1. Многопоточное программирование. Понятия процесс и поток. Методы передачи данных между потоками и процессами. Синхронизация потоков. Synchronized vs Lock.
2. Клиент-серверное приложение. Веб приложение. Front-end и Back-end программирование. Web-server. Http-server. Proxy-server. Статические и динамические ресурсы.
3. Сервер приложений IBM WAS. Apach Tomcat. Масштабируемость и кластеризация. Понятие кластера. Баласировка запросов.
4. Обзор JavaEE. Обзор компонентов стандарта: Servlets, JSP, EJB, JPA, JSTL, JSF, JNDI, JMS, JTA, JAAS
5. Понятие Servlet. Жизненный цикл сервлета. Методы HpptServelet. Пример реализация пользовательского сервлета.
6. Понятие Filter, Listener. Жизненный цикл Filter и Listener. Порядок вызова методов объекта, реализующий Filter и Listener. Примеры реализации Filter, Listener.
7. Регистрация Servlet, Filter, Listener при помощи аннотаций и в web.xml. Файл web.xml. Формат. Основные теги. Пример web.xml
8. Servler и JSP. Компиляция JSP в Servlet. Пример JSP и Servlet с одинаковой логикой. Перенаправление на JSP. Сравнение методов forward() и sendRedirect().
9. JSP. Скриплеты. Коментарии в jsp. HTML. Порядок генерации контента на основе jsp страницы. Пример jsp.
10.Session, request, cookies. Жизненные циклы. Атрибуты и параметры Request. Доступ из Servlet и jsp. Передача данных из сервлета в jsp. Пример.
11. Библиотеки тегов. JSP, JSTL. Пример использования. Порядок генерации java-кода из jsp со сложными тегами. Пользовательские теги. Создание, регистрация, Использование. Пример пользовательского тега.
12. JPA. Принципы. Реализации. Hibernate. Пример класса и связанной таблицы.
13. Hibernate. Аннотации. Обработка связей Один-к-Одному, Один-ко-Многим, Многие-к-Одному и Многие-ко-Многим. Пример добавления записи в таблицу, изменение записи в таблице.
14. Hibernate. Использование HQL & нативного SQL. Создание запросов на чтение данных из таблиц. Criteria. Пример запроса чтения данных с условием.
15. JDBC connection. Регистрация драйвера. Создание Connection к базе данных. DriverManager. Обеспечение транзакционности. Пример создания Connection и вставки данных в таблицу.
16. JDBC connection. Работа с Объектом типа Statment, PreparedStatment и CallableStatment. Пример изменения записи в таблице и вызова хранимой процедуры.
17. JDBC connection. Обработка результата вызова sql запроса. ResultSet. Изменение данных через ResultSet. Пример.
18. Spring. Компоненты фреймворка. Использование JDBCTemplate для доступа к базе. Пример.
19. Spring. Связывание объектов в Spring. Способы инициализации связанного объекта. Получение объекта. Синглтоны и сессионные объекты. Пример xml и java-кода.
20. Обзор Аспектно-ориентированного программирования. Подходы реализации АОП. JAspect & Spring AOP. Применение ас пектов в Spring. Точки перехвата вызова метода. Описание правил перехвата методов. Пример.
21. Реализация транзакционности в Spring. Реализация кеширования в Spring. Примеры.
22. Strats2. Библиотеки фреймворка. Подключение Strats2 к проекту. Создание Action. Методы обработки validate() & exectue().Порядок регистрации и обработки сообщений и ошибок. Пример класса Action.

23. Strats2. Перенаправление обработки в jsp. Генерация ответа в формате JSON. Strats.xml. Понятие пакета в strats.xml. Interceptor. Порядок обработки и применение Interceptor. Пример класса Interceptor.
24. JSON. Формат. Генерация на стороне сервера и обработка на стороне клиента. Использование в Servlet и Struts2. Примеры.
25. EJB. Типы EJB. Аннотации. Применение при создании веб приложения. Пример EJB.
26. JMS. Обработка с очередями сообщений. Реализация IBM WebSphere MQ. Подключение к очереди. Пример добавления и чтения сообщения из очереди. Если кто-то разберется, что именно тут надо дописать, то допишите плз.
27. JavaScript. Чувствительность к регистру. Юникод. Точки с запятой. Типы данных. null. NaN. undefined. Глобальный объект. Преобразование типов.
28. JavaScript. Объявление переменной. Область видимости. Функции. ООП. "use strict". Работа с объектами: создание, удаление, редактирование свойств. Prototype. Массивы: foreach, every, some, map, reduce, filter, indexOf. Объекты, подобные массивам. this. const.
29. jQuery. Базовые селекторы (a[href~=http://], a[href*=jquery], a[href$=.pdf], li:has(a), a:not(:hidden), a:first, a:odd, a:even, li:last-child, a:only-child, li:eq(2), li:gt(2), li:lt(7), :disabled, :selected, :visible). Создание новых элементов. .size() .get(0) .filter() .slice() .children() .contents() .next() .parents() .prev() .siblings() .find() .contains() .is(selector) .closest(). Цепочки вызовов. .each().attr() .removeAttr()
30. jQuery. .css(), .width(), .height() .addClass(), .removeClass(), .toggleClass(), .hasClass() .html(), .text() .append(), appendTo(), prepend(), prependTo(), before(), after() .remove(), .clone() .val(), .submit() .bind(), one(), on() - unbind(), off() .trigger(), .change() .hide(), show(), toggle(), fadeIn(), fadeOut(), fadeTo(), slideDown(), slideUp(), slideToggle() - stop() .animate() $.trim(), $,each(), $.grep(), $.map(), $.inArray(), $.makeArray(), $.unique(), $.extend(), $.getScript(). Ссоздание собственного плагина для jQuery

1. Многопоточное программирование. Понятия процесс и поток. Методы передачи данных между потоками и процессами. Синхронизация потоков. Synchronized vs Lock.

При помощи многопоточности можно выделить в приложении несколько потоков, которые будут выполнять различные задачи одновременно. Если у нас графическое приложение, которое посылает запрос к какому-нибудь серверу или считывает и обрабатывает огромный файл, то без многопоточности у нас бы блокировался графический интерфейс на время выполнения задачи. А благодаря потокам мы можем выделить отправку запроса или любую другую задачу, которая может долго обрабатываться, в отдельный поток. Поэтому большинство реальных приложений, которые многим из нас приходится использовать, практически не мыслимы без многопоточности.

Процесс (*process*) создается, когда программа загружается в память на выполнение (после запуска приложения пользователем). Процесс — это автономная выполняющаяся программа.

Поток (*thread* — нить процесса) — это независимая последовательность выполняемых команд процессора (фрагмент кода, который выполняется автономно и ориентирован на решение некоторой задачи).

Операционная система, имея доступ ко всем областям памяти, играет роль посредника в информационном обмене прикладных потоков. При необходимости в обмене данными поток обращается с запросом к ОС. ОС, пользуясь своими привилегиями, создает различные системные средства связи. Многие из средств межпроцессного обмена данными выполняют также и функции синхронизации: в том случае, когда данные для процесса-получателя отсутствуют, последний переводится в состояние ожидания средствами ОС, а при поступлении данных от процесса-отправителя процесс-получатель активизируется.

Передача может осуществляться несколькими способами:

- разделяемая память;
- канал (pipe, конвейер) — псевдофайл, в который один процесс пишет, а другой читает;
- сокеты — поддерживаемый ядром механизм, скрывающий особенности среды и позволяющий единообразно взаимодействовать процессам, как на одном компьютере, так и в сети;
- почтовые ящики (только в Windows), однонаправленные, возможность широковещательной рассылки;
- вызов удаленной процедуры, процесс **A** Может вызвать процедуру в процессе **B**, и получить обратно данные.

Существует два способа организовать двунаправленное соединение с помощью каналов: безымянные и именованные каналы. Канал представляет собой буфер в оперативной памяти, поддерживающий очередь байт по алгоритму FIFO. Для программиста этот буфер выглядит как безымянный файл, в который можно писать и читать, осуществляя тем самым обмен данными. Обмениваться данными могут только родственные процессы, точнее процессы, которые имеют общего прародителя, создавшего данный конвейер. Связано это с тем, что конвейер не имеет имени, обращение к нему происходит по дескриптору, который имеет локальное для каждого процесса значение.

Процессы выполняются в собственном адресном пространстве и защищены от воздействия друг на друга средствами операционной системы. Потоки лишены подобной защиты. Любой поток в рамках одного процесса может получить доступ и внести изменения в данные, которые являются разделяемыми ресурсами, т.е. которые каждый поток считает принадлежащими только ему. Основной механизм, применяемый

в многопоточном программировании для обеспечения корректной работы потоков с разделяемыми ресурсами — ***синхронизация потоков***. Java на уровне языка обеспечивает поддержку синхронизации потоков.

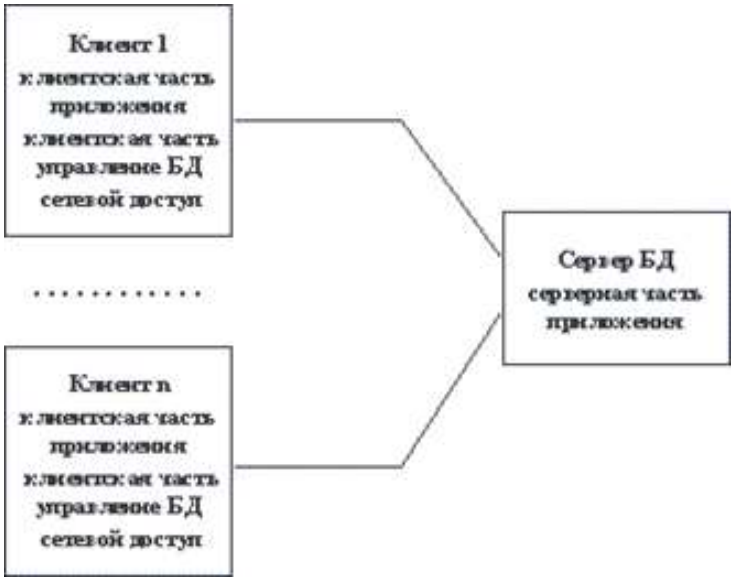
Эффект синхронизации достигается за счёт применения ***механизма блокировки (lock)***. Каждому объекту в Java ставится в соответствие *свойство блокировки* ***lock***. Lock — специальная переменная в области памяти, выделяемой Java-машиной при создании объекта. Потоки взаимодействуют в соответствии с протоколом, согласно которому любому действию над объектом должен предшествовать *захват права на блокировку объекта*. Если поток захватил такое право (*установил блокировку объекта*), то другие потоки не могут использовать объект, пока право не будет возвращено потоком-владельцем,то есть пока блокировка не будет освобождена. Право доступа к блокировке объекта обеспечивается в Java с помощью ключевого слова ***synchronized***.

Использование ключевого слова synchronized гарантирует, что разделяемый ресурс потоков будет одновременно использоваться только одним потоком.

synchronized гарантирует, что потоки пришедшие к синхронизируемому коду становятся в очередь и будут захватывать блокировку в порядке очереди. Использование Lock это не гарантирует. Использование synchronized уменьшает возможность допущения ошибки, ибо сама jvm строит код, который гарантирует то, что блокировка в случае чего будет освобождена. В то же время Lock предоставляет больше гибкости, таких как методы tryLock, lockInterrupt, lockTimeout и т.д., а также позволяет освободить блокировку в scope, отличном от того, в котором блокировка была захвачена. В то же время synchronized позволяет внутри себя использовать на объектах синхронизации методы wait, notify, notify all.

In the single-threaded test, synchronized() was about 7.5x faster on average than Lock.lock(). In the two-threaded test, synchronized() was still the clear winner, about 2x faster on average.

2. Клиент-серверное приложение. Веб приложение. Front-end и Back-end программирование. Web-server. Http-server. Proxy-server. Статические и динамические ресурсы.



Под клиент-серверным приложением будем понимать информационную систему, основанную на использовании серверов баз данных .Общее представление информационной системы в архитектуре "клиент-сервер" показано на рисунке.

На стороне клиента выполняется код приложения, в который обязательно входят компоненты, поддерживающие интерфейс с конечным пользователем, производящие отчеты, выполняющие другие специфичные для приложения функции.

Клиентская часть приложения взаимодействует с клиентской частью программного обеспечения управления базами данных, которая, фактически, является индивидуальным представителем СУБД для приложения.

Веб-приложение — клиент-серверное приложение, в котором клиентом выступает браузер, а сервером — веб-сервер. Логика веб-приложения распределена между сервером и клиентом, хранение данных осуществляется, преимущественно, на сервере, обмен информацией происходит по сети. Одним из преимуществ такого подхода является тот факт, что клиенты не зависят от конкретной операционной системы пользователя, поэтому веб-приложения являются кроссплатформенными сервисами.

front-end разработка – это создание клиентской части сайта. Front-end разработчик занимается версткой шаблона сайта и созданием пользовательского интерфейса.

backend (бэкэнд) — серверная часть (работа с БД, обработка данных, и т.д.), в общем все, чего клиент не видит.

Веб-сервер — сервер, принимающий HTTP-запросы от клиентов, обычно веб-браузеров, и выдающий им HTTP-ответы, как правило, вместе с HTML-страницей, изображением, файлом, медиа-потоком или другими данными.

Веб-сервером называют как программное обеспечение, выполняющее функции веб-сервера, так и непосредственно компьютер , на котором это программное обеспечение работает

HTTP (англ. *HyperText Transfer Protocol* — «протокол передачи гипертекста») — протокол прикладного уровня передачи данных (изначально — в виде гипертекстовых документов в формате HTML, в настоящий момент используется для передачи произвольных данных). Основой HTTP является технология «клиент-сервер», то есть предполагается существование потребителей (клиентов), которые инициируют соединение и посылают запрос, и поставщиков (серверов), которые ожидают

соединения для получения запроса, производят необходимые действия и возвращают обратно сообщение с результатом.

Прокси-сервер (от англ. *proxy* — «представитель, уполномоченный») — сервер (комплекс программ) в компьютерных сетях, позволяющий клиентам выполнять косвенные запросы к другим сетевым службам. Сначала клиент подключается к прокси-серверу и запрашивает какой-либо ресурс (например, e-mail), расположенный на другом сервере. Затем прокси-сервер либо подключается к указанному серверу и получает ресурс у него, либо возвращает ресурс из собственного кэша (в случаях, если прокси имеет свой кэш). В некоторых случаях запрос клиента или ответ сервера может быть изменён прокси-сервером в определённых целях. Прокси-сервер позволяет защищать компьютер клиента от некоторых сетевых атак и помогает сохранять анонимность клиента.

Ресурсы могут быть статическими и динамическими. Статические ресурсы(HTML, JavaScript, изображения и т.д.) устанавливается только один раз. А динамические ресурсы (сервлеты, JSP, а также java-код;) могут меняться в течение работы программы.

Смысл разделения динамического и статического содержания в том, что статические ресурсы могут находиться под управлением HTTP-сервера, в то время как динамические нуждаются в движке (Servlet Engine) и в большинстве случаев в доступе к уровню данных.

Рекомендуется разделить и разрабатывать параллельно две части приложения: Web-приложение, состоящее только из динамических ресурсов, и Web-приложение, состоящее только из статических ресурсов.

· *Разделение динамического и статического содержания.*

Возможность разделить логику приложения и дизайн Web-страницы снижает сложность разработки Web-приложений и упрощает их поддержку.

3. Сервер приложений IBM WAS. Apach Tomcat. Масштабируемость и кластеризация. Понятие кластера. Балансировка запросов.

IBM WebSphere Application Server помогает создавать, развертывать и запускать приложения с гибкими, безопасными и Java EE-сертифицированными средами выполнения, начиная от легковесных производственных сред и заканчивая средами корпоративного уровня. Вы можете использовать WebSphere Application Server локально, в общедоступных, частных и гибридных облаках и выбрать одну из нескольких версий по подходящей цене.

WebSphere Application Server (WAS). WebSphere относится к категории middleware — промежуточного программного обеспечения, которое позволяет приложениям электронного бизнеса (e-business) работать на разных платформах на основе веб-технологий.

WebSphere Application Server доступен в трех версиях:

- WebSphere Application Server Liberty Core — Легковесная рабочая среда для быстрой разработки и развертывания облачных и веб-приложений.
- WebSphere Application Server Base — Гибкая и безопасная серверная среда выполнения Java для корпоративных приложений, которая предоставляет модели программирования и обладает высокой производительностью и избыточностью.
- WebSphere Application Server Network Deployment — Улучшенная гибкая среда выполнения для крупномасштабных и важных приложений, обеспечивающая практически постоянную готовность и предоставляющая функции Intelligent Management для открытых сред и сред z/OS.

Tomcat (в старых версиях — **Catalina**) — контейнер сервлетов с открытым исходным кодом, разрабатываемый Apache Software Foundation. Реализует спецификацию сервлетов и спецификацию JavaServer Pages (JSP) и JavaServer Faces (JSF). Написан на языке Java.

Tomcat позволяет запускать веб-приложения, содержит ряд программ для самоконфигурирования.

Tomcat используется в качестве самостоятельного веб-сервера, в качестве сервера контента в сочетании с веб-сервером Apache HTTP Server, а также в качестве контейнера сервлетов в серверах приложений JBoss и GlassFish.

Одно из самых современных направлений в области создания вычислительных систем–это *кластеризация*. По производительности и коэффициенту готовности кластеризация представляет собой альтернативу симметричным мультипроцессорным системам.

Кластер – это группа взаимно соединенных вычислительных систем (узлов), работающих совместно, составляя единый вычислительный ресурс и создавая иллюзию наличия единственной ВМ. В качестве узла кластера может выступать как однопроцессорная ВМ, так и ВС типа SMP или MPP. Каждый узел в состоянии функционировать самостоятельно и отдельно от кластера. Архитектура кластерных вычислений сводится к объединению нескольких узлов высокоскоростной сетью. Наряду с термином «кластерные вычисления» часто применяются такие названия, как:*кластер рабочих станций*(workstation cluster), *гипервычисления* (hypercomputing), *параллельные вычисления на базе се-ти*(network-basedconcurrentcomputing).

Перед кластерами ставятся две задачи:

- достичь большой вычислительной мощности;
- обеспечить повышенную надежность ВС.
- готовность
- масштабируемость

Кластер — это система произвольных устройств (серверы, дисковые накопители, системы хранения и пр.), обеспечивающих отказоустойчивость на уровне 99,999%, а также удовлетворяющая «четырем правилам».

Первый коммерческий кластер создан корпорацией DEC в начале 80-х годов прошлого века.

В качестве узлов кластеров могут использоваться как одинаковые ВС (гомогенные кластеры), так и разные (гетерогенные кластеры). По своей архитектуре кластерная ВС является слабо связанной системой.

Преимущества, достигаемые с помощью кластеризации:

- **Абсолютная масштабируемость**. Возможно создание больших кластеров, превосходящих по вычислительной мощности даже самые производительные одиночные ВМ. Кластер в состоянии содержать десятки узлов, каждый из которых представляет собой мультиплексор.
- **Наращиваемая масштабируемость**. Кластер строится так, что его можно наращивать, добавляя новые узлы небольшими порциями.
- **Высокий коэффициент готовности**. Поскольку каждый узел кластера– самостоятельная ВМ или ВС, отказ одного из узлов не приводит к потере работоспособности кластера. Во многих системах отказоустойчивость автоматически поддерживается программным обеспечением.
- **Превосходное соотношение цена/производительность**. Кластер любой производительности можно создать, соединяя стандартные ВМ, при этом его стоимость будет ниже, чем у одиночной ВМ с эквивалентной вычислительной мощностью.

На уровне аппаратного обеспечения кластер – это просто совокупность независимых вычислительных систем, объединенных сетью. При соединении машин в кластер почти всегда поддерживаются прямые межмашинные связи. Решения могут быть простыми, основывающимися на аппаратуре Ethernet, или сложными с высокоскоростными сетями с пропускной способностью в сотни мегабайтов в секунду (система RS/6000 SP компании IBM, системы фирмы Di- gital на основе Memory Channel,ServerNet корпорации Compaq).

