, что разрешает грязное, неповторяющееся и фантомное чтения
- **TRANSACTION_READ_COMMITTED** - означает, что любое изменение,сделанное в транзакции, не видно вне ее, пока она не сохранена. Это предотвращает грязное чтение, но разрешает неповторяющееся и фантомное
- **TRANSACTION_REPEATABLE_READ** - запрещает грязное и неповторяющееся чтение, но фантомное разрешено
- **TRANSACTION_SERIALIZABLE** - определяет, что грязное, неповторяющееся и фантомное чтения запрещены

Какие есть типов чтения транзакций

Для транзакций существует несколько типов чтения:

- Грязное чтение (dirty reads) происходит, когда транзакциям разрешено видеть несохраненные изменения данных. Иными

словами, изменения, сделанные в одной транзакции, видны вне ее до того, как она была сохранена. Если изменения не будут сохранены, то, вероятно, другие транзакции выполняли работу на основе некорректных данных;

- Неповторяющееся чтение (nonrepeatable reads) происходит, когда транзакция А читает строку, транзакция Б изменяет эту строку, транзакция А читает ту же строку и получает обновленные данные;
- Фантомное чтение (phantom reads) происходит, когда транзакция А считывает все строки, удовлетворяющие WHERE-условию, транзакция Б вставляет новую или удаляет одну из строк, которая удовлетворяет этому условию, транзакция А еще раз считывает все строки, удовлетворяющие WHERE-условию, уже вместе с новой строкой или недосчитавшись старой.

Log4j

~~Что такое логгер~~

Из чего состоит логгер

Любой регистратор событий состоит из трех элементов:

- Регистратора - `logger`
- Вывода - `appender`
- Форматера для вывода - `layout`

Какие есть уровни сообщении в log4j

Приоритетов может быть 6:

- **FATAL** - произошла фатальная ошибка - у этого сообщения наивысший - приоритет
- **ERROR** - в программе произошла ошибка
- **WARN** - предупреждение в программе что-то не так
- **INFO** - информация.
- **DEBUG** - детальная информация для отладки
- **TRACE**– трассировка всех сообщений в указанный аппендер

Что такое `Appender`

Вывод регистратора может быть направлен в различные места назначения: файл, консоль и каждому из них соответствует класс, реализующий интерфейс `org.apache.log4j.Appender`. Кроме того, вывод в базу данных можно произвести с помощью класса `JDBCAppender`, в журнал событий ОС - `NTEventLogAppender`, на SMTP-сервер - `SMTPAppender`.

Если логгер - это та точка, откуда уходят сообщения в коде, то аппендер - это та точка, куда они приходят в конечном итоге.

Список таких точек, поддерживаемых Log4J:

- Вывод в консоль
- Файлы (несколько различных типов)
- База данных (JDBC)
- ~~Темы (topics) JMS~~
- NT Event Log
- SMTP
- Socket
- Syslog
- Telnet
- `java.io.Writer` или `java.io.OutputStream`

Какие основные аппендеры Log4j

Основными аппендерами, использующимися наиболее широко, являются файловые аппендеры. Их есть несколько типов:

```
org.apache.log4j.FileAppender
org.apache.log4j.RollingFileAppender
org.apache.log4j.DailyRollingFileAppender
```

Вывод регистратора может иметь различный формат. Каждый формат представлен классом, производным от `Layout`. Все методы класса `Layout` предназначены только для создания подклассов.

`org.apache.log4j.SimpleLayout` - наиболее простой вариант. На выходе читается уровень вывода и сообщение.
`org.apache.log4j.HTMLLayout` - данный компоновщик форматирует сообщения в виде HTML-страницы.
`org.apache.log4j.xml.XMLLayout` - формирует сообщения в виде XML. `org.apache.log4j.PatternLayout` и `org.apache.log4j.EnhancedPatternLayout` используют шаблонную строку для форматирования выводимого сообщения.

Maven

Зависимости Зависимости - следующая очень важная часть pom.xml - тут хранится список всех библиотек (зависимостей) которые используюся в проекте. Каждая библиотека идентифицируется также как и сам проект - тройкой groupId, artifactId, version (GAV). Объявление зависимостей заключено в тэг `<dependencies>...</dependencies>`

```
                    <dependency>
                        <groupId>junit</groupId>
                        <artifactId>junit</artifactId>
                        <version>4.4</version>
                        <scope>test</scope>
                    </dependency>
                    <dependency>
                        <groupId>org.powermock</groupId>
                        <artifactId>powermock-reflect</artifactId>
                        <version>${version}</version>
                    </dependency>
                    <dependency>
                        <groupId>org.javassist</groupId>
                        <artifactId>javassist</artifactId>
                        <version>3.13.0-GA</version>
                        <scope>compile</scope>
                    </dependency>
                </dependencies>
```

Как вы могли заметить, кроме GAV при описании зависимости может присутствовать тэг . Scope задаёт, для чего библиотека используется. В данном примере говорится, что библиотека с GAV junit:junit:4.4 нужна только для выполнения тестов.

Тэг Тэг не обязательный, т. к. существуют значения по умолчанию. Этот раздел содержит информацию по самой сборке: где находятся исходные файлы, где ресурсы, какие плагины используются. Например:

```
    ```<build>
    <outputDirectory>target2</outputDirectory>
     <finalName>ROOT</finalName>
    <sourceDirectory>src/java</sourceDirectory>
       <resources>
           <resource>
               <directory>${basedir}/src/java</directory>
               <includes>
               <include>**/*.properties</include>
               </includes>
           </resource>
       </resources>
            <plugins>
                <plugin>
                    <groupId>org.apache.maven.plugins</groupId>
                    <artifactId>maven-pmd-plugin</artifactId>
                    <version>2.4</version>
                </plugin>
            </plugins>
        </build>```
```

Давайте рассмотрим этот пример более подробно.

