# Java Core

## Принципы ООП

**Наследование** — это свойство системы, позволяющее описать новый класс на основе уже существующего с частично или полностью заимствующейся функциональностью. Класс, от которого производится наследование, называется базовым, родительским или суперклассом. Новый класс — потомком, наследником или производным классом. Процесс, в результате которого один объект получает свойства другого, именуется *наследованием*.

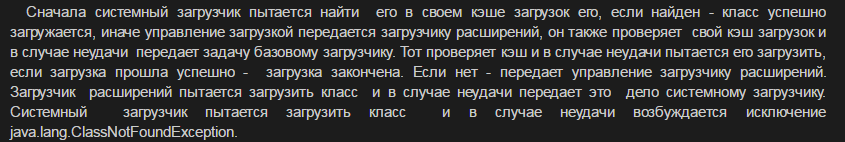
**Инкапсуляция** — это свойство системы, позволяющее объединить данные и методы, работающие с ними, в классе и скрыть детали реализации от пользователя. Инкапсуляцию можно считать защитной оболочкой, которая предохраняет код и данные от произвольного доступа со стороны другого кода, находящегося снаружи оболочки. Доступ к коду и данным, находящимся внутри оболочки, строго контролируется тщательно определенным интерфейсом.

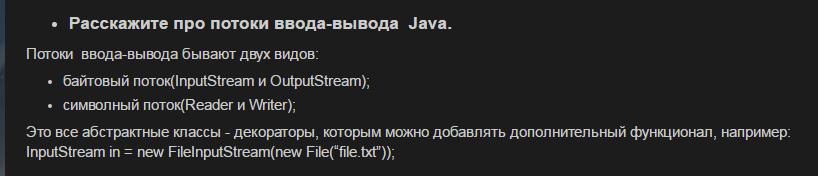
**Абстракция** — способ выделения самых значимых характеристик объекта, при этом менее значимые отбрасываются. В ООП абстракция — работа только со значимыми характеристиками.

**Полиморфизм** — способность объектов самим определять, какие методы они должны применить в зависимости от того, где именно в коде они находятся. То есть, объект может изменяться в зависимости от своего местоположения и действовать по-разному, что позволяет не заводить лишних структур. Иначеговоря: один интерфейс — множество решений. Пример Энимал и собаки один итерфейс.

Универсальный пример полиморфизма Generic и Наследловлание. Комопнент у которого есть функционал кторый может корректно работать с разными типами. Пример: AL<**T**> c Int, String и т.д. или

наследование drawAll(**List**<**Figure**>) : figure





## Типы данных

Язык Java определяет восемь *элементарных* типов данных: byte , short , int ,

long, char, float , double и boolean .

Элементарные типы называют также *простыми*,

long 64 от -9223372036854775808 до 9223372036854775807

int 32 от -2147483648 до 2147483647

short 16 от -32768 до 32767

byte 8 от -128 до 127

char 16

В ссылочные типы входят все классы, интерфейсы, массивы. Описанный выше тип String также относится к ссылочным типам. Этот класс из стандартной библиотеки Java.

 Вспомним [прошлый урок](http://study-java.ru/uroki-java/urok-5-sozdanie-i-vyzov-klassa-v-java/), где мы создали класс Cat, а потом создали переменнуюourcat типа Cat при создании экземпляра класса.

 Cat ourcat = new Cat();

## Класс Object

Класса **Object** - корень иерархии классов. Каждый класс имеет **Object** как суперкласс. Все объекты, включая массивы, реализуют методы этого класса.

Конструктор:  
**Object()**

Методы:

* **Protected Object clone()** - Создает и возвращает копию этого объекта
* **protected void finalize()** - Вызывается сборщиком мусора когда нет больше ссылок на объект
* **Class getClass()** - Возвращает класс во время выполнения
* **Int hashCode()** - Возвращает hash код для объекта
* **Void notify()** - Возобновляет единичный поток, который ожидает на объектном мониторе
* **Void notifyAll()** - Возобновляет все потоки, которые ожидают на объектном мониторе
* **String toString()** - Возвращает строковое представление объекта
* **Void wait()** - Остановка текущего потока пока другой поток не вызовет notify() или notifyAll метод для этого объекта
* public Boolean equals(Object obj)

### EqualsиHashcode

Переопределяют equals в том случае если необходимо, чтобы разные объекты с одинаковым содержимым рассматривались как равные, то надо переопределить equals. И вследствии этого необходимо переопределить hashcode, так как может возникнуть ситуация что для логически одинаковых объектов может быть разный hashcode.

**Public native int hashCode**();

При вычислении хэш-кода для объектов класса Object по умолчанию используется [Park-MillerRNG](http://en.wikipedia.org/wiki/Park-Miller_random_number_generator) алгоритм. В основу работы данного алгоритма положен генератор случайных чисел. Это означает, что при каждом запуске программы у объекта будет разный хэш-код.

В итоге, в терминах Java, хеш-код — это целочисленный результат работы метода, которому в качестве входного параметра передан объект. Хеширование применяется в следующих случаях:

* при построении [ассоциативных массивов](https://ru.wikipedia.org/wiki/%D0%90%D1%81%D1%81%D0%BE%D1%86%D0%B8%D0%B0%D1%82%D0%B8%D0%B2%D0%BD%D1%8B%D0%B9_%D0%BC%D0%B0%D1%81%D1%81%D0%B8%D0%B2" \o "Ассоциативный массив);
* при поиске дубликатов в сериях наборов данных;
* при построение уникальных идентификаторов для наборов данных;
* при вычислении [контрольных сумм](https://ru.wikipedia.org/wiki/%D0%9A%D0%BE%D0%BD%D1%82%D1%80%D0%BE%D0%BB%D1%8C%D0%BD%D0%B0%D1%8F_%D1%81%D1%83%D0%BC%D0%BC%D0%B0) от данных (сигнала) для последующего обнаружения в них ошибок (возникших случайно или внесённых намеренно), возникающих при хранении и/или передаче данных;
* при сохранении паролей *в системах защиты* в виде хеш-кода (для восстановления пароля по хеш-коду требуется функция, являющаяся [обратной](https://ru.wikipedia.org/wiki/%D0%9E%D0%B1%D1%80%D0%B0%D1%82%D0%BD%D0%B0%D1%8F_%D1%84%D1%83%D0%BD%D0%BA%D1%86%D0%B8%D1%8F) по отношению к использованной хеш-функции);
* при выработке [электронной подписи](https://ru.wikipedia.org/wiki/%D0%AD%D0%BB%D0%B5%D0%BA%D1%82%D1%80%D0%BE%D0%BD%D0%BD%D0%B0%D1%8F_%D0%BF%D0%BE%D0%B4%D0%BF%D0%B8%D1%81%D1%8C) (на практике часто подписывается не само сообщение, а его «хеш-образ»);
* и др.

Ситуация, когда у *разных*объектов *одинаковые*хеш-коды называется — коллизией. Вероятность возникновения коллизии зависит от используемого алгоритма генерации хеш-кода.

Одинаковые объекты — это объекты одного класса с одинаковым содержимым полей.

* для одного и того-же объекта, хеш-код всегда будет одинаковым;
* если объекты одинаковые, то и хеш-коды одинаковые ;
* если хеш-коды равны, то входные объекты не всегда равны (коллизия);
* если хеш-коды разные, то и объекты гарантированно разные;

Эквивалентность и хеш-код тесно связанны между собой, поскольку хеш-код вычисляется на основании содержимого объекта (значения полей) и если у двух объектов одного и того же класса содержимое одинаковое, то и хеш-коды должны быть одинаковые.

**Public Boolean equals**(Object obj){ **return** (**this** == obj); }

Теперь понято, почему Object.equals() работает не так как нужно, ведь он сравнивает ссылки, а не содержимое объектов.  
Далее на очереди hashCode(), который тоже работает не так как полагается.

В частности после переобпределения, метод equals() должен обладать следующими свойствами:

* Симметричность: Для двух ссылок, a и b, a.equals(b) тогда и только тогда, когда b.equals(a)
* Рефлексивность: Для всех ненулевых ссылок, a.equals(a)
* Транзитивность: Если a.equals(b) и b.equals(c), то тогда a.equals(c)
* Консистентность или устойчивость тот случай когда для одних и тех же входных значенией должно возращаться одно и то же.

**publicinthashCode**() {

**finalint** prime = 31;

**int** result = 1;

result = prime \* result + varA;

result = prime \* result + varB;

**return** result;

}

**public** **boolean** equals(Object obj) {

             // проверяет не равен ли obj – null

**if**(obj == **null**) {**return** **false**;}

// проверяет является ли obj объектом App

**if**(!(obj **instanceof** App)){**return** **false**;}

             App obj1 = (App) obj;

             // сравнивает поля экземпляров класса

**return** str1.equals(obj1.str1)

&& str2.equals(obj1.str2)

&& num == obj1.num;

       }

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

    {

             App app1 = **new** App();

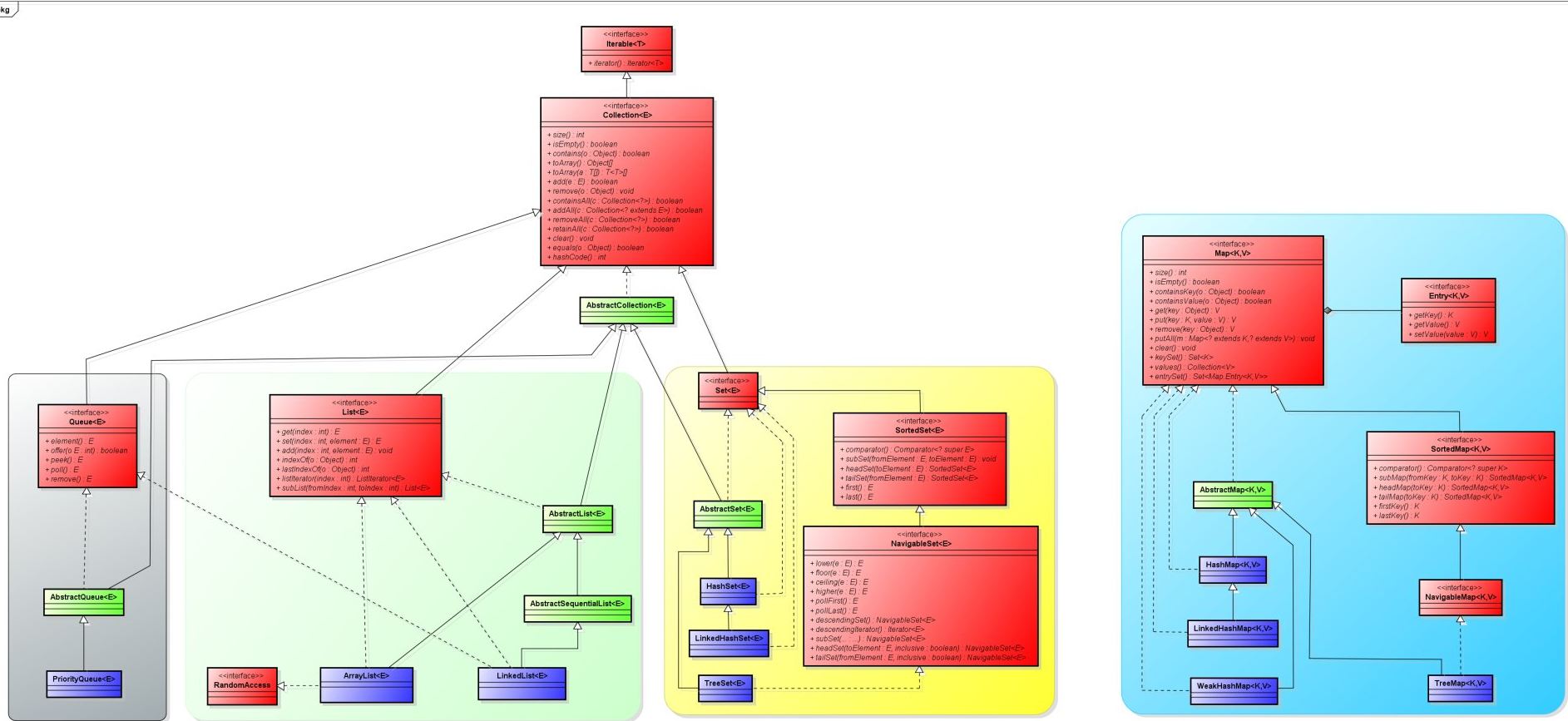
App app2 = **new** App();

System.*out*.println( app1.equals(app2) );

    }

}

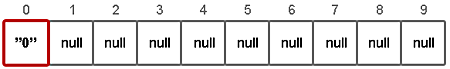
## Collections



ArrayList*— реализует интерфейс List*

Добавление элементов

list.add("0");

[[](https://habrastorage.org/storage1/3720b293/cf78ff93/c9343de2/f9e8c1b7.png)](https://habrastorage.org/storage1/3720b293/cf78ff93/c9343de2/f9e8c1b7.png)  
  
Внутри метода **add(value)** происходят следующие вещи:  
  
1) проверяется, достаточно ли места в массиве для вставки нового элемента;

ensureCapacity(size + 1);

2) добавляется элемент в конец (согласно значению **size**) массива.

elementData[size++] = element;

Весь метод **ensureCapacity(minCapacity)** рассматривать не будем, остановимся только на паре интересных мест. Если места в массиве не достаточно, новая емкость рассчитывается по формуле **(oldCapacity \* 3) / 2 + 1**. Второй момент это копирование элементов. Оно осуществляется с помощью **native** метода **System.arraycopy()**, который написан не на Java.

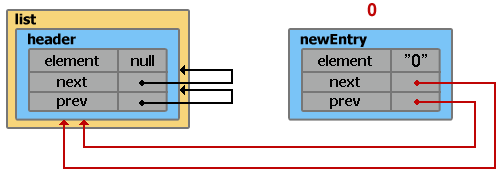
При добавлении 11-го элемента, проверка показывает что места в массиве нет. Соответственно создается новый массив и вызывается**System.arraycopy()**.