Узлы кластера контролируют работоспособность друг друга и обмениваются специфической информацией. Контроль работоспособности осуществляется с помощью специального сигнала, называемого *heart-beat*(«сердцебиение»). Этот сигнал передается узлами кластера друг другу, чтобы подтвердить их нормальное функционирование.

Неотъемлемой частью кластера является специализированное программное обеспечение (ПО), на которое возлагается задача обеспечения бесперебойной работы при отказе одного или нескольких узлов. Такое ПО производит перераспределение вычислительной нагрузки при отказе одного или нескольких узлов кластера, а также восстановление вычислений при сбое в узле. Кроме того, при наличии в кластере совместно используемых дисков кластерное ПО поддерживает единую файловую систему.

Балансировка нагрузки

Существует несколько распространенных подходов в организации распределения нагрузки:

- **круговой DNS**, когда для распределения нагрузки используется DNS-сервер
- **аппаратное распределение нагрузки**, когда используется прибор, схожий по своим функциям с маршрутизатором
- **программное распределение нагрузки**. Например программа "TCP/IP Network Load Balancing" от Microsoft.
- **смешанные схемы**, когда используется комбинация аппаратных и программных средств

4. Обзор JavaEE. Обзор компонентов стандарта: Servlets, JSP, EJB, JPA, JSTL, JSF, JNDI, JMS, JTA, JAAS

С помощью архитектуры Java EE (Java™ Platform, Enterprise Edition) можно создавать распределенные веб-приложения и приложения J2EE. Данная архитектура освобождает от системных задач и помогает сосредоточиться на представлении данных и логике приложения

Java EE 5 Platform Enterprise Edition (Java EE) ускоряет разработку приложений и делает ее более удобной, чем в предыдущих версиях. Java EE 5 является заменой J2EE 1.4. Инструменты продукта поддерживают обе версии. Java EE 5 значительно проще в использовании:

- Меньше время разработки
- Ниже сложность приложения
- Выше производительность приложения

Java EE 5 предоставляет упрощенную программную модель, включая следующие средства:

- Настройка прямо в коде приложения с помощью аннотаций (файлы описания теперь необязательны)
- Добавление зависимостей, скрывающее создание и поиск ресурсов от кода приложения
- Java persistence API (JPA) позволяет управлять данными, не прибегая к SQL или JDBC
- Применение обычных объектов Java для объектов EJB и веб-служб

В Java EE 5 более простые правила упаковки приложений J2EE:

- Веб-приложения упаковываются в файлы .WAR
- Адаптеры ресурсов упаковываются в файлы .RAR
- Приложения J2EE упаковываются в файлы .EAR
- Каталог lib содержит общие файлы .JAR
- Файл .JAR с Main-Class является приложением-клиентом
- Файл .JAR с аннотацией @Stateless является приложением EJB
- Многим простым приложениям не требуются файлы описания, включая
    - Приложения EJB (файлы .JAR)
    - Веб-приложения, в которых применяется только технология JSP
    - Приложения-клиенты
    - Приложения J2EE (файлы .EAR)

Java EE 5 обеспечивает более простой доступ к ресурсам с помощью добавления зависимостей:

- В шаблоне проектирования Вставка зависимости внешняя сущность автоматически поставляет зависимости объекта.
    - Объекту не нужно запрашивать эти ресурсы явно
- В Java EE 5 добавление зависимостей может применяться ко всем ресурсам, которые требуются компоненту
    - Процесс создания и поиска ресурсов скрыт от кода приложения
- Добавление зависимостей может применяться во всех элементах технологии Java EE 5:
    - Контейнеры EJB
    - Веб-контейнеры
    - Клиенты
    - Веб-службы

Общие сведения о Java EE

Java EE (Enterprise Edition) представляет собой широко используемую платформу, содержащую набор взаимосвязанных технологий, которые существенно сокращают стоимость и сложность разработки, развертывания многоуровневых серверных приложений, а также управления ими. Платформа Java EE основана на платформе Java SE и предоставляет набор интерфейсов API (интерфейсов разработки приложений) для разработки и запуска портируемых, надежных, масштабируемых и безопасных серверных приложений.

Java EE в числе прочих содержит следующие компоненты:

- Enterprise JavaBeans (EJB): управляемая серверная архитектура компонентов, используемая для инкапсуляции бизнес-логики приложения. Технология EJB позволяет осуществлять быструю и упрощенную разработку распределенных, транзакционных, безопасных и переносимых приложений, основанных на технологии Java.
- Интерфейс API сохранения состояния Java (Java Persistence API, JPA): инфраструктура, позволяющая разработчикам управлять данными с помощью объектно-реляционного сопоставления (ORM) в приложениях, созданных на платформе Java.

Сервлет является интерфейсом Java, реализация которого расширяет функциональные возможности сервера. Сервлет взаимодействует с клиентами посредством принципа запрос-ответ.

Хотя сервлеты могут обслуживать любые запросы, они обычно используются для расширения веб-серверов. Для таких приложений технология Java Servlet определяет HTTP-специфичные сервлет классы.

Пакеты javax.servlet и javax.servlet.http обеспечивают интерфейсы и классы для создания сервлетов.

JSP (JavaServer Pages) — технология, позволяющая веб-разработчикам создавать содержимое, которое имеет как статические, так и динамические компоненты. Страница JSP содержит текст двух типов: статические исходные данные, которые могут быть оформлены в одном из текстовых форматов HTML, SVG, WML, или XML, и JSP- элементы, которые конструируют динамическое содержимое. Кроме этого могут использоваться библиотеки JSP-тегов, а также EL (Expression Language), для внедрения Java-кода в статичное содержимое JSP-страниц.

Код JSP-страницы транслируется в Java-код сервлета с помощью компилятора JSP-страниц Jasper, и затем компилируется в байт-код виртуальной машины java (JVM). Контейнеры сервлетов, способные исполнять JSP-страницы, написаны на платформонезависимом языке Java. JSP-страницы загружаются на сервере и управляются из структуры специального Java server packet, который называется Java EE Web Application. Обычно страницы упакованы в файловые архивы .war и .ear.

JSP является платформонезависимой, переносимой и легко расширяемой технологией для разработки веб-приложений

Enterprise JavaBeans (также часто употребляется в виде аббревиатуры EJB) — спецификация технологии написания и поддержки серверных компонентов, содержащих бизнес-логику. Является частью Java EE.

Эта технология обычно применяется, когда бизнес-логика требует как минимум один из следующих сервисов, а часто все из них:

- поддержка сохранности данных (persistence); данные должны быть в сохранности даже после остановки программы, чаще всего достигается с помощью использования базы данных
- поддержка распределённых транзакций
- поддержка параллельного изменения данных и многопоточность
- поддержка событий
- поддержка именования и каталогов (JNDI)
- безопасность и ограничение доступ а к данным
- поддержка автоматизированной установки на сервер приложений
- удалённый доступ

Каждый EJB-компонент является набором Java-классов со строго регламентированными правилами именования методов (верно для EJB 2.0, в EJB 3.0 за счет использования аннотаций выбор имён свободный). Бывают трёх основных типов:

- *объектные* (Entity Bean) — перенесены в спецификацию Java Persistence API
- *сессионные* (Session Beans), которые бывают
    - stateless (без состояния)
    - stateful (с поддержкой текущего состояния сессии)
    - singleton (один объект на все приложение; начиная с версии 3.1)
- *управляемые сообщениями* (Message Driven Beans) — их логика является реакцией на события в системе

Java Persistence API (*JPA*) — API, входящий с версии Java 5 в состав платформ Java SE и Java EE, предоставляет возможность сохранять в удобном виде Java-объекты в базе данных.

Существует несколько реализаций этого интерфейса, одна из самых популярных использует для этого Hibernate. JPA реализует концепцию ORM.

Поддержка сохранности данных, предоставляемая *JPA*, покрывает области:

- непосредственно API, заданный в пакете `javax.persistence`;
- платформо-независимый объектно-ориентированный язык запросов Java Persistence Query Language;
- метаинформация, описывающая связи между объектами.
- Генерация DDL для сущностей

Стандартная библиотека тегов JSP (англ. *JavaServer Pages Standard Tag Library*, JSTL) — расширение спецификации JSP, добавляющее библиотеку JSP тегов для общих нужд, таких как разбор XML данных, условная обработка, создание циклов и поддержка интернационализации.

JSTL является альтернативой такому виду встроенной в JSP логики, как скриплеты, то есть прямые вставки Java кода. Использование стандартизованного множества тегов предпочтительнее, поскольку получаемый код легче поддерживать и проще отделять бизнес-логику от логики отображения.

JavaServer Faces (JSF) — это фреймворк для веб-приложений, написанный на Java. Он служит для того, чтобы облегчать разработку пользовательских интерфейсов для Java EE-приложений. В отличие от прочих MVC-фреймворков, которые управляются запросами, подход JSF основывается на использовании компонентов. Состояние компонентов пользовательского интерфейса сохраняется когда пользователь запрашивает новую страницу и затем восстанавливается, если запрос повторяется. Для

отображения данных обычно используется JSP, Facelets, но JSF можно приспособить и под другие технологии, например XUL.

Технология JavaServer Faces включает:

- Набор API для представления компонент пользовательского интерфейса (UI) и управления их состоянием, обработкой событий и валидацией вводимой информации, определения навигации, а также поддержку интернационализации (i18n) и доступности (accessibility).
- Специальная библиотека JSP тегов для выражения интерфейса JSF на JSP странице. В JSF 2.0 в качестве обработчика представления используется технология Facelets которая пришла на замену JSP.

Java Naming and Directory Interface (JNDI) — это Java API, организованный в виде службы каталогов, который позволяет Java-клиентам открывать и просматривать данные и объекты по их именам.

API предоставляет:

- механизм ассоциации (связывания) объекта с именем;
- интерфейс просмотра директорий для выполнения общих запросов;
- интерфейс событий, который позволяет определить клиентам, когда элементы директории были изменены;
- LDAP-расширение для поддержки дополнительных возможностей LDAP-сервисов.

Java Message Service (JMS) — стандарт промежуточного ПО для рассылки сообщений, позволяющий приложениям, выполненным на платформе Java EE, создавать, посылать, получать и читать сообщения. Часть Java Platform, Enterprise Edition.

Коммуникация между компонентами, использующими JMS, асинхронна (процедура не дожидается ответа на своё сообщение) и независима от исполнения компонентов.

JMS поддерживает две модели обмена сообщениями: «от пункта к пункту» и «издатель-подписчик».

Модель «от пункта к пункту» характеризуется следующим:

- Каждое сообщение имеет только одного адресата
- Сообщение попадает в «почтовый ящик», или «очередь» адресата и может быть прочитано когда угодно. Если адресат не работал в момент отсылки сообщения, сообщение не пропадёт.
- После получения сообщения адресат посылает извещение.

Модель «издатель-подписчик» характеризуется следующим:

- Подписчик подписывается на определённую «тему»
- Издатель публикует своё сообщение. Его получают все подписчики этой темы
- Получатель должен работать и быть подписан в момент отправки сообщения

Java Transaction API, сокращенно JTA — Java API для транзакций. Определяет взаимодействие между менеджером транзакций и другими участниками распределённой транзакционной системы. JTA — спецификация, разработанная в рамках Java Community Process в качестве JSR 907. JTA обеспечивает:

- разделение границ транзакции
- API к стандарту X/Open XA, описывающему взаимодействие ресурсов в транзакциях.

Сервис Аутентификации и Авторизации Java (англ. *Java Authentication and Authorization Service*, сокр. *JAAS*) — стандарт API Java SE и реализация системы информационной безопасности PAM.

Главной целью JAAS является отделение аутентификации и авторизации пользователей от основной программы, чтобы управлять ими независимо от программы. Прошлый механизм аутентификации работал лишь на основе того откуда получен исполняемый код (например, локальному коду предоставлялось больше привилегий, чем коду полученному из интернета), но современная версия JAAS, также использует информацию о том, кто именно запустил код. JAAS является полностью расширяемым, тем самым он позволяет создавать собственные механизмы проверки подлинности и авторизации.

5. **Понятие Servlet. Жизненный цикл сервлета. Методы HttpServelet. Пример реализация пользовательского сервлета.**

Сервлет является интерфейсом Java, реализация которого расширяет функциональные возможности сервера. Сервлет взаимодействует с клиентами посредством принципа запрос-ответ.

Хотя сервлеты могут обслуживать любые запросы, они обычно используются для расширения веб-серверов. Для таких приложений технология Java Servlet определяет HTTP-специфичные сервлет классы.

Пакеты javax.servlet и javax.servlet.http обеспечивают интерфейсы и классы для создания сервлетов.

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.

методы

- doGet , если сервлет поддерживает HTTP GET запросов
- doPost , для запросов HTTP POST
- doPut , для запросов HTTP PUT
- doDelete , для HTTP DELETE запросов инициализации и уничтожить , чтобы управлять ресурсами, которые проводятся для жизни сервлета
- getServletInfo , который сервлет использует для предоставления информации о себе

```
import java.io.IOException;
import java.io.PrintWriter;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.servlet.http.HttpSession;

public class NewServlet extends HttpServlet {
```

```
    @Override
    protected void doGet(HttpServletRequest request,
HttpServletResponse response) throws ServletException, IOException {

        // Параметр
        String parameter = request.getParameter("parameter");
        // Старт HTTP сессии
        HttpSession session = request.getSession(true);
        session.setAttribute("parameter", parameter);

        response.setContentType("text/html;charset=UTF-8");
        PrintWriter out = response.getWriter();
        try {
            out.println("<html>");
            out.println("<head>");
            out.println("<title>Заголовок</title>");
            out.println("</head>");
            out.println("<body>");
            out.println("<h1>Пример сервлета"+parameter+"</h1>");
            out.println("</body>");
            out.println("</html>");
        } finally {
            out.close();
        }
    }
    @Override
    public String getServletInfo() {
        return "Пример сервлета";
    }

}
```

6. Понятие Filter, Listener. Жизненный цикл Filter и Listener. Порядок вызова методов объекта, реализующий Filter и Listener. Примеры реализации Filter, Listener.

Сервлетный фильтр, в соответствии со спецификацией, это Java-код, пригодный для повторного использования и позволяющий преобразовать содержание HTTP-запросов, HTTP-ответов и информацию, содержащуюся в заголовках HTML. Сервлетный фильтр занимается предварительной обработкой запроса, прежде чем тот попадает в сервлет, и/или последующей обработкой ответа, исходящего из сервлета.

Основой для формирования фильтров служит интерфейс *javax.servlet.Filter*, который реализует три метода:

- void *init* (FilterConfig config) throws ServletException;
- void *destroy ()*;
- void *doFilter* (ServletRequest request, ServletResponse response, FilterChain chain) throws IOException, ServletException;

Жизненный цикл и порядок вызова методов объекта фильтра:

Метод *init* вызывается прежде, чем фильтр начинает работать, и настраивает конфигурационный объект фильтра. Метод *doFilter* выполняет непосредственно работу фильтра. Таким образом, сервер вызывает *init* один раз, чтобы запустить фильтр в работу, а затем вызывает *doFilter* столько раз, сколько запросов будет сделано непосредственно к данному фильтру. После того, как фильтр заканчивает свою работу, вызывается метод *destroy.*

Пример реализации фильтра:

```
public class MyFilter implements Filter {
	private FilterConfig filterConfig;

	@Override
	public void init(FilterConfig filterConfig) throws
ServletException {
		this.filterConfig = filterConfig;
	}

	@Override
	public void doFilter(ServletRequest request, ServletResponse
response, FilterChain chain) throws IOException, ServletException {
		//код фильтра
		chain.doFilter(request, response);  // вызываем следующий
фильтр
	}

	@Override
	public void destroy() {
		this.filterConfig = null;
	}
}
```

Слушатель (listener) - это объект, уведомляемый о событии. Он должен быть зарегистрирован источником событий и реализовывать методы для получения и обработки уведомлений.

Жизненный цикл слушателя:

- Определить для заданного события класс, реализующий связанный с ним интерфейс. В существующее определение класса можно добавить строку "implements типListener" или создать новый класс, реализующий данный интерфейс.
- Необходимо реализовать каждый метод данного интерфейса.
- После определения реализации интерфейса необходимо создать экземпляр реализации и ассоциировать его с компонентом. Только после этого наблюдатель (слушатель) сможет получать уведомления о возникновении соответствующих событий.

Пример реализации слушателя:

```
interface AListener {
  public void doEvent();
}

class A {
  AListener listeners[];
  public void addListener(AListener listener) {
    //Запоминаем listener
  }

  public void doSomething(){
    //Делаем что-то о чем требуется оповестить всех слушателей
    for( int i = 0; i < listeners.length; i++ ) {
      listeners[i].doEvent(); //class A не знает кто его слушает
    }
  }
}

class B implements AListener {
  public void doEvent(){
    //Что-то случилось в классе A
  }
}

//...
A a = new A();
B b = new B();
B c = new B();

a.addListener(b);
a.addListener(c);
a.doSomething(); //Оба объекта b и c выполняют doEvent
```

Servlet Filter is used for monitoring request and response from client to the servlet, or to modify the request and response, or to audit and log.

Servlet Listener is used for listening to events in a web containers, such as when you create a session, or place an attribute in an session or if you passivate and activate in another container, to subscribe to these events you can configure listener in `web.xml`, for example `HttpSessionListener`.

7. Регистрация Servlet, Filter, Listener при помощи аннотаций и в web.xml. Файл web.xml. Формат. Основные теги. Пример web.xml

Регистрация сервлетов и фильтров происходит добавлением **аннотаций, таких как @WebServlet, @WebFilter и @WebListener**, к соответствующим классам. Это не только упрощает кодирование классов сервлетов, фильтров и слушателей, но и делает необязательным дескриптор развертывания web.xml.

@WebServlet - аннотация уровня класса. Указывает, что аннотированный класс является сервлетом, и содержит некоторые метаданные о сервлете.

@WebFilter и **@WebListener** - аннотации, используемые для объявления фильтров и слушателей в классах веб-приложения.

У всех аннотаций могут указываться в скобках параметры. Нужно заметить, что дескриптор развертывания (web.xml) имеет приоритет над аннотациями. Другими словами, он может переопределить конфигурацию, заданную механизмом аннотации.

Servlet web.xml

```
<web-app>
   <servlet>
           <servlet-name>MyServlet</servlet-name>

           <servlet-class>samples.MyServlet</servlet-class>
   </servlet>

   <servlet-mapping>
           <servlet-name>MyServlet</servlet-name>
           <url-pattern>/MyApp</url-pattern>
   </servlet-mapping>
...
 </web-app>
```

Filter web.xml

```
<filter>
        <filter-name>FilterName</filter-name>
        <filter-class>FilterConnect</filter-class>
        <init-param>
                <param-name>active</param-name>
                <param-value>true</param-true>
        </init-param>
</filter>
<filter-mapping>
        <filter-name>FilterName</filter-name>
        <servlet-name>ServletName</servlet-name>
</filter-mapping>
```

Listener web.xml

```
<listener>
    <listener-class>example</listener-class>
</listener>
```

Web.xml (дескриптор развертывания) - конфигурационный файл артефакта, который будет развернут в контейнере сервлетов. Этот конфигурационный файл

указывает параметры развертывания для модуля или приложения с определенными настройками, параметры безопасности и описывает конкретные требования к конфигурации. Для синтаксиса файлов дескриптора развертывания используется язык XML.