определяет, откуда maven будет брать файлы исходного кода. По умолчанию это src/main/java, но вы можете определить, где это вам удобно. Директория может быть только одна (без использования специальных плагинов)

и вложенные в неё тэги определяют, одну или несколько директорий, где хранятся файлы ресурсов. Ресурсы в отличие от файлов исходного кода при сборке просто копируются . Директория по умолчанию src/main/resources

определяет, в какую директорию компилятор будет сохранять результаты компиляции - *.class файлы. Значение по умолчанию - target/classes
```

- имя результирующего jar (war, ear..) файла с соответствующим типу расширением, который создаётся на фазе package. Значение по умолчанию — artifactId-version.

Репозитории - это место где хранятся артефакты: jar файлы, pom -файлы, javadoc, исходники. Существуют:

Локальный репозиторий по умолчанию он расположен в /.m2/repository - персональный для каждого пользователя. центральный репозиторий который расположен в http://repo1.maven.org/maven2/ и доступен на чтение для всех пользователей в интернете. Внутренний "Корпоративный" репозиторий- дополнительный репозиторий, один на несколько пользователей.

Локальный репозиторий Локальный репозиторий по умолчанию расположен в /.m2/repository. Здесь лежат артефакты которые были скачаны из центрального репозитория либо добавлены другим способом. Например если вы наберёте команду

Центральный репозиторий Чтобы самому каждый раз не создавать репозиторий, сообщество для Вас поддерживает центральный репозиторий. Если для сборки вашего проекта не хватает зависимостей, то они по умолчанию автоматически скачиваются с http://repo1.maven.org/maven2. В этом репозитории лежат практически все опенсорсные фреймворки и библиотеки.

Самому в центральный репозиторий положить нельзя. Т.к. этот репозиторий используют все, то перед тем как туда попадают артефакты они проверяются, тем более что если артефакт однажды попал в репозиторий, то по правилам изменить его нельзя.

Для поиска нужной библиотеки очень удобно пользоваться сайтами http://mavenrepository.com/ и http://findjar.com/

Корпоративный репозиторий Если вы хотите создать свой репозиторий, содержимое которого вы можете полностью контролировать(как локальный), и сделать так, чтобы он был доступен для нескольких человек, вам будет полезен корпоративный репозиторий. Доступ к артефактам можно ограничивать настройками безопасности сервера так, что код ваших проектов не будет доступен извне.

Чтобы добавить репозиторий в список, откуда будут скачиваться зависимости, нужно добавить секцию repositories в pom.xml, например:

```
```<project>
  <repositories>
    <repository>
      <id>my-company-repo</id>
      <url>http://my-company-site.ru/repo</url>
    </repository>
  </repositories>
</project>```
```

Существуют несколько реализаций серверов - репозиториев maven. Наиболее известные это artifactory, continuum, nexus.

Основные фазы сборки проекта

1. compile Компилирование проекта
2. test Тестирование с помощью JUnit тестов
3. package Создание .jar файла или war, ear в зависимости от типа проекта
4. integration-test Запуск интеграционных тестов
5. install Копирование .jar (war , ear) в локальный репозиторий
6. deploy публикация файла в удалённый репозиторий

Мавен изначально создавался , принимая во внимание портируемость. Но довольно часто приложение приходится запускать в разном окружении: например, для разработки используется одна база данных, в рабочем сервере используется другая. при этом могут понадобиться разные настройки, разные зависимости и плагины. Для этих целей в maven используются профайлы.

Давайте определим два профайла: один для разработки, другой для производственного сервера. Для разработки вполне подойдёт база hsqldb, которая хранит все данные в памяти. На производственном сервере же используется база данных postgres, которая сохраняет все данные на диск. В профайлах для каждой конфигурации определены свои проперти database.url и зависимости для разных jdbc драйверов.

Ниже приведён пример объявления таких профайлов.

```<?xml version="1.0" encoding="UTF-8"?>
```
```

(1) development jdbc:hsqldb:mem:testdb org.hsqldb hsqldb 2.0.0 (2) productionServer jdbc:postgresql://databseserver/database postgresql postgresql 9.0-801.jdbc4 ``` цифрами 1 и 2 обозначены начала объявления профайлов. каждый профайл имеет идентификатор в данном случае development и productionServer. Внутри тэга содержатся все те же объявления что и внутри : properties, dependencies, и др. Вот полный список тегов которые могут содержаться внутри профайлов: ``` * * * * * * * * * * тэг, который может содержать о о о о ``` При сборке проекта в тестах произошла ошибка. Как мне собрать проект без запуска тестов? Для запуска сборки без выполнения тестов добавьте -Dmaven.test.skip=true к командной строке запуска maven: ```mvn install -Dmaven.test.skip=true``` Как запустить только один тест? Для запуска только одного теста добавьте -Dtest=[Имя класса] к командной строке запуска maven. Например ```mvn install -Dtest=ru.apache-maven.utils.ConverterTest``` Компилятор Компилятор - основной плагин который используется практически во всех проектах. Он доступен по умолчанию, но практически в каждом проекте, его приходится переобъявлять т.к. настройки по умолчанию не очень подходящие. Пример использования: org.apache.maven.plugins maven-compiler-plugin 2.0.2 1.6 1.6 UTF-8 В этом примере в кофигурации используется версия java 1.6 (source - версия языка на котором написана программа; target - версия java машины которая будет этот код запускать) и указано что кодировка исходного кода программы UTF-8. По умолчанию версии java - 1.3 а кодировка - та которая у операционной системы по умолчанию. Вообще у плагина есть две цели compiler:compile и compiler:testCompile compiler:compile - компилирует основную ветку исходников и по умолчанию связана с фазой compile compiler:testCompile - компилирует тесты и по умолчанию связана с фазой test-compile. Кроме приведёных настроек для компилятора можно задать следующие параметры: verbose true или false fork запусть компиляцию в отдельной jvm executable путь к javac compilerVersion meminitial maxmem debug compilerArgument задать аргуметы в одной коммандной строке-verbose -bootclasspath ${java.home}\lib\rt.jar compilerArguments задать аргуметы в коммандной строке пораздельно в тегах verbose, bootclasspath и др. compilerId позволяет задать язык программирования исходного кода, например csharp maven-surefire-plugin - плагин для запуска тестов maven-surefire-plugin - плагин который запускает тесты и генерирует отчёты по результатам их выполнения. По умолчанию отчёты сохраняются в ${basedir}/target/surefire-reports и находятся в двух форматах - txt и xml. maven-surefire-plugin содержит единственную цель surefire:test тесты можно писать используя как JUnit так и TestNG. по умолчанию запускаются все тесты с такими именами * "**/Test*.java" - включает все java файлы которые начинаются с "Test" и расположены в поддиректориях. * "**/*Test.java" - включает все java файлы которые заканчиваются на "Test" и расположены в поддиректориях. * "**/*TestCase.java" - включает все java файлы которые заканчиваются на "TestCase" и расположены в поддиректориях. Чтобы вручную добавлять или удалять классы тестов можно посмотреть здесь http://maven.apache.org/plugins/maven-surefire-plugin/examples/inclusion-exclusion.html. Запустить отдельный тест можно так: mvn -Dtest=TestCircle test имейте в виду что если вы хотите отладить тест в среде разработки то в конфигурации плагина нужно выставить never либо запускать тесты с remotedebug mvn -Dmaven.surefire.debug test Пропустить выполнение тестов можно true или mvn install -DskipTests чтобы пропустить ещё и компиляцию тестов вызовите maven так: mvn install -Dmaven.test.skip=true Наследование Наследование позволяет избавиться от дублирования описаний. Давайте рассмотрим как использовать наследование при описании maven проектов. Предположим, что у нас несколько проектов и в каждом мы используем библиотеку log4j для логгирования приложения. Сначала создадим проект-предок. Весь проект-предок будет состоять из одного файла pom.xml 4.0.0 ru.apache-maven parent 1.0-SNAPSHOT pom
как вы заметили тэг packaging содержит значение pom. Теперь давайте внутри что-нибудь объявим. Например добавим log4j log4j 1.2.16 </dependency

и установим в локальный репозиторий: cd parent mvn install.

Если посмотреть список зависимостей, то он будет такой: $ mvn dependency:tree - log4j:log4j:jar:1.2.16:compile

Теперь можно создать один или несколько проектов-потомков:

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
<modelVersion>4.0.0</modelVersion>

<!--<groupId>ru.apache_maven</groupId>-->
<artifactId>testproj1</artifactId>
<!--<version>1.0-SNAPSHOT</version>-->
<parent>
    <groupId>ru.apache_maven</groupId>
    <artifactId>parent</artifactId>
    <version>1.0-SNAPSHOT</version>
</parent>
<packaging>jar</packaging>
```

```
            <dependencies>
            </dependencies>
        </project>
```

