**Конспект лекций по курсу «Разработка Web-приложений»**

**Web-приложения и их особенности**

**Распределенная система** – это собрание независимых компьютеров (узлов), соединенных сетью и программным обеспечением, обеспечивающим их совместное функционирование.

**Причины создания распределенных приложений**

* + - 1. **Необходимость совместного использования общих ресурсов** (данных, устройств, приложений).
      2. **Улучшение функциональных характеристик системы** (производительности, повышения надежности и устойчивости к сбоям, упрощение и удешевление за счет специализации компонентов, отношения цена/производительность).

**Принципы построения**

1. **Функциональное разделение.** Узлы выполняют различные задачи (клиент / сервер, хост / терминал, сбор данных / обработка данных). Решение – создание разделяемых сервисов, которые управляют набором ресурсов и предоставляют услуги пользователям.
2. **Естественное разделение.** Разделение определяется задачей (система обслуживания сети супермаркетов, сеть для обеспечения коллективной работы). Функциональная нагрузка одинаковая у однотипных узлов.
3. **Балансировка нагрузки.** Функциональность может повторяться. Задачи назначаются на процессоры таким образом, чтобы нагрузка была равномерной. Такой подход обеспечивает наибольшую эффективность использования вычислительных ресурсов.

**Последствия распределенности**

1. **Параллельность.** Параллельное выполнение (вопросы синхронизации, «гонки потоков»), совместное использование ресурсов (коллизии при доступе к данным, сервисам, устройствам), типичные проблемы (взаимные блокировки, ненадежные коммуникации).
2. **Нет “глобального” времени** (асинхронная передача сообщений, ограниченная точность синхронизации часов).
3. **Нет состояния системы.** В распределенной системе нет ни одного процесса, который бы знал текущее глобальное состояние системы. Это следствие параллелизма и механизма передачи данных.
4. **Сбои.** Процессы выполняются автономно, изолированно. Неудачи отдельных процессов могут остаться необнаруженными. Отдельные процессы могут не подозревать об общесистемном сбое. Сбои происходят чаще, чем в централизованной системе. Новые причины сбоев (которых не было в монолитных системах). Сетевые сбои изолируют процессы и фрагментируют систему на изолированные части.

**Требования к распределенным системам**

1. Открытость.
2. Безопасность.
3. Масштабируемость.
4. Механизмы обработки ошибок и восстановления после сбоев.
5. Методы решения проблем параллелизма.
6. Прозрачность.
7. Управляемость.
8. **Открытость.** Гарантирует расширяемость, возможность повторного использования, использование открытых протоколов и стандартов. Важными факторами являются наличие четких спецификаций и полной документации, опубликованные интерфейсы, тестирование и проверка на многих платформах.
9. **Безопасность.** Физическая распределенность означает возможность доступа злоумышленников к компонентам. Необходимо обеспечить посылку важной информации по сети безопасно и эффективно. Это достигается, например, с помощью аутентификации (процедуры проверки подлинности данных) и авторизации (предоставление определённому лицу или группе лиц прав на выполнение определённых действий; а также процесс проверки данных прав при попытке выполнения этих действий), шифрования данных.
10. **Масштабируемость.** Распределенная система масштабируема, если она остается эффективной при увеличении числа обслуживаемых пользователей или ресурсов. Возникает проблема стоимости физических ресурсов (растет при увеличении числа пользователей) и потери производительности (увеличиваются с ростом размера данных и количества пользователей). Для «обхода узких мест» используется децентрализация алгоритмов (Пример – Domain Name Service (DNS)), тиражирование и кэширование данных.
11. **Обработка сбоев.** Сбои возникают чаще, чем в централизованных системах, но обычно носят локальный характер. Обработка сбоев включает в себя: диагностику (например, ошибки передачи могут быть обнаружены с помощью контрольных сумм. Однако, в ряде случаев, может быть невозможна. Например, невозможно определить, удаленный сервер не работает или просто очень загружен?), маскирование (многие сбои могут быть скрыты, однако, в ряде случаев маскирование может быть невозможно или не всегда приводит к правильным последствиям) и восстановление.
12. **Прозрачность.** Это сокрытие гетерогенной (разнородной) и распределенной структуры системы таким образом, чтобы пользователю система представлялась монолитной. Степень прозрачности взаимосвязана с производительностью. Различают:

* **Прозрачность доступа.**  
  Скрывается разница в представлении данных и доступе к ресурсам.
* **Прозрачность расположения.**  
  Скрывается местоположения ресурса.
* **Прозрачность переноса.**  
  Скрывается факт перемещения ресурса в другое место.
* **Прозрачность смены местоположения.**  
  Скрывается факт перемещения ресурса в другое место в процессе обработки.
* **Прозрачность репликации.**  
  Скрывается факт и особенности репликации (механизм [синхронизации](https://ru.wikipedia.org/wiki/%D0%A1%D0%B8%D0%BD%D1%85%D1%80%D0%BE%D0%BD%D0%B8%D0%B7%D0%B0%D1%86%D0%B8%D1%8F_(%D0%B8%D0%BD%D1%84%D0%BE%D1%80%D0%BC%D0%B0%D1%82%D0%B8%D0%BA%D0%B0)) содержимого нескольких копий объекта (например, содержимого [базы данных](https://ru.wikipedia.org/wiki/%D0%91%D0%B0%D0%B7%D0%B0_%D0%B4%D0%B0%D0%BD%D0%BD%D1%8B%D1%85))) ресурса.
  + - * **Прозрачность параллелизма.**Возможность нескольким процессам параллельно работать с ресурсами, не оказывая влияния друг на друга.
      * **Прозрачность обработки ошибок.**Защита программных компонентов от сбоев, произошедших в других программных компонентах; восстановление после сбоев.