Основные теги:

context-param - элемент, содержащий объявление параметров инициализации
filter - определяет класс фильтра и его атрибуты инициализации
filter-mapping - определяет соответствие фильтра
listener - определяет слушателя в приложении
servlet - содержит объявляемые данные сервлета
servlet-mapping - определяет соответствие между сервлетом и URL паттерном
welcome-file-list - список элементов welcome-file
welcom-file - имя файла, используемое имя файла по умолчанию

Пример web.xml:

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns="http://xmlns.jcp.org/xml/ns/javaee"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
http://xmlns.jcp.org/xml/ns/javaee/web-app_3_1.xsd"
         version="3.1">
    <filter>
        <filter-name>struts2</filter-name>                    <filter-
class>org.apache.struts2.dispatcher.ng.filter.StrutsPrepareAndExecuteF
ilter</filter-class>
    </filter>
    <filter-mapping>
        <filter-name>struts2</filter-name>
        <url-pattern>/*</url-pattern>
    </filter-mapping>
    <welcome-file-list>
        <welcome-file>index.jsp</welcome-file>
    </welcome-file-list>
</web-app>
```

8. Servlet и JSP. Компиляция JSP в Servlet. Пример JSP и Servlet с одинаковой логикой. Перенаправление на JSP. Сравнение методов forward() и sendRedirect().

Код JSP-страницы транслируется в Java-код сервлета с помощью компилятора JSP-страниц Jasper, и затем компилируется в байт-код виртуальной машины java (JVM). Контейнеры сервлетов, способные исполнять JSP-страницы, написаны на платформонезависимом языке Java. JSP-страницы загружаются на сервере и управляются из структуры специального Java server packet, который называется Java EE Web Application. Обычно страницы упакованы в файловые архивы .war и .ear.

JSP является платформонезависимой, переносимой и легко расширяемой технологией для разработки веб-приложений.

Жизненный цикл JSP

Страница JSP обслуживает запросы, как сервлет. Следовательно, жизненный цикл и многие возможности страниц JSP (в частности, динамические аспекты) определяются технологией **Servlet**.

Когда запрос отображается на страницу JSP, он обрабатывается специальным сервлетом, который сначала проверяет, не старше ли сервлет страницы JSP, чем сама страница JSP. Если это так, он переводит страницу JSP в класс сервлета и компилирует класс. При разработке Web-приложения одним из преимуществ страниц JSP перед сервлетами является то, что процесс построения (*компиляции страницы JSP в сервлет*) выполняется автоматически.

Трансляция и компиляция страницы JSP

На фазе трансляции каждый тип данных в странице JSP интерпретируется отдельно. Шаблонные данные трансформируются в код, который будет помещать данные в поток, возвращающий данные клиенту. Элементы JSP трактуются следующим образом:

- директивы, используемые для управления тем, как Web-контейнер переводит и выполняет страницу JSP;
- скриптовые элементы вставляются в класс сервлета страницы JSP;
- элементы в форме <jsp:XXX ... /> конвертируются в вызов метода для компонентов **JavaBeans** или вызовы API Java Servlet.

И фаза трансляции, и фаза компиляции могут порождать ошибки, которые будут выведены только, когда страница будет в первый раз запрошена

Когда страница оттранслирована и откомпилирована, сервлет страницы JSP в основном следует жизненному циклу сервлета, описанному на странице сервлета:

1. Если экземпляр сервлета страницы JSP не существует, контейнер:
   - загружает класс сервлета страницы JSP;
   - создает экземпляр класса сервлета;
   - инициализирует экземпляр сервлета вызовом метода jspInit.
2. Вызывает метод _jspService, передавая ему объекты запроса и отклика.

Если контейнеру нужно удалить сервлет страницы JSP, он вызывает метод jspDestroy.

```
<html>
<head>,
<title>Пример</title>
</head>
 <body>
<form enctype="multipart/form-data" action="servlet.java" method="post">
```

```
 Имя <input name="myname" type="text">
  <input type="submit" value="Отправить">
</form>
</body>
</html>

  protected void doPost(HttpServletRequest request, HttpServletResponse
response)
            throws ServletException, IOException{
            request.setCharacterEncoding("utf8
            String myName = request.getParameter("myname");
}
```

отличия методов forward() и sendRedirect()?

- RequestDispatcher forward() используется для проброски того же самого запроса к другому ресурсу, в то время как ServletResponse sendRedirect() это двух шаговый метод. Во втором методе веб приложение возвращает ответ клиенту с status code 302 (redirect) с ссылкой для отправки запроса. Запрос посылает полностью новый запрос.
- forward() обрабатывается внутри контейнера, а sendRedirect() обрабатывается браузером.
- Необходимо использовать forward() для организации доступа внутри одного и того же приложения, т.к. он быстрее sendRedirect(), которому требуется дополнительная сетевая работа.
- В методе forward() браузер не знает о фактически обрабатываемом ресурсе и URL в строке остается прежним. В sendRedirect() методе URL адрес изменяется на пробрасываемый ресурс.
- В методе forward() нельзя использовать для внедрения сервлета в другой контекст. Для этого можно использовать только sendRedirect().

9. JSP. Скриплеты. Коментарии в jsp. HTML. Порядок генерации контента на основе jsp страницы. Пример jsp.

Технология Java Server Pages (JSP) предназначена для создания специальной серверной компоненты web-приложения, называемой jsp-страницей и обладающей одновременно свойствам html-страницы и сервлета. В самом первом приближении jsp-страница – это html-странница с вкраплениями java-кода. Как и в случае с сервлетом для исполнения jsp-страницы требуется специальный контейнер (JSP Engine), который отвечает за разбор (parsing) страницы JSP и преобразование ее в сервлет, генерирующий при исполнении html-код.

Скриплеты JSP

Скриплеты должны содержать фрагменты кода на языке скрипта, который указывается в атрибуте **language** директивы page(в нашем случае это язык Java). Тег JSP применяемый для скриплетов имеет следующий синтаксис:

<% скрипт на языке Java %>

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);
%>
```

Код внутри скриплета вставляется в том виде, как он был записан. Весь статический HTML (текст шаблона) до или после скриплета конвертируется при помощи оператора print.

скриплеты не обязательно должны содержать завершённые фрагменты Java, и что оставленные открытыми блоки могут оказать влияние на статический HTML вне скриплета.

Комментарии используются для пояснения исходного текста программы. В JSP-страницах комментарии можно разделить на две группы:

- комментарии исходного кода JSP
- комментарии HTML-разметки.

Комментарии исходного кода JSP отмечаются специальной последовательностью символов: <%-- в начале и --%> в конце комментария

Комментарии HTML-разметки оформляются в соответствии с правилами языка HTML. Данный вид комментариев рассматривается JSP-компилятором как статический текст и помещается в выходной HTML-документ. JSP-выражения внутри HTML-комментариев исполняются. Пример HTML-комментария:

<!-- Дата создания страницы: <%= new java.util.Date() %> →

HTML – это язык разметки. **HTML-страница** — это набор вложенных элементов, по своей структуре напоминающей дерево.

JSP страница

Как правило, JSP страница хранится в отдельном файле с расширением .jsp. Большая часть содержимого JSP страницы преобразуется в сервлете в набор инструкций out.println(). Пример JSP страницы:

```
<%@ taglib uri="/exttags.tld" prefix="dscat" %>
<%@ page import = "xbcat.util.*"  %>
```

```
<dscat:pageheader>All Customers</dscat:pageheader>
<jsp:useBean id="pagescroll" class="ru.view.bean.ScrollPage" scope="session">
</jsp:useBean>
<jsp:setProperty name="pagescroll" property="PageSize" param="psize" />
<html><body>
<%
  Vector menu=pagescroll.getMenu();
  if( pagescroll.page.size() > 0 ) {
%>
<table width="100%" border="0">
<tr>
<td>
  <%= pagescroll.getTotalPages() %>
</td>
<td align="right">
<% if(!pagescroll.isSinglePage()) {
    for(int i=0; i<menu.size(); i++) {
      String href = ((ScrollMenu)menu.elementAt(i)).m_Href;
      String name = ((ScrollMenu)menu.elementAt(i)).m_Name;
      if( href != null ) { %>
    <a href="<%= href %>"><%= name %></a>
<% } else { %>
    <%= name %>
<%    }
    }
  } %>
</td>
</tr>
</table></body></html>
```

Динамическая составляющая JSP страницы представлена тремя типами специальных элементов: директивами, action и скриптами. Подробнее каждый из них рассматривается в соответствующем разделе.

10.Session, request, cookies. Жизненные циклы. Атрибуты и параметры Request. Доступ из Servlet и jsp. Передача данных из сервлета в jsp. Пример.

Сеанс (сессия) – соединение между клиентом и сервером, устанавливаемое на определенное время, за которое клиент может отправить на сервер сколько угодно запросов. Сеанс устанавливается непосредственно между клиентом и Web-сервером. Каждый клиент устанавливает с сервером свой собственный сеанс.

Сеансы используются для обеспечения хранения данных во время нескольких запросов Web-страницы или на обработку информации, введенной в пользовательскую форму в результате нескольких HTTP-соединений (например, клиент совершает несколько покупок в интернет-магазине; студент отвечает на несколько тестов в системе дистанционного обучения).

Чтобы открыть новый сеанс, используется метод **getSession()** интерфейса **HttpServletRequest**. Метод извлекает из переданного в сервлет запроса объект сессии класса **HttpSession**, соответствующий данному пользователю. Сессия содержит информацию о дате и времени создания и последнего обращения к сессии, которая может быть извлечена с помощью методов

getCreationTime() и **getLastAccessedTime()**.

Если для метода **getSession(boolean param)** входной параметр равен **true**, то сервлет-контейнер проверяет наличие активного сеанса, установленного с данным клиентом. В случае успеха метод возвращает дескриптор этого сеанса. В противном случае метод устанавливает новый сеанс:

HttpSession se = request.getSession(true);

после чего начинается сбор информации о клиенте.

Чтобы сохранить значения переменной в текущем сеансе, используется метод **setAttribute()** класса **HttpSession**, прочесть – **getAttribute()**, удалить – **removeAttribute()**. Список имен всех переменных, сохраненных в текущем сеансе, можно получить, используя метод **Enumeration getAttributeNames()**, работающий так же, как и соответствующий метод интерфейса **HttpServletRequest**.

Метод **String getId()** возвращает уникальный идентификатор, который получает каждый сеанс при создании. Метод **isNew()** возвращает **false** для уже существующего сеанса и **true** – для только что созданного.

Завершить сеанс можно методом **invalidate()**. Сеанс уничтожает все связи с объектами, и данные, сохраненные в старом сеансе, будут потеряны для всех приложений.

Для хранения информации на компьютере клиента используются возможности класса **Cookie**.

Cookie – это небольшие блоки текстовой информации, которые сервер посылает клиенту для сохранения в файлах cookies. Клиент может запретить браузеру прием файлов cookies. Браузер возвращает информацию обратно на сервер как часть заголовка HTTP, когда клиент повторно заходит на тот же Web-ресурс. Cookies могут быть ассоциированы не только с сервером, но и также с доменом – в этом случае браузер посылает их на все серверы указанного домена. Этот принцип лежит в основе одного из протоколов обеспечения единой идентификации пользователя (Single Signon), где серверы одного домена обмениваются идентификационными маркерами (token) с помощью общих cookies.

Чтобы послать cookie клиенту, сервлет должен создать объект класса **Cookie**, указав конструктору имя и значение блока, и добавить их в объект-response. Конструктор использует имя блока в качестве первого параметра, а его значение – в качестве второго.

Cookie cookie = new Cookie("myid", "007");
response.addCookie(cookie);

Извлечь информацию cookie из запроса можно с помощью метода **getCookies()** объекта **HttpServletRequest**, который возвращает массив объектов, составляющих этот файл.

Cookie[] cookies = request.getCookies();

После этого для каждого объекта класса **Cookie** можно вызвать метод **getValue()**, который возвращает строку **String** с содержимым блока cookie. В данном случае этот метод вернет значение "007".

Объект **Cookie** имеет целый ряд параметров: путь, домен, номер версии, время жизни, комментарий. Одним из важнейших является срок жизни в секундах от момента первой отправки клиенту. Если параметр не указан, то cookie существует только до момента первого закрытия браузера. Для запуска следующего приложения можно использовать сервлет из примера # 1 этой главы, вставив в метод **performTask()**следующий код:

CookieWork.setCookie(resp); *// добавление cookie*
CookieWork.printToBrowser(resp, req); *// извлечение cookie*

request

Это объект HttpServletRequest, связанный с запросом, который позволяет вам обращаться к параметрам запроса (через метод getParameter), типу запроса (GET, POST, HEAD, и т.д..), и входящим HTTP заголовкам (cookies, Referer, и т.д..). Проще говоря, request является подклассом ServletRequest и может отличаться от HttpServletRequest если используется протокол отличный от HTTP, что на практике практически никогда не встречается.

session

Это объект типа HttpSession, связанный с запросом. Сессии создаются автоматически, и эта переменная существует даже если нет ссылок на входящие сессии. Единственным исключением является ситуация, когда вы отключаете использование сессий используя атрибут session директивы page. В этом случае ссылки на переменную session приводят к возникновению ошибок при трансляции JSP страницы в сервлет.

11. Библиотеки тегов. JSP, JSTL. Пример использования. Порядок генерации java-кода из jsp со сложными тегами. Пользовательские теги. Создание, регистрация, Использование. Пример пользовательского тега.

Библиотека тегов представляет собой *java-классы*, реализующие определенную бизнес-логику в соответствии с *Tag Libraries Interface* (интерфейс библиотеки тегов). Структура тегов напоминает структуру сервлетов, которые могут быть многократно использованы в течение цикла жизни тегов. Библиотека тегов включает один или несколько тегов-классов и XML-дескриптор, содержащий описание тегов и параметров, используемых тегами. Код java-класса, оформленный в виде тега для выполнения определенных действий, скрывает от пользователя набор операций, определяющих его функциональность. При использовании библиотеки тегов разработка *java-кода* на странице JSP заключается в описании пользовательских дескрипторов и определением необходимых атрибутов тега *(java-класса)*.

При компиляции страницы JSP в сервлет пользовательские теги преобразуются в действия над объектами серверной стороны. Интерфейсы и классы, с помощью которых создаются пользовательские теги, располагаются в пакете *javax.servlet.jsp.tagext*.

JSP (JavaServerPages) — технология, позволяющая веб-разработчикам создавать содержимое, которое имеет как статические, так и динамические компоненты. Страница JSP содержит текст двух типов: статические исходные данные, которые могут быть оформлены в одном из текстовых форматов HTML, SVG, WML, или XML, и JSP-элементы,которые конструируют динамическое содержимое. Кроме этого могут использоваться библиотеки JSP-тегов, а также EL (Expression Language), для внедрения Java-кода в статичное содержимое JSP-страниц. Код JSP-страницы транслируется в Java-код сервлета с помощью компилятора JSP-страниц Jasper, и затем компилируется в байт-код виртуальной машины java (JVM). Контейнеры сервлетов, способные исполнять JSP-страницы, написаны на платформонезависимом языке Java. JSP-страницы загружаются на сервере и управляются из структуры специального Java server packet, который называется Java EE Web Application.

Обычно страницы упакованы в файловые архивы .war и .ear. JSP является платформонезависимой, переносимой и легко расширяемой технологией для разработки веб-приложений

```
<!doctype html public "-//w3c//dtd html 4.0 transitional//en">
  <%@ page contentType="text/html; charset=windows-1251" %>
  <%@ page import="java.util.*, java.text.*" %>
  <html>
      <head>
            <title>Простейшая страница JSP</title>
            <meta http-equiv="Content-Type" content="text/html;
charset=windows-1251">
      </head>
      <body>
            Добро пожаловать! Сегодня <%= getFormattedDate () %>
      </body>
  </html>
  <%!
      String getFormattedDate ()
      {
            SimpleDateFormat sdf = new SimpleDateFormat ("dd.MM.yyyy
hh:mm:ss");
            return sdf.format (new Date ());
      }
  %>
```

JSTL – стандартная библиотека тегов JSP.

Подключается следующим образом:

```
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
```

Пример:

```
<c:if test="${10 > 9}">
   <p>True<p>
</c:if>
```

Тэги JSTL 1.1 делятся на 4 категории:

1. Основные:
   - Общего назначения: <c:out>, <c:set>, <c:remove>, <c:catch>
   - Условия: <c:if>, <c:choose>, <c:when>, <c:otherwise>
   - Относящиеся к URL: <c:import>, <c:url>, <c:redirect>, <c:param>
   - Итерации: <c:forEach>, <c:forTokens>
2. Библиотеки форматирования:
   - Интернационализация: <fmt:message>, <fmt:setLocale>, <fmt:bundle>, <fmt:setBundle>, <fmt:param>, <fmt:requestEncoding>
   - Форматирование: <fmt:timeZone>, <fmt:setTimeZone>, <fmt:formatNumber>, <fmt:parseNumber>, <fmt:parseDate>
3. Библиотеки «SQL»:
   - Работа с БД: <sql:query>, <sql:update>, <sql:setDataSource>, <sql:param>, <sql:dateParam>
4. Библиотеки XML:
   - Общие XML операции: <x:parse>, <x:out>, <x:set>
   - XML-контроль: <x:if>, <x:choose>, <x:when>, <x:otherwise>, <x:forEach>
   - Трансформации: <x:transform>, <x:param>

Пользовательский тег представляет собой определенный пользователем элемент JSP-языка. Когда JSP-страница, содержащая пользовательский тег, транслируется в сервлет, тег преобразовывается в операции над объектом, называемым обработчиком тега. Затем Web-контейнер вызывает эти операции во время выполнения сервлета JSP-страницы.

Пользовательские теги могут:

- Быть настроенными при помощи атрибутов, переданных из вызывающей страницы.
- Обращаться ко всем объектам, доступным JSP-странице.
- Изменять ответ, генерируемый вызывающей страницей.
- Взаимодействовать между собой. Вы можете создать и инициализировать компонент JavaBeans, создать переменную, обращающуюся к этому компоненту в одном теге, и затем использовать этот компонент в другом теге.
- Быть вложенными, разрешая сложные взаимодействия в JSP-странице.
- Пользовательские теги JSP пишутся с использованием синтаксиса XML. Они имеют начальный тег, конечный тег и, возможно, тело:

```
<tt:tag>
   body
</tt:tag>
```

Пользовательский тег без тела: `<tt:tag />`

Простой тег не содержит тела и атрибутов: `<tt:simple />`

Для объявления, что JSP-страница будет использовать теги, определенные в библиотеке тегов, при помощи помещения директивы taglib в страницу перед использованием какого-либо пользовательского тега: `<%@ taglib uri="/WEB-INF/tutorial-template.tld" prefix="tt" %>`

Для определения тега необходимо:

- Разработать обработчик тега и вспомогательный класс для тега
- Объявить тег в дескрипторе библиотеки тегов

Обработчик тега представляет собой объект, вызываемый Web-контейнером для вычисления пользовательского тега во время выполнения JSP-страницы, которая ссылается на тег. Обработчики тегов должны реализовывать интерфейсы либо Tag, либо BodyTag. Интерфейсы могут использоваться для существующего объекта Java, чтобы преобразовать его в обработчик тега. Обработчик тега имеет доступ к API, позволяющему взаимодействовать с JSP-страницей.

Если тег является вложенным, обработчик тега имеет также доступ к обработчику (называемому родитель), связанному с внешним тегом.

Дескриптор библиотеки тегов (TLD) представляет собой XML-документ, описывающий библиотеку тегов. TLD содержит общую информацию и информацию о каждом теге, содержащемся в библиотеке.

Обработчик для простого тега должен реализовывать методы doStartTag и doEndTag интерфейса Tag. Метод doStartTag вызывается при обработке начального тега. Этот метод возвращает SKIP_BODY, поскольку простой тег не имеет тела. Метод doEndTag вызывается при обработке конечного тега. Метод doEndTag должен возвращать EVAL_PAGE, если оставшаяся часть страницы должна быть вычислена; в противном случае, он должен возвращать SKIP_PAGE.

Простой тег `<tt:simple />` может быть реализован следующим обработчиком тега:

```
public SimpleTag extends TagSupport {
   public int doStartTag() throws JspException {
     try {
     pageContext.getOut().print("Hello.");
     } catch (Exception ex) {
     throw new JspTagException("SimpleTag: " +
            ex.getMessage());
     }
     return SKIP_BODY;
   }
   public int doEndTag() {
     return EVAL_PAGE;
   }
}
```

12. JPA. Принципы. Реализации. Hibernate. Пример класса и связанной таблицы.

JPA (Java Persistence API) это спецификация Java EE и Java SE, описывающая систему управления сохранением java объектов в таблицы реляционных баз данных в удобном виде. Сама Java не содержит реализации JPA, однако существует много реализаций данной спецификации от разных компаний (открытых и нет). Это не единственный способ сохранения java объектов в базы данных (ORM систем), но один из самых популярных в Java мире.