Как видно в примере, мы объявили первое на что стоит обратить внимание — это то что groupId и version закомментированы. Они теперь необязательны и берутся по умолчанию такие же как и у parent проекта, хотя можно задать значения и отличные от тех которые по умолчанию. Второе и самое важное, если посмотреть список зависимойстей $ mvn dependency:tree - log4j:log4j:jar:1.2.16:compile

то можно увидеть log4j, хотя мы в pom.xml проекта мы log4j не объявляли. Он унаследовался от проекта-предка. Кроме зависимостей в проекте-предке часто объявляют плагины:

maven-compiler-plugin нужен в каждом проекте! properties репозитории

Свойства Для того чтобы можно было легче писать и настраивать проект в pom.xml можно использовать свойства. Можно рассматривать свойства просто как переменные.

Есть:

Переменные объявленные внутри pom.xml Предопределённые переменные. Переменные объявленные во внешнем файле Переменные объявленные внутри pom.xml Давайте начнём с самого простого объявим свойства и сами будем их использовать. Свойства можно объявить и использовать так:

```
<properties>
    <temp.directory>/tmp</temp.directory>
</properties>
<build>
<outputDirectory>${temp.directory}<outputDirectory>
```

Свойства помогают избавиться от дублирования информации В примере

```
    <properties>
        <jetty.version>6.1.25</jetty.version>
    </properties>
    <dependency>
        <groupId>org.mortbay.jetty</groupId>
        <artifactId>jetty</artifactId>
        <version>${jetty.version}</version>
        <scope>provided</scope>
    </dependency>
    <dependency>
        <groupId>org.mortbay.jetty</groupId>
        <artifactId>jetty-util</artifactId>
        <version>${jetty.version}</version>
        <scope>provided</scope>
    </dependency>
    <dependency>
        <groupId>org.mortbay.jetty</groupId>
        <artifactId>jetty-management</artifactId>
        <version>${jetty.version}</version>
        <scope>provided</scope>
    </dependency>
```

использование свойства jetty.version позволяет избавиться от дублирования и уменьшет вероятность ошибок при апгрейде. Предопределёные переменные Предопределённые переменные можно разделить на несколько видов. Встроенные свойства

${basedir} директория где лежит pom.xml ${version} тоже самое что и ${project.version} или ${pom.version} Свойства проекта На все свойства в pom.xml, можно сослаться с помощью префиксов project. или pom. Ниже приведён пример некоторых часто используемых элементов.

${project.build.directory} ваша "target" директория,или тоже самое ${pom.project.build.directory} ${project.build.outputDirectory} путь к директории куда компилятор складывает файлы по умолчанию "target/classes" ${project.name} или ${pom.name} имя Вашего проекта ${project.version} или ${pom.version} версия Вашего проекта. Настройки пользователя Можно получить

доступ к свойствам settings.xml с помощью префикса settings. ,например:

${settings.localRepository} путь к локальному репозиторию пользователя. Переменные окружения Для доступа к переменным окружения используйте префикс env. Примеры:

${env.M2_HOME} путь . ${java.home} specifies the path to the current JRE_HOME environment use with relative paths to get for example: ${java.home}../bin/java.exe Системные свойства System.properties Доступ к системным свойствам возможен напрямую. Для просмотра переменных можно воспользоваться maven-echo-plugin.

Переменные объявленные во внешнем файле Для того чтобы загрузить переменные из внешнего файла удобнее всего использовать maven-properties-plugin Давайте рассмотрим его работу.

```
<plugin>
    <groupId>org.codehaus.mojo</groupId>
    <artifactId>maven-properties-plugin</artifactId>
    <version>1.0-SNAPSHOT</version>
    <executions>
        <execution>
            <phase>initialize</phase>
            <goals>
                <goal>read-project-properties</goal>
            </goals>
            <configuration>
                <files>
                    <file>src/config/app.properties</file>
                </files>
            </configuration>
        </execution>
    </executions>
</plugin>
```

В данном примере объявляется плагин, он отрабатывает на стадии initialize и загружает свойства из src/config/app.properties.