Итоги

— Быстрый доступ к элементам по индексу за время O(1);  
— Доступ к элементам по значению за линейное время O(n);  
— Медленный, когда вставляются и удаляются элементы из «середины» списка;  
— Позволяет хранить любые значения в том числе и null;

LinkedList *— реализует интерфейс List.*

Каждый раз при добавлении нового элемента, по сути выполняется два шага:  
  
1) создается новый новый экземпляр класса **Entry**

[[](https://habrastorage.org/storage1/789c9810/a8612e8a/20246617/7fe32bf0.png)](https://habrastorage.org/storage1/789c9810/a8612e8a/20246617/7fe32bf0.png)  
  
2) переопределяются указатели на предыдущий и следующий элемент

Итоги

— Из LinkedList можно организовать стэк, очередь, или двойную очередь, со временем доступа O(1);  
— На вставку и удаление из середины списка, получение элемента по индексу или значению потребуется линейное время O(n). Однако, на добавление и удаление из середины списка, используя ListIterator.add() и ListIterator.remove(), потребуется O(1);  
— Позволяет добавлять любые значения в том числе и null. Для хранения примитивных типов использует соответствующие классы-оберки;

HashMap *— основан на хэш-таблицах, реализует интерфейс Map*

Новоявленный объект hashmap, содержит ряд свойств:

* **table** — Массив типа **Entry[]**, который является хранилищем ссылок на списки (цепочки) значений;
* **loadFactor** — Коэффициент загрузки. Значение по умолчанию 0.75 является хорошим компромиссом между временем доступа и объемом хранимых данных;
* **threshold** — Предельное количество элементов, при достижении которого, размер хэш-таблицы увеличивается вдвое. Рассчитывается по формуле **(capacity \* loadFactor)**;
* **size** — Количество элементов HashMap-а;

В конструкторе, выполняется проверка валидности переданных параметров и установка значений в соответствующие свойства класса. Словом, ничего необычного.

**Разрешение коллизий**

Разрешение коллизий при помощи цепочек.

Каждая ячейка массива *H* является указателем на [связный список](https://ru.wikipedia.org/wiki/%D0%A1%D0%B2%D1%8F%D0%B7%D0%BD%D1%8B%D0%B9_%D1%81%D0%BF%D0%B8%D1%81%D0%BE%D0%BA) (цепочку) пар ключ-значение, соответствующих одному и тому же хеш-значению ключа. Коллизии просто приводят к тому, что появляются цепочки длиной более одного элемента.

Операции поиска или удаления элемента требуют просмотра всех элементов соответствующей ему цепочки, чтобы найти в ней элемент с заданным ключом. Для добавления элемента нужно добавить элемент в конец или начало соответствующего списка, и, в случае, если коэффициент заполнения станет слишком велик, увеличить размер массива *H* и перестроить таблицу.

При предположении, что каждый элемент может попасть в любую позицию таблицы *H* с равной вероятностью и независимо от того, куда попал любой другой элемент, среднее [время работы](https://ru.wikipedia.org/wiki/%D0%A1%D0%BB%D0%BE%D0%B6%D0%BD%D0%BE%D1%81%D1%82%D1%8C_%D0%B0%D0%BB%D0%B3%D0%BE%D1%80%D0%B8%D1%82%D0%BC%D0%B0)операции поиска элемента составляет [Θ](https://ru.wikipedia.org/wiki/%D0%9E-%D0%BD%D0%BE%D1%82%D0%B0%D1%86%D0%B8%D1%8F)(1 + *α*), где *α* — коэффициент заполнения таблицы.

Открытая адресация

Пример хеш-таблицы с открытой адресацией и линейным пробированием, получающейся при вставке элементов в левой колонке сверху вниз.

В массиве *H* хранятся сами пары ключ-значение. Алгоритм вставки элемента проверяет ячейки массива *H* в некотором порядке до тех пор, пока не будет найдена первая свободная ячейка, в которую и будет записан новый элемент. Этот порядок вычисляется на лету, что позволяет сэкономить на памяти для указателей, требующихся в хеш-таблицах с цепочками.

Алгоритм поиска просматривает ячейки хеш-таблицы в том же самом порядке, что и при вставке, до тех пор, пока не найдется либо элемент с искомым ключом, либо свободная ячейка (что означает отсутствие элемента в хеш-таблице).

Удаление элементов в такой схеме несколько затруднено. Обычно поступают так: заводят булевый флаг для каждой ячейки, помечающий, удален элемент в ней или нет. Тогда удаление элемента состоит в установке этого флага для соответствующей ячейки хеш-таблицы, но при этом необходимо модифицировать процедуру поиска существующего элемента так, чтобы она считала удалённые ячейки занятыми, а процедуру добавления — чтобы она их считала свободными и сбрасывала значение флага при добавлении.

**Последовательности проб**

* [Линейное пробирование](https://ru.wikipedia.org/w/index.php?title=%D0%9B%D0%B8%D0%BD%D0%B5%D0%B9%D0%BD%D0%BE%D0%B5_%D0%BF%D1%80%D0%BE%D0%B1%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5&action=edit&redlink=1): ячейки хеш-таблицы последовательно просматриваются с некоторым фиксированным интервалом *k* между ячейками (обычно *k* = 1), то есть *i*-й элемент последовательности проб — это ячейка с номером (hash(*x*) + *ik*) mod *N*. Для того, чтобы все ячейки оказались просмотренными по одному разу, необходимо, чтобы *k* было [взаимно-простым](https://ru.wikipedia.org/wiki/%D0%92%D0%B7%D0%B0%D0%B8%D0%BC%D0%BD%D0%B0%D1%8F_%D0%BF%D1%80%D0%BE%D1%81%D1%82%D0%BE%D1%82%D0%B0) с размером хеш-таблицы.
* [Квадратичное пробирование](https://ru.wikipedia.org/w/index.php?title=%D0%9A%D0%B2%D0%B0%D0%B4%D1%80%D0%B0%D1%82%D0%B8%D1%87%D0%BD%D0%BE%D0%B5_%D0%BF%D1%80%D0%BE%D0%B1%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5&action=edit&redlink=1): интервал между ячейками с каждым шагом увеличивается на константу. Если размер хеш-таблицы равен степени двойки (*N* = 2*p*), то одним из примеров последовательности, при которой каждый элемент будет просмотрен по одному разу, является:

hash(*x*) mod *N*, (hash(*x*) + 1) mod *N*, (hash(*x*) + 3) mod *N*, (hash(*x*) + 6) mod *N*, …

* [Двойное хеширование](https://ru.wikipedia.org/w/index.php?title=%D0%94%D0%B2%D0%BE%D0%B9%D0%BD%D0%BE%D0%B5_%D1%85%D0%B5%D1%88%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5&action=edit&redlink=1): интервал между ячейками фиксирован, как при линейном пробировании, но, в отличие от него, размер интервала вычисляется второй, вспомогательной хеш-функцией, а значит, может быть различным для разных ключей. Значения этой хеш-функции должны быть ненулевыми и взаимно-простыми с размером хеш-таблицы, что проще всего достичь, взяв [простое число](https://ru.wikipedia.org/wiki/%D0%9F%D1%80%D0%BE%D1%81%D1%82%D0%BE%D0%B5_%D1%87%D0%B8%D1%81%D0%BB%D0%BE) в качестве размера, и потребовав, чтобы вспомогательная хеш-функция принимала значения от 1 до *N* — 1.

LinkedHashMap*расширяет класс HashMap и реализует интерфейс Map*

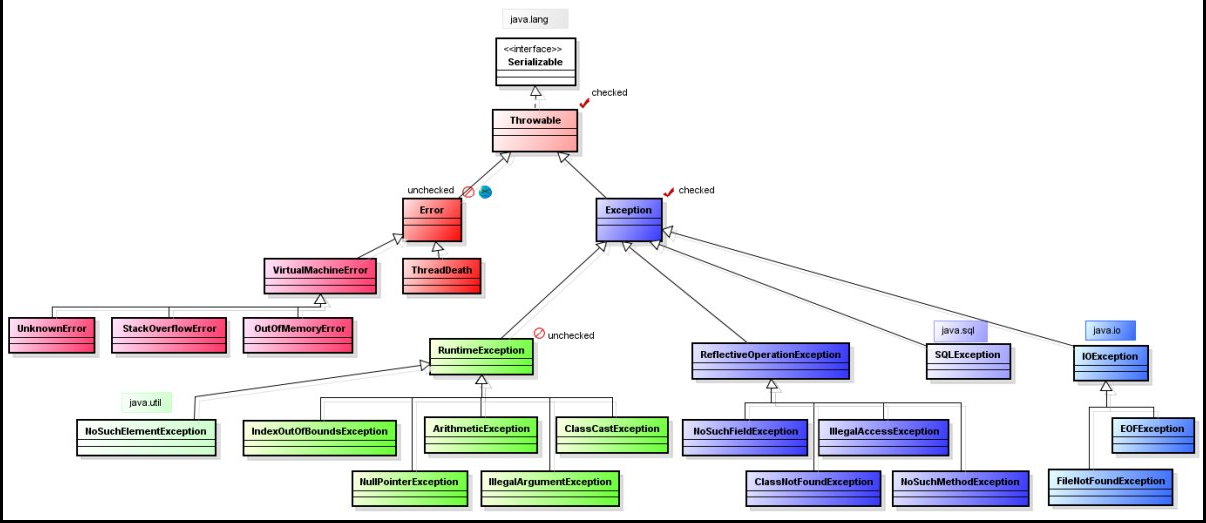
Только что созданный объект **linkedHashMap**, помимо свойств унаследованных от **HashMap** (такие как table, loadFactor, threshold, size, entrySet и т.п.), так же содержит два доп. свойства:

* **header** — «голова» двусвязного списка. При инициализации указывает сам на себя;
* **accessOrder** — указывает каким образом будет осуществляться доступ к элементам при использовании итератора. При значении**true** — по порядку последнего доступа (об этом в [конце](https://habrahabr.ru/post/129037/#accessOrderTrue) статьи). При значении **false** доступ осуществляется в том порядке, в каком элементы были вставлены.

[TreeMap](http://docs.oracle.com/javase/8/docs/api/java/util/TreeMap.html) — реализация Map основанная на красно-чёрных деревьях. Как и LinkedHashMap является упорядоченной. По-умолчанию, коллекция сортируется по ключам с использованием принципа "[naturalordering](http://docs.oracle.com/javase/8/docs/api/java/lang/Comparable.html)", но это поведение может быть настроено под конкретную задачу при помощи объекта Comparator, которые указывается в качестве параметра при создании объекта TreeMap.

Красно-чёрное дерево — двоичное дерево поиска, в котором каждый узел имеет атрибут цвет, принимающий значения красный или черный. В дополнение к обычным требованиям, налагаемым на двоичные деревья поиска, к красно-чёрным деревьям применяются следующие требования:  
  
1. Узел либо красный, либо чёрный.  
2. Корень — чёрный. (В других определениях это правило иногда опускается. Это правило слабо влияет на анализ, так как корень всегда может быть изменен с красного на чёрный, но не обязательно наоборот).  
3. Все листья(NIL) — черные.  
4. Оба потомка каждого красного узла — черные.  
5. Всякий простой путь от данного узла до любого листового узла, являющегося его потомком, содержит одинаковое число черных узлов.  
  
[WeakHashMap](http://docs.oracle.com/javase/8/docs/api/java/util/WeakHashMap.html) — реализация хэш-таблицы, которая организована с использованием [weakreferences](http://docs.oracle.com/javase/8/docs/api/java/lang/ref/WeakReference.html). Другими словами, GarbageCollector автоматически удалит элемент из коллекции при следующей сборке мусора, если на ключ этого элеметна нет жёстких ссылок.

## Exceptions



То есть если в вашем коде есть участок который может бросить checked исключение то вы обязаны либо заключить его в конструкцию try/catch либо объявить в сигнатуре метода "throwsSomeException" но в данном случае обработка исключения делегируется на уровень выше. В любом случае его нужно будет перехватить. В противном случае программа просто не скомпилируется.

Почему я выделил **Throwable, Error, Exception и RuntimeException.**

Throwable - *проверяемое*исключение

Error -*не проверяемое*

Exception -*проверяемое*

RuntimeException - *не проверяемое*

Все остальные исключения наследуют это свойство от данных классов. Например SQLException является наследником Exception и оно проверяемое. *NullPointerException*- наследник RuntimeException и оно не проверяемое.

Так почему не все исключения являются проверяемыми? Дело в том, что если проверять каждое место, где теоретически может быть ошибка, то ваш код сильно разрастется, станет плохо читаемым. Например в любом месте, где происходит деление чисел, нужно было бы проверять на *ArithmeticException*, потому что возможно деление на ноль. Эту опцию(пере отлавливать не проверяемые исключения) создатели языка оставили программисту на его усмотрение.

**public** **class** SuperClass {     
    **public** **void** start() **throws** IOException{   
        **throw** **new** IOException("Not able to open file");   
    }   
}   
**public** **class** SubClass **extends** SuperClass{     
    **public** **void** start() **throws** Exception{   
        **throw** **new** Exception("Not able to start");   
    }   
}

Он не скомпилируется.

Дело в том, что метод start() был переопределен в сабклассе и в сигнатуре был указан более общий класс исключения. Согласно [правилам переопределения методов](http://java67.blogspot.com/2012/08/what-is-method-overriding-in-java-example-tutorial.html), это действие не допустимо.

Можно лишь сужать класс исключения:

Система не может предусмотреть все исключения, иногда вам придётся создать собственный тип исключения для вашего приложения. Вам нужно наследоваться от **Exception**(напомню, что этот класс наследуется от **Trowable**) и переопределить нужные методы класса **Throwable**.