У JPA существуют разные реализации:

- Hibernate
- Oracle TopLink
- Apache OpenJPA

Hibernate - это механизм отображения в реляционной базе данных объектов java. Hibernate не только заботится об отражении классов Java в таблицы БД (и типов данных Java в типы данных SQL), но также обеспечивает запрос данных и поисковые средства и может значительно сократить время разработки которое тратится на ручное написание SQL и JDBC кода.

```java
package a1s.learn.dao;
import a1s.learn.domain.User;
import java.util.List;
import javax.persistence.EntityManager;
import
javax.persistence.PersistenceContext;
import
org.springframework.stereotype.Repository;

@Repository
public class UserDAO implements
BaseDAO<User> {
    @PersistenceContext
    protected EntityManager emf;

    public List<User> findAll(){
        return emf.createQuery("from Users
u").getResultList();
    }
}
```

```java
package a1s.learn;

import a1s.learn.dao.UserDAO;
import a1s.learn.domain.User;
import java.util.List;
import
org.springframework.context.Application
ext;
import
org.springframework.context.support.Cla
thXmlApplicationContext;

public class App
{
    public static void main( String[] a
)
    {
        ApplicationContext ctx =
(ApplicationContext)new
ClassPathXmlApplicationContext("app-
context.xml");

        UserDAO
dao=(UserDAO)ctx.getBean("userDao");
        List<User> users = dao.findAll(

        for(User u:users) {

System.out.println(u.toString());
        }
    }
}
```

13. Hibernate. Аннотации. Обработка связей Один-к-Одному, Один-ко-Многим, Многие-к-Одному и Многие-ко-Многим. Пример добавления записи в таблицу, изменение записи в таблице. (хз где взять)

Hibernate — библиотека для языка программирования Java, предназначенная для решения задач объектно-реляционного отображения (object-relational mapping — ORM). Она представляет собой свободное программное обеспечение с открытым исходным кодом (open source), распространяемое на условиях GNU Lesser General Public License. Данная библиотека предоставляет легкий в использовании каркас (фреймворк) для отображения объектно-ориентированной модели данных в традиционные реляционные базы данных.

Целью Hibernate является освобождение разработчика от значительного объёма сравнительно низкоуровневого программирования по обеспечению хранения объектов в реляционной базе данных. Разработчик может использовать Hibernate как в процессе проектирования системы классов и таблиц «с нуля», так и для работы с уже существующей базой данных.

Hibernate не только решает задачу связи классов Java с таблицами базы данных (и типов данных Java с типами данных SQL), но и также предоставляет средства для автоматической генерации и обновления набора таблиц, построения запросов и обработки полученных данных и может значительно уменьшить время разработки, которое обычно тратится на ручное написание SQL- и JDBC-кода. Hibernate автоматизирует генерацию SQL-запросов и освобождает разработчика от ручной обработки результирующего набора данных и преобразования объектов, максимально облегчая перенос (портирование) приложения на любые базы данных SQL.

Hibernate обеспечивает прозрачную поддержку сохранности данных (persistence) для «POJO» (то есть для стандартных Java-объектов); единственное строгое требование для сохраняемого класса — наличие конструктора по умолчанию (без параметров). Для корректного поведения в некоторых приложениях требуется также уделить внимание методам *equals()* и *hashCode()*.

Аннотации Hibernate	
@Audited	org.hibernate.envers.Audited
@NotAudited	org.hibernate.envers.NotAudited
@Type	org.hibernate.annotations.Type

@Audited — Включает аудит сущности (отслеживание версий).

@NotAudited — указывает, что изменения этого свойства отслеживать не нужно. По умолчанию Hibernate пытается отслеживать изменения в ассоциациях.

@Type указывает на тип свойства

При **связи один-к-одному** каждая запись в одной таблице напрямую связана с отдельной записью в другой таблице. Для того, чтобы связать сущности отношением один-к-одному в Hibernate используется аннотация @OneToOne. В целом, может быть 3 варианта ее использования:

- связанные сущности используют одно и тоже значение первичного ключа;

- внешний ключ определяется полем одной из сущностей (это поле в БД должно быть уникальным для имитации отношения один-к-одному);
- используется таблица для хранения ссылки между двумя сущностями (ограничение уникальности должно быть установлено на каждом из полей для того, чтобы соответствовать кратности один-к-одному).

Один ко многим

Ассоциация один-ко-многим возникает тогда, когда каждой записи в таблице А, соответствует множество записей в таблице Б, но каждая запись в таблице Б имеет лишь одну соответствующую запись в таблице А.

Связь **многие-ко-многим** используется, когда записи из таблицы А может соответствовать 0 или более записей в таблице Б и наоборот для каждой записи из Б есть 0 или более записей таблицы А.

@ManyToMany

Многие к одному

аннотация @ManyToOne

14.Hibernate. Использование HQL & нативного SQL. Создание запросов на чтение данных из таблиц. Criteria. Пример запроса чтения данных с условием.

Hibernate — библиотека для языка программирования Java, предназначенная для решения задач объектно-реляционного отображения (object-relational mapping — ORM). Она представляет собой свободное программное обеспечение с открытым исходным кодом (open source), распространяемое на условиях GNU Lesser General Public License. Данная библиотека предоставляет легкий в использовании каркас (фреймворк) для отображения объектно-ориентированной модели данных в традиционные реляционные базы данных. Hibernate совместима с JSR-220/317 и предоставляет стандартные средства JPA.

HQL (Hibernate Query Language) – это объектно-ориентированный язык запросов, который крайне похож на SQL.

Отличие между HQL и SQL состоит в том, что SQL работает с таблицами в базе данных и их столбцами, а HQL – с сохраняемыми объектами (Persistent Objects) и их полями (аттрибутами класса).

Hibernate трнаслирует HQL – запросы в понятные для БД SQL – запросы, которые и выполняют необходимые нам действия в БД.

Мы также имеем возможность использовать обычные SQL – запросы в Hibernate используя Native SQL, но использование HQL является более предпочтительным.

Ключевые слова языка HQL:

FROM

Если мы хотим загрузить в память наши сохраняемые объекты, то мы будем использовать ключевое слово FROM. Вот пример его использования:

```
Query query = session.createQuery("FROM Developer");
List developers = query.list();
```

Developer – это POJO – класс Developer.java, который ассоциргван с таблицей в шагей БД.

INSERT

Мы используем ключевое слово INSERT, в том случае, если хотим добавить запись в таблицу нашей БД.

Вот пример использования этого ключевого слова:

```
Query query = session.createQuery("INSERT INTO Developer
(firstName,
lastName, specialty, experience)");
```

UPDATE

Ключевое слово UPDATE используется для обновления одного или нескольких полей объекта.

Вот так это выглядит на практике:

```
Query query = session.createQuery(UPDATE Developer SET
experience =: experience WHERE id =: developerId);
query.setParameter("expericence", 3);
```

DELETE

Это ключевоеслово используется для удаления одного или нескольких объектов. Пример использования:

```
Query query = session.createQuery("DELETE FROM Developer
WHERE id = :developerId");
query.setParameter("developerId", 1);
```

SELECT

Если мы хотим получить запись из таблицы нашей БД, то мы должны использоваь ключевое слово SELECT. Пример использования:

```
Query query = session.createQuery("SELECT D.lastName FROM
Developer D");
List developers = query.list();
```

ORDER BY

Для того, чтобы отсортировать список объектов, полученных в результате запроса, мы должны применить ключевое слово ORDER BY. Нам необходимо указать параметр, по которому список будет отсортирован и тип сортировки – по возрастанию (ASC) или по убыванию (DESC). В виде кода это выгллядит так:

```
Query query = session.createQuery("FROM Developer D WHERE
experience > 3 ORDER BY D.experience DESC");
```

Методы агрегации

Язык запросов Hibernate (HQL) поддерживает различные методы агрегации, которые доступны и в SQL. HQL поддерживает следующие методы:

- Минимальное значение данного свойства. min(имя свойства)
- Максимальное значение данного свойства. max(имя свойства)
- Сумма всех значений данного свойства. sum(имя свойства)
- Среднее арифметическое всех значений данного свойства. avg(имя свойства)
- Какое количество раз данное свойство встречается в результате. count(имя свойства)

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.

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

Вот некоторые области применения Criteria API:

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

Создаем запрос, который звучит как: вытащить все данные из ContactEntity, где дата соответствует промежутку startDate — endDate и отсортировать результат по дате. (Criteria)

```
private static List getNameCriteria(Calendar startDate, Calendar endDate,
Session session) {
        Criteria criteria = session.createCriteria(ContactEntity.class);
        if(startDate != null) {
            criteria.add(Restrictions.ge("birthDate", startDate.getTime()));
            /*@DEPRECATED
            criteria.add(Expression.ge("birth_date", startDate.getTime()));
            */
        }
        if(endDate != null) {
            criteria.add(Restrictions.le("birthDate", endDate.getTime()));
        }
        criteria.addOrder(Order.asc("birthDate"));
        return criteria.list();
    }
```

15. JDBC connection. Регистрация драйвера. Создание Connection к базе данных. DriverManager. Обеспечение транзакционности. Пример создания Connection и вставки данных в таблицу.

JDBC - это прикладной программный интерфейс (API) Java для выполнения SQL-запросов.

JDBC предоставляет стандартный API для разработчиков, использующих базы данных.

Компоненты JDBC

- Driver Manager
  - ★ предоставляет средства для управления набором драйверов баз данных;
  - ★ предназначен для выбора базы данных и создания соединения с БД.
- Драйвер
  - ★ обеспечивает реализацию общих интерфейсов для конкретной СУБД
  - ★ и конкретных протоколов.
- Соединение (Connection)
  - ★ Сессия между приложением и драйвером базы данных.
- Запрос
  - ★ SQL запрос на выборку или изменение данных.
- Результат
  - ★ Логическое множество строк и столбцов таблицы базы данных.
- Метаданные
  - ★ Сведения о полученном результате и об используемой базе данных.

Использование JDBC

Последовательность действий:

1. Загрузка класса драйвера базы данных.
2. Установка соединения с БД.
3. Создание объекта для передачи запросов.
4. Выполнение запроса.
5. Обработка результатов выполнения запроса.
6. Закрытие соединения.

Загрузка драйвера базы данных:

- В общем виде: Class.forName([location of driver]);
- Для MySQL: Class.forName("org.gjt.mm.mysql.Driver");

Согласно принятому соглашению классы JDBC-драйверов регистрируют себя сами при помощи Driver Manager во время своей первой загрузки.

Установка соединения с базой данных:

Объект Connection представляет собой соединение с БД. Сессия соединения включает в себя выполняемые SQL-запросы и возвращаемые через соединение результаты.

Приложение может открыть одно или более соединений с одной или несколькими БД.

Класс DriverManager содержит список зарегистрированных классов Driver и обеспечивает управление ими, и при вызове метода getConnection он проверяет каждый драйвер и ищет среди них тот, который "умеет" соединятся с БД, указанной в URL. Метод connect() драйвера использует этот URL для установления соединения.

Вызов метода DriverManager.getConnection(…) - стандартный способ получения соединения Методу передается строка, содержащая "URL". Класс DriverManager пытается найти драйвер, который может соединиться к БД с помощью данного URL.

Пример создания соединения и вставки данных:

```
public class ExecuteUpdateToDBExample {
  public static void main(String[] args) {
  Connection con = null;
  Statement st = null;
  ResultSet rs = null;
  try {
      Class.forName("org.gjt.mm.mysql.Driver");
      con = DriverManager.getConnection(
    "jdbc:mysql://127.0.0.1/test\","root", "123456");
      st = con.createStatement();
      int countRows = st.executeUpdate(
"INSERT INTO students (name, id_group) VALUES (\"Баба-Яга\",123456)");
      System.out.println(countRows);
  } catch (SQLException ex) {
            System.out.println(ex.toString());
      } finally {
            if (rs != null ){rs.close();}
            if (st != null){ st.close();}
            if (con != null) {con.close();}
      }
  }
}
```

Транзакционность:

Необходимо заключить код соединения и выполнения запросов как показано в примере:

```
dbConnection.setAutoCommit(false);
...
//код соединения и выполнения запросов
...
dbConnection.commit();
```

16. JDBC connection. Работа с Объектом типа Statment, PreparedStatment и CallableStatment. Пример изменения записи в таблице и вызова хранимой процедуры.

<u>Про JDBC смотреть 15 вопрос!!!</u>

В JDBC есть три класса для отправления SQL-запросов в БД и три метода в интерфейсе Connection определяют экземпляры этих классов:

- <u>Statement</u>: создается методом createStatement. Объект Statement используется при простых SQL-запросах (без параметров).
- <u>PreparedStatement</u>: Метод prepareStatement используется для SQL-выражений с одним или более входным (IN-) параметром простых SQL-выражений, которые исполняются часто. Для компиляции SQL запроса, в котором отсутствуют конкретные значения, используется метод prepareStatement(String sql), возвращающий объект PreparedStatement. Подстановка реальных значений происходит с помощью методов setString(), setInt() и подобных им. Выполнение запроса производится методами executeUpdate(), executeQuery(). PreparedStatement - оператор предварительно откомпилирован, поэтому он выполняется быстрее обычных операторов ему соответствующих.
- <u>CallableStatement</u>: создается методом prepareCall. Объекты CallableStatement используются для выполнения т.н. хранимых процедур - именованных групп SQL-запросов, наподобие вызова подпрограммы.

Пример(вызов хранимой процедуры):

```
Connection con = null;
CallableStatement cs = null;
ResultSet rs = null;
try {
   Class.forName("org.gjt.mm.mysql.Driver");
   con = DriverManager.getConnection(
     "jdbc:mysql://127.0.0.1/test","root", "123456");
     //вызов хранимой процедуры
   cs = con.prepareCall("{call calcstudents(?)}");
   cs.registerOutParameter(1,java.sql.Types.INTEGER);
   cs.execute();
   int empName = cs.getInt(1);
System.out.println("Count students - " + empName);
     // запись данных
    String sql = "INSERT INTO students(name,id_group)
VALUES(?,?)";
    PreparedStatement ps = con.prepareStatement(sql);
    ps.setString(1, "Кукушнин");
    ps.setInt(2, 851001);
    ps.executeUpdate();
} …
```

17. JDBC connection. Обработка результата вызова sql запроса. ResultSet. Изменение данных через ResultSet. Пример.

Про JDBC смотреть 15 вопрос!!!

Метод executeQuery возвращает объект типа ResultSet с построчными результатами выполнения запроса.

```
ResultSet rs = stat.executeQuery("SELECT * FROM Books");
```

Для построчного анализа результатов выполнения запроса используется приведенный ниже цикл.

```
while (rs.next()){}
```

При обработке отдельной строки нужно с помощью специальных методов получить содержимое каждого столбца.

```
String isbn = rs.getString(1);
float price = rs.getDouble("Price");
```

Для каждого типа данных языка Java предусмотрен отдельный метод извлечения данных, например getString и getDouble. Каждый из них имеет два способа представления, основанных на числовом и строковом типе аргумента. При использовании числового аргумента метод извлечет данные из столбца с указанным аргументом-номером. Например, метод rs.getString(1) возвратит значение из первого столбца текущей строки. При использовании строкового аргумента метод извлечет данные из столбца с указанным аргументом-именем. Например, метод rs.getDouble("Price") возвратит значение из столбца с именем Price. Первый способ на основе числового аргумента более эффективен, но строковые аргументы позволяют создать более читабельный и простой для сопровождения код. Каждый метод извлечения данных выполняет преобразование типа, если указанный тип не соответствует фактическому типу.

Методы, используемые для обновления данных:

- void updateRow() - Обновляет текущую строку объекта ResultSet и базовую таблицу базы данных;
- void insertRow() Обновляет текущую строку объекта ResultSet и базовую таблицу базы данных;
- void updateString() Обновляет специфицированный столбец, заданный строковым значением;
- void updateInt() Обновляет специфицированный столбец, заданный целым значением.

Пример:

```
String query = "SELECT * FROM Books";
ResultSet rs = stat.executeQuery(query);
while (rs.next()){
  if (...){
    double increase = 10.2;
    double price = rs.getDouble("Price");
    rs.updateDouble("Price", price + increase);
    rs.updateRow();
  }
```

```
}
```

18. Spring. Компоненты фреймворка. Использование JDBCTemplate для доступа к базе. Пример.

Spring Framework (или коротко Spring) — универсальный фреймворк с открытым исходным кодом для Java-платформы.

Spring Framework предоставляет бóльшую свободу Java-разработчикам в проектировании; кроме того, он предоставляет хорошо документированные и лёгкие в использовании средства решения проблем, возникающих при создании приложений корпоративного масштаба.

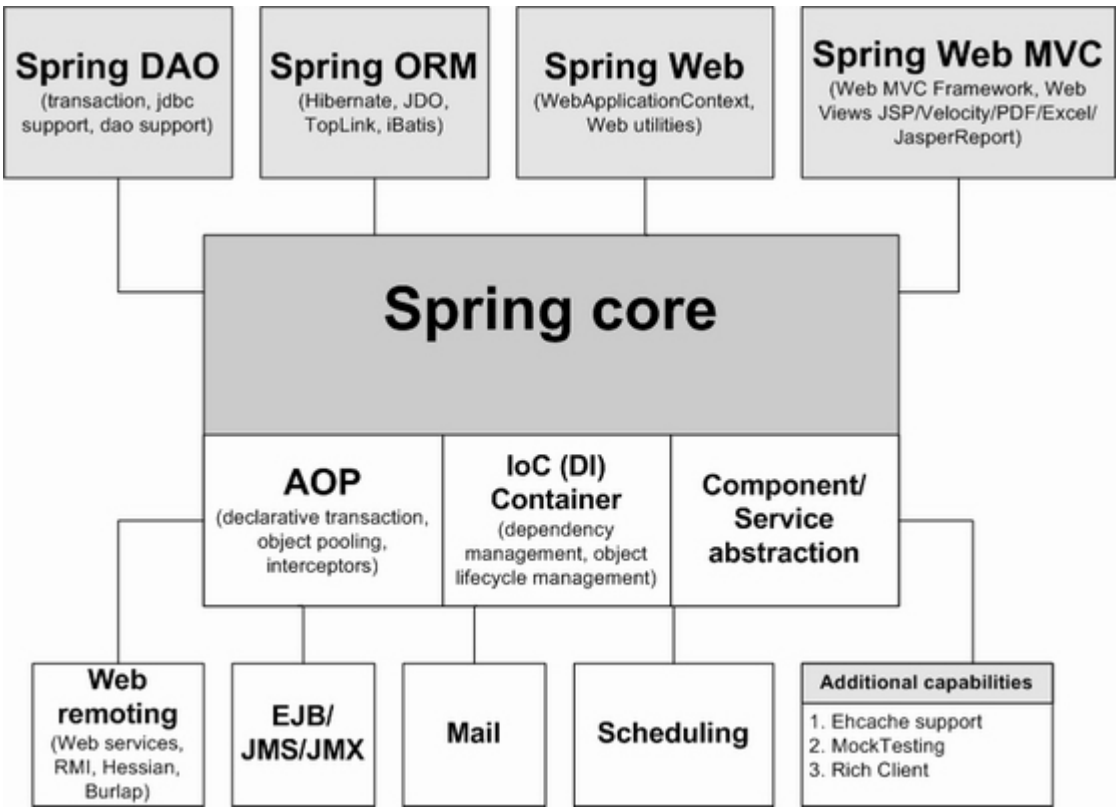
Между тем, особенности ядра Spring Framework применимы в любом Java-приложении, и существует множество расширений и усовершенствований для построения веб-приложений на Java Enterprise платформе. По этим причинам Spring приобрёл большую популярность и признаётся разработчиками как стратегически важный фреймворк.