Java Unit Testing Interview Questions

Что такое Unit Testing

Модульное тестирование или *Unit Testing* - процесс проверки на корректность функционирования отдельных частей исходного кода программы путем запуска тестов в искусственной среде.

Что такое интеграционные тесты (Integration Testing)

Интеграционные тесты - это тесты, проверяющие работоспособность двух или более модулей системы, но в совокупности - то есть нескольких объектов как единого блока.

Что такое Fixture

Фикстура (Fixture) - состояние среды тестирования, которое требуется для успешного выполнения тестового метода. Может быть представлено набором каких-либо объектов, состоянием базы данных, наличием определенных файлов, соединений и прочее

Какие аннотации используются для создания фикстур?

Предусмотрено четыре аннотации две для фикстур уровня класса и две для уровня метода.

- `@BeforeClass` - запускается только один раз при запуске теста
- `@Before` - запускается перед каждым тестовым методом
- `@After` - запускается после каждого метода
- `@AfterClass` - запускается после того, как отработали все тесты

Для чего нужна аннотация `@Ignore`?

Аннотация `@Ignore` заставляет инфраструктуру тестирования проигнорировать данный тестовый метод. Аннотация предусматривает наличие комментария о причине игнорирования теста.

Чем стаб (Stub) отличается от мока (Mock)

Stub - объекты, которые предоставляют заранее заготовленные ответы на вызовы во время выполнения теста и обычно не отвечающие ни на какие другие вызовы, которые не требуются в тесте. Также могут запоминать какую-то дополнительную информацию о количестве вызовов, параметрах и возвращать их потом для проверки Используется для "затычки" сервисов, методов, классов и т.д. Абсолютно все равно что они вернут при работе или сколько раз произойдет вызов.

Mock - объекты, которые заменяют реальный объект в условиях теста и позволяют проверять вызовы своих членов как часть системы. Содержат заранее запрограммированные ожидания вызовов, которые они ожидают получить. Применяются в основном для *interaction (behavioral) testing* Используется для подмены результатов вызова функций в юнит тестах, для учета количества вызовов функций и просто ожидания их вызовов. Используется в области Assert юнит теста

Serialization

Что такое сериализация

Сериализация - это процес чтения или записи объекта. Это процесс сохранения состояния объекта и считывание этого состояния. Для реализации сериализации нужен интерфейс - маркер Serializable. Обратная операция - перевод байтов в объект, называется десериализацией.

Как исключить поля из сериализации

Для того чтоб исключить поля из сериализуемого потока, необходимо пометить поле модификатором `transient`.

`transient` что значит

Свойства класса, помеченные модификатором `transient` ,не сериализуются. Обычно в таких полях хранится промежуточное состояние объекта, которое, к примеру,проще вычислить, чем сериализовать, а затем десериализавать. Другой пример такого поля - ссылка на экземпляр объекта, который не требует сериализации или не может быть сериализован.

Как изменить стандартное поведение сериализации/ десериализации

В большинстве случаев мы не определяем поведение вручную, а полагаемся на стандартную реализацию, и очень не удобно постоянно переопределять какие-то методы сериализации + постоянно следить за добавлением новых полей, добавлять их в методы. Ну и специально для этих целей есть `Externalizable`.

Тем не менее, мы знаем, что можно изменить стандартное поведение сериализации предопределив и поместив в свои файлы классов два метода:

```
private void writeObject(ObjectOutputStream out) throws IOException;
private void readObject(ObjectInputStream in) throws IOException, ClassNotFoundException;
```

Обратите внимание, что оба метода объявлены как `private`, поскольку это гарантирует что методы не будут переопределены или перезагружены. Весь фокус в том, что виртуальная машина при вызове соответствующего метода автоматически проверяет, не были ли они объявлены в классе объекта. Виртуальная машина в любое время может вызвать `private` методы вашего класса, но другие объекты этого сделать не смогут. Таким образом обеспечивается целостность класса и нормальная работа протокол сериализации.

Вы создали класс, чей родительский класс сериализуемый, но при этом вы не хотите чтобы ваш класс был сериализуемым, как остановить сериализацию

Сделать этого напрямую нельзя, если класс родитель реализует `Serializable`, то и созданный вами новый класс также будет реализовать его. Чтобы остановить автоматическую сериализацию вы можете применить private методы для создания исключительной ситуации `NotSerializableException`. Вот как это можно сделать:

```
private void writeObject(ObjectOutputStream out) throws IOException {
    throw new NotSerializibleException("Serialization Error!");
}
private void readObject(ObjectInputStream in) throws IOException {
```

```
    throw new NotSerializibleException("Serialization Error!");
}
```

Любая попытка записать или прочитать этот объект теперь приведет к возникновению исключительной ситуации. Запомните, если методы объявлены как `private`, никто не сможет модифицировать ваш код не изменяя исходный код класса. Java не позволяет переопределять такие методы.

Как создать собственный протокол сериализации

Вместо реализации интерфейса `Serializable`, вы можете реализовать интерфейс `Externalizable`, который содержит два метода:

```
public void writeExternal(ObjectOutput out) throws IOException;
public void readExternal(ObjectInput in) throws IOException, ClassNotFoundException;
```

Для создания собственного протокола нужно просто переопределить эти два метода. В отличие от двух других вариантов сериализации, здесь ничего не делается автоматически. Протокол полностью в ваших руках. Хотя это и наиболее сложный способ, при этом он наиболее контролируемый.

Какая роль поля serialVersionUID в сериализации

Поле `private static final long serialVersionUID` содержит уникальный идентификатор версии сериализованного класса. Оно вычисляется по содержимому класса - полям, их порядку объявления, методам, их порядку объявления. Соответственно, при любом изменении в классе это поле поменяет свое значение.

Это поле записывается в поток при сериализации класса. Кстати, это, пожалуй, единственный известный случай, когда `static` поле сериализуется.