**Эта программа создает пользовательский**

**classMyExceptionextendsException {**

**private int detail;**

**гиписключения.**

**MyException(int a) {**

**detail = a;**

}

**public String toString()**

**return "MyException['**

}

Часть исключений может обрабатывать сама система. Но можно создать собственные исключения при помощи оператора **throw**. Код выглядит так:

throw экземпляр\_Throwable

Вам нужно создать экземпляр класса **Throwable** или его наследников. Получить объект класса **Throwable** можно в операторе **catch** или стандартным способом через оператор **new**.

Поток выполнения останавливается непосредственно после оператора **throw** и другие операторы не выполняются. При этом ищется ближайший блок **try/catch** соответствующего исключению типа.

## Многопоточность

Процесс — это совокупность кода и данных, разделяющих общее виртуальное адресное пространство. Процессы изолированы друг от друга, поэтому прямой доступ к памяти чужого процесса невозможен (взаимодействие между процессами осуществляется с помощью специальных средств). Для каждого процесса ОС создает так называемое «виртуальное адресное пространство», к которому процесс имеет прямой доступ. Это пространство принадлежит процессу, содержит только его данные и находится в полном его распоряжении. Операционная система же отвечает за то, как виртуальное пространство процесса проецируется на физическую память.

Один поток («нить» или «трэд») – это одна единица исполнения кода. Каждый поток последовательно выполняет инструкции процесса, которому он принадлежит, параллельно с другими потоками этого процесса.

Бывает многозадачность на потоках и на процессах.

* Если t  является экземпляром класса Thread, чей поток в данный момент продолжает выполняться, то

t.join();

приведёт к приостановке выполнения текущего потока до тех пор, пока поток t  не завершит свою работу.

* **sleep**

Задержка потока на заднное время. Время задается в миллисекундах и наносекундах

public static void sleep(long millis);

Задержка потока на заднное время. Время задается в миллисекундах и наносекундах

public static void sleep(long millis, int nanos);

* ВызовThread.interrupt()  устанавливает этот флаг. Когда поток проверяет наличие прерывания вызовов Thread.interrupted(), то флаг статуса прерывания сбрасывается. Нестатический метод isInterrupted(), который используется одним потоком для проверки статуса прерывания другого потока, не меняет флаг статуса прерывания.

Поток может быть в следующем состоянии: созданный, запущенный, блокированный, остановленный.

“монитор”.

Несколько нитей могут мешать друг другу при обращении к одним и тем же данным. Для решения этой проблемы придуман мьютекс (он же монитор). Он имеет два состояния — объект занят и объект свободен. Монитор(мьютекс) — высокоуровневый механизм взаимодействия и синхронизации процессов, обеспечивающий доступ к неразделяемым ресурсам.

http://javastudy.ru/category/interview/

https://jsehelper.blogspot.ru/2016/01/blog-post\_59.html

<https://habrahabr.ru/post/162017/>

https://habrahabr.ru/company/luxoft/blog/157273/

<https://urvanov.ru/2016/05/27/java-8-%D0%BC%D0%BD%D0%BE%D0%B3%D0%BE%D0%BF%D0%BE%D1%82%D0%BE%D1%87%D0%BD%D0%BE%D1%81%D1%82%D1%8C/>

## String

**Метод append()** — обновляет значение объекта, который вызвал метод. Этот метод в Java принимает boolean, char, int, long, Strings (StringBuffer). и т.д.

StringBuilder – без синхронайз, StringBuffer - с синхронайзом.

<https://habrahabr.ru/post/260767/>

<https://www.youtube.com/watch?v=GuCfBOGGkW0>

<http://info.javarush.ru/translation/2014/02/07/Java-String-%D0%92%D0%BE%D0%BF%D1%80%D0%BE%D1%81%D1%8B-%D0%BA-%D1%81%D0%BE%D0%B1%D0%B5%D1%81%D0%B5%D0%B4%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8E-%D0%B8-%D0%BE%D1%82%D0%B2%D0%B5%D1%82%D1%8B-%D0%BD%D0%B0-%D0%BD%D0%B8%D1%85-.html>

## Интерфейс и абстрактный класс

концептуальное различие:  
абстрактный класс (как и любой класс) описывает СУЩНОСТЬ т.е. "предмет", тогда как интерфейс описывает КОНТРАКТ (поведение или свойства в отрыве от самого предмета). Наследование(реализация) интерфейсов почти всегда предпочтительнее наследования классов, т.к. более гибко, и допускает множественность реализации (множественное наследование для классов в Java отсутствует).

Абстрактный класс может реализовывать методы; интерфейс не может реализовывать методы.

Интерфейс может только описывать константы и методы, но не реализовывать их. Все методы интерфейса по-умолчанию являются публичными (public) и абстрактными (abstract), а поля - publicstaticfinal.

В Java класс может наследоваться (реализовывать) от многих интерфейсов, но только от одного абстрактного класса.

плюс можно добавить, что абстрактный класс может содержать в себе конструкторы.

## GC

Давайте начнём с того, как JVM использует память. В JVM память делится на два сегмента – Heap и PermanentGeneration.

Permanent generation используется только JVM для хранения необходимых данных, в том числе метаданные о созданных объектах и стек с примитивами. При каждом создании объекта JVM будет «класть» некоторый набор данных в PG.

Heap – основной сегмент памяти, где хранятся все ваши объекты. Heap делится на два подсегмента, OldGeneration и NewGeneration. NewGeneration в свою очередь делится на Eden и два сегмента Survivor.

ReferencecountingСуть подхода состоит в том, что каждый объект имеет счетчик. Счетчик хранит информацию о том, сколько ссылок указывает на объект. Kогда ссылка уничтожается, счетчик уменьшается. Если значение счетчика равно нулю, - объект можно считать мусором и память можно очищать.

В "Tracing" главная идея состоит в мысли: "Живые объект - те до которых мы можем добраться с корневых точек (GCRoot), все остальные - мусор. Все что доступно с живого объекта - также живое". Самое простое java приложение будет иметь такие корневые точки:

* Локальные переменные внутри main метода, параметры main метода.
* Поток который выполняет main.
* Статические переменные класса, внутри которого находится main метод.

Самый простой алгоритм для garbagecollection называется CopyCollection, и работает он так, как показано на диаграмме. На первом этапе Mark помечаются неиспользуемые объекты (красные). На втором (Copy) объекты, которые ещё нужны (d) копируется в сегмент survivor – квадрат справа. Сегментов Survivor два, и они меньше Eden. Теперь все объекты, которые мы хотим, чтобы они были сохранены, скопированы в Survivor, и JVM просто удаляет всё из Eden. На этом всё.

Следующий алгоритм называется Mark-Sweep-CompactCollection. У алгоритма три этапа:  
  
1) «Mark»: помечаются неиспользуемые объекты (красные).  
  
2) «Sweep»: эти объекты удаляются из памяти. Обратите внимание на пустые слоты на диаграмме.  
  
3) «Compact»: объекты размещаются, занимая свободные слоты, что освобождает пространство на тот случай, если потребуется создать «большой» объект.

<http://ggenikus.github.io/blog/2014/05/04/gc>

<https://habrahabr.ru/post/269621/>

## Сериализация

В Листинге 1 только одна вещь отличается от создания нормального класса, это реализация интерфейса java.io.Serializable. Интерфейс Serializable это интерфейс-маркер; в нём не задекларировано ни одного метода. Но говорит сериализующему механизму, что класс может быть сериализован.  
  
Теперь у нас есть всё необходимое для сериализации объекта, следующим шагом будет фактическая сериализация объекта. Она делается вызовом метода writeObject() класса java.io.ObjectOutputStream, как показано в листинге

public static void main(String args[]) throws IOException {  
  FileOutputStreamfos = new FileOutputStream("temp.out");  
  ObjectOutputStreamoos = new ObjectOutputStream(fos);  
  TestSerialts = new TestSerial();  
  oos.writeObject(ts);  
  oos.flush();  
  oos.close();  
}

public static void main(String args[]) throws IOException {  
  FileInputStreamfis = new FileInputStream("temp.out");  
  ObjectInputStreamoin = new ObjectInputStream(fis);  
  TestSerialts = (TestSerial) oin.readObject();  
  System.out.println("version="+ts.version);  
}

Восстановление объекта происходит с помощью вызова метода oin.readObject(). В методе происходит чтение набора байт из файла и создаие точной копии графа оригинального объекта. oin.readObject() может прочитать любой сериализованный объект, поэтому необходимо полученный объект приводить к конкретному типу.

Второй способ – это тоже реализация интерфейса, но уже другого: java.io.Externalizable. В отличие от java.io.Serializable, он содержит два метода, которые необходимо реализовать – writeExternal(ObjectOutput) и readExternal(ObjectInput). В этих методах как раз и находится логика сериализации/десериализации.

То есть **при десериализации вызывается конструктор без параметров родительского НЕсериализуемого класса**. И если такого конструктора не будет – при десериализации возникнет ошибка. Конструктор же дочернего объекта, того, который мы десериализуем, не вызывается, как и было сказано выше.

Так ведут себя стандартные механизмы при использовании Serializable. При использовании же Externalizable ситуация иная. Сначала *вызывается конструктор без параметров*, а потом уже на созданном объекте вызывается методreadExternal, который и вычитывает, собственно, все свои данные. Потому – **любой реализующий интерфейс Externalizable класс обязан иметь public конструктор без параметров!** Более того, поскольку все наследники такого класса тоже будут считаться реализующими интерфейс Externalizable, у них тоже должен быть конструктор без параметров!

Пойдем дальше. Существует такой модификатор поля как transient. Он означает, что это поле *не должно* быть сериализовано. Однако, как вы сами понимаете, указание это действует только на стандартный механизм сериализации. При использовании Externalizable никто не мешает сериализовать это поле, равно как и вычитать его. Если поле объявлено transient, то при десериализации объекта оно принимает значение по умолчанию.

Еще один достаточно тонкий момент. При стандартной сериализации поля, имеющие модификатор static, *не сериализуются*. Соответственно, после десериализации это поле значения не меняет. Разумеется, при реализацииExternalizable сериализовать и десериализовать это поле никто не мешает, однако я крайне не рекомендую этого делать, т.к. это может привести к трудноуловимым ошибкам.

Поля с модификатором final сериализуются как и обычные. За одним исключением – их невозможно десериализовать при использовании Externalizable. Ибо final-поля должны быть инициализированы в конструкторе, а после этого вreadExternal изменить значение этого поля будет невозможно. Соответственно – если вам необходимо сериализовать объект, имеющий final-поле, вам придется использовать только стандартную сериализацию.

Еще один момент, который многие не знают. При стандартной сериализации учитывается порядок объявления полей в классе. Во всяком случае, так было в ранних версиях, в JVM версии 1.6 реализации Oracle уже порядок неважен, важны тип и имя поля. Состав же методов с очень большой вероятностью повлияет на стандартный механизм, при том, что поля могут вообще остаться теми же. Чтобы этого избежать, есть следующий механизм. В каждый класс, реализующий интерфейс Serializable, на стадии компиляции добавляется еще одно поле – **privatestaticfinallong** serialVersionUID. Это поле 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 мне случай, когда static-поле сериализуется. При десериализации значение этого поля сравнивается с имеющимся у класса в виртуальной машине. Если значения не совпадают – инициируется исключение наподобие этого:

Ссылки на объекты после десериализации отличаются от ссылок до нее. Иначе говоря, при сериализации/десериализации объект *был скопирован*. Этот метод используется иногда для клонирования объектов.

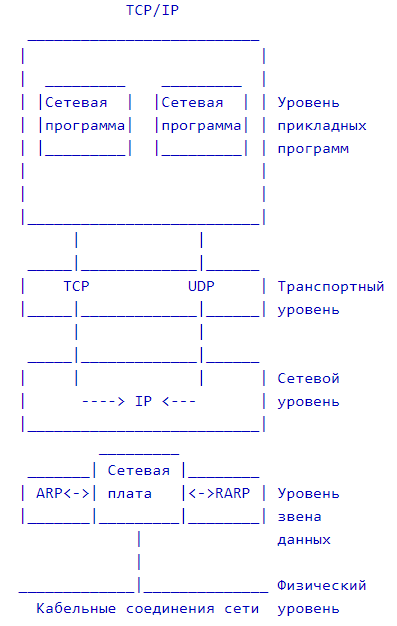
Второй вывод, более сущеcтвенный. При сериализации/десериализации нескольких объектов, имеющих перекрестные ссылки, эти ссылки остаются действительными после десериализации. Иначе говоря, если до сериализации они указывали на один объект, то после десериализации они тоже будут указывать на один объект.

Зачем вообще нужна расширенная сериализация? Ответ прост. Во-первых, она дает гораздо большую гибкость. Во-вторых, зачастую она может дать немалый выигрыш по объему сериализованных данных. В-третьих, существует такой аспект как производительность.

Минус Externalizable: И прежде всего это **нарушение целостности графа**. Поскольку протокол сериализации не используется – контроль целостности остается на самом разработчике.

http://www.skipy.ru/technics/serialization.html

## TCP/IP



# Other

## Вложенные классы

Вложенные классы делятся на два вида: статические и не статические. Вложенные классы, объявленные как статические называются вложенными статическими (staticnestedclasses). Не статические называются внутренними (innerclasses).

Как статические переменные и методы, вложенный (статический) класс связан с внешним **классом**. И так же как и статические методы не может напрямую обращаться к полям объекта внешнего класса(толко создав объект), может обращаться к статическим полям. Внешний класс не видит поля вложенного.

Доступ к вложенному классу осуществляется с помощью следующей конструкции:

|  |  |
| --- | --- |
| 1 | OuterClass.StaticNestedClass |

Синтаксис создания объекта вложенного класса:

|  |  |
| --- | --- |
| 1  2 | OuterClass.StaticNestedClassnestedObject =       newOuterClass.StaticNestedClass(); |

Внутренние (не статические) классы, как переменные и методы связаны с **объектом** внешнего класса. Внутренние классы так же имеют прямой доступ к полям внешнего класса. Такие классы не могут содержать в себе статические методы и поля. Внутренние классы не могут существовать без экземпляра внешнего. Для создания объекта:

|  |  |
| --- | --- |
| 1 | OuterClass.InnerClassinnerObject = outerObject.newInnerClass(); |

Свойства локального класса**:**

*Локальный класс* – это класс, объявленный в блоке Java кода. Обычно локальный класс определяется в методе, но он также может быть объявлен в инициализаторе экземпляра класса. Поскольку все блоки Java кода находятся внутри определения класса, то все локальные классы вложены в окружающие классы.