Spring Framework может быть рассмотрен как коллекция меньших фреймворков или фреймворков во фреймворке. Большинство этих фреймворков может работать независимо друг от друга, однако они обеспечивают большую функциональность при совместном их использовании. Эти фреймворки делятся на структурные элементы типовых комплексных приложений:

1)Spring Core — основа ядра фрейморка

2)Spring DATA — фреймворк способный облегчить работу с базами данных
3)Spring MVC — мощный фреймворк, использующий шаблон проектирования model-view-contoller
4)Spring Web Flow — разработка веб-приложений с сложной навигацией
5)Spring Security — защита и контроль доступа в приложении

Архитектура Spring представлена следующей схемой


Spring DAO
(transaction, jdbc support, dao support)
Spring ORM
(Hibernate, JDO, TopLink, iBatis)
Spring Web
(WebApplicationContext, Web utilities)
Spring Web MVC
(Web MVC Framework, Web Views JSP/Velocity/PDF/Excel/ JasperReport)
Spring core
AOP
(declarative transaction, object pooling, interceptors)
IoC (DI) Container
(dependency management, object lifecycle management)
Component/ Service abstraction
Web remoting
(Web services, RMI, Hessian, Burlap)
EJB/ JMS/JMX
Mail
Scheduling
Additional capabilities
1. Ehcache support
2. MockTesting
3. Rich Client

Ядро

1. IoC контейнер

В основе Spring лежит паттерн Inversion of control. Применительно к легковесным контейнерам, основная идея этого паттерна заключается в устранении зависимости компонентов или классов приложения от конкретных реализаций вспомогательных интерфейсов и делегировании полномочий по управлению созданием нужных реализаций IoC контейнеру.

Основные преимущества IoC контейнеров:

1. управление зависимостями
2. упрощение повторного использования классов или компонентов
3. упрощение unit-тестирования
4. более "чистый" код (Классы больше не занимаются инициализацией вспомогательных объектов. Не стоит, конечно "перегибать палку", управляя созданием абсолютно всех объектов через IoC. В IoC контейнер лучше всего выносить те интерфейсы, реализация которых может быть изменена в текущем проекте или в будущих проектах.)

Spring гораздо больше, чем просто IoC контейнер. Framework упрощает разработку J2EE проектов, реализуя низкоуровневые и наиболее часто используемые части корпоративного приложения.

Концепция, лежащая в основе инверсии управления, часто выражается "голливудским принципом": "Не звоните мне, я вам сам позвоню". IoC переносит ответственность за выполнение действий с кода приложения на фреймворк. В отношении конфигурирования это означает, что если в традиционных контейнерных архитектурах наподобие EJB, компонент может вызвать контейнер и спросить: "где объект X, нужный мне для работы?", то в IoC сам контейнер выясняет, что компоненту нужен объект X, и предоставляет его компоненту во время исполнения. Контейнер делает это, основываясь на подписях методов (таких, как свойства JavaBean) и, возможно, конфигурационных данных в формате XML.

AOP

Одним из ключевых компонентов Spring является AOP framework. Основные задачи, которые решаются с помощью AOP в Spring:

1. Декларативное управление транзакциями.
2. Организация пулов объектов.
3. Написание собственных аспектов.

Наиболее удобный и гибкий способ управления транзакциями это декларативное управление транзакциями. Декларативное управление транзакциями избавляет код от зависимости от фреймворка или конкретного механизма управления транзакциями.

> Роль шаблона – убрать весь повторяющийся код, чтобы разработчик мог сосредоточиться на бизнес логике. Spring обрабатывает доступ к данным, используя шаблоны и callback. Код, такой как открытие и закрытие соединения, выполняется шаблоном, а переменная часть кода, обработка результата, например, обрабатывается callback.
>
> Spring поддерживает несколько шаблонов доступа к данным для различных механизмов сохранения:
>
> - JdbcTemplate
> - NamedParameterJdbcTemplate
> - SimpleJdbcTemplate
> - HibernateTemplate
> - JpaTemplatея

На данный момент мы ищем способ взаимодействия с базой через JDBC. Давайте сначала попробуем JdbcTemplate. Этот шаблон автоматически обрабатывает

управление ресурсами, обработку исключений и управление транзакциями. Теперь наша реализация класса для JdbcDAO выглядит так:

```
public class StudentJdbcDao implements StudentDao{
  private JdbcTemplate jdbcTemplate;
  public void setJdbcTemplate(JdbcTemplate jdbcTemplate){
     this.jdbcTemplate = jdbcTemplate;
  }
  public void saveStudent(Student student) {
     jdbcTemplate.update
              ("insert into STUDENT (name) values (?)",new Object[] {student.getName()} );
  }
}
```

вопрос теперь в том, как получить jdbcTemplate. Сделаем это при помощи Spring.

```
<bean id="jdbcTemplate" class="org.springframework.jdbc.core.JdbcTemplate">
      <property name="dataSource" ref="dataSource" />
</bean>
<bean id="dataSource" class="org.springframework.jdbc.datasource.DriverManagerDataSource"
   destroy-method="close">
  <property name="driverClassName" value="org.hsqldb.jdbcDriver" />
  <property name="url" value="jdbc:hsqldb:hsql://localhost" />
  <property name="username" value="sa" />
  <property name="password" value="" />
</bean>
```

19. Spring. Связывание объектов в Spring. Способы инициализации связанного объекта. Получение объекта. Синглтоны и сессионные объекты. Пример xml и java-кода.

Spring Framework (или коротко Spring) — универсальный фреймворк с открытым исходным кодом для Java-платформы.

Spring Framework предоставляет бóльшую свободу Java-разработчикам в проектировании; кроме того, он предоставляет хорошо документированные и лёгкие в использовании средства решения проблем, возникающих при создании приложений корпоративного масштаба.

Между тем, особенности ядра Spring Framework применимы в любом Java-приложении, и существует множество расширений и усовершенствований для построения веб-приложений на Java Enterprise платформе. По этим причинам Spring приобрёл большую популярность и признаётся разработчиками как стратегически важный фреймворк.

Spring Framework может быть рассмотрен как коллекция меньших фреймворков или фреймворков во фреймворке. Большинство этих фреймворков может работать независимо друг от друга, однако они обеспечивают большую функциональность при совместном их использовании. Эти фреймворки делятся на структурные элементы типовых комплексных приложений:

1)Spring Core — основа ядра фрейморка

2)Spring DATA — фреймворк способный облегчить работу с базами данных
3)Spring MVC — мощный фреймворк, использующий шаблон проектирования model-view-contoller
4)Spring Web Flow — разработка веб-приложений с сложной навигацией
5)Spring Security — защита и контроль доступа в приложении

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

Доступны два способа для получения основных объектов контейнера внутри бина:

- Реализовать один из Spring*Aware (ApplicationContextAware, ServletContextAware, ServletConfigAware и др.) интерфейсов.
- Использовать автоматическое связывание @Autowired в спринг. Способ работает внутри контейнера спринг.

```
@Autowired
ServletContext servletContext;
```

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

Существует четыре вида связывания в спринг:

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

20. Обзор Аспектно-ориентированного программирования. Подходы реализации АОП. JAspect & Spring AOP. Применение аспектов в Spring. Точки перехвата вызова метода. Описание правил перехвата методов. Пример.

Аспектно-ориентированное программирование (АОП) возникло как ответ на вопрос о том, как "правильно" декомпозировать большую программу на модули.

Термины АОП

- аспект (aspect);
- совет (advice);
- точка соединения (join point);
- срез (pointcut);
- внедрение (introduction).

```
// Аспект состоит из именованного среза и рекомендации.
aspect Logging {
  // Именованный срез задает соответствие точкам
  // соединения программы, вызовам методов.
  pointcut move():
    call(void FigureElement.setXY(int,int)) ||
    call(void Point.setX(int))              ||
    call(void Point.setY(int));
  // Рекомендация печатает сообщение на экран до
  // выполнения соответствующей данному именованному
  // срезу точки соединения программы.
  before(): move() {
    System.out.println("about to move");
  }
}
```

Одним из основных понятий аспектно-ориентированного программирования является **точка соединения** (англ. *join point*). Существуют различные определения данного понятия [3, 4, 8, 9, 14–16]. В данной статье под точкой соединения будет пониматься программная конструкция, которая может быть связана с описанием некоторой части сквозной функциональности программы. Типичными примерами точек соединения служат вызов функции и определение структуры.

Срез (англ. *pointcut*) – это описание набора точек соединения, логически объединенных по некоторому условию. Например, с помощью среза можно описать все вызовы функций выделения памяти (таких как *malloc, calloc* и т.д.).

Еще одним понятием АОП является **рекомендация** (англ. *advice*). Посредством рекомендации задается набор действий, которые должны быть выполнены для точек соединения, описанных срезом[3]. Например, для такой точки соединения, как вызов функции, в рекомендации могут быть записаны инструкции по выводу в журнал сообщения, содержащего значение, которое возвращает данная функция.

Посредством срезов и рекомендаций АОП позволяет выделить описание части сквозной функциональности программы в отдельные модули, так называемые **аспекты**[4] (англ. *aspect*).

Традиционно реализации АОП поддерживают следующие срезы, описывающие динамические события[8]:

- **call** – вызов функции;
- **execution** – выполнение функции, которое происходит после перехода управления из некоторой функции, вызывающей данную;
- **set** – присваивание значения переменной или полю;
- **get** – использование переменной или поля.

Стоит отметить, что традиционно АОП предоставляет возможность неявного задания точек соединения по их контексту, например:

- **infile** – все точки соединения из некоторого файла;
- **infunc** – все точки соединения из некоторой функции;
- **cflow** – все точки соединения, которые встречаются в контексте выполнения точек соединения другого среза.

AspectJ — аспектно-ориентированное расширение языка Java, созданное компанией PARC. Язык доступен в проектах Eclipse Foundation как отдельно, так и в составе среды разработки Eclipse.

AspectJ расширяет синтаксис Java, то есть все программы, написанные на Java будут корректными программами AspectJ, но не наоборот, так как могут включать специальные конструкции, называемые аспектами, которые могут содержать несколько частей, недоступных обычным классам.

- Методы расширения позволяют программисту добавлять методы, поля и интерфейсы в существующие классы. Данный пример добавляет метод acceptVisitor(см. шаблон проектирования посетитель) в код класса Point:

```
aspect VisitAspect {
    void Point.acceptVisitor(Visitor v) {
        v.visit(this);
    }
}
```

Spring AOP

- написан на чистом Java/C#;
- не использует сторонних компиляторов;
- урезанная поддержка АОП;
- интеграция с AspectJ;
- поддержка аннотаций @AspectJ;
- часть Spring Framework.

Spring AOP: Примеры

Ограничение прав доступа:

```
<sec:protect-pointcut
    expression="execution(* ru.novotelecom.xxx.Subscription.*(..))"
    access="ROLE_SUBSCRIPTION_MANAGER" />
```

Или

```
@Aspect
public class SecurityManager {

    @Before("execution(* ru.novotelecom.xxx.Subscription.*(..))")
    public void doAccessCheck() {... }
}
```

- Основная цель АОП — выноса «общей» (сквозной) функциональности «за скобки» (модуляризация сквозной функциональности);
- Для Java AOP доступен через проект AspectJ, для .NET – через PostSharp;
- Наиболее простая и проверенная реализация AOP – Spring AOP.

21. Реализация транзакционности в Spring. Реализация кеширования в Spring. Примеры.

Под **транзакцией** понимается ряд действий, которые воспринимаются системой, как единый пакет, т.е. или все действия проходят успешно, или все откатываются на исходные позиции.

Существует два вида транзакций: *глобальные и локальные.*

Глобальные транзакции позволяют работать с несколькими транзакционными ресурсами, как правило, реляционными базами данных и очередью сообщений. Сервер приложений управляет глобальными транзакциями через JTA, который является громоздким API(частично из-за его модели исключений) . Кроме того, JTA UserTransaction как правило должен быть получен из JNDI, а это означает, что также необходимо использовать JNDI для того, чтобы использовать JTA . Очевидно, что использование глобальных транзакций будет ограничивать любое потенциальное повторное использование кода приложения, так как JTA, как правило, доступен только на сервере приложений.

Локальные транзакции являются ресурсо-специфичными, такие как транзакции связанные с JDBC соединением. Недостатком локальных транзакций является то, что они работают только с одним источником. Например, код, который управляет транзакций с использованием соединения JDBC не может работать в рамках глобальной транзакции JTA. Поскольку сервер приложений не участвует в управлении транзакциями, он не может помочь обеспечить корректность по нескольким ресурсам. Другим недостатком является то, что локальные транзакции являются инвазивными к модели программирования.

Стратегия транзакций опред интерфейсом org.springframework.transaction.PlatformTransactionManager

Программист часто работает с СУБД типа HSQLDB или MySQL, установленные на его личном компьютере, и как правило запускает Tomcat в качестве сервера приложений (если конечно речь не идет о разработке EJB компонент). В этом случае для доступа к базе данных создается локальный экземпляр класса, реализующего интерфейс *DataSource*, и *DataSourceTransactionManager* управляет транзакциями, то есть конфигурационный файл имеет приблизительно следующий вид:

```
<bean id="dataSource"
    class="org.springframework.jdbc.datasource.DriverManagerDataSource">
    <property name="username" value="${jdbc.username}"/>
    <property name="password" value="${jdbc.password}" />
    <property name="driverClassName" value="${jdbc.driverClassName}" />
    <property name="url" value="${jdbc.url}" />
</bean>
<bean id="transactionManager"
    class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
    <property name="dataSource" ref="dataSource"/>
</bean>
```

В промышленной эксплуатации приложение как правило исполняется в полновесном сервере приложений (как например Oracle Weblogic), и там для получения *DataSource* и менеджера транзакций используется JNDI поиск и соответствующая часть конфигурационного файла принимает следующий вид:

```
<tx:jta-transaction-manager />
<jee:jndi-lookup id="dataSource" jndi-name="java:comp/env/jdbc/dataSource"/>
```

Разумеется эти две конфигурации не могут одновременно быть активными, поскольку уже только одинаковые идентификаторы управляемых компонент вызовут проблемы.

С версии 3.1, Spring Framework позволяет без особых усилий подключить к своему Java/Spring проекту кэширование.

Допустим, у нас есть такая сущность:

```
public class Transaction {

    private final long id;
    private final long amount;

    // constructor & getters
}
```

И следующий интерфейс для DAO:

```
public interface TransactionDao {

    Transaction getById(long id);
    void save(Transaction tx);
}
```

Очевидно, что поиск по ключу можно закэшировать. Для этого в Spring 3.1 появилась аннотация @Cacheable. Добавим реализацию этого интерфейса.

```
public class TransactionDaoImpl implements TransactionDao {
  private final List<Transaction> txs = new CopyOnWriteArrayList<>();
  @Override
  @Cacheable("transactions")
  public Transaction getById(long id) {
    // inefficient search imitation
    for (Transaction tx : txs) {
      if (tx.getId() == id) {
        return tx;
      }
    }
    return null;
  }
  @Override
  public void save(Transaction tx) {
    txs.add(tx);
  }
}
```

На метод getById(long id) была навешена аннотация @Cacheable. В качестве value в неё передаётся имя кэша, в который значение будет добавлено, и в котором будет осуществляться поиск до вызова метода. Если мы не хотим, чтобы перед вызовом Spring лез в кэш, а просто добавлял в него результат выполнения, необходимо использовать аннотацию @CachePut.

Далее, свяжем всё это при помощи Spring и включим кэширование. Я использую Java конфигурацию:

```
@Configuration
@EnableCaching
public class AppConfig {
  @Bean
```

```
    public TransactionDao transactionDao() {
        return new TransactionDaoImpl();
    }
    @Bean
    public CacheManager cacheManager() {
        SimpleCacheManager cacheManager = new SimpleCacheManager();
        cacheManager.setCaches(Collections.singleton(new ConcurrentMapCache("transactions")));
        return cacheManager;
    }
}
```

Для того, чтобы включить кэширование, нам надо добавить аннотацию @EnableCaching (XML аналог - <cache:annotation-driven>) и добавить инстанс CacheManager в контекст. Для примера, я подключил SimpleCacheManager, но в реальном приложении лучше использовать Ehcache (в Spring есть обёртки), тем более настраивается он также просто.

В main я просто сохраняю объекты, а затем пытаюсь забрать один (должен быть самый последний в коллекции) дважды. В первый раз после вызова метода он попадёт в кэш, во второй раз метод не должен быть вызван.

```
public static void main(String[] args) {
    ApplicationContext applicationContext = new AnnotationConfigApplicationContext(AppConfig.class);
    TransactionDao txDao = applicationContext.getBean(TransactionDao.class);
    for (int i = 0; i < 100000; i++) {  txDao.save(new Transaction(i, i + 1000));   }
    long start = System.nanoTime();
    txDao.getById(100000);
    long end = System.nanoTime();
    System.out.println("Search without cache: " + (end - start));
    start = System.nanoTime();
    txDao.getById(100000);
    end = System.nanoTime();
    System.out.println("Search from cache: " + (end - start));
}
```

Результат:

Search without cache: 9765142

Search from cache: 797937

Цифры различаются на порядок :).

В большинстве случаев, нам также необходимо этот кэш инвалидировать (удалять из него неактуальные значения). Вот простой пример. Добавим метод update(Transaction tx) в наш DAO:

```
public void update(Transaction tx) {
    for (int i = 0; i < txs.size(); i++) {
        if (txs.get(i).getId() == tx.getId()) {
            txs.set(i, tx);
        }
    }
}
```

В данном случае, если мы выполним код:

```
System.out.println("Amount of the tx: " + txDao.getById(1L).getAmount());
txDao.update(new Transaction(1, 5000L));
System.out.println("Amount of the tx: " + txDao.getById(1L).getAmount());
```

мы увидим, что значение не изменилось. Для решения таких проблем в Spring есть аннотация @CacheEvict(). Добавим её:

```
@CacheEvict(value = "transactions", allEntries = true)
public void update(Transaction tx) {...}
```

Как и в других Spring-Cache аннотациях, value - это имя кэша. Параметр allEntries "говорит" Spring, чтобы он обновил весь кэш. Получается не очень эффективно. Улучшить это можно следующим образом:

```
@Override
@CacheEvict(value = "transactions", key = "#tx.id")
public void update(Transaction tx) {
   for (int i = 0; i < txs.size(); i++) {
      if (txs.get(i).getId() == tx.getId()) {
         txs.set(i, tx);
      }
   }
}
```

В данном случае мы явно указываем ключ, по которому у нас осуществляется кэширование (этот же параметр можно использовать в @Cacheable и других аннотациях) и Spring обновит только один объект.

22. Struts2. Библиотеки фреймворка. Подключение Strats2 к проекту. Создание Action. Методы обработки validate() & exectue().Порядок регистрации и обработки сообщений и ошибок. Пример класса Action.

Struts2 - это фреймворк, ориентированный на действия (Action), которые являются основным элементом данного фреймворка, его ядром. Действия обрабатывают запросы пользователей и определяют результат его выполнения для представления информации.

Struts2 - назначает каждому действию свой уникальный URL и набор результатов. Фреймворк включает пять констант, которые определяют результат выполнения действия и которые можно использовать в приложении (константы) определены в классе ActionSupport) :

- public static final String ERROR "error";
- public static final String INPUT "input";
- public static final String LOGIN "login";
- public static final String NONE "none";
- public static final String SUCCESS "success".

Если этих констант будет недостаточно, то можно дополнительно определить любые свои.

Действие в Struts2 представляет простой Java-класс (Action-класс), который должен наследовать свойства класса org.apache.struts2.interceptor.ServletRequestAware. Класс **ServletRequestAware** содержит виртуальную функцию **execute()**, возвращающую строковое значение. Именно в этой функции определяется результат выполнения действия, или результат обработки запроса HttpServletRequest.

Действия в Struts2 выполняют функцию обработки и передачи данных для дальнейшего их представления в JSP-странице или других вариантах отображения страницы пользователю. Кроме того, действие указывает фреймворку какой результат необходимо отобразить пользователю.

Struts2, как и любой фреймворк, представляет из себя набор Java библиотек, которые необходимо добавить в WEB проект. Для работы приложения с фреймворком Struts2 необходимо подключить фильтр (FilterDispatcher), выполняющий роль контролера в MVC. Сама настройка приложения выполняется при помощи файлов web.xml, struts.xml и struts.properties.

Минимальный набор требуемых библиотек Struts2

- struts2-core*.jar - ядро Struts2;
- xwork-*.jar - фреймворк построен на базе XWork;
- ognl*.jar - используется в качестве языка выражений OGNL;
- freemarker*.jar - кастом теги JSP написаны на freemarker;
- commons-fileupload-*.jar - библиотека загрузки файлов;
- commons-io-*.jar - библиотека для работы с операциями ввода вывода;
- commons-logging-*.jar - Struts 2 использует commons logging API для протоколирования.