В чем проблема сериализации Singleton

Проблема в том что после десериализации мы получим другой объект. Таким образом, сериализация дает возможность создать `Singleton` еще раз, что не совсем не нужно. Конечно можно запретить сериализовать `Singleton`, но это, фактически, уход от проблемы, а не ее решение.

Решение же заключается в следующем. В классе определяется метод со следующей сигнатурой:

```
ANY-ACCESS-MODIFIER Object readResolve() throws ObjectStreamException
```

Модификатор доступа может быть `private`, `protected` и по умолчанию (`default`). Можно, наверное, сделать его и `public`, но смысла я в этом не вижу. Назначение этого метода - возвращать замещающий объект вместо объекта, на котором он вызван.

Object Oriented Programming

Что такое ООП

ООП - методология программирования, основанная на представлении программного продукта в виде совокупности объектов, каждый из которых является экземпляром конкретного класса. ООП использует в качестве базовых элементов взаимодействие объектов.

Что такое объект

Объект - именованная модель реальной сущности, обладающая конкретными значениями свойств и проявляющая свое поведение, обладающая именем, набором данных (полей и свойств объекта), физически находящихся в памяти компьютера, и методов, имеющих доступ к ним.

Объект - конкретный экземпляр класса.

Назовите основные принципы ООП

Принято считать, что объектно-ориентированное программирование строится на 4 основных принципах (раньше их было всего 3). Эти принципы:

- Абстракция
- Инкапсуляция
- Наследование
- Полиморфизм

Что такое наследование

Наследование это процесс благодаря которому один объект может приобрести свойства другого объекта (наследование всех свойств одного объекта другим) и добавлять новые.

```
class Dog extends Animal
{...}
```

Терминология: Суперкласс -> Подкласс Родительский -> Дочерний

Что такое полиморфизм? Каковы проявления полиморфизма в Java

Полиморфизм (от греческого polymorphos) - это свойство, которое позволяет одно и то же имя использовать для решения двух или более схожих, но технически разных задач.

Целью полиморфизма, применительно к объектно-ориентированному программированию, является использование одного имени для задания общих для класса действий. Выполнение каждого конкретного действия будет определяться типом данных.

В более общем смысле, концепцией полиморфизма является идея "один интерфейс, множество методов". Это означает, что можно создать общий интерфейс для группы близких по смыслу действий.

Что такое инкапсуляция

Инкапсуляция - это сокрытие реализации класса и отделение его внутреннего представления от внешнего (интерфейса), это свойство которое позволяет закрыть доступ к полям и методам класса другим классам, а предоставлять им доступ только через интерфейс(метод). При использовании объектно-ориентированного подхода не принято применять прямой доступ к свойствам

какого-либо класса из методов других классов. Для доступа к свойствам класса принято задействовать специальные методы этого класса для получения и изменения его свойств.

Что такое абстракция

Абстракция - это выделение общих характеристик объекта,исключая набор незначительных.

С помощью принципа абстракции данных, данные преобразуются в объекты. Данные обрабатываются в виде цепочки сообщений между отдельными объектами. Все объекты проявляют свои уникальные признаки поведения. Огромный плюс абстракции в том, что она отделяет реализацию объектов от их деталей, что в свою очередь позволяет управлять функциями высокого уровня через функции низкого уровня.

В чем преимущества объектно-ориентированных языков программирования

Они представляют реальные объекты в жизни, например, Машина, Джип, Счет в банке и так далее. Инкапсуляция, наследование и полиморфизм делает его еще мощнее.

Как использование объектно-ориентерованного подхода улучшает разработку программного обеспечения

Основные преимущества:

- повторное использование кода(наследование)
- реальное отображение предметной области. Объекты соответствуют реальности

Имеется выражение "является" и "имеет". Что они подразумевают в плане принципов ООП? В чем разница между композицией и агрегацией

- является - наследование
- имеет - композиция

В качестве примера предположим что у нас есть классы Строение, Дом и Ванная комната. Так вот Дом является строением, что нельзя сказать про Ванну, которая не является домом. А вот Дом имеет\включает в себя Ванну. Если вы хотите использовать повторно код,то не обязательно использовать наследование. Если нет отношения "является", то лучше тогда использовать композицию для повторного использования кода.

Не используйте наследование для получение полиморфизма, если нет ключевой зависимости "является". Используйте интерфейсы для полиморфизма.

Из спецификации можно узнать, что:

- Ассоциация обозначает связь между объектами
- Агрегация и композиция это частные случаи ассоциации

Агрегация предполагает, что объекты связаны взаимоотношением "part-of" (часть). **Композиция** более строгий вариант агрегации. Дополнительно к требованию part-of накладывается условие, что "часть" не может одновременно принадлежать разным "хозяевам",и заканчивает своё существование вместе с владельцем.

Например:

- мотоцикл -> сумка с багажём - ассоциация. *Отношение: имеет*.
- мотоцикл -> колесо - композиция. *Отношение: имеет*
- группа по интересам -> человек - агрегация. человек часть группы, но может принадлежать нескольким разным группам.

Что вы подразумеваете под полиморфизмом, инкапсуляцией и динамическим связыванием

Полиморфизм означает способность переменной данного типа, которая ссылается на объекты разных типов, при этом вызывается метод, характерный для конкретного типа ссылки на объект.

В чем преимущество полиморфизма? Он позволяет добавлять новые классы производных объектов, не нарушая при этом код вызова. Также использование полиморфизма называют динамическим связыванием объектов.

Рассмотрим пример полиморфизма:

Имеется классы: Figure, Circle и Triangle. Circle и Triangle наследуется от Figure соответственно. Каждый класс имеет метод draw(). В Circle и Triangle этот метод переопределен.

Так вот, создаем объект с типом Figure и присваиваем ей ссылку на объект типа Circle и вызываем на этом объекте метод draw(). В итоге вызывается метод класса Circle, а не класса Figure как ожидалось.

```
Figure  f = new Circle();
f.draw();
```

Также вместо класса родителя Figure к примеру можно использовать интерфейс Figure, определив там метод рисовать. Этот интерфейс мы имплементируем в классах Circle, Triangle. Далее на интерфейсе создаем объект и присваиваем ему ссылку на объект какого-то из реализующих этот интерфейс классов.