Определяющей характеристикой локального класса является вложенность для блока кода. Как и локальная переменная, локальный класс действителен только в области видимости окружающего блока.

* Подобно вложенным нестатическим классам, локальные классы связаны с окружающим экземпляром и имеют доступ ко всем членам, включая private члены окружающего класса.
* Локальный класс нельзя объявить с каким-либо модификатором доступа. Эти модификаторы используются только для членов класса; они не доступны для объявления локальных переменных или классов.
* Как и нестатические вложенные классы, и по тем же причинам, локальные классы не могут иметь static поля, исключение составляют константы, объявленные как static final.

## Анонимные классы

*Анонимный класс* – локальный класс без имени. Анонимный класс определяется и инициализируется в едином выражении с помощью оператора new. Может своего рода сделать экземпляр интерфейса или абстрактного класса и тут же обработать его в блоке.

<http://pr0java.blogspot.ru/2015/09/5.html>

## Модификаторы

* **private** члены класса доступны только внутри класса
* **package-private или default (по умолчанию)** члены класса видны внутри пакета
* **protected** члены класса доступны внутри пакета и в классах-наследниках
* **public** члены класса доступны всем

### **Static**

* Применяется к внутренним классам, методам, переменным и логическим блокам
* Статические переменные инициализируются во время загрузки класса
* Статические переменные едины для всех объектов класса (одинаковая ссылка)
* Статические методы имеют доступ только к статическим переменным
* К статическим методам и переменным можно обращаться через имя класса
* Статические блоки выполняются во время загрузки класса
* Не static методы не могут быть переопределены как static
* Локальные переменные не могут быть объявлены как static
* Абстрактные методы не могут быть static
* Static поля не сериализуются (только при реализации интерфейса Serializable)
* Только static переменные класса могут быть переданы в конструктор с параметрами, вызывающийся через слово super(//параметр//) или this(//параметр//)

### **Abstract**

* Применяется только для методов и классов
* У абстрактных методов нет тела метода
* Является противоположностью final: final класс не может наследоваться, abstract класс обязан наследоваться
* Класс должен быть объявлен как abstract если:

1. он содержит хотя бы один абстрактный метод
2. он не предоставляет реализацию наследуемых абстрактных методов
3. он не предоставляет реализацию методов интерфейса, реализацию которого он объявил
4. необходимо запретить создание экземпляров класса

### **Final**

### Поля не могут быть изменены, методы переопределены

* Классы нельзя наследовать
* Этот модификатор применяется только к классам, методам и переменным (также и к локальным переменным)
* Аргументы методов, обозначенные как final, предназначены только для чтения, при попытке изменения будет ошибка компиляции
* Переменные final не инициализируются по умолчанию, им необходимо явно присвоить значение при объявлении или в конструкторе, иначе – ошибка компиляции
* Если final переменная содержит ссылку на объект, объект может быть изменен, но переменная всегда будет ссылаться на тот же самый объект
* Также это справедливо и для массивов, потому что массивы являются объектами, – массив может быть изменен, а переменная всегда будет ссылаться на тот же самый массив
* Если класс объявлен final и abstract (взаимоисключающие понятия), произойдет ошибка компиляции
* Так как final класс не может наследоваться, его методы никогда не могут быть переопределены

**Конструктор** не может быть static, abstract или final

# Java EE

## Continious Integration and continuous delivery

<https://ru.atlassian.com/continuous-delivery/principles/continuous-integration-vs-delivery-vs-deployment>

## SOLID



1. Например, представьте себе модуль, который составляет и печатает отчёт. Такой модуль может измениться по двум причинам. Во-первых, может измениться само содержимое отчёта. Во-вторых, может измениться формат отчёта.
2. Предположим, что у нас есть объект SmtpMailer. Для логирования своих действий он использует Logger, который записывает информацию в текстовые файлы. А потом в БД.
3. Поведение списка DoubleList отличается от типичных реализаций IList. Получается, что наш DoubleList не может быть заменен базовым типом. Это и есть нарушение принципа замещения Лисков.
4. Разработчик Алекс создал интерфейс «отчет», и добавил два метода: generateExcel() и generatedPdf(). Теперь клиент А хочет использовать этот интерфейс, но он намерен использовать отчеты только в PDF формате, а не в Excel.
5. Модули верхнего уровня не должны зависеть от модулей нижнего уровня. Оба должны зависеть от абстракции.

Абстракции не должны зависеть от деталей. Детали должны зависеть от абстракций.

Другими словами, следует разрабатывать программное обеспечение таким образом, что различные модули были автономными, и соединялись друг с другом с помощью абстракции. Классическое использование этого принципа можно наблюдать на примере [Spring framework.](http://howtodoinjava.com/spring-3/)В рамках Spring framework все модули выполнены в виде отдельных компонентов, которые могут работать вместе. Они выполнены настолько автономно, что могут использоваться в других программных модулях, кроме Spring framework с такой же легкостью.

<http://blog.byndyu.ru/2009/12/blog-post.html>

<http://info.javarush.ru/translation/2013/08/06/%D0%9F%D1%8F%D1%82%D1%8C-%D0%BE%D1%81%D0%BD%D0%BE%D0%B2%D0%BD%D1%8B%D1%85->%D0%BF%D1%80%D0%B8%D0%BD%D1%86%D0%B8%D0%BF%D0%BE%D0%B2-%D0%B4%D0%B8%D0%B7%D0%B0%D0%B9%D0%BD%D0%B0-%D0%BA%D0%BB%D0%B0%D1%81%D1%81%D0%BE%D0%B2-S-O-L-I-D-%D0%B2-Java.html

## Методологии

<https://habrahabr.ru/company/edison/blog/313410/>

<https://geekbrains.ru/posts/methodologies>

<https://netology.ru/blog/scrum-kanban>

<https://www.youtube.com/watch?v=TPrj-AMJ4Ds>

## Agile

Манифест:

* Общение важнее процесоов и инструментов
* Рабочий продукт важнее документации
* Диалог с заказчиком важнее утверждения контрактов
* Изменения важнее следования первоночальному плану

Цикл:



Спринт:

Длина спринта одинаковая от 3-4 недель.

Пожелания заказчика пишется в бэклог, затем уточняется план спринтов, ежедневно проводятся встечи с командой(scrum) где контролируется ход работ. В конце каждого спринта заказчик видит и анализирует конкретный результат что бы внести корректировки.

МЖП:

Концетнтрация на минимальном жизнеспособном продукте что бы проверить какие даработки необходимы а какие функции излишние, что бы показать пользователям.

Приемущества:

* Детальное задание не обязательно.
* Разработка под контролем.
* Цена ниже ввиду отсутсвия рисков залаженых в фиксируемую стоимость.

**UML** – описывает характер отношений между сущностями

**Делегирование** – задачу пример метод sin() : int больше 50% работы делает Math.sin()

**Ассоциация** – это когда класс связан с каким то другим классом хоть как то, пример класс Врач с классом страховка.

**Агрегирование** – часть целого они могут быть в разных группах(не зависимы), пример Student может быть частью многих кружков и групп.

**Композиция** – часть целого которая очень сильно привязана к целому. Печень часть организма. Ноды принадлежащие LinkedList.

## Patterns

**Структурные** шаблоны определяют отношения между классами и объектами, позволяя им работать совместно.

* **Мост** — структурный шаблон проектирования, используемый в проектировании программного обеспечения чтобы разделять абстракцию и реализацию так, чтобы они могли изменяться независимо. Представим, что у вас есть сайт с разными страницами, и вам надо разрешить пользователям менять их тему. *Создаем клас, а вконструктор отдаем тему и контент.*
* **Декоратор** — структурный шаблон проектирования, предназначенный для динамического подключения дополнительного поведения к объекту. *Добавляет новый функционал Кофе простое – а можно с молоком сливками и пр.*
* **Фасад** — структурный шаблон проектирования, позволяющий скрыть сложность системы путём сведения всех возможных внешних вызовов к одному объекту, делегирующему их соответствующим объектам системы. *Создаешь отдельный клас, а потом в методе или конструкторе инициализируешь всю цепочку подводную.*
* **Заместитель(Proxy)** — структурный шаблон проектирования, который предоставляет объект, который контролирует доступ к другому объекту, перехватывая все вызовы (выполняет функцию контейнера). *AOP в спринге, как пример логер.*
* **Адаптер** — структурный шаблон, который позволяет использовать класс, реализующий нужные функции, но имеющий неподходящий интерфейс. *Создает новый интрефейс имплеминтируя существующий и дополняет его.*

**Порождающие** шаблоны предоставляют механизмы инициализации, позволяя создавать объекты удобным способом.

* Простая фабрика. Собирает нужный объект. Мои лягушки.
* Фабричный метод Собирает нужные классы.
* Абстрактная фабрика. Фабрика фабрик. Пример JDBC
* Singleton

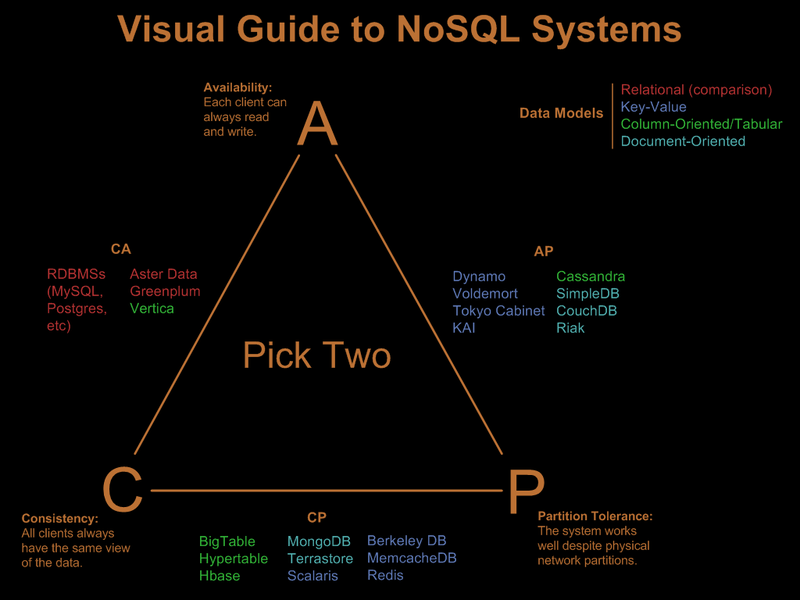


**Поведенческие** шаблоны используются для того, чтобы упростить взаимодействие между сущностями.

* **Стратегия** — поведенческий шаблон, который позволяет выбрать поведение программы в процессе выполнения в зависимости от контекста путем инкапсуляции нескольких алгоритмов в разных классах. *Для примера, в зависимости от ситуаций может обновить что то, а если не существует сущности может ее создать.*
* **Наблюдатель** — поведенческий шаблон проектирования, также известен как «подчинённые» (Dependents). Создает механизм у класса, который позволяет получать экземпляру объекта этого класса оповещения от других объектов об изменении их состояния, тем самым наблюдая за ними. *Пример слушателя мышки клави и тп. Оповещает по событию.*
* **Итератор** — поведенческий шаблон проектирования. Представляет собой объект, позволяющий получить последовательный доступ к элементам объекта-агрегата без использования описаний каждого из агрегированных объектов. *Пример ResaulSet.*

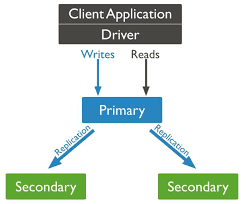
<https://tproger.ru/translations/design-patterns-for-beginners/>

## Mongo

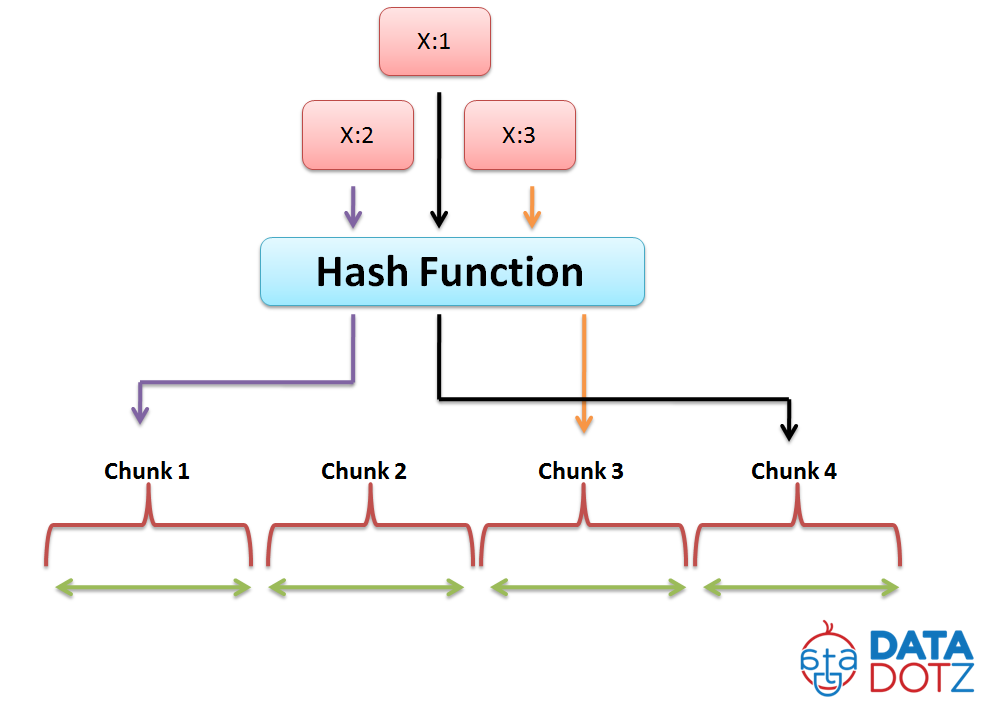


BSON – как JSON

Репликация – это разделение на согласованность для работоспособности сервера в сети. Если сервер один падает, его заменяет другой.(теневой сервер, орбитр) Число серверов должно быть нечетное, что бы решались конфликты.