Если для работы WEB приложения появится необходимость работать с формой, ТО БЛЯТЬ НЕ ИСПОЛЬЗУЙТЕ ЕЕ то необходимо еще установить библиотеку commons-beanutil-*.jar, для работы с регулярными выражениями - oro-.*.jar. При работе с шаблонами страницы не обойтись без библиотек struts-tiles-*.jar, tiles-*.jar. Таким образом функциональные свойства WEB приложения определяют пакет используемых библиотек.

Все библиотеки фреймворков в WEB-приложениях располагаются в директории **WEB-INF/lib.**

Подключение FilterDispatcher к приложению

Библиотека Struts2подключается к WEB приложению В дескрипторе приложения web.xml

```
 <filter>
      <filter-name>struts2</filter-name>
      <filter-class>org.apache.struts2.dispatcher.FilterDispatcher</filter-class>
  </filter>

  <filter-mapping>
      <filter-name>struts2</filter-name>
      <url-pattern>/*</url-pattern>
  </filter-mapping>
```

FilterDispatcher перехватывает все обращения браузера клиента к серверу, подобно сервлетному фильтру

Как выполняется проверка данных в Struts2 ?

Фреймворк в начале обработки запроса пользователя просматривает список всех подключенных перехватчиков (интерцепторов). Если интерцептор проверки присутствует, то он вызывается первым. Для разработчиков необходимо только создать Action-класс, наследующий ActionSupport, и переопределить метод **validate**, в котором описать всю логику проверки. Метод **validate** определен в интерфейсе com.opensymphony.xwork2.Validateable, который реализуется классом ActionSupport. Реализованные механизмы класса **ActionSupport** позволяют отобразить результаты проверки пользователю.

```
public void validate ()
      {
            if ( isEmptyString ( name ))
                 addFieldError ( "name", "Не указано имя" );
            if ( isEmptyString ( login ))
                 addFieldError ( "login", "Не указан логин" );
            if ( isEmptyString ( password ))
                 addFieldError ( "password", "Не указан пароль" );
            if ( isEmptyString ( confirm ))
                 addFieldError ( "confirm", "Не указано подтверждение пароля"
);
            if ( !isEmptyString ( password ) && !isEmptyString ( confirm ))
            {
                 if(!password.equals(confirm)) {
                     addActionError ( "Не совпадают пароль и его
подтверждение" );
            }
      }
```

Основная функция метода **validate** в данном действии заключается в проверке введенных оператором данных. Здесь следует обратить внимание на два метода в **ActionSupport**, которые позволяют сообщать об ошибках. Это метод **addActionError**, который сообщает об ошибке при выполнении действия, и **addFieldError**, сообщающий об ошибке при заполнении определенного поля. Таким образом, если не заполнено имя пользователя, то логично вызвать addFieldError, а если, например, пароль и его

подтверждение не совпадают, то логичнее выдать сообщение об ошибке действия, так как нельзя понять, в каком именно поле мог ошибиться оператор.

```
package examples.greeting;

import com.opensymphony.xwork2.Action;
import com.opensymphony.xwork2.ActionSupport;

public class GreetingAction extends ActionSupport
{
    private String name;

    public String execute () throws Exception
    {
        if (( getName () != null ) && ( getName ().trim ().length () > 0 ))
            return Action.SUCCESS;
        else
            return Action.NONE;
    }

    public String getName
    {
        return name;
    }

    public void setName (String name)
    {
        this.name = name;
    }
}
```

Действие GreetingAction наследует все свойства класса **com.opensymphony.xwork2.ActionSupport** и переопределяет функцию **execute**. Как только будет обращение к действию GreetingAction, фреймворк Struts2 сразу же вызовет **execute**, которая проверит поле "name", и, в зависимости от состояния, вернет либо "success", либо "none".

Листинг JSP-страницы Entry.jsp

```
<%@ page language="java" contentType="text/html; charset=utf-8" pageEncoding="utf-8"%>
<%@ taglib prefix="s" uri="/struts-tags" %>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html>
  <head>
      <title>Приглашение</title>
  </head>

  <body>
    <h1>Приглашение пользователя</h1>

    Введите имя и нажмите кнопку "Войти"

    <s:form action="Greeting"

          <s:textfield label="Ваше имя" name="name"></s:textfield>

          <s:submit value="Войти"></s:submit>

    </s:form>
  </body>
</html>
```

На данной странице в строке <%@ taglib prefix="s" uri="/struts-tags" %> подключаются теги Struts2, после чего обращение к объектам фреймворка производится через префикс "s".

На странице несколько тегов Struts2. Первый тег - это **<s:form ... >**, который указывает какое действие будет отвечать за обработку данных. Далее установлено текстовое поле**<s:textfield ... >**, которое определено как "name". При нажатии на кнопку <s:submit ... > "Войти" Struts2 вызывает действие Greeting.

Листинг файла конфигурации struts.xml

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

<!DOCTYPE struts PUBLIC "-//Apache Software Foundation//DTD Struts Configuration 2.0//EN"
       "http://struts.apache.org/dtds/struts-2.0.dtd">

<struts>

    <package name="Default" namespace="/" extends="struts-default">

        <action name="Greeting" class="examples.greeting.GreetingAction">
           <result name="success">/Wellcome.jsp</result>
           <result name="none">/Empty.jsp</result>
        </action>

    </package>

</struts>
```

В файле конфигурации Struts2 определено, что действие Greeting связано с классом examples.greeting.GreetingAction. В зависимости от результата (result) выполнения Action-класса будет открыта либо страница Wellcome.jsp, либо Empty.jsp.

23. Strats2. Перенаправление обработки в jsp. Генерация ответа в формате JSON. Strats.xml. Понятие пакета в strats.xml. Interceptor. Порядок обработки и применение Interceptor. Пример класса Interceptor.

Actionы в Struts 2 управляют результатом обработки запроса к серверу. Этим результатом может быть, например, JSP страница или обычный html, JSON, XML и т.д. При получении запроса по какому-то URL в Actione вызывается определённый, заданный в struts.xml метод, к которому предъявляется следующий список требований:

- метод не имеет входных параметров;
- метод возвращает строковое значение.

После того, как action вернул строку, внутри тега <action>, описывающий Action, который был вызван, ищется тег <result> с атрибутом name, значение которого равно той строке, которую вернул Action. Если такого результата не было найдено, то поиск продолжается в глобальных результатах пакета, в котором тег <action> был определён. Как только такой <result> был найден, дальнейшая обработка зависит от типа отдаваемого контента.

В случае с jsp, строка помещённая внутри тега <result> используется как путь к jsp, которая отдаётся пользователю.

В случае возврата json, никаких строк внутри <result> не предполагается, а возвращается сериализованный в json объект Action, который был вызван в ответ на пользовательский запрос.

Именно возращённое строковое значение в совокупности с struts.xml и определяет что вернёт данный Action.

Дополнительная информация касательно генерации ответа в формате JSON:

1. Для возврата и принятия данных в формате JSON, необходимо подключить к проекту struts-json-plugin;
2. Теги <result>, которые предполагают возврат json, должны иметь аттрибут type, равный json (<result type=”json” />).
3. В случае когда нужно не только вернуть данные в формате JSON, но и принять, необходимо подключить interceptor с именем “json”.
4. Управлять данными, которые будут возвращены в ответ на запрос можно как используя ключевое слово вызвать Action прописанный внутри него, необходимо добавить transient на полях объектов, которые не должны быть сериализованы, так и с использованием тега <param> с именами includeProperties или excludeProperties, внутри последних задаётся регулярные выражения для определения полей объектов, которые должны быть включены в сериализованные данные или наоборот не должны соответственно.

Struts.xml - файл с настройками фреймворка. В нём также описываются Interceptorы, Actionы. Все Actionы распределены по пакетам.

Пример struts.xml:

```
<!DOCTYPE struts PUBLIC … >
<struts>
   <constant name="struts.devMode" value="true" />
   <package name="helloworld" extends="struts-default">
      <action name="hello"
            class="com.tutorialspoint.struts2.HelloWorldAction"
            method="execute">
            <result name="success">/HelloWorld.jsp</result>
      </action>
      <-- more actions can be listed here -->
   </package>
```

```
    <-- more packages can be listed here -->
</struts>
```

Тег <struts> - корневой тег, внутри которого описываются пакеты (тег <package). **Пакеты** - инструмент для обеспечения модульности при описании Actionов. Если ваш проект условно можно разделить на 3 части: бизнес часть, часть для персонала, API для мобильного приложения, вы можете создать соответственно 3 пакета и хранить Actionы внутри них. Тег <package> имеет следующие аттрибуты:

Attribute	**Description**
name (required)	Уникальный идентификатор пакета
extends	Указывает от какого пакета данный пакет унаследован
namesapce	Namespace пакета. Если указан namespace, то для того, чтобвить namespace в url к Actiony (типа localhost/namespace/action)

На каждый url сервера определяется тег <action>. Аттрибуты:

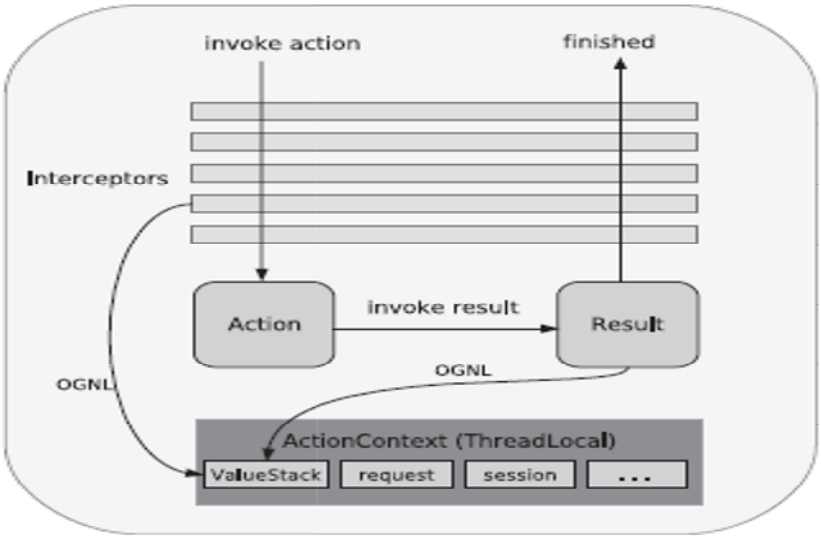
Attribute	**Description**
name (required)	Имя Actiona
class	Полное имя класса-Actiona, если не определён, то используется класс ActionSupport
method	Метод, который вызывается у класса-Actiona. Если не определён, ты вызывается метод execute.

Результат определяет что будет возвращено клиенту после того, как Action отработал. Action может иметь несколько Resultов. У Resulta есть атрибут name, который описан выше. В случае отсутствия считается, что name=”success”. Также имеется атрибут type. По стандарту он равен “default”.

Перехватчики могут выполнять огромный объем работы: разбор параметров запроса, загрузку исходных данных, обработку исключений, проверку корректности данных. Перехватчики являются для действий в Struts2 тем же, чем фильтры для сервлетов.

Перехватчики представляют собой Java-классы, наследующие (implements) свойства интерфейса **Interceptor**. Подключаются перехватчики к WEB-приложению на уровне файла конфигурации *struts.xml.*

На следующем рисунке представлена схема обработки запроса, поступившего от пользователя. FilterDispatcher уже выполнил свою миссию и передал управление фреймворку для выполнения соответствующего действия, к которому подключено несколько интерцепторов. Согласно данной структуре **Interceptors** включаются в процесс обработки перед выполнением действия. Потом управление передается действию, по окончанию которого управление вновь возвращается к интерцептору для завершения своих функций.



Для создания класса-интерсептора, этот класс должен реализовать interface Interceptor:

```
public interface Interceptor extends Serializable{
   void destroy();
   void init();
   String intercept(ActionInvocation invocation) throws Exception;
}
```

Метод init вызывается при создании intercepora, там должны быть прописано инициализация класса.

Метод Intercept вызывается при непосредственной обработки запроса. Один и тот же interceptor может использоваться в разных запросах, поэтому метод intercept должен быть потокобезопасен.

Метод destroy вызывается, когда interceptor больше не нужен. В нём должно происходить освобождение ресурсов и т.д.

Для того, чтобы зарегистрировать свой интерсептор используется тег <interceptor>, внутри которого указывается его имя и класс, который реализует этот interceptor. Пример: <intercepor name="myinterceptor" class="MyInterceptor" />.

Чтобы включить интерептор или стек интерсепторов в стек обработки Actiona необходимо внутри <action> использовать тег <interceptor-ref> с именем interceptora, которое было указано при его регистрации.

Пример класса-интерсептора:

```
public class MyInterceptor extends AbstractInterceptor {
   public String intercept(ActionInvocation invocation)throws
Exception{
      /* do some pre-processing */
      String output = "Pre-Processing";
      System.out.println(output);

      /* let us call action or next interceptor */
      String result = invocation.invoke();

      /* do some post-processing */
      output = "Post-Processing";
      System.out.println(output);

      return result;
   }
}
```

24. JSON. Формат. Генерация на стороне сервера и обработка на стороне клиента. Использование в Servlet и Struts2. Примеры.

JSON - простой, основанный на использовании текста, способ хранить и передавать структурированные данные. С помощью простого синтаксиса вы можете легко хранить все, что угодно, начиная от одного числа до строк, массивов и объектов, в простом тексте. Также можно связывать между собой массивы и объекты, создавая сложные структуры данных.

После создания строки JSON, ее легко отправить другому приложению или в другое место сети, так как она представляет собой простой текст. JSON имеет следующие преимущества:

- Он компактен.
- Его предложения легко читаются и составляются как человеком, так и компьютером.
- Его легко преобразовать в структуру данных для большинства языков программирования (числа, строки, логические переменные, массивы и так далее)
- Многие языки программирования имеют функции и библиотеки для чтения и создания структур JSON.
- Название JSON означает JavaScript Object Notation (представление объектов JavaScript). Как и представляет имя, он основан на способе определения объектов (очень похоже на создание ассоциативных массивов в других языках) и массивов.

В простейшем случае JSON позволяет преобразовывать данные, представленные в объектах JavaScript, в строку, которую можно легко передавать от одной функции к другой или – в случае асинхронного приложения – от Web-клиента к серверной программе. Строка выглядит немного замысловато (скоро вы увидите несколько примеров), но зато её легко может интерпретировать JavaScript. JSON также позволяет формировать более сложные структуры, чем простые пары «имя/значение». Например, можно представлять массивы и сложные объекты, а не только простые списки ключей и значений.

JSON – это родной формат для JavaScript. Это значит, что для работы с JSON-данными в JavaScript нам не нужен какой-нибудь специальный инструментарий (toolkit) или API. Например, можно довольно легко создать новую JavaScript-переменную и затем непосредственно присвоить ей строку с данными, отформатированными в JSON. Вот так:

```
var people =
  { "programmers": [
    { "firstName": "Brett", "lastName":"McLaughlin", "email":
"brett@newInstance.com" },
    { "firstName": "Jason", "lastName":"Hunter", "email":
"jason@servlets.com" },
    { "firstName": "Elliotte", "lastName":"Harold", "email":
"elharo@macfaq.com" }
   ],
  "authors": [
    { "firstName": "Isaac", "lastName": "Asimov", "genre": "science fiction"
},
    { "firstName": "Tad", "lastName": "Williams", "genre": "fantasy" },
    { "firstName": "Frank", "lastName": "Peretti", "genre": "christian
fiction" }
   ],
  "musicians": [
```

```
    { "firstName": "Eric", "lastName": "Clapton", "instrument": "guitar" },
    { "firstName": "Sergei", "lastName": "Rachmaninoff", "instrument":
"piano" }
   ]
  }
```

Доступ к данным

Хотя это может показаться неочевидным, но эта длинная строка сверху – обычный массив, и если вы когда-нибудь работали с массивами в JavaScript, то сможете легко получить доступ к данным. Фактически компоненты имени массива можно разделить просто точками. Так, для доступа к фамилии (lastname) первого элемента в списке программистов (programmers) в JavaScript-программе можно использовать такой код:

people.programmers[0].lastName;

Заметьте, что индексация массива начинается с нуля. Доступ к искомому полю данных осуществляется примерно так: мы начинаем с people; затем движемся к элементу programmers и указываем, что нас интересует первая запись ([0]); наконец, мы получаем доступ к значению по ключу lastName. В результате мы получаем строку "McLaughlin".

Ниже представлено еще несколько примеров для этой же переменной.

people.authors[1].genre // возвратит строку "fantasy"

people.musicians[3].lastName // результат не определён (undefined).

Код ссылается на 4-й элемент в массиве, которого не существует

people.programmers.[2].firstName //возвратит строку "Elliotte"

Пользуясь этим простым синтаксисом, можно работать с любыми структурами JSON-форматированных данных, и всё это без привлечения каких-либо дополнительных инструментариев (toolkit'ов) или API для JavaScript.

Модификация JSON-данных

Так же как мы получили доступ к данным с помощью точек и скобок, как показано выше, мы можем легко модифицировать данные в нашей переменной:

people.musicians[1].lastName = "Rachmaninov";

Это всё, что нужно сделать, чтобы изменить данные в переменной, после того как мы преобразовали JSON-данные в объект JavaScript.

Обратное преобразование в строку

Конечно же, наши изменения были бы почти бесполезными, если бы мы не могли легко конвертировать данные обратно в текстовый формат. К счастью, в JavaScript это также довольно тривиальная задача:

String newJSONtext = people.toJSONString();

25. EJB. Типы EJB. Аннотации. Применение при создании веб приложения. Пример EJB.

Технологию EJB (Enterprise Java Beans) можно рассматривать с двух точек зрения: как фреймворк, и как компонент.

С точки зрения компонента **EJB** - это всего-лишь надстройка над POJO-классом, описываемая с помощью аннотации. **Существует три типа компонентов EJB:**

1. session beans - используется для описания бизнесс-логики приложения
2. message-driven beans - так же используется для бизнесс-логики
3. entities - используется для хранения данных

С точки зрения фреймворка EJB - это технология, предоставляющая множество готовых решений (управление транзакциями, безопасность, хранение информации и т.п.) для вашего приложения.

Session beans

Вызываются пользователем для совершения какой-либо бизнесс-операции. Существует 2 типа session-beans: stateless и stateful.

Stateful-бины автоматически сохраняют свое состояние между разными клиентскими вызовами. Типичным примером stateful-бина является корзина в интернет-магазине.

Stateless-бины используются для реализации бизнесс-процессов, которые могут быть завершены за одну операцию.Так же на основе stateless-бинов проектируются web-сервиса.

Message-driven beans

Так же как и session beans используются для бизнесс-логики. Отличие в том, что клиенты никогда не вызывают MDB напрямую. Обычно сервер использует MDB в асинхронныхзапросах.

Entities и Java Persistence API

Одним из главным достоинством EJB3 стал новый механизм работы с persistence - возможность автоматически сохранять объекты в реляционной БД используя технологию объектно-реляционного маппинга (ORM).

В контексте EJB3 persistence провайдер - это ORM-фреймворк, который поддерживает EJB3 Java Persistence API (JPA). JPA определяет стандарт для:

- конфигурации маппинга сущностей приложения и их отображения в таблицах БД;
- EntityManager API - стандартный API для CRUD (create, read, update, delete) операций над сущностями;
- Java Persistence Query Language (JPQL) - для поиска и получения данных приложения;

Можно сказать, что session beans - это "глаголы" приложения, в то время как entities - это "существительные".

EntityManager - это инерфейс, который связывает класс сущности приложения и его представление в БД. EntityManager знает как нужно добавлять сущности в базу, обновлять их, удалять, а так предоставляет механизмы для настройки производительности, кэширования, транзакций и т.д.

Аннотация записывается так:

@<имя аннотации>(<список парамет-значение>) Аннотации:

- **Stateless** - говорит контейнеру, что класс будет stateless session bean. Для него контейнер обеспечит безопасность потоков и менеджмент транзакций. Дополнительно, вы можете добавить другие свойства, например прозрачное управление безопасностью и перехватчики событий;

- **Local** - относится к интерфейсу и говорит, что bean реализующий интерфейс доступен локально
- **Remote** - относится к интерфейсу и говорит, что bean доступен через RMI (Remote Method Invocation)
- **EJB** - применятеся в коде, где мы используем bean.
- **Stateful** - говорит контейнеру, что класс будет stateful session bean.
- **Remove** - опциональная аннотация, которая используется с stateful бинами. Метод, помеченный как Remove говорит контейнеру, что после его исполнения нет больше смысла хранить bean, т.е. его состояние сбрасывается. Это бывает критично для производительности.
- **Entity** - говорит контейнеру, что класс будет сущностью БД
- **Table**(name="...") - указывает таблицу для маппинга
- **Id**, **Column** - параметры маппинга
- **WebService** - говорит, что интерфейс или класс будет представлять вэб-сервис.

Интерфейс может быть помечен как **Local**, что сделает классы, реализующие этот интерфейс, классами локальной бизнесс-логики. Локальные интерфейсы не требуют никаких дополнительные действий при реализации.

В противном случае интерфейс может быть помечен как **Remote**, что обеспечит возможность работы **RMI**. Обычно такой интерфейс расширяет интерфейс Remote, но это не обязательно.
Если вас интересует функциональность и Local и Remote интерфейсов - вот интересный пример из "EJB 3 in Action":