Это удобно например если у нас есть некий метод:

```
public void drawShape(Figure f){
f.draw();
}
```

Обратите внимание что в метод мы передаем параметр с типом интерфейса, т.е. мы не знаем какой именно тип объекта будет, но реализация будет такая же. Далее мы можем просто создать еще класс, к примеру Trapezoid,и имплементировать интерфейс Figure и просто передать экземпляр класса в метод, ничего не меняя в реализации и вызове.

Наследование это включение поведения(методы) и состояния(поля) базового класса в производный от него. В результате этого мы избегаем дублирования кода и процесс исправления ошибок в коде также упрощается.

В джава есть два вида наследования:

- наследование классов. Каждый наследник может иметь толко одного родителя
- наследование интерфейсов. Интерфейс может иметь сколько угодно родителей

Некоторые тонкие ньюансы по поводу наследования интрефейсов и классов. Мы имеем два интерфейса с одинаковыми по имени полями. Имплементируем эти интерфейсы на каком-то классе. Как нам вызвать поля этих интерфейсов?

У нас неоднозначность. Необходимо объект класса привести к нужному интерфейсу.

```
Class c = new Class();
((InterfaceOne) c).field;
```

Хорошо, что будет если мы имеем метод с одинаковой сигнатурой в интерфейсах и реализуем эти интерфейсы на классе. Как нам в классе реализовать два метода с одинаковой сигнатурой?

Никак, мы просто реализовываем один общий метод в классе. Это является недостатком, так как нам может потребоваться разная реализация.

И третий случай: У нас есть класс и интерфейс с одинаковым по сигнатуре методом. Мы наследуемся от этого класса и имплементируем этот интерфейс. Что нам нужно делать? ведь необходимо реализовать метод интерфейса по всем правилам.

И вот тут интересно, компилятор не выдает ошибок, так как метод уже у нас реализован в классе родителе.

Jmeter

Jmeter Functions & Variables

syntax function : ${__functionName(args...)} syntax variable : ${variableName}

==========

FUNCTIONS:

${__log("message")}* *${__time(dd mm yyyy)} show time *${__threadNum}* show number of thread *${__intSum(2,3)}* sum numbers in bracests

...Functions -> function helper

JSON

Что такое JSON

JSON (JavaScript Object Notation) - простой формат обмена данными, удобный для чтения и написания как человеком, так и компьютером. Он основан на JavaScript. JSON - текстовый формат, полностью независимый от языка реализации, но он использует соглашения, знакомые программистам C-подобных языков, таких как C, C++, C#, Java, JavaScript, Perl, Python и многих других. Эти свойства делают JSON идеальным языком обмена данными.

Что такое JSON Schema

JSON Schema - это стандарт описания структур данных в формате JSON. Использует синтаксис JSON. Схемы, описанные этим стандартом, имеют MIME "application/schema+json". Стандарт удобен для использования при валидации и документировании структур данных, состоящих из чисел, строк, массивов и структур типа ключ-значение.

Что такое JSON объект

JSON объект - неупорядоченный набор пар ключ/значение. Объект начинается с { (открывающей фигурной скобки) и заканчивается } (закрывающей фигурной скобкой). Каждое имя сопровождается : (двоеточием), пары ключ/значение разделяются , (запятой).

Какие есть правила синтаксиса JSON объекта (массива)

Есть несколько основных правил для создания строки JSON:

- Строка JSON содержит либо массив значений, либо объект (ассоциативный массив пар имя/значение).
- Массив заключается в квадратные скобки ([и]) и содержит разделенный запятой список значений.
- Объект заключается в фигурные скобки ({ и }) и содержит разделенный запятой список пар имя/значение.
- Пара имя/значение состоит из имени поля, заключенного в двойные кавычки, за которым следует двоеточие (:) и значение поля.

Чтобы включить двойные кавычки в строку, нужно использовать обратную косую черту: \". Так же, как и во многих языках программирования, можно помещать управляющие символы и шестнадцатеричные коды в строку, предваряя их обратной косой чертой.

Ниже приводится пример оформления заказа в формате JSON:

```
{"menu": {
  "id": "file",
  "value": "File",
  "popup": {
    "menuitem": [
      {"value": "New", "onclick": "CreateNewDoc()"},
      {"value": "Open", "onclick": "OpenDoc()"},
      {"value": "Close", "onclick": "CloseDoc()"}
    ]
  }
}}
```

Какие типы данных, поддерживаются в JSON

Значение в массиве или объекте может быть:

- Числом (целым или с плавающей точкой)
- Строкой (в двойных кавычках)
- Логическим значением (true или false)
- Другим массивом (заключенным в квадратные скобки)
- Другой объект (заключенный в фигурные скобки)
- Значение null

Каковы недостатки JSON

Недостатками JSON являются:

- Трудно читается и анализируется пользователем, нет визуальности
- Нет синтаксиса для задания типа объекта
- Много синтаксического мусора

Что такое JSONP

JSONP или "JSON with padding" является расширением JSON, позволяющим выполнять в единообразном стиле асинхронные запросы к сервисам, расположенным на другом домене - операцию, запрещённую в типичных веб-браузерах из-за политики ограничения домена.

Какой MIME-тип в JSON

`application/json`

Для чего используется JSON

Наиболее частое распространенное использование JSON - пересылка данных между сервером и браузером, а так же для хранения данных. Обычно данные JSON доставляются с помощью AJAX, который позволяет обмениваться данными браузеру и серверу без необходимости перезагружать страницу.

Какие преимущества использования JSON

Сравнительные преимущества JSON:

- В разы меньший объем данных (экономия трафика, плюс к скорости работы сайта)
- Меньшая загрузка процессора и клиента, и сервера
- Почти неограниченные возможности расширения (т.к. можно выполнить ф-цию)
- Его предложения легко читаются и составляются как человеком, так и компьютером.
- Его легко преобразовать в структуру данных для большинства языков программирования (числа, строки, логические переменные, массивы и так далее)
- Многие языки программирования имеют функции и библиотеки для чтения и создания структур JSON.