Шардирование – это как будут делиться сервера. Разделение по диапозонам ключей. Или еще по hash cod, надо осторожнее так как коллизии и фрагментация.



Плюсы:

* Используется для неустоявшихся огромных данных.
* Легко расширяется на нескольких серверах в отличии от реляционных баз.

Минусы:

* Нет транзакций, но етсь как в гите признак версионности.
* Нет JOIN данные денормализированы

Сreate – insert() db.students.insert({name: “Dasha”, nickname:”Pretty”, group:”PI-401”})

Read – find() db.students.find() -> “\_id”: OdjectID(“56cc..”) в ответе завшиврованы в строке Id процесса id машины timestamp за с чет этого дв аразных процесса

Update – Update() не кофликтуют.

Delete – remove()

<https://www.youtube.com/watch?v=CZKl22peGJk&t=914s&list=PLENtfSPQbIknDa_JFGR4oYYWi70xCl9ZS&index=61>

## Maven

В депенденси существует транзитивная зависимость, в следствии этого существует так же проблема разных версий одной библиотеки. Жизненный цикл состоит из фаз. Фазы вызывают работу плагинов у плагина есть цели(goals) у них разные задачи(task). Так же существует пом родитель(суперпом). Он в каждом поме по умолчанию.

Из минусов – Если стандартная структура проекта то все работает норм, если что то поменять то надо заморачиваться.

## Servlet

**Сервлет** является интерфейсом [Java](https://ru.wikipedia.org/wiki/Java), реализация которого расширяет функциональные возможности [сервера](https://ru.wikipedia.org/wiki/%D0%A1%D0%B5%D1%80%D0%B2%D0%B5%D1%80_(%D0%BF%D1%80%D0%B8%D0%BB%D0%BE%D0%B6%D0%B5%D0%BD%D0%B8%D0%B5)). Сервлет взаимодействует с клиентами посредством принципа запрос-ответ.

Жизненный цикл сервлета состоит из следующих шагов:

1. В случае отсутствия сервлета в контейнере.
   1. Класс сервлета загружается контейнером.
   2. Контейнер создает экземпляр класса сервлета.
   3. Контейнер вызывает метод init(). Этот метод инициализирует сервлет и вызывается в первую очередь, до того, как сервлет сможет обслуживать запросы. За весь жизненный цикл метод init() вызывается только один раз.
2. Обслуживание клиентского запроса. Каждый запрос обрабатывается в своем отдельном потоке. Контейнер вызывает метод service() для каждого запроса. Этот метод определяет тип пришедшего запроса и распределяет его в соответствующий этому типу метод для обработки запроса. Разработчик сервлета должен предоставить реализацию для этих методов. Если поступил запрос, метод для которого не реализован, вызывается метод родительского класса и обычно завершается возвращением ошибки инициатору запроса.
3. В случае если контейнеру необходимо удалить сервлет, он вызывает метод destroy(), который снимает сервлет из эксплуатации. Подобно методу init(), этот метод тоже вызывается единожды за весь цикл сервлета.

сервлет контейнера - прочитать запрос клиента, расшифровать его и, в соответствиии с расшифровкой, передать работу сервлету, отвечающему за этот тип запрашиваемой информации.

### Интерфейс Servlet

Объединяет все эти модули то, что они сквозным образом связанны между собою с помощью интерфейса javax.servlet.Servlet

Посмотрим на этот интерфейс. В нём указано всего 5 методов:

public void init(ServletConfig config) throws ServletException

Этот метод вызывается, чтобы проинформировать сервлет о том, что он включён как модуль для обслуживания запросов клиента. Параметр config разделяет интерфейс javax.servlet.ServletConfig, несущий информацию об окружении сервера, имени сервлета, начальных параметрах и прочих плюшках. Об интерфейсе javax.servlet.ServletConfig будет рассказано чуть далее. Предполагается, что после вызова этой функции, сервлет аккуратно сохранит этот config у себя в переменной и будет выдавать его с помощью другого метода:

publicServletConfiggetServletConfig()

Получив системную информацию с помощью "getServletConfig()", сервер может захотеть узнать имя автора, дату создания, прочую информацию о сервлете, что и достигается вызовом

publicStringgetServletInfo()

Чтобы обработать запрос и получить результат его обработки, используется функция

public void service(ServletRequest request, ServletResponse response)

throwsServletException, java.io.IOException

В этой функции коду, который будет обрабатывать данные, передаются два инструмента: один - для получения данных от сервера, другой - для отправки результата работы сервлета. Соответственно это параметры request и response, разделяющие интерфейсы javax.servlet.ServletRequest и javax.servlet.ServletResponse Вся работа с данными ведётся именно через эти интерфейсы, так что далее поговорим о них подробнее.

После того, как сервер перестал нуждаться в этом модуле вызывается метод

publicvoiddestroy()

который и завершает все операции с объектом сервлета.

### Интерфейс ServletConfig

4 метода, имена которых говорят сами за себя, составляют суть интерфейса javax.servlet.ServletConfig:

public String getServletName()

public ServletContext getServletContext()

public String getInitParameter(String name)

public java.util.Enumeration getInitParameterNames()

Думаю, назначение всех функция понятно, кроме

publicServletContextgetServletContext()

Этот метод возвращает ссылку на очень полезный инструмент для работы с сервером:

### Сервлеты и многопоточность (multithreading)

Контейнер сервлетов имеет пул нитей (thread), которые он будет диспетчеризировать для обработки запросов клиентов. Они вполне подходят для случая, когда два клиентских запроса поступают одновременно и должны одновременно обработаться вашим методом service( ). Поэтому метод service( ) должен быть выполнен безопасным для многопоточности способом. Любой доступ к общим ресурсам (файлам, базам данных) должен быть гарантировано использоваться с ключевым словом synchronized. Один запрос 1 поток, обращаются к одному и тому же экземпляру сервлета, так как сервлет - **Singleton**. Либо много экземпляров но тогда кластер серверов(деприкейт).

## JSP

 По сути JSP при первом обращении преобразуется в сервлет и работает уже как сервлет.

JavaServerPages (JSP) позволяют вам отделить динамическую часть ваших страниц от статического HTML. Вы, как обычно, пишете обычный код в HTML, используя для этого любую программу для создания Web страниц. Затем вы заключаете динамическую часть кода в специальные таги, большинство которых начинаются с "<%" и завершаются "%>".

Код JSP-страницы транслируется в Java-код сервлета с помощью компилятора JSP-страниц [Jasper](https://ru.wikipedia.org/wiki/Apache_Tomcat#Jasper)

Структура таких страниц может состоять из пяти конструкций: [HTML](https://ru.wikipedia.org/wiki/HTML), комментарии, скриптовые элементы, директивы и действия.

*Элементы скриптов* позволяют вам указать код на языке Java, который впоследствии станет частью в конечный сервлет,*директивы* дадут вам возможность управлять всей структурой сервлета, а *действия* служат для задания существующих используемых компонентов, а также для контроля поведением движка JSP.

### **Скриптовые элементы**

Спецификация JSP различает три типа скриптовых элементов:

* Объявления <%! одна или несколько деклараций %>
* Выражения <%= одно выражение %>
* Скриплеты <% скриплет %>

Объявления JSP позволят вам задавать переменные, методы, внутренние классы и так далее.

<%!**private**int accessCount =0;%>

Выражения JSP применяются для того, чтобы вставить значения Java непосредственно в вывод

Текущее время:<%=**new** java.util.Date()%>

Скриплеты JSP дают возможность вставить любой код в метод сервлета, который будет создан при обработке страницы, позволяя использовать большинство конструкций Java. Скриплеты также имеют доступ к тем же заранее определённым переменным, что и выражения. Поэтому, например, для вывода значения на страницу необходимо использовать заранее определённую переменную out.

<%String queryData = request.getQueryString();

out.println("Дополнительные данные запроса: "+ queryData);

%>

### **Директивы JSP**

JSP страница может послать сообщение соответствующему контейнеру с указаниями действий, которые необходимо провести.

<%@ page import=”java.util.\*, com.myclasses.\*” buffer=”15kb” %>

### **Действия**

Действия JSP используют конструкции с синтаксисом [XML](https://ru.wikipedia.org/wiki/XML) для управления работой движка сервлета. Вы можете динамически подключать файл, многократно использовать компоненты [JavaBeans](https://ru.wikipedia.org/wiki/JavaBeans), направить пользователя на другую страницу или сгенерировать [HTML](https://ru.wikipedia.org/wiki/HTML) для [Javaplugin](https://ru.wikipedia.org/w/index.php?title=Java_plugin&action=edit&redlink=1" \o "Java plugin (страница отсутствует)).

* ***jsp:declaration*** — Объявление, аналогичен тегу <%! … %>;
* ***jsp:scriptlet*** — Скриптлет, аналогичен тегу <% … %>;
* ***jsp:expression*** — Выражение, аналогичен тегу <%= … %>;
* ***jsp:text*** — Вывод текста;

<jsp:useBean id="имя" class="пакет.class"/>

Появилась JSTL – JSPStandardTagLibrary – библиотека стандартных тэгов JSP, в которой достаточно большое количество очень удобных тэгов, позволяющих работать с данными — циклы, условия и прочая.

По сути это упрощение конструкций JSP, которого хотелось достичь.

Давайте рассмотрим несложный пример того, как можно написать и задействовать свой собственный тэг. Для этого нам потребуется три шага:

1. Написать TLD-файл (Tag Library Definition) – файл описателя библиотеки тэгов.
2. Написать класс для реализации самого тэга
3. Исправить HTML-файл

Итак. Файл TLD имеет расширение .tld и обычно развертывается внутри каталога WEB-INF в каталоге вашего приложения. Хотя можно разместить и в другом. Если у вас много таких файлов, то для них можно предусмотреть отдельный каталог. Вот наш вариант hello.tld.

**ExpressionLanguage** (EL) — скриптовый язык выражений, который позволяет получить доступ к [Java](https://ru.wikipedia.org/wiki/Java) компонентам ([JavaBeans](https://ru.wikipedia.org/wiki/JavaBeans" \o "JavaBeans)) из [JSP](https://ru.wikipedia.org/wiki/JavaServer_Pages).

## XML

HTML и XML похожи на первый взгляд

Основное различие в том, что XML – это метаязык, то есть он предназначен для определения других языков.

XML документ всегда представляет собой дерево

Поэтому в XML деревья представляются гораздо лучше, чем в реляционной модели хранения данных

* Альтернатива реляционной модели хранения данных
* Мультиплатформенный формат для передачи данных
* Отделение семантики данных от их представления
* XML чувствителен к регистру
* В XML нельзя «опускать» закрывающие тэги
* В XML часто встречаются тэги, одновременно открывающие и закрывающие **<img src="coffeecup.png"/>**
* В XML значения атрибутов должны быть заключены в кавычки
* В XML все атрибуты должны иметь значения

XML документ является Valid, если

* + Была проведена его валидация с использованием некой схемы, описывающей допустимую структуру тегов, типы данных и прочее. При этом отклонений от схемы выявлено не было.
  + Распространенные схемы: DTD, XSD(Полная поддержка пространств имен, можно подключать другие схемы), RELAX NG
  + Иногда XML содержит эту схему в себе.

XPath

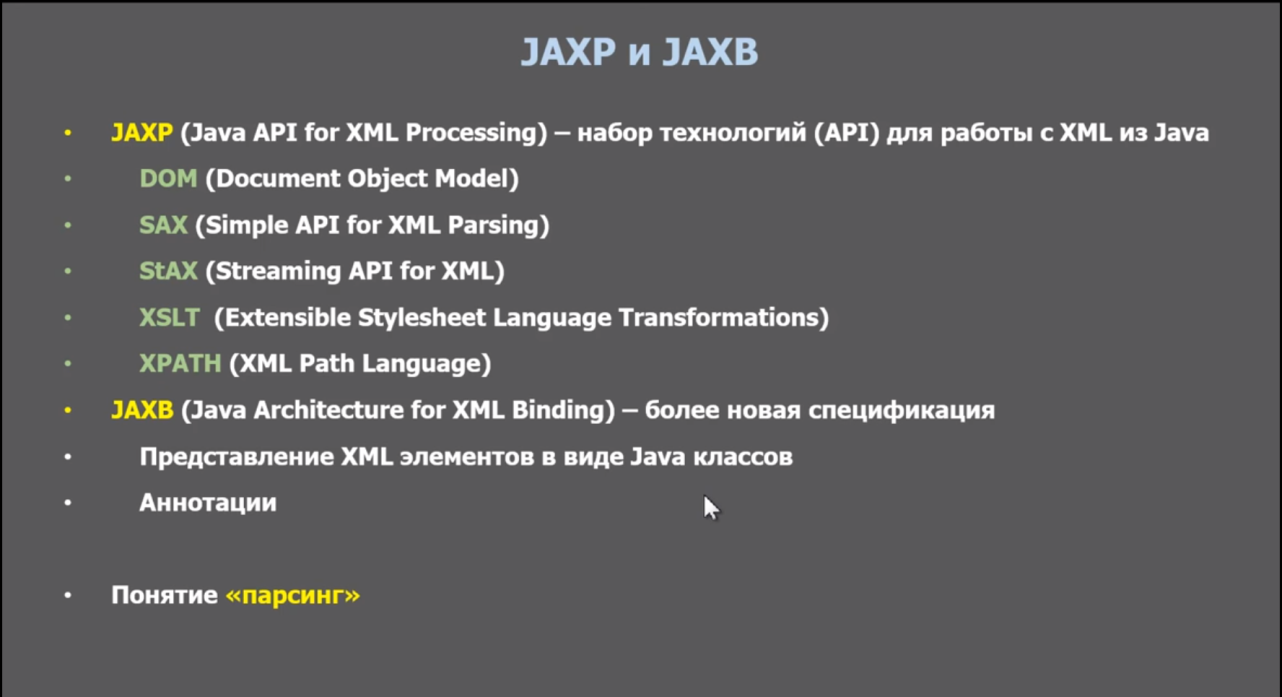
* Реализует навигацию по DOM в XML-документе
* Выполняет для XML ту же роль, что и SQL для реляционных баз данных
* Входит в состав языка запросов Xquery
* XPath-пути похожи на пути файловой системы
* Уязвим для Xpath-иньекций :
  + Не обеспечивает разграничения прав доступа
  + Можно одним запросом вытянуть все данные
  + XPath-иньекции – относительно новый способ атак