```
1   public interface BidManager{
2      void addBid(Bid bid);
3      List<Bid> getBids(Item item);
4   }
5
6   @Local
7   public interface BidManagerLocal extends BidManager {
8      void cancelBid(Bid bid);
9   }
10
11  @Remote
12  public interface BidManagerRemote extends BidManagerLocal {}
13
14  @WebService
15  public interface BidManagerWS extends BidManager {}
```

26. JMS. Обработка с очередями сообщений. Реализация IBM WebSphere MQ. Подключение к очереди. Пример добавления и чтения сообщения из очереди.(тут как бы через сервлеты, наверное, надо писать пример кода, но я не знаю)https://m.habrahabr.ru/post/176209/?mobile=yes

Java Message Service (JMS) — стандарт промежуточного ПО для рассылки сообщений, позволяющий приложениям, выполненным на платформе Java EE, создавать, посылать, получать и читать сообщения.

Обработка с очередями сообщений

Асинхронные транзакции, которые выполняются посредством каналов публикации или служб предприятия, используют очереди Java Message Service (JMS) для обмена данными с внешней системой.

При входящей обработке получения сообщение немедленно записывается в очередь JMS и источник вызова службы высвобождается из транзакции. Сообщение обрабатывается во входящей очереди JMS посредством бизнес-объектов программы и сохраняется в базе данных. Сообщения остаются во входящей очереди, пока они не будут успешно обработаны или удалены из очереди. Обычная стратегия реализации входящей очереди - выделить очереди и потребителей очередей на отдельный сервер или кластер серверов. Эта стратегия гарантирует, что входящая обработка сообщений не повлияет на производительность пользователей программы.

При исходящей обработке сообщения, отправленные посредством канала публикации, записываются в очередь JMS, а пользователь, который инициировал сообщение, высвобождается из транзакции. Сообщение обрабатывается в исходящей очереди JMS с использованием сконфигурированной конечной точки и отправляется внешней программе. Сообщения остаются в исходящей очереди, пока они не будут успешно доставлены внешней программе или удалены из очереди.

Есть три очереди сообщений по умолчанию:

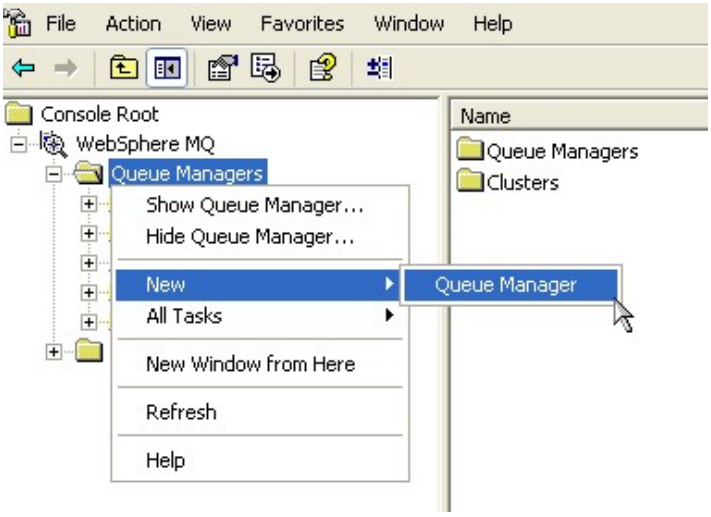
- Одна исходящая последовательная очередь
- Одна входящая последовательная очередь
- Одна входящая непрерывная очередь

Реализация IBM WebSphere MQ

IBM WebSphere MQ представляет собой средство для управления обменом данными в распределенной системе. Одна из главных особенностей таких программ – это гарантированная доставка сообщений, даже если в данный момент времени компьютер-получатель недоступен по каким-либо причинам. IBM WebSphere MQ представляет собой средство для управления обменом данными в распределенной системе. Одна из главных особенностей таких программ – это гарантированная доставка сообщений, даже если в данный момент времени компьютер-получатель недоступен по каким-либо причинам.

Создание менеджера очередей

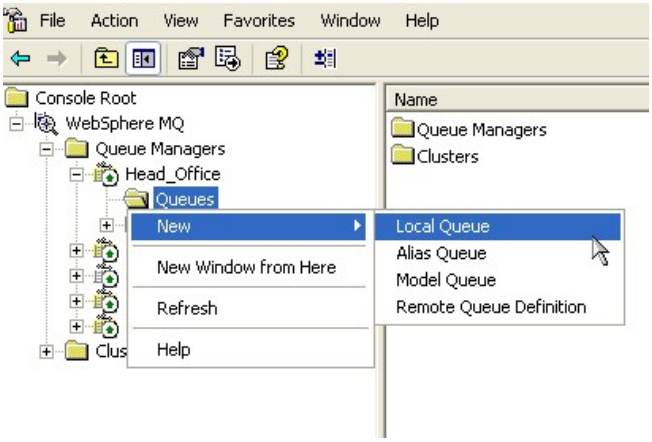
В иерархическом списке в левой части окна нужно установить курсор на узел Queue Managers и после нажатия правой кнопки мыши выбрать **New**. Запустится мастер создания нового менеджера очередей.



В первом окошке необходимо указать уникальное имя менеджера очереди латинскими символами (например: "Head_Office") и нажать кнопку **Next**. Все остальные настройки остаются без изменений, кроме последнего окна, где нужно указать номер порта, например: 1515. Аналогично создается менеджер очередей на удаленном компьютере.

Создание очередей

В менеджере очередей находится папка **Queue**. Для создания новой очереди достаточно правой кнопкой мыши щелкнуть на эту папку и выбрать пункт меню **New – Local Queue**, указать имя новой очереди, а также на вкладке **Cluster** выбрать **Shared in cluster** и указать кластер, который будет управлять этой очередью.



Аналогичные действия выполняются и на удаленном компьютере.

27. JavaScript. Чувствительность к регистру. Юникод. Точки с запятой. Типы данных. null. NaN. undefined. Глобальный объект. Преобразование типов.

JavaScript *(ECMAScript)* изначально создавался для того, чтобы сделать web-странички «живыми». Программы на этом языке называются *скриптами*. В браузере они подключаются напрямую к HTML и, как только загружается страничка – тут же выполняются.

JavaScript может выполняться не только в браузере, а где угодно, нужна лишь специальная программа – интерпретатор. Во все основные браузеры встроен интерпретатор JavaScript, именно поэтому они могут выполнять скрипты на странице. Но, разумеется, JavaScript можно использовать не только в браузере. Это полноценный язык, программы на котором можно запускать и на сервере, и даже в стиральной машинке, если в ней установлен соответствующий интерпретатор.

Что умеет JavaScript?

- Создавать новые HTML-теги, удалять существующие, менять стили элементов, прятать, показывать элементы и т.п.
- Реагировать на действия посетителя, обрабатывать клики мыши, перемещения курсора, нажатия на клавиатуру и т.п.
- Посылать запросы на сервер и загружать данные без перезагрузки страницы (эта технология называется "AJAX").
- Получать и устанавливать cookie, запрашивать данные, выводить сообщения…
- …и многое, многое другое!

Что не умеет JavaScript?

- JavaScript не может читать/записывать произвольные файлы на жесткий диск, копировать их или вызывать программы. Он не имеет прямого доступа к операционной системе. (Современные браузеры могут работать с файлами, но эта возможность ограничена специально выделенной директорией – «песочницей).
- JavaScript, работающий в одной вкладке, не может общаться с другими вкладками и окнами, за исключением случая, когда он сам открыл это окно или несколько вкладок из одного источника (одинаковый домен, порт, протокол). Есть способы это обойти (но это вам не надо).
- Из JavaScript можно легко посылать запросы на сервер, с которого пришла страница. Запрос на другой домен тоже возможен, но менее удобен, т. к. и здесь есть ограничения безопасности.

Чувствительность к регистру

Все ключевые слова пишутся в нижнем регистре. Чувствителен к регистру, т.е. переменные Abb и abb – разные.

Юникод (https://learn.javascript.ru/string#кодировка-юникод)

Юникод — стандарт кодирования символов, позволяющий представить знаки почти всех письменных языков.

Все строки имеют внутреннюю кодировку Юникод.

Неважно, на каком языке написана страница, находится ли она в windows-1251 или utf-8. Внутри JavaScript-интерпретатора все строки приводятся к единому «юникодному» виду. Каждому символу соответствует свой код.

Есть метод для получения символа по его коду: *String.fromCharCode(code)*

Точки с запятой (этот пункт я просто расписала случаи, когда все может поломаться, не поставив вы точку с запятой. Можете забить. Главное сказать, что их лучше ставить.)

При написании кода на javascript точки с запятой ставятся после каждой инструкции (за исключением, for, function, if, switch, try и while). Точку с запятой во многих случаях можно не ставить, если есть переход на новую строку.

Так тоже будет работать:

alert('Привет')

alert('Мир')

В этом случае JavaScript интерпретирует переход на новую строчку как разделитель команд и автоматически вставляет «виртуальную» точку с запятой между ними.

Однако, важно то, что «во многих случаях» не означает «всегда»!

Например, запустите этот код:

alert(3 +

1
+ 2);

Выведет 6.

То есть, точка с запятой не ставится. Почему? Интуитивно понятно, что здесь дело в «незавершённом выражении», конца которого JavaScript ждёт с первой строки и поэтому не ставит точку с запятой. И здесь это, пожалуй, хорошо и приятно.

Но в некоторых важных ситуациях JavaScript «забывает» вставить точку с запятой там, где она нужна.

Таких ситуаций не так много, но ошибки, которые при этом появляются, достаточно сложно обнаруживать и исправлять.

Такой код работает: *[1, 2].forEach(alert)*. Он выводит по очереди 1, 2.

Важно, что вот такой код уже работать не будет:

alert("Сейчас будет ошибка")

[1, 2].forEach(alert)

Выведется только первый alert, а дальше – ошибка. Потому что перед квадратной скобкой JavaScript точку с запятой не ставит, а как раз здесь она нужна (упс!). Если её поставить, то всё будет в порядке:

alert("Сейчас будет ошибка");

[1, 2].forEach(alert)

Поэтому в JavaScript рекомендуется точки с запятой ставить. Сейчас это, фактически, стандарт, которому следуют все большие проекты.

Типы данных. null. NaN. undefined. (их всего 6, если будет интересно к каждому типу кинула ссылку)

Число «number» (https://learn.javascript.ru/number)

var n = 123;

n = 12.345;

n = new Number(32); // так делать плохо, здесь n не число, а объект, про \то можно умолчать.

Единый тип число используется как для целых, так и для дробных чисел.

Существуют специальные числовые значения *Infinity (бесконечность)* и *NaN (ошибка вычислений).*

Например, бесконечность *Infinity* получается при делении на ноль:

Ошибка вычислений *NaN* будет результатом некорректной математической операции, например:

*alert("нечисло" * 2); // NaN*

Эти значения формально принадлежат типу «число», хотя, конечно, числами в их обычном понимании не являются.

Строка «string» (https://learn.javascript.ru/string)

var str = "Мама мыла раму";

str = 'Одинарные кавычки тоже подойдут';

str = new String("hello"); // так делать плохо, здесь str формально не строка, а объект

В JavaScript одинарные и двойные кавычки равноправны. Можно использовать или те или другие. Тип символ не существует, есть только строка.

Булевый (логический) тип «boolean»

У него всего два значения: true (истина) и false (ложь).

var t = true;

var f = new Boolean(false); //опять же не очень способ

Специальное значение «null»

Значение null не относится ни к одному из типов выше, а образует свой отдельный тип, состоящий из единственного значения null:

var age = null;

В JavaScript null не является «ссылкой на несуществующий объект» или «нулевым указателем», как в некоторых других языках. Это просто специальное значение, которое имеет смысл «ничего» или «значение неизвестно». В частности, код выше говорит о том, что возраст age неизвестен.

Специальное значение «undefined»

Значение undefined, как и null, образует свой собственный тип, состоящий из одного этого значения. Оно имеет смысл «значение не присвоено».

Если переменная объявлена, но в неё ничего не записано, то её значение как раз и есть undefined:

var x;

alert(x); // выведет "undefined"

Можно присвоить undefined и в явном виде, хотя это делается редко:

var x = 123;

x = undefined;

В явном виде undefined обычно не присваивают, так как это противоречит его смыслу. Для записи в переменную «пустого» или «неизвестного» значения используется null.

Объекты «object» (но JS, такой язык, что все типы представляются как объекты. Если кому интересно..могу пояснить как это вообще происходит. Но если нет, то просто опустите это предложение, когда будете отвечать)

Первые 5 типов называют «примитивными».

Особняком стоит шестой тип: «объекты».

Он используется для коллекций данных и для объявления более сложных сущностей.

Объявляются объекты при помощи фигурных скобок {...}, например:

var user = { name: "Вася" };

Глобальный объект (https://learn.javascript.ru/global-object)

Механизм работы функций и переменных в JavaScript очень отличается от большинства языков.

Глобальными называют переменные и функции, которые не находятся внутри какой-то функции. То есть, иными словами, если переменная или функция не находятся внутри конструкции function, то они – «глобальные».

В JavaScript все глобальные переменные и функции являются свойствами специального объекта, который называется «глобальный объект» (global object).

В браузере этот объект явно доступен под именем window. Объект *window* одновременно является глобальным объектом и содержит ряд свойств и методов для работы с окном браузера.

В других окружениях, например Node.JS, глобальный объект может быть недоступен в явном виде, но суть происходящего от этого не изменяется, поэтому далее для обозначения глобального объекта мы будем использовать "window".

Присваивая или читая глобальную переменную, мы, фактически, работаем со свойствами *window*.

Например:

var a = 5; // объявление var создаёт свойство window.a
alert(window.a); // 5

Создать переменную можно и явным присваиванием в window:

window.a = 5;
alert(a); // 5

Еще следует сказать, что конструкции for, if... не влияют на видимость переменных.

Преобразование типов.

К Boolean

В Javascript преобразование типа к boolean можно сделать следующими способами:

1) *!!myVar*
2) *Boolean(myVar)*

К Number

1) *+myVar*
2) *Number(myVar);*
3) *parseInt(myVar).* Функция parseInt ("к целому") преобразует свой аргумент в целочисленное значение

К Object

1) *var oNum = new Number(3);*
2) *var oStr = new String("1.2");*
3) *var oBool = new Boolean(true);*

К String

1) Каждый объект обладает методом *toString*, который вызывается автоматически каждый раз, когда требуется строковое представление объекта.
2) Добавить пустую строку: *var t = 12 + “”; // “12”*

В преобразовании типов есть свои подводные камни, все они здесь конечно не описаны. Но, опять же, если интересно..обращайтесь.

Значение	Преобразование в:			
	Строку	Число	Булевское	Объект
undefined	“undefined”	NaN	false	typeError
null	“null”	0	false	typeError

true false	“true” “false”	1 0		new Boolean(true) new Boolean(false)
“” (пустая строка) “1.2” “one” “-10” “+10” “011” “0xff”		0 1.2 NaN -10 10 11 255	false true true true true true true	new String(“”) new String(“1.2”) new String(“one”) new String(“-10”) new String(“+10”) new String(“011”) new String(“0xff”)
0 -0 NaN Infinity -Infinity 3	“0” “0” “NaN” “Infinity” “-Infinity” “3”		false false false true true true	new Number(0) new Number(-0) new Number(NaN) new Number(Infinity) new Number (-Infinity) new Number(3)
{} (любой объект) [] (пустой массив) [9] (1 числовой элемент) [0,1,2,3] (массив) function(){}(любая функция)	Смотри внизу “” “9” "0,1,2,3" код функции	NaN 0 9 NaN NaN	true true true true true	

28. JavaScript. Объявление переменной. Область видимости. Функции. ООП. "use strict". Работа с объектами: создание, удаление, редактирование свойств. Prototype. Массивы: foreach, every, some, map, reduce, filter, indexOf. Объекты, подобные массивам. this. const.

Объявление переменной.

Для *объявления* используется ключевое слово var:
var t; //теперь переменная принимает значение undefined
После объявления, можно записать в переменную данные:
var t;
t = 'Hello, world'; //ну или любой другой тип.
Эти данные будут сохранены в соответствующей области памяти.
Для краткости можно совместить объявление переменной и запись данных:
var t = 'Hello, world';
Можно даже объявить несколько переменных сразу:
var t = 1, b = {a: 1}, c = 'hello';

Область видимости.

Переменные в Javascript бывают глобальными и локальными. Глобальная переменная доступна везде, локальная – только в текущей области видимости. Здесь главное помнить, что блочной области видимости в JavaScript нет. Единственное, что может задать какое-либо окружение для переменных, это *функции*. Или можно вот так, для понимания: *Глобальными* называют переменные и функции, которые не находятся внутри какой-то функции. То есть, иными словами, если переменная или функция не находятся внутри конструкции function, то они – «глобальные».

Одно из основных понятий JS – замыкание. Это определение объясняется с нескольких сторон. Но именно для конкретного пункта я объясню одну. Замыканием является функция и все переменные, до которых она может достучаться. Суть в том, то, что если в функцие используется какая-то переменная, но она не объявляена в данной функцие, JS ищет ее выше по вложенности данной функции, вплоть до глобального объекта.

Функции

Как уже раньше говорилось, а может и нет, функции – это объекты. Ну вообще все в JS это объекты.
Функции могут выполнять несколько ролей:

1) Обычная функция
```
function add(a, b) {
    return a+b;
}
```
2) Функция – метод в объекте
```
var Rectangle = {
  x: 2,
  y: 4,
  perimeter : function() {
      return (this.x+this.y)*2;
  }
}
Rectangle.perimeter(); // 12
```
3) Функции – конструкторы
```
function Rectangle(x, y) {
   this.x = x;
   this.y = y;
}
var rect = new Rectangle();
```

Два способа объявления функций:

1) ***function declaration***
```
function add(a, b) {
    return a+b;
}
```
Такая функция может вызываться до ее объявления, т.е. если мы напишем *var t = add(1,2);* до функции выше,она все равно отработет.
2) ***function expression***
```
var add = function(a, b) {
    return a+b;
```

}

Вот здесь, фокус, как в первом случае, уже не сработает. Если интересно почему, можете подумать, а потом мне написать:)

Функции также бывают анонимные и именованные. Проще это показать на функции типа function expression. Более сложные примеры Вам наверное ни к чему.

1) Анонимные

```
var add = function(a, b) {
    return a+b;
}
```

2) Именованные

```
var add = function add(a, b) {
    return a+b;
}
```

Заметили в чем разница? В данном случае именованные функции в основном используются для рекурсии. Ну еще такой маленький лайфхак: если функции не называть, а в них далее будет ошибка, Вы просто не поймете, где это произошло. Потому что в консоли браузера будет что-то типа «ошибка в анонимной функции», на английском разумеется)

ООП. Работа с объектами

В JavaScript нет классов. Объект в javascript представляет собой обычный ассоциативный массив или, иначе говоря, "хэш". Он хранит любые соответствия "ключ => значение" и имеет несколько стандартных методов.

Следующие два варианта создания объекта эквивалентны:

```
var o = new Object()
var o = {}
```

Добавление свойств

Есть два синтаксиса добавления свойств в объект. Первый - точка, второй - квадратные скобки:

```
o.test = 5
o["test"] = 5
```

Квадратные скобки используются в основном, когда название свойства находится в переменной:

```
var name = 'test'
o[name] = 5
```

Доступ к свойствам:

```
alert(o.test)
alert(o['test'])
```

Если у объекта нет такого свойства, то результат будет 'undefined'

Удаление свойств

Удаляет свойство оператор delete:

```
o.test = 5
delete o.test
o['bla'] = true
```

Добавление метода

Как и в других языках, у объектов javascript есть методы.

Например, создадим объект rabbit с методом run

```
var rabbit = {}
rabbit.run = function(n) {
   alert("Пробежал "+n+" метров!")
}
```

В JS можно реализовать наследование. Но это сложно очень описать и объяснить. Если опять же интересно, то можно почитать здесь (https://learn.javascript.ru/class-inheritance), ну или написать мне конечно:)

“use strict”

Для того, чтобы перевести код в режим полного соответствия современному стандарту (ES5 и выше), нужно указать специальную директиву use strict. Эта директива не поддерживается IE9-.

Директива выглядит как строка "use strict"; или 'use strict'; и ставится в начале скрипта.

Например:

```
"use strict";
// этот код будет работать по современному стандарту ES5
...
```

Отменить действие use strict никак нельзя

Не существует директивы no use strict или подобной, которая возвращает в старый режим.

Если уж вошли в современный режим, то это дорога в один конец.

use strict для функций

use strict также можно указывать в начале функций, тогда строгий режим будет действовать только внутри функции.

Prototype (вообще лучше почитать здесь: https://learn.javascript.ru/prototype)

Реализуется наследование через неявную(внутреннюю) ссылку одного объекта на другой, который называется его прототипом и в спецификации обозначается [[prototype]]. Это свойство обычно скрыто от программиста.

К нему можно обратиться через:

a.prototype или *a.__proto__* (тут два нижних подчеркивания перед proto и после. Здесь есть разница. Но Вам это не столь важно)

Когда вы ставите функции Animal свойство Animal.prototype = XXX – вы этим декларируете: "все новые объекты класса Animal будут иметь прототип XXX".

Массивы: foreach, every, some, map, reduce, filter, indexOf.

Javascript поддерживает два вида структуры "массив":

1. Ассоциативный массив (хеш), где данные хранятся по произвольному ключу.
2. Числовой массив Array, где данные хранятся по номерам.

forEach

Метод *«arr.forEach(callback[, thisArg])»* используется для перебора массива.

Он для каждого элемента массива вызывает функцию callback.

Этой функции он передаёт три параметра callback(item, i, arr):

item – очередной элемент массива.

i – его номер.

arr – массив, который перебирается.

every/some

Эти методы используются для проверки массива.

1) Метод *«arr.every(callback[, thisArg])»* возвращает true, если вызов callback вернёт true для каждого элемента arr.
2) Метод *«arr.some(callback[, thisArg])»* возвращает true, если вызов callback вернёт true для какого-нибудь элемента arr.

Map

Метод «*arr.map(callback[, thisArg])*» используется для трансформации массива.

Он создаёт новый массив, который будет состоять из результатов вызова *callback(item, i, arr)* для каждого элемента arr.

Filter

Метод «*arr.filter(callback[, thisArg])*» используется для фильтрации массива через функцию.

Он создаёт новый массив, в который войдут только те элементы arr, для которых вызов *callback(item, i, arr)* возвратит true.

reduce

Метод «*arr.reduce(callback[, initialValue])*» используется для последовательной обработки каждого элемента массива с сохранением промежуточного результата.

Метод reduce используется для вычисления на основе массива какого-либо единого значения, иначе говорят «для свёртки массива». Он применяет функцию callback по очереди к каждому элементу массива слева направо, сохраняя при этом промежуточный результат.

Аргументы функции *callback(previousValue, currentItem, index, arr):*

previousValue – последний результат вызова функции, он же «промежуточный результат».

currentItem – текущий элемент массива, элементы перебираются по очереди слева-направо.

index – номер текущего элемента.

arr – обрабатываемый массив.

IndexOf(value)

Возвращает индекс элемента массива, который равен value

Объекты, подобные массивам

Все объекты в JS являются ассоциативными массивами. Т.е. если объявить объект: *var o = {a: 1, b: 2, c: 3};* то потом к полям мы можем обращаться через квадратные скобки и записанное в них поле в виде строки: *o['a'], o['b'], o['c'].*

this (тут если что-то будет не так пишите, это я взяла из той шпоры. Просто если начну писать то это надолгои на много выйдет)

Глобальный контекст

В глобальном контексте выполнения (за пределами каких-либо функций), this ссылается на глобальный объект вне зависимости от использования в строгом или нестрогом режиме.

console.log(this.document === document); // true

В контексте функции

В пределах функции значение this зависит от того, каким образом вызвана функция.

Простой вызов

```
function f1(){
  return this;
}
f1() === window; // global object
```

В этом случае значение this не устанавливается вызовом. Так как этот код написан не в строгом режиме, значением this всегда должен быть объект, по умолчанию – глобальный объект.

Строгий режим

```
function f2(){
  "use strict"; // see strict mode
  return this;
}
f2() === undefined;
```

В строгом режиме, значение this остается тем значением, которое было установлено в контексте исполнения. Если такое значение не определено, оно остается undefined.

В методе объекта

Когда функция вызывается как метод объекта, используемое в этой функции ключевое слово this принимает значение объекта, по отношению к которому вызнан метод.

В следующем примере, когда вызвано свойство o.f() , внутри функции this привязано к объекту o.

```
var o = {
  prop: 37,
  f: function() {
    return this.prop;
  }
};
console.log(o.f()); // logs 37
```

const

Объявление const задаёт константу, то есть переменную, которую нельзя менять:

```
const apple = 5;
apple = 10; // ошибка
```

Заметим, что если в константу присвоен объект, то от изменения защищена сама константа, но не свойства внутри неё:

```
const user = {
  name: "Вася"
};
user.name = "Петя"; // допустимо
user = 5; // нельзя, будет ошибка
```

То же самое верно, если константе присвоен массив или другое объектное значение.

29. **jQuery. Базовые селекторы (a[href~=http://], a[href*=jquery], a[href$=.pdf], li:has(a), a:not(:hidden), a:first, a:odd, a:even, li:last-child, a:only-child, li:eq(2), li:gt(2), li:lt(7), :disabled, :selected, :visible). Создание новых элементов. .size() .get(0) .filter() .slice() .children() .contents() .next() .parents() .prev() .siblings() .find() .contains() .is(selector) .closest(). Цепочки вызовов. .each().attr() .removeAttr()**

jQuery — библиотека JavaScript, фокусирующаяся на взаимодействии JavaScript и HTML.

Селекторами называют строчные выражения, с помощью которых задаются условия поиска элементов DOM на странице.

[attribute ~= value]- Соответствует всем элементам с атрибутом *attribute*, содержащим слово *value* (именно слово, а не просто подстроку. То есть вхождение value должно содержать с обоих сторон разделители: пробелы или начало/конец строки). Если *value* состоит из нескольких слов, между которыми есть пробелы, то нужно заключать *value* в кавычки. Если *value* не содержит пробелов — кавычки не обязательны.

[attribute *= value] - Соответствует всем элементам, у которых значение атрибута *attribute* содержит *value*. Если *value* состоит из нескольких слов, между которыми есть пробелы, то нужно заключать *value* в кавычки. Если *value* не содержит пробелов — кавычки не обязательны.

[attribute $= value] - Соответствует всем элементам, у которых значение атрибута *attribute* заканчивается на *value*. Если *value* состоит из нескольких слов, между которыми есть пробелы, то нужно заключать *value* в кавычки. Если *value* не содержит пробелов — кавычки не обязательны.

:has(selector) - Соответствует элементам, которые обладают потомками, удовлетворяющими селектору *selector*.

:not(selector) - Исключает элементы удовлетворяющие селектору *selector* из найденных элементов.

:hidden - Соответствует всем скрытым элементам страницы. Элемент считается скрытым в следующих случаях:

- Его css-свойство display равно none
- Он является элементом формы с type="hidden"
- Его высота или ширина равна 0
- Он находится внутри невидимого элемента и поэтому тоже невидим на странице.

Элементы с css-свойством visibility равным hidden, а так же элементы с нулевой прозрачностью, считаются видимыми, поскольку они продолжают занимать место на странице. Если на элементе выполняется анимация, делающая его невидимым, то статус «скрытости» он получит сразу после ее завершения.

:first - Соответствует первому элементу, из всех выбранных с помощью селектора *someSelector*.

:odd- Соответствует элементам с нечетными номерами позиций, в наборе выбранных элементов.

:even - Соответствует элементам с четными номерами позиций, в наборе выбранных элементов.

:last-child - Соответствует элементам, которые идут последними в своих непосредственных предках.

:only-child - Соответствует элементам, которые являются единственными в своих непосредственных предках.

eq(index) - Соответствует элементу, занимающему позицию под номером *index*, среди уже выбранных с помощью селектора someSelector элементов. Нумерация элементов начинается с 0.

:gt(n) - Фильтрует набор выбранных элементов, оставляя только те, индекс которых превышает *n*. Не забывайте, что индексирование начинается с 0.

:lt(n)- Фильтрует набор выбранных элементов, оставляя только те, индекс которых меньше *n*. Не забывайте, что индексирование начинается с 0.

:disabled - Соответствует всем заблокированным элементам формы (элементы с атрибутом disabled).

:selected - Соответствует всем элементам со статусом selected. Это могут быть выбранные элементы типа <option>. Для поиска выбранных checkbox и radioэлементов, этот селектор НЕ подойдет.

:visible - Соответствует всем видимым элементам страницы. Элемент считается **не**видимым в следующих случаях:

- Его css-свойство display равно none
- Он является элементом формы с type="hidden"
- Его высота или ширина равна 0
- Он находится внутри невидимого элемента и поэтому тоже невидим на странице.

Элементы с css-свойством visibility равным hidden, а так же элементы с нулевой прозрачностью, считаются видимыми, поскольку они продолжают занимать место на странице.

.size() - Возвращяет количество выбранных элементов.

.get([index]) - Возвращяет DOM-элементы, хранящиеся в объекте jQuery. Если указан необязательный параметр *index*, то будут возвращен один объект, который идет в наборе под номером *index* (нумерация начинается с нуля). Если не указывать этот параметр, то будет возвращен массив со всеми DOM-элементами объекта jQuery.

.filter() - Фильтрует набор выбранных элементов.

.slice() - Фильтрует набор выбранных элементов, оставляя только те элементы, чьи индексы лежат в определенной области (например от 0 до 5).

.children() - Находит все дочерние элементы у выбранных элементов. При необходимости, можно указать селектор для фильтрации.

.contents() - Находит все дочерние элементы у выбранных элементов. В результат, помимо элементов, включается и текст.

.next() - Осуществляет поиск элементов, лежащих непосредственно после заданных элементов (по одному для каждого из заданных).

.parents() - Осуществляет поиск всех предков выбранных элементов, то есть, не только прямых родителей, но и прародителей, прапрародителей и так далее, до начало дерева DOM.

.prev() - Для каждого из выбранных элементов находит предшествующий ему элемент (но только если он лежит на том же уровне иерархии DOM).

.siblings() - Осуществляет поиск элементов, являющихся соседними для выбранных элементов (под соседними понимаются элементы, которые имеют общего родителя). При этом, сами выбранные элементы в результат не включаются.

.find() - Осуществляет поиск элементов внутри уже выбранных элементов.

.is() - Проверяет, соответствует ли хотя бы один из выбранных элементов определенному условию (оно зависит от заданного параметра: если задан селектор, то условием будет соответствие селектору; если задан объект jQuery, то условие — наличие в нем выбранных элементов; при передаче в.is() элемента DOM, будет проверяться его наличие среди выбранных элементов; и наконец если указать в качестве параметра функцию, то проверка условия будет возложена на нее). Возвращает значение типа boolean (true или false).

.closest() - Для каждого из выбранных элементов, closest() будет искать ближайший подходящий элемент из числа следующих: сам выбранный элемент, его родитель, его прародитель, и так далее, до начало дерева DOM.

$.contains() - Проверяет, содержится ли один элемент страницы внутри другого.

.each() - Выполняет заданную функцию для каждого из выбранных элементов в отдельности. Это дает возможность обрабатывать выбранные элементы отдельно друг от друга.

.attr() - Возвращает или изменяет значение атрибутов у выбранных элементов страницы.

.removeAttr() - Метод для удаления атрибутов (таких как id, class, title и.т.д) у элементов страницы.

Создание новых элементов:

CreateElement:

```
1   r aOne = document.createElement('a');
```

```
2   ne.href = "http://google.com"
3   ne.innerText = "Гугли!"
4   cument.getElementById('wrapper').appendChild(aOne);
```

Не самый лучший вариант. Когда пример усложняется, следует постоянно думать о кроссбраузерности, да и кода многовато.

Очень часто встречающийся подход с использованием jQuery:

```
1   '#wrapper').append('<a href="http://google.com">Гугли!</a>');
```

Этот вариант ужасен. Никогда не используйте его с динамическим контентом.

Третий вариант самый красивый, кроссбраузерный и безопасный:

```
1   '<a>', { href: 'http://google.com', text: 'Гугли!'}).appendTo('#wrapper');
```

Цепочки вызовов

Цепочкой методов называют последовательный вызов нескольких методов jQuery. Например:

```
$("div").parent().css("heigth", "10px").fadeTo(0, 0.5).addClass("divOwner");
// в результате будут найдены родители всех div-элементов, им будет установлена высота в 10 пикселей,
// прозрачность на 50%, и добавлен класс ''divOwner''.
```

30. jQuery. .css(), .width(), .height() .addClass(), .removeClass(), .toggleClass(), .hasClass() .html(), .text() .append(), appendTo(), prepend(), prependTo(), before(), after() .remove(), .clone() .val(), .submit() .bind(), one(), on() - unbind(), off() .trigger(), .change() .hide(), show(), toggle(), fadeIn(), fadeOut(), fadeTo(), slideDown(), slideUp(), slideToggle() - stop() .animate() $.trim(), $,each(), $.grep(), $.map(), $.inArray(), $.makeArray(), $.unique(), $.extend(), $.getScript(). Ссоздание собственного плагина для jQuery

jQuery — библиотека JavaScript, фокусирующаяся на взаимодействии JavaScript и HTML.

.css() - Возвращает или изменяет значения css-величин у выбранных элементов страницы.

.width() - Функция возвращают ширину элемента без учета отступов и толщины рамки. Кроме этого, с помощью width(), можно установить новое значение ширины.

.height() - Функция возвращает высоту элемента без учета отступов и толщины рамки. Кроме этого, с помощью height(), можно установить новое значение высоты.

.addClass() - Добавляет класс(ы) выбранным элементам страницы. Если из этих элементов некоторые уже принадлежат каким-то классам, то новый (новые) класс добавится к существующим, а не заменит их.

.removeClass() - Удаляет заданные классы у элементов на странице. Функция имеет несколько вариантов использования:

.toggleClass() - Добавляет или удаляет заданный класс(ы) по принципу переключателя (добавляет, если элемент не содержит класса, и удаляет, если класс есть).

.hasClass() - Проверяет наличие класса у элементов страницы. Метод имеет один вариант использования:

.html() - Возвращает или изменяет html-содержимое выбранных элементов страницы. Если таких элементов несколько, то значение будет взято у первого.

.text() - Возвращает или изменяет текстовое содержимое выбранных элементов страницы.. Если таких элементов несколько, метод возвратит строку, в которой будет содержимое всех элементов, расположенное через пробел.

.append() .appendTo() - Функции добавляют содержимое в конец элементов.

```
$(R).append(V); // Добавляет V в R

$(R).appendTo(V); // Добавляет R в V
```

.prepend() .prependTo() - Функции добавляют содержимое в начало определенных элементов.

.before() - Функциz помещает заданное содержимое перед определенными элементами страницы.

.after() - Функция вставляют заданное содержимое сразу после определенных элементов страницы.

.remove() - Метод для удаления элементов страницы.

.clone() - Метод создает копии выбранных элементов страницы и возвращает их в виде объекта jQuery. Элементы копируются вместе со всеми содержащимися внутри них элементами (так называемое глубокое копирование).

.val() - Метод позволяет получать и изменять значения элементов форм. Для элементов input это значение атрибута value; для списков выбора (select) – значение value выбранного элемента (в случае множественного выбора – массив значений); в случае с textarea, метод .val() будет работать непосредственно с содержимым тега textarea.

.submit() - Устанавливает обработчик отправки формы на сервер, либо запускает это событие.

.bind() - Устанавливает обработчик события на выбранные элементы страницы. Обработчик не сработает на элементах, появившихся после его установки.

.one() - Устанавливает обработчик события на выбранные элементы страницы, который сработает только по одному разу, на каждом из элементов.

.on() - Универсальный метод для установки обработчиков событий на выбранные элементы страницы.

.unbind() - Метод необходим для удаления обработчиков событий, установленных на выбранных элементах методами bind(), one() или методами с узким назначением (click(), focus() и.т.д).

.off() - Удаляет с выбранных элементов страницы обработчики событий, установленные с помощью метода .on().

.trigger() - Вызывает событие у выбранных элементов, что приводит к запуску обработчиков этого события.

.change() - Устанавливает обработчик изменения заданного элемента формы, либо, запускает это событие.

.show() .hide() - С помощью этих функций можно плавно показывать и скрывать выбранные элементы на странице, за счет изменения размера и прозрачности. Отметим, что после скрытия элемента, его css-свойство display становится равным none, а перед появлением, оно получает свое прежнее значение обратно.

.toggle() - Поочередно выполняет одно из нескольких заданных действий.

.fadeIn() .fadeOut() - С помощью этих функций можно показывать и скрывать выбранные элементы на странице, за счет плавного изменения прозрачности. Отметим, что после скрытия элемента, его css-свойство display автоматически становится равным none, а перед появлением, оно получает свое прежнее значение обратно.

.fadeTo() - Изменяет уровень прозрачности у выбранных элементов на странице. Позволяет менять прозрачность плавно.

.slideDown() .slideUp() - С помощью этих функций можно показывать и скрывать выбранные элементы на странице, за счет плавного разворачивания и сворачивания. Отметим, что после скрытия элемента, его css-свойство display становится равным none, а перед появлением, оно получает свое прежнее значение обратно.

.slideToggle() - Вызов этого метода приводит к плавному сворачиванию (если элемент развернут) или разворачиванию (если элемент свернут) выбранных элементовна странице. Отметим, что после скрытия элемента, его css-свойство display становится равным none, а перед появлением, оно получает свое прежнее значение обратно.

.animate() - Выполняет заданную пользователем анимацию, с выбранными элементами. Анимация происходит за счет плавного изменения CSS-свойств у элементов

.stop() - Останавливает выполнение текущей анимации.

.trim() - Удаляет символы пробелов, табов и переносов строк из начала и конца строки.

.each() - Выполняет заданную функцию для каждого из выбранных элементов в отдельности. Это дает возможность обрабатывать выбранные элементы отдельно друг от друга.

.grep() - Ищет в заданном массиве элементы, удовлетворяющие условиям фильтрующей функции. Возвращает массив с найденными элементами (в исходный массив изменения не вносятся).

.map() - Выполняет заданную функцию для каждого из выбранных элементов в отдельности. Значения, полученные в результате выполнения этой функции образуют новый набор в виде объекта jQuery, именно его и возвращает метод map.

.inArray() - Ищет заданный элемент в массиве. Возвращает индекс этого элемента или -1 в случае его отсутствия.

.makeArray() - Конвертирует массивоподобные объекты, в массивы.

.unique() - Сортирует массив с DOM-элементами, выстраивая их в порядке расположения в DOM, а так же удаляя повторения.

.extend() - Объединяет содержимое двух или более заданных javascript-объектов. Результат объединения записывается в первый из этих объектов (он же будет возвращен функцией, после ее выполнения).

.getScript() - Делает запрос к серверу без перезагрузки страницы, с запросом javascript файла. При получении запрошенного файла, код внутри него будет автоматически выполнен.

Создание собственного плагина:

```
(function( $ ) {
  $.fn.myPlugin = function() {
```

```
        // Тут пишем функционал нашего плагина

    };
})(jQuery);
```