Какая функция используется для преобразования текста JSON в объект

Чтобы преобразовать текст JSON в объект используется функция `eval()`.

Что такое JSON Parser

Вызов `JSON.parse(str)` превратит строку с данными в формате JSON в JavaScript-объект/массив/значение, возможно с преобразованием получаемого в процессе разбора значения.

Что такое JSON-RPC

JSON-RPC(сокр. от англ. JavaScript Object Notation Remote Procedure Call - JSON-вызов удалённых процедур) - протокол удалённого вызова процедур, использующий JSON для кодирования сообщений. Это очень простой протокол (очень похожий на XML-RPC), определяющий только несколько типов данных и команд. JSON-RPC поддерживает уведомления (информация, отправляемая на сервер, не требует ответа) и множественные вызовы.

Что такое JSON-RPC-Java

Реализация протокола JSON-RPC на Java

Какова роль `JSON.stringify()`

Метод `JSON.stringify()` преобразует объекты JavaScript в строку в формате JSON, возможно с заменой значений, если указана функция замены, или с включением только определённых свойств, если указан массив замены. Используется, когда нужно из JavaScript передать данные по сети.

Как парсить JSON в JavaScript

```
var json = '{"name": "Andrew", "sn": "Cohen", "age": "24"}';
var obj = JSON.parse(json);
```

Как создать JSON объект из JavaScript

```
var obj = {};
    obj['name'] = 'Andrew';
    obj['sn'] = 'Cohen';
    obj['age'] = 24;
```

Валидациия JSON в JavaScript

```
function isValidJSON(jsonData) {
    try {
        JSON.parse(jsonData)
        return true;
    }
    catch (e) {
        return false;
    }
}

var json = '{"name": "Andrew", "sn": "Cohen", "age": "24"}';
isValidJSON(json);
```

UML

Что такое UML

Унифицированный язык моделирования (Unified Modeling Language) - графический язык визуализации, специфицирования, конструирования и документирования программного обеспечения.

Что такое Нотации и метамодель в UML

Нотация - совокупность графических объектов, которые используются в моделях. В качестве примера на диаграмме показано, как в нотации диаграммы класса определяют понятия и предметы типа «класс», «ассоциацция»,«множественность» и прочее.

Метамодель - диаграмма, определяющая нотацию. Метамодель помогает понять, что такое хорошо организованная, синтаксически правильная, модель.

Основные понятия

К основным понятиям UML относятся:

- Сущности - абстракции, являющиеся основными элементами модели
- Отношения - связывают различные сущности
- Диаграммы - группируют представляющие интерес совокупности сущностей

Какие есть типы диаграмм

Структурные Диаграммы

- Диаграммы классов (class diagrams) предназначены для моделирования структуры объектно-ориентированных приложений - классов, их атрибутов и заголовков методов, наследования, а также связей классов друг с другом;
- Диаграммы компонент (component diagrams) используются при моделировании компонентной структуры распределенных приложений; внутри каждая компонента может быть реализована с помощью множества классов;
- Диаграммы объектов (object diagrams) применяются для моделирования фрагментов работающей системы, отображая реально существующие в runtime экземпляры классов и значения их атрибутов;
- Диаграммы композитных структур (composite structure diagrams) используются для моделирования составных структурных элементов моделей - кооперaций, композитных компонент и т.д.;
- Диаграммы развертывания (deployment diagrams) предназначены для моделирования аппаратной части системы, с которой ПО непосредственно связано (размещено или взаимодействует);
- Диаграммы пакетов (package diagrams) служат для разбиения объемных моделей на составные части, а также (традиционно) для группировки классов моделируемого ПО, когда их слишком много.

Поведенческие Диаграммы

- Диаграммы активностей (activity diagrams) используются для спецификации бизнес-процессов, которые должно автоматизировать разрабатываемое ПО, а также для задания сложных алгоритмов;
- Диаграммы случаев использования(use case diagrams) предназначены для "вытягивания" требований из пользователей, заказчика и экспертов предметной области;
- Диаграммы конечных автоматов (state machine diagrams) применяются для задания поведения реактивных систем;
- Диаграммы взаимодействий (interaction diagrams):
- Диаграммы последовательностей (sequence diagrams) используются для моделирования временных аспектов внутренних и

внешних протоколов ПО;

- Диаграммы схем взаимодействия (interaction overview diagrams) служат для организации иерархии диаграмм последовательностей;
- Диаграммы коммуникаций (communication diagrams) являются аналогом диаграмм последовательностей, но по-другому изображаются (в привычной, графовой, манере);
- временные диаграммы (timing diagrams) являются разновидностью диаграмм последовательностей и позволяют в наглядной форме показывать внутреннюю динамику взаимодействия некоторого набора компонент системы.

Какие отношение обобщения реализуется при наследовании классов

Реализацией (Realization) называется отношение между классификаторами (классами, интерфейсами), при котором один описывает контракт (интерфейс сущности), а другой гарантирует его выполнение.

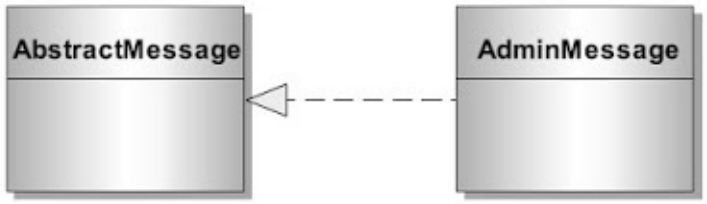


Рис. 6. *Отношение реализации*

Ассоциация (Association) показывает, что объект одного класса связан с объектом другого класса и отражает некоторое отношение между ними.

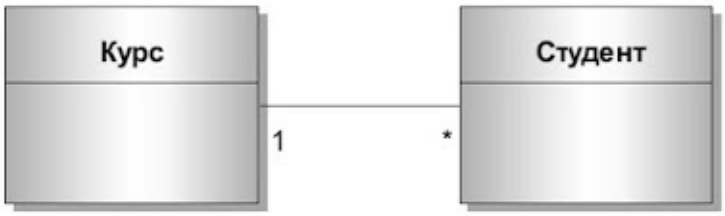


Рис. 7. *Отношение ассоциации*

Агрегация - ассоциация, моделирующая взаимосвязь «часть/целое» между классами, которые в то же время могут быть равноправными. Оба класса при этом находятся на одном концептуальном уровне, и ни один не является более важным, чем другой.