**Критически важными являются:**

* + Прозрачность доступа.
  + Прозрачность расположения.
  + Прозрачность паралеллизма.

1. **Управляемость.** Распределенные ресурсы не имеют центральной точки управления. Локальная оптимизация не всегда означает глобальную оптимизацию (нужно учитывать гетерогенность и другие особенности, нужен глобальный взгляд на проблему, он не всегда возможен (есть системы, никому конкретно не принадлежащие)).

**Сложности при реализации**

1. **Выбор архитектуры.** Важнейшие характеристики системы очень сильно зависят от выбранной архитектуры. Практически невозможно отказаться от принятых в начале проектирования неверных решений.   
   **Решение:** использование широко известных шаблонов (образцов) архитектур.
2. **Гетерогенность среды (разнородность).** Различаются:
   * Сетевые инфраструктуры.
   * Оборудование (например, Intel и Motorolla).
   * Программное обеспечение (например, UNIX sockets и Winsock calls).
   * Языки программирования.
   * Представления данных.

Различные компоненты системы выполняются на различных платформах. Различия должны быть скрыты. Интерфейсы и реализация могут быть разными, но базовые концепции обычно неизменны. Средства борьбы с гетерогенностью – стандарты.

**Решения:**

* Использование распространенных открытых стандартов и протоколов.
* Использование промежуточного программного обеспечения (middleware).

Middleware: промежуточный программный слой:

* + Позволяет гетерогенным узлам взаимодействовать.
  + Определяет однородную вычислительную модель.
  + Поддерживает один или несколько языков программирования.
  + Обеспечивает поддержку распределенных приложений (вызов удаленных объектов, удаленный вызов SQL, распределенную обработку транзакций).

Примеры: CORBA, JavaEE, Microsoft DCOM.

1. **Сложность развертывания.**

* Фрагментация (разделение приложения на модули развертывания).
* Конфигурация (связь модулей друг с другом (зависимости)).
* Размещение (выгрузка модулей в целевую систему, распределение вычислительных модулей между узлами (статическое или динамическое), использование специализированных инструментальных средств).

1. **Сложность отладки.** Нет глобального состояния. Параллельность приводит к неповторяемости (невоспоизводимости) результатов выполнения. Компоненты распределены по разным узлам.

**Модели распределенных систем**

1. **Модели архитектуры.** Модель архитектуры распределенной системы должна содержать решение двух проблем: физическое размещение компонентов между узлами и взаимодействие между компонентами. Реальные функции отдельных компонентов не указываются, указываются:
   * Расположение (размещение по узлам).
   * Шаблоны распределения данных и задач по их обработке.
   * Взаимодействие компонентов (роли компонентов, шаблоны взаимодействия).
2. **Другие модели.** Формальное описание различных параметров и свойств системы, модели взаимодействия, обработки ошибок, безопасности, и т.д.

**Архитектура**

* Определяет разделение системы на наиболее крупные составные части.
* Определяет конструктивные решения, которые после их принятия с трудом поддаются изменению.
* Отображает общий взгляд разработчиков на результаты проектирования системы: идентификация главных компонентов системы, способов их взаимодействия, выбор основополагающих решений, не подлежащих изменению в будущем.

**Основные принципы архитектуры**

* **Согласованность.** Частичное знание системы позволяет предсказать остальное.
* **Ортогональность.** Функции независимы и специфицированы по отдельности.
* **Соответствие.** Включаются только функции, соответствующие существенным требованиям к системе, нет ненужных функций.
* **Экономичность.** Отсутствие дублирования.
* **Прозрачность.** Функции должны быть известны пользователю.
* **Общность.** Если функция должна быть введена, ее следует вводить в таком виде, чтобы она отвечала как можно большему числу назначений.
* **Открытость.** Можно использовать функцию иначе, чем это предполагалось при проектировании.
* **Полнота.** Введенные функции должны с учетом экономических и технологических ограничений как можно полнее соответствовать требованиям и пожеланиям пользователя.

**Преодоление сложности:**

1. **Использование типовых решений - образцов проектирования.** Каждое типовое решение описывает некую повторяющуюся проблему и ключ к ее разгадке, причем таким образом, что вы можете пользоваться этим ключом многократно, ни разу не придя к одному и тому же результату.

**Структура типовых решений**

* Название решения.
* Назначение (аннотация).
* Мотивация, применимость.
* Принцип действия (структура, участники, отношения).
* Результаты.
* Реализация.

1. **Разделение системы на слои (расслоение).** Основная идея: независимость нижележащих уровней от вышележащих. Основная задача: уменьшать сложность систем, разделяя их на слои и сервисы:
   * **Слой (уровень):** группа сильно связанных и закрытых элементов, реализующих одну функциональность.
   * **Сервис:** функциональность, обеспечиваемая для вышестоящего слоя.

Рисунок 1 – Монолитная система

Логика

представления

Бизнес-логика

Логика

доступа к данным

База данных

Рисунок 2 – Двухслойная система

Логика

представления

Бизнес-логика

Логика

доступа к данным

База данных

Рисунок 3 – Трехслойная система

Логика

представления

Бизнес-логика

База данных

Логика

доступа к данным

**Три основных слоя**

* Слои:
  + представление (presentation).
  + домен (domen) – предметная область, бизнес-логика.
  + работа с данными (data source).
* Рекомендуется различать:
  + **Слой** (layer) – логическое разделение.
  + **Уровень** или **Ярус** (tier) – физическое разделение.

**Итоги**

* Распределенная система – это автономные (но соединенные средой передачи данных) узлы. Взаимодействие осуществляется посредством передачи сообщений.
* Много доводов в пользу того, что распределенные системы нужны и их нужно уметь строить.
* Распределенные системы существуют и их нужно уметь развивать и