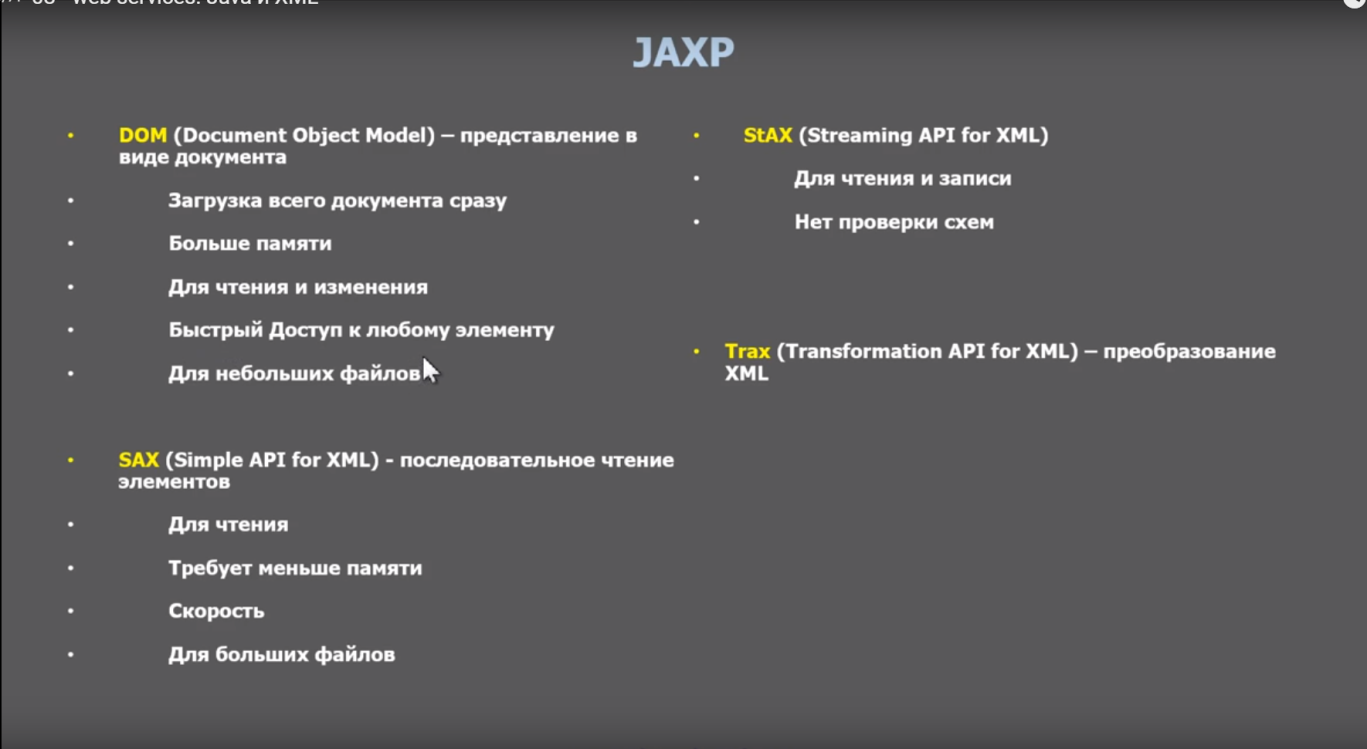
StAX

* StAX по принципу использования во многом похож на SAX.
* Отличие в том, что SAX основан на шаблоне Observer (Listener), в то время как использование StAX напоминает работу с итератором

JAXB

* Гибкий API для маппинга между XML-структурами и классами Java
* Можно создавать как классы по XML-схеме, так и схему по Java классам





## JDBC

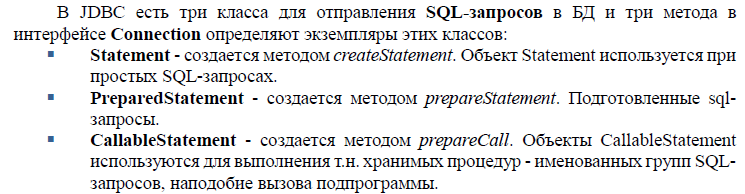
JDBC – это интерфейс позволяющий JAVA рабоать с раличными СУБД.

Типы драйверов:

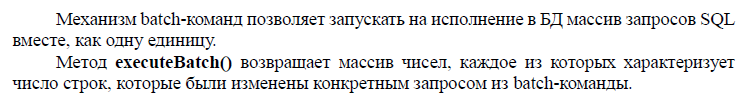
* 1. Мост JDBC-ODBC: Транслирует JDBC в ODBC и для взаимодействия с базой данных использует драйвер ODBC. Компания Sun включила в состав JDK один такой драйвер — мост JDBC/ODBC. Однако для его использования требуется соответствующим образом установить и конфигурировать ODBC-драйвер. В первом выпуске JDBC этот мост предполагалось использовать только для тестирования, а не для рабочего применения. В настоящее время уже имеется большое количество более удачных драйверов.
  2. Нативный API/частичный JAVA драйвер: Создается преимущественно на языке Java и частично на собственном языке программирования, который используется для взаимодействия с клиентским API базы данных. Для использования такого драйвера нужно помимо библиотеки Java установить специфический для данной платформы код.
  3. Сетевой протокол: Создается только на основе библиотеки Java, в которой используется независимый от базы данных протокол взаимодействия сервера и базы. Этот протокол позволяет транслировать запросы в соответствии со спецификой конкретной базы. Если код, зависящий от базы данных, находится только на сервере, развертывание существенно упрощается.
  4. Нативный API/чистый JAVA драйвер: Представляет собой библиотеку Java, которая транслирует JDBC-запросы непосредственно в протокол конкретной базы данных.

1. Мы загружаем драйвер, и он автоматически зарегистрируется себя для использования вместе с JDBC.
2. Для соединения с базой данных используют интерфейс Connection:

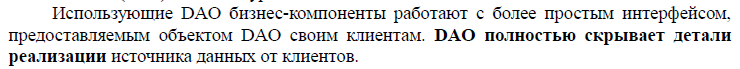
**Connection** dbh = DriverManager.getConnection(url, user, passwd);



1. Обрабатываем ResaultSet.

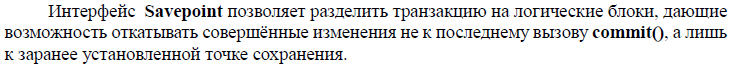


DatabaseConnectionPool (dbcp) — Он подразумевает, что в нашем распоряжении имеется некоторый набор («пул») соединений к базе анных. Когда новый пользователь запрашивает доступ к БД, ему выдаётся уже открытое соединение из этого пула. Если все открытые соединения уже заняты, создаётся новое. Как только пользователь освобождает одно из уже существующих соединений, оно становится доступно для других пользователей. Если соединение долго не используется, оно закрывается.



Вершиной иерархии **DAO** является абстрактный класс или интерфейс с описанием общих методов, которые будут использоваться при взаимодействии с базой данных. Как правило, это методы поиска, удаление по ключу, обновление и т.д.

Транзакция состоит из одного или более выражений, которые после выполнения либо все фиксируются (commit), либо все откатываются назад (rollback). При вызове метода commit или rollback текущая транзацкия заканчивается и начинается другая.



Плоские, или традиционные, транзакции, характеризуются четырьмя классическими свойствами: атомарности, согласованности, изолированности, долговечности (прочности) — ACID (Atomicity, Consistency, Isolation, Durability). Иногда традиционные транзакции называют ACID-транзакциями. Упомянутые выше свойства означают следующее:

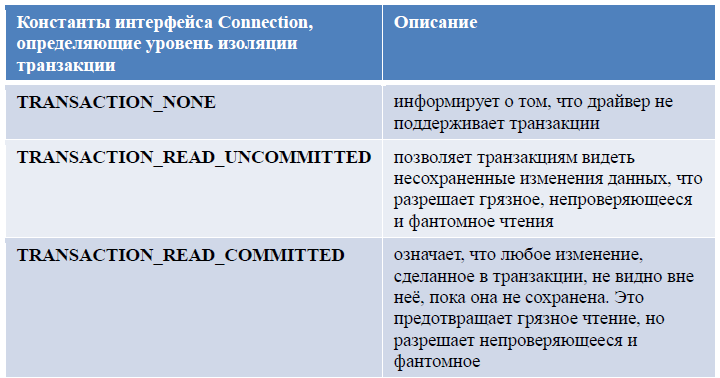
* *Свойство атомарности*(Atomicity) выражается в том, что транзакция должна быть выполнена в целом или не выполнена вовсе.
* *Свойство согласованности*(Consistency) гарантирует, что по мере выполнения транзакций данные переходят из одного согласованного состояния в другое — транзакция не разрушает взаимной согласованности данных. Согласованность является более широким понятием. Например, в банковской системе может существовать требование равенства суммы, списываемой с одного счёта, сумме, зачисляемой на другой. Это бизнес-правило и оно не может быть гарантировано только проверками целостности, его должны соблюсти программисты при написании кода транзакций. Если какая-либо транзакция произведёт списание, но не произведёт зачисления, то система останется в некорректном состоянии и свойство согласованности будет нарушено.
* *Свойство изолированности*(Isolation) означает, что конкурирующие за доступ к базе данных транзакции физически обрабатываются последовательно, изолированно друг от друга, но для пользователей это выглядит так, как будто они выполняются параллельно.
* *Свойство долговечности*(Durability) трактуется следующим образом: если транзакция завершена успешно, то те изменения в данных, которые были ею произведены, не могут быть потеряны ни при каких обстоятельствах (даже в случае последующих ошибок).

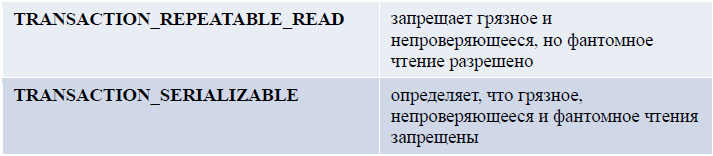
При [параллельном выполнении транзакций](https://ru.wikipedia.org/wiki/%D0%9F%D0%B0%D1%80%D0%B0%D0%BB%D0%BB%D0%B5%D0%BB%D1%8C%D0%BD%D1%8B%D0%B5_%D0%B2%D1%8B%D1%87%D0%B8%D1%81%D0%BB%D0%B8%D1%82%D0%B5%D0%BB%D1%8C%D0%BD%D1%8B%D0%B5_%D1%81%D0%B8%D1%81%D1%82%D0%B5%D0%BC%D1%8B) возможны следующие проблемы:

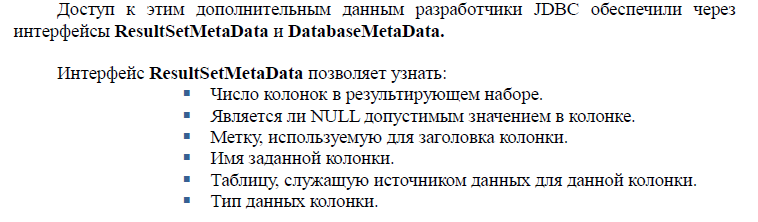
* потерянное обновление ([англ.](https://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA" \o "Английский язык) *lost update*) — при одновременном изменении одного блока данных разными транзакциями теряются все изменения, кроме последнего;
* «грязное» чтение ([англ.](https://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA" \o "Английский язык) *dirty read*) — чтение данных, добавленных или изменённых транзакцией, которая впоследствии не подтвердится (откатится);
* неповторяющееся чтение ([англ.](https://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA" \o "Английский язык) *non-repeatable read*) — при повторном чтении в рамках одной транзакции ранее прочитанные данные оказываются изменёнными;
* фантомное чтение ([англ.](https://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA" \o "Английский язык) *phantom reads*) — одна транзакция в ходе своего выполнения несколько раз выбирает множество строк по одним и тем же критериям. Другая транзакция в интервалах между этими выборками добавляет или удаляет строки или изменяет столбцы некоторых строк, используемых в критериях выборки первой транзакции, и успешно заканчивается. В результате получится, что одни и те же выборки в первой транзакции дают разные множества строк.