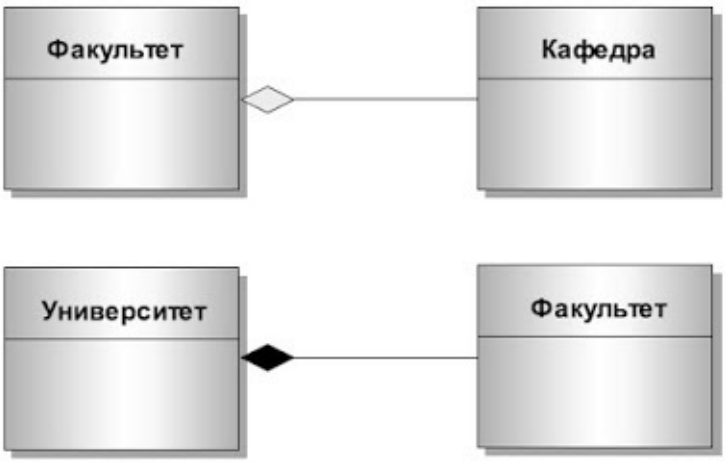


Рис. 8. *Отношения коллективной агрегации и композиции*

SQL Interview Questions

Что такое SQL

SQL (*structured query language* - формальный не процедурный язык программирования, применяемый для создания, модификации и управления данными в произвольной реляционной базе данных, управляемой соответствующей системой управления базами данных (СУБД).

Какие есть типы JOIN

INNER JOIN - внутреннее соединение. В результирующем наборе присутствуют только записи, значения связанных полей в которых совпадают. **LEFT JOIN** - левое внешнее соединение. В результирующем наборе присутствуют все записи из первой таблицы и соответствующие им записи из второй таблицы. Если соответствия нет, поля из второй таблицы будут пустыми. **RIGHT JOIN** - правое внешнее соединение. В результирующем наборе присутствуют все записи из второй таблицы и соответствующие им записи из первой таблицы. Если соответствия нет, поля из первой таблицы будут пустыми. **FULL JOIN** - полное внешнее соединение. Комбинация двух предыдущих. В результирующем наборе присутствуют все записи из первой таблицы и соответствующие им записи из второй таблицы. Если соответствия нет - поля из второй таблицы будут пустыми. Записи из второй таблицы, которым не нашлось пары в первой таблице, тоже будут присутствовать в результирующем наборе. В этом случае поля из первой таблицы будут пустыми. **CROSS JOIN** - Результирующий набор содержит все варианты комбинации строк из первой таблицы и второй таблицы. Условие соединения при этом не указывается.

Примеры различий LEFT JOIN и RIGHT JOIN

(*!NeedsWork*)

Table1

Key1	Field1
1	A
2	B
3	C

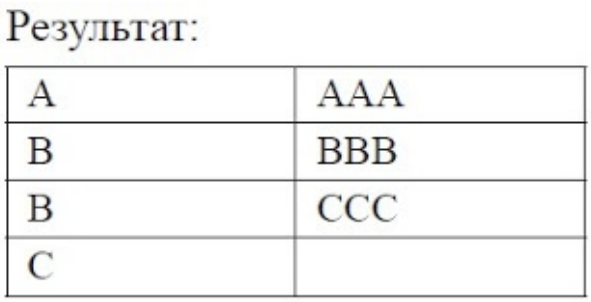
Table2

Key2	Field2
1	AAA
2	BBB
2	CCC
4	DDD

LEFT JOIN

```
SELECT Table1.Field1, Table2.Field2
FROM Table1
LEFT JOIN Table2
ON Table1.Key1 = Table2.Key2
```

Результат:



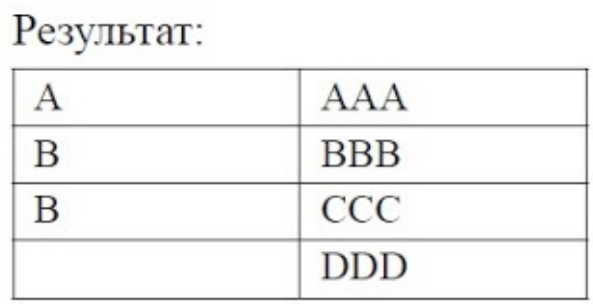
A	AAA
B	BBB
B	CCC
C	

Результат:

RIGHT JOIN

```
SELECT Table1.Field1, Table2.Field2
FROM Table1
RIGHT JOIN Table2
ON Table1.Key1 = Table2.Key2
```

Результат:



A	AAA
B	BBB
B	CCC
	DDD

Результат:

Для чего используется слово HAVING

Ключевое слово HAVING определяет условие, которое затем применяется к групам строк. Следовательно, это предложение имеет тот же смысл для группы строк, что и предложение WHERE в отношении соодержимого соответствующей таблицы. Синтаксис предложения HAVING: `HAVING condition` где condition содержит агрегатные функции или константы.

Важно понимать, что секции HAVING и WHERE взаимно дополняют друг друга. Сначала с помощью ограничений WHERE формируется итоговая выборка, затем выполняется разбивка на группы по значениям полей, заданных в GROUP BY. Далее по каждой группе вычисляется групповая функция и в заключение накладывается условие HAVING. Пример:

```
SELECT DeptNum, MAX(SALARY)
FROM Employees
GROUP BY DeptNum
HAVING MAX(SALARY) > 1000
```

В приведенном примере в результат попадут только отделы, максимальная зарплата в которых превышает 1000

Что такое DLL

DDL(Data Definition Language) - Команды определения структуры данных. В состав DDL-группы входят команды, позволяющие определять внутреннюю структуру базы данных. Перед тем, как сохранять данные в БД, необходимо создать в ней таблицы и, возможно, некоторые другие сопутствующие объекты

Пример некоторых DDL-команд: `CREATE TABLE` - Создание новой таблицы `DROP TABLE` - Удалить существующую таблицу