<https://bourabai.ru/dbt/dbms/1101.htm>

[https://ru.wikipedia.org/wiki/%D0%A3%D1%80%D0%BE%D0%B2%D0%B5%D0%BD%D1%8C\_%D0%B8%D0%B7%D0%BE%D0%BB%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%BD%D0%BE%D1%81%D1%82%D0%B8\_%D1%82%D1%80%D0%B0%D0%BD%D0%B7%D0%B0%D0%BA%D1%86%D0%B8%D0%B9#%D0%9F%D1%80%D0%BE%D0%B1%D0%BB%D0%B5%D0%BC%D1%8B\_%D0%BF%D0%B0%D1%80%D0%B0%D0%BB%D0%BB%D0%B5%D0%BB%D1%8C%D0%BD%D0%BE%D0%B3%D0%BE\_%D0%B4%D0%BE%D1%81%D1%82%D1%83%D0%BF%D0%B0\_%D1%81\_%D0%B8%D1%81%D0%BF%D0%BE%D0%BB%D1%8C%D0%B7%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5%D0%BC\_%D1%82%D1%80%D0%B0%D0%BD%D0%B7%D0%B0%D0%BA%D1%86%D0%B8%D0%B9](https://ru.wikipedia.org/wiki/%D0%A3%D1%80%D0%BE%D0%B2%D0%B5%D0%BD%D1%8C_%D0%B8%D0%B7%D0%BE%D0%BB%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%BD%D0%BE%D1%81%D1%82%D0%B8_%D1%82%D1%80%D0%B0%D0%BD%D0%B7%D0%B0%D0%BA%D1%86%D0%B8%D0%B9#%D0%9F%D1%80%D0%BE%D0%B1%D0%BB%D0%B5%D0%BC%D1%8B_%D0%BF%D0%B0%D1%80%D0%B0%D0%BB%D0%BB%D0%B5%D0%BB%D1%8C%D0%BD%D0%BE%D0%B3%D0%BE_%D0%B4%D0%BE%D1%81%D1%82%D1%83%D0%BF%D0%B0_%D1%81_%D0%B8%D1%81%D0%BF%D0%BE%D0%BB%D1%8C%D0%B7%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B)







## KAFKA

<https://habr.com/ru/company/piter/blog/352978/>

<https://habr.com/ru/post/354486/>

## JMS

<http://dit.isuct.ru/Publish_RUP/tech.j2ee/guidances/concepts/java_messaging_service_jms_84F49452.html>

<http://java-online.ru/javax-jms.xhtml>

<https://jsehelper.blogspot.com/2016/12/jms.html>

## REST

<https://habr.com/ru/post/259067/>

## JPA

<https://habr.com/ru/post/265061/>

## Mockito

<https://habr.com/ru/post/444982/>

## Spring

Spring это легковесный opensourceJ2EEFramework, разработка которого началась в феврале 2003 года, альтернатива и замена модели EnterpriseJavaBean. Позволяет создавать множество приложений без привязки какой то конкретной сфере.

В основе всего фреймворка Spring лежит концепция InversionofControl / DependencyInjection —[“Обращение контроля”](http://ru.wikipedia.org/wiki/%D0%9E%D0%B1%D1%80%D0%B0%D1%89%D0%B5%D0%BD%D0%B8%D0%B5_%D0%BA%D0%BE%D0%BD%D1%82%D1%80%D0%BE%D0%BB%D1%8F), облегчает переносимость модулей.

IOC – принцип где сами объекты не знают как и с кем будут работать. Контроль лежит на контейнере. (Основан на ServiceLocator специальный локатор кторый определяет с какми объектом мы будем работать.)

**DI**– способ внедрения объектов через конструкторы, сетеры, методы.(шаблон для проектирования и реализации IOC)

* Основная цель – упрощение разработки любых приложений на java, разгрузка кода.
* Так как основа IOC програмный кодстановится проще, связь между объектами слабее.
* Каждый объект занимается своим делом и используются POJO объекты.
* Ипользование принципов ООП на полную мощность за счет абстарагирования и использования интрефейсов.
* Доп логика из вне на основе AOP, много готовых модулей (работа с БД, логироввание, транзацкии и тп.)
* Упрощенная конфигурация приложения за счет настроечного XML.

IOC

* Сначала создаем обстракцию, без указания объектов
* Затем подставляем реализацию.
* Подключаем доп аспекты, если нужно

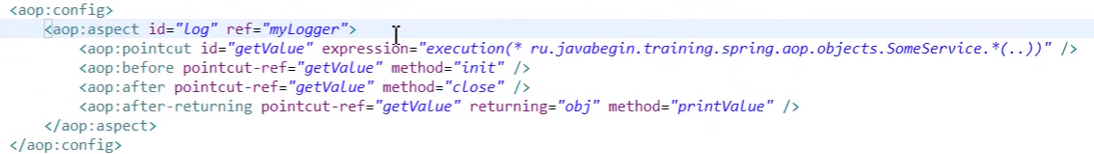
Контейнер

* Управляет зависимостями
* Связывает объекты между собой
* Управляет их жизненым циклом
* **DependencyInjection** – в объект внедряется ссылка на другой объект

AOP

* Разделение основного функционала и дополнительного – без перемешивания их между собой
* «Сквозной функционал» Кеширование, логирование, транзакции, безопастность и пр.(иначе бы пришлось дописывать методы в каждый метод бизнес логики.)
* Аспект – функциональность, напрямую к бизнес логике, можно внедрять по необходимости.
* Аспекты можно использовать в любых проектах
* Аспект это комбинация совета и срезов.  представляет собой какой либо сквозной функционал, аспект логирования транзакций и тп.
* Advice(Совет) - когда и что именно нужновыполнить в точке соединения.
* JoinPoint (Точка соединения) в каком месте при каком действии нужно выпорлнить эдвайс(в спринге только execution)
* PointCut (Срез точек соединеня) область применения или масив точек соединення (сужение).

Аспект –набор эдвайсов для каждого из которых определены точки соединеня и область применения.



Типы Эдвайсов

* Before до выполнения точки сопряжения.
* After после выполнения
* Afterreturning в случае успешного выполнения
* After throwing в случае exception
* Around advice до и после

В использовании AOP объект автоматически оборачивается в прокси шаблон и работет уже через него. Поверх объекта, логирование например.

Проект SpringSecurity 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.

Weaving (Вплетение) процесс добавления аспектов к объектам во время загрузки выполнения и пр. В СПРИНГЕ во время RunTime(библиотека CGLIP)

IOC контейнер это  простой механизм для предоставления зависимостей компонента и управления этими зависимостями на протяжении всего их жизненного цикла.

Термин **бин** в Spring используется для ссылки на **любой компонент, управляемый контейнером**.

Для интеграции Hibernate в Spring необходимо подключить зависимости, а так же настроить файл конфигурации Spring.

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().

### @Autowired?

Если же опустить все умные слова, то полезную функциональность аннотации @Autowired можно описать следующим образом... Используя эту аннотацию, не нужно заботиться о том, как лучше всего передать классу или bean'у экземпляр другого bean'a. Фреймворк Spring сам найдет нужный bean и подставит его значение в свойство, которое отмечено аннотацией @Autowired. Существует четыре вида связывания в спринг:

* autowire**byName,**
* autowire**byType,**
* autowire **by constructor,**
* autowiring by**@Autowired and @Qualifier annotations** можно отметить конкретного кандидата для автозаполнения если кандидатов несколько.

Существует несколько способов работы с классами в Spring:

XML файл;

Анотации;

Сканирование(Component-scan) Можно использовать внутри кода аннотации @Component, @Service, @Repository, @Controller для указания классов в качестве спринг бинов.

### значение конфигурационный файл SpringBean

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

### Какие вы знаете различные scope у SpringBean?

В Spring предусмотрены различные области времени действия бинов:

1. **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.
2. **prototype** — создается новый экземпляр при каждом запросе**.**
3. **request** —  аналогичен prototype, но название служит пояснением к использованию бина в веб приложении**.**Создается новый экземпляр при каждом HTTP request.
4. **session** — новый бин создается в контейнере при каждой новой HTTP сессии.
5. **global-session**: используется для создания глобальных бинов на уровне сессии для Portlet приложений.

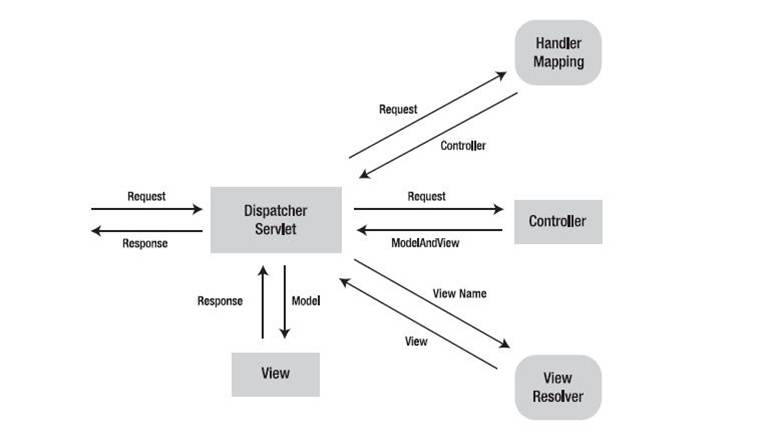
AspectJ де-факто является стандартом реализации АОП. Реализация АОП от Spring имеет некоторые отличия:

* SpringAOP немного проще, т.к. нет необходимости следить за процессом связывания.
* SpringAOP поддерживает аннотации AspectJ, таким образом мы можем работать в спринг проекте похожим образом с AspectJ проектом.
* SpringAOP поддерживает только proxy-based АОП и может использовать только один тип точек соединения — MethodInvocation. AspectJ поддерживает все виды точек соединения.
* Недостатком SpringAOP является работа только со своими бинами, которые существуют в SpringContext.

Рассмотрим один из самых главных разделов фреймворка Spring — SpringMVC.

Фреймворк SpringWebmodel-view-controller (MVC) или по нашему модель-представление-контроллер построен вокругDispatcherServlet, который распределяет запросы по обработчикам. В нём настраивается мэппинг запросов, локали, временные зоны и многое другое. Обработчик по умолчанию строится на аннотациях @Controller и @RequestMapping, которые предоставляют широкий набор гибких методов для обработки запросов. После версии Spring 3.0. механизм @Controller так же позволяет создавать RESTful веб сайты и приложения, используя аннотацию @PathVariable и другие возможности.

DispatcherServlet  выражение из шаблона проектирования «FrontController».

DispatcherServlet — это обычный сервлет (наследуется от базового класса HttpServlet) и его также необходимо описывать в web.xml

* Вначале DispatcherServlet (диспетчер сервлетов) получает запрос, далее он смотрит свои настройки, чтобы понять какой контроллер использовать (на рисунке HandlerMapping).
* После получения имени контроллера запрос передается в него (на рисунке Controller). В контроллере происходит обработка запроса и обратно посылается ModelAndView (модель — сами данные; view (представление) — как эти данные отображать).
* DispatcherServlet на основании полученного ModelAndView ищет какое представление ему использовать (ViewResolver) и получает в ответе имя представления View
* В представление передаются данные (model) и обратно, если необходимо, посылается ответ от представления.

Ещё раз основы шаблона mvc:

* Model — представление данных, сами данные
* View — представление, вид, отображение
* Controller — управление, связь между моделью и видом.

**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.

**ViewResolver** — распознаватель представлений. Интерфейс ViewResolver в SpringMVC (из пакета org.springframework.web.servlet) поддерживает распознавание представлений на основе логического имени, возвращаемого контроллером.

Интерфейс MultipartResolver используется для загрузки файлов

В SpringMVC интерфейс HandlerExceptionResolver (из пакета org.springframework.web.servlet) предназначен для работы с непредвиденными исключениями, возникающими во время выполнения обработчиков. По умолчанию DispatcherServletрегистрирует класс DefaultHandlerExceptionResolver (из пакета org.springframework.web.servlet.mvc.support). Этот распознаватель обрабатывает определенные стандартные исключения SpringMVC, 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.

### Что вы знаете о SpringMVCInterceptor и как он используется?

Перехватчики в Spring (SpringInterceptor) являются аналогом ServletFilter 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 handlermethod. 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 handlermethod, 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.

### Как использовать TomcatJNDIDataSource в веб-приложении Spring?

Для использования контейнера сервлетов настроенного на использование JNDIDataSource, необходимо задать соответствующее свойство в файле конфигурации и затем внедрять его как зависимость. Далее мы можем использовать объект JdbcTemplate для выполнения операций с базами данных.

### 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 настройкуtransactionManager для DataSource.

### SpringJdbcTemplate

Объявляют бин DataSourse из стандартного пакета спринга DriverManagerDataSource. Уазываем проперти даные для подключения.

Затем создание JDBC объекта с передачаей конструкотору datasource. И спомощью него делаем запросы.

### Как использовать TomcatJNDIDataSource в веб-приложении Spring?

Для использования контейнера сервлетов настроенного на использование JNDIDataSource, необходимо задать соответствующее свойство в файле конфигурации и затем внедрять его как зависимость. Далее мы можем использовать объект JdbcTemplate для выполнения операций с базами данных.

### Transaction Spring

<http://akorsa.ru/2016/08/kak-na-samom-dele-rabotaet-transactional-spring/>

## Hibernate

В двух словах ORM — это отображение объектов какого-либо объектно-ориентированного языка в структуры реляционных баз данных. Именно объектов, таких, какие они есть, со всеми полями, значениями, отношениями м/у друг другом.  
ORM-решением для языка [Java](http://ru.wikipedia.org/wiki/Java), является технология [Hibernate](http://ru.wikipedia.org/wiki/Hibernate_(%D0%B1%D0%B8%D0%B1%D0%BB%D0%B8%D0%BE%D1%82%D0%B5%D0%BA%D0%B0)), которая не только заботится о связи Java классов с таблицами базы данных (и типов данных Java в типы данных [SQL](http://ru.wikipedia.org/wiki/Sql)), но также предоставляет средства для автоматического построения запросов и извлечения данных и может значительно уменьшить время разработки, которое обычно тратится на ручное написание SQL и [JDBC](http://ru.wikipedia.org/wiki/Jdbc)кода. Hibernate генерирует SQL вызовы и освобождает разработчика от ручной обработки результирующего набора данных и конвертации объектов, сохраняя приложение портируемым во все SQL базы данных.

Hibernate является одним из самых востребованных ORM фреймворков для Java. И вот почему:

1. Hibernate устраняет множество спагетти кода (повторяющегося), который постоянно преследует разработчика при работе с JDBC. Скрывает от разработчика множество кода, необходимого для управления ресурсами и позволяет сосредоточиться на бизнес логике.
2. Hibernate поддерживает XML так же как и JPA аннотации, что позволяет сделать реализацию кода независимой.
3. Hibernate 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).
4. Hibernate — широко распространенный opensource проект. Благодаря этому доступны тысячи открытых статей, примеров, а так же документации по использованию фреймворка.
5. Hibernate легко интегрируется с другими JavaEE фреймворками, например, SpringFramework поддерживает встроенную интеграцию с Hibernate.
6. Hibernate поддерживает ленивую инициализацию используя proxy объекты и выполняет запросы к базе данных только по необходимости.
7. Hibernate поддерживает разные уровни cache, а следовательно может повысить производительность.
8. Важно, что Hibernate может использовать чистый SQL, а значит поддерживает возможность оптимизации запросов и работы с любым сторонним вендором БД и его фичами.

#### Каковы преимущества Hibernate над JDBC?

Hibernate имеет ряд преимуществ перед JDBCAPI:

1. Hibernate удаляет множество повторяющегося кода из JDBCAPI, а следовательно его легче читать, писать и поддерживать.
2. Hibernate поддерживает наследование, ассоциации и коллекции, что не доступно в JDBCAPI.
3. Hibernate неявно использует управление транзакциями. Большинство запросов нельзя выполнить вне транзакции. При использовании JDBCAPI для управления транзакциями нужно явно использовать commit и rollback.
4. JDBCAPI 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 блокиtry-catch.
5. HibernateQueryLanguage (HQL) более объектно ориентированный и близкий к Java язык программирования, чем SQL в JDBC.
6. Hibernate поддерживает кэширование, а запросы JDBC — нет, что может понизить производительность.
7. Конфигурация Hibernate позволяет использовать JDBC вроде соединения по типу JNDIDataSource для пула соединений. Это важная фича для энтерпрайз приложений, которая полностью отсутствует в JDBCAPI.
8. 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.

#### 4. Назовите некоторые важные интерфейсы Hibernate.

1. **SessionFactory (org.hibernate.SessionFactory)** — неизменяемый потокобезопасный объект с компилированным маппингом для одной базы данных. Необходимо инициализировать SessionFactory всего один раз. Экземпляр SessionFactory используется для получения объектов Session, которые используются для операций с базами данных.
2. **Session (org.hibernate.Session)** — однопоточный короткоживущий объект, который предоставляет связь между объектами приложения и базой данных. Он оборачивает JDBC java.sql.Connection и работает как фабрика для org.hibernate.Transaction. Разработчик должен открывать сессию по необходимости и закрывать ее сразу после использования. Экземпляр Session является интерфейсом между кодом в java приложении и hibernateframework и предоставляет методы для операций CRUD.
3. **Transaction (org.hibernate.Transaction)** — однопоточный короткоживущий объект, используемый для атомарных операций. Это абстракция приложения от основных JDBC или JTA транзакций. org.hibernate.Session может занимать несколькоorg.hibernate.Transaction в определенных случаях.

#### 5. 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.

#### 6. Что такое Hibernate mapping file?

Файл отображения (mappingfile) 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).

#### 7. 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:

1. **javax.persistence.Entity**: используется для указания класса как entitybean.
2. **javax.persistence.Table**: используется для определения имени таблицы из БД, которая будет отображаться на entitybean.
3. **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.
4. **javax.persistence.Id**: определяет primary key в entity bean.
5. **javax.persistence.EmbeddedId**: используется для определения составного ключа в бине.
6. **javax.persistence.Column**: определяет имя колонки из таблицы в базе данных.
7. **javax.persistence.GeneratedValue**: задает стратегию создания основных ключей. Используется в сочетании сjavax.persistence.GenerationType enum.
8. **javax.persistence.OneToOne**: задает связь один-к-одному между двумя сущностными бинами. Соответственно есть другие аннотации OneToMany, ManyToOne и ManyToMany.
9. **org.hibernate.annotations.Cascade**: определяет каскадную связь между двумя entity бинами. Используется в связке сorg.hibernate.annotations.CascadeType.
10. **javax.persistence.PrimaryKeyJoinColumn**: определяет внешний ключ для свойства. Используется вместе с org.hibernate.annotations.GenericGenerator и org.hibernate.annotations.Parameter.

#### 8. Что вы знаете о HibernateSessionFactory и как его сконфигурировать?

SessionFactory является фабрикой классов и используется для получения объектов session. SessionFactory отвечает за считывание параметров конфигурации Hibernate и подключение к базе данных. Обычно в приложении имеется только один экземпляр SessionFactory и потоки, обслуживающие клиентские запросы, получают экземпляры session с помощью объекта SessionFactory. Внутреннее состояние SessionFactory неизменно (immutable). Internal state (внутреннее состояние) включает в себя все метаданные об Object/ Relational Mapping и задается при создании SessionFactory.

SessionFactory также предоставляет методы для получения метаданных класса и статистики, вроде данных о втором уровне кэша, выполняемых запросах и т.д.

#### 9. Является ли HibernateSessionFactory потокобезоспансым?

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

#### 10. Как получить HibernateSession и что это такое?

Объект HibernateSession является связью между кодом java приложения и hibernate. Это основной интерфейс для выполнения операций с базой данных. Жизненный цикл объекта session связан с началом и окончанием транзакции. Этот объект предоставляет методы для CRUD (create, read, update, delete) операций для объекта персистентности. С помощью этого экземпляра можно выполнять HQL, SQL запросы и задавать критерии выборки.

#### 11. Является ли HibernateSession потокобезоспаным?

Объект HibernateSession не является потокобезопасным. Каждый поток должен иметь свой собственный объект Session и закрывать его по окончанию.

#### 12. В чем разница между openSession и getCurrentSession?

Hibernate SessionFactorygetCurrentSession() 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.

Метод Hibernate SessionFactory 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.

#### 13. Какая разница между методами HibernateSessionget() и load()?

Hibernatesession обладает различными методами для загрузки данных из базы данных. Наиболее часто используемые методы для этого — get() и load().

* get() загружает данные сразу при вызове, в то время как load() использует прокси объект и загружает данные только тогда, когда это требуется на самом деле. В этом плане load() имеет преимущество в плане ленивой загрузки данных.
* load() бросает исключение, когда данные не найдены. Поэтому его нужно использовать только при уверенности в существовании данных.
* Нужно использовать метод get(), если необходимо удостовериться в наличии данных в БД.

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

Hibernate использует кэширование, чтобы сделать наше приложение быстрее. Кэш Hibernate может быть очень полезным в получении высокой производительности приложения при правильном использовании. Идея кэширования заключается в сокращении количества запросов к базе данных.

Кэш первого уровня Hibernate связан с объектом Session. Кэш первого уровня у Hibernate включен по умолчанию 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 объекта сессии все кэшированные объекты будут потеряны.

#### 15. Как настроить кэш второго уровня в Hibernate с помощью EHCache?

EHCache является лучшим выбором для организации кэширования второго уровня в хибернейт. Для настройки второго уровня кэширования в хибернейт требуется выполнить несколько шагов.

* Добавить зависимость hibernate-ehcache в проект.
* Добавить несколько записей в конфигурационный файл Hibernate.
* Создать файл конфигурации EHCache.

16. Каковы существуют различные состояния у entitybean?

1. **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().
2. **Persistent**: когда объект связан с уникальной сессией он находится в состоянии persistent (персистентности). Любой экземпляр, возвращаемый методами get() или load() находится в состоянии persistent.
3. **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().

#### 17. Как используется вызов метода HibernateSessionmerge()?

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.

#### 18. В чем разница между Hibernatesave(), 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().

#### 19. Что произойдет, если будет отсутствовать конструктор без аргументов у EntityBean?

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.

#### 20. В чем разница между sortedcollection и orderedcollection? Какая из них лучше?

При использовании алгоритмов сортировки из CollectionAPI для сортировки коллекции, то он вызывает отсортированный список (**sortedlist**). 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 CriteriaAPI и команду orderbyдля получения отсортированного списка (**orderedlist**). Orderedlist является лучшим выбором к sortedlist, т.к. он использует сортировку на уровне базы данных. Она быстрее и не может привести к утечке памяти.

21. Какие типы коллекций в Hibernate вы знаете?

1. Bag; Set; List ; Array; Map

#### 22. 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 «joinfetch«, позволяющая загружать данные немедленно (не lazy).
* Использовать чистый SQL запрос с командой join.

#### 23. Почему мы не должны делать Entityclass как final?

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

#### 24. Что вы знаете о HQL и каковы его преимущества?

HibernateFramework поставляется с мощным объектно-ориентированным языком запросов — HibernateQueryLanguage (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 ru.javastudy.MyClass отличен от  ru.javastudy.MyCLASS. Запросы HQL кэшируются (это как плюс так и минус).

Подробнее можете посмотреть в [Hibernate — HibernateQuery (HQL) примеры: SELECT, INSERT, UPDATE, DELETE](http://javastudy.ru/hibernate/hibernate-hql-examples/) и в общем разделе [Hibernate](http://javastudy.ru/frameworks/hibernate/).

#### 25. 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. QueryCache полезны только для часто выполняющихся запросов с повторяющимися параметрами. Для начала необходимо добавить эту запись в файле конфигурации Hibernate:Уже внутри кода приложения для запроса применяется метод setCacheable(true), как показано ниже:

26. Можем ли мы выполнить нативный запрос SQL (sqlnative) 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:

#### 27. 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.

#### 28. Что такое NamedSQLQuery?

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 NativeSQL. Создать именованный запрос можно с помощью JPA аннотаций @NamedQuery, @NamedNativeQuery или в конфигурационном файле отображения (mappingfiles).

[Hibernate — примеры именованных запросов NamedQuery](http://javastudy.ru/hibernate/hibernate-namedquery/).

#### 29. Каковы преимущества NamedSQLQuery?

Именованный запрос Hibernate позволяет собрать множество запросов в одном месте, а затем вызывать их в любом классе. Синтаксис NamedQuery проверяется при создании sessionfactory, что позволяет заметить ошибку на раннем этапе, а не при запущенном приложении и выполнении запроса. NamedQuery 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).

#### 30. Расскажите о преимуществах использования HibernateCriteriaAPI.

HibernateCriteriaAPI является более объектно-ориентированным для запросов, которые получают результат из базы данных. Для операций update, delete или других DDL манипуляций использовать CriteriaAPI нельзя. Критерии используются только для выборки из базы данных в более объектно-ориентированном стиле.

Вот некоторые области применения CriteriaAPI:

* CriteriaAPI поддерживает проекцию, которую мы можем использовать для агрегатных функций вроде sum(), min(), max() и т.д.
* CriteriaAPI может использовать ProjectionList для извлечения данных только из выбранных колонок.
* CriteriaAPI может быть использована для join запросов с помощью соединения нескольких таблиц, используя методы createAlias(), setFetchMode() и setProjection().
* CriteriaAPI поддерживает выборку результатов согласно условиям (ограничениям). Для этого используется метод add() с помощью которого добавляются ограничения (Restrictions).
* CriteriaAPI позволяет добавлять порядок (сортировку) к результату с помощью метода addOrder().

#### 31. Как логировать созданные HibernateSQL запросы в лог-файлы?

Для логирования запросов SQL добавьте в файл конфигурации Hibernate строчку: Отметьте, что это необходимо использовать на уровне Development или Testing и должно быть отключено в продакшн.

#### 32. Что вы знаете о Hibernate прокси и как это помогает в ленивой загрузке (lazyload)?

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.

#### 33. Как реализованы отношения в Hibernate?

Реализовать отношение one-to-one, one-to-many, many-to-many можно с помощью JPA аннотаций или конфигурирования xml файла. За примерами посетите раздел [Hibernate](http://javastudy.ru/frameworks/hibernate/).

#### 34. Как управлять транзакциями с помощью Hibernate?

Hibernate вообще не допускает большинство операций без использования транзакций. Поэтому после получения экземпляра session отSessionFactory 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.

#### 35. Что такое каскадные связи (обновления) и какие каскадные типы есть 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:

Обратите внимание, что есть некоторые различия между enumCascadeType в Hibernate и в JPA. Поэтому обращайте внимание какой пакет вы импортируете при использовании аннотации и константы типа. Наиболее часто используемые CascadeType перечисления описаны ниже.

1. **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.
2. **ALL**: Cascades save, delete, update, evict, lock, replicate, merge, persist. В общем — всё.
3. **SAVE\_UPDATE**: Cascades save и update. Доступно только для hibernate.
4. **DELETE**: передает в Hibernate native **DELETE** действие. Только для hibernate.
5. **DETATCH**, **MERGE**, **PERSIST**, **REFRESH** и **REMOVE** – для простых операций.
6. **LOCK**: передает в Hibernate native **LOCK** действие.
7. **REPLICATE**: передает в Hibernate native **REPLICATE** действие.

#### 36. 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.

Может быть полезно посетить [раздел Log4j2](http://javastudy.ru/tools/log4j/).

#### 37. Как использовать JNDIDataSource сервера приложений с HibernateFramework?

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 и использовать указания свойства JNDIDataSource:

38. 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.

## Other

Потоки

<https://habrahabr.ru/company/luxoft/blog/157273/>

<https://habrahabr.ru/post/132884/>

<http://www.duct-tape-architect.ru/?p=294>

<https://habrahabr.ru/post/116363/>

<https://www.ibm.com/developerworks/ru/library/l-java_universe_multithreading_tasks/index.html>

<https://tproger.ru/translations/java8-concurrency-tutorial-1/>

<https://tproger.ru/translations/java8-concurrency-tutorial-2/>

<https://javadevblog.com/java-lock-teoriya-i-primer-ispol-zovaniya-concurrency-lock.html>

<https://urvanov.ru/2016/05/27/java-8-%D0%BC%D0%BD%D0%BE%D0%B3%D0%BE%D0%BF%D0%BE%D1%82%D0%BE%D1%87%D0%BD%D0%BE%D1%81%D1%82%D1%8C/>

JMS & JNDI

<https://jsehelper.blogspot.ru/2016/12/jms.html>

<http://java-online.ru/jndi.xhtml>

<http://dit.isuct.ru/Publish_RUP/tech.j2ee/guidances/concepts/java_messaging_service_jms_84F49452.html>

Java 8

https://habrahabr.ru/post/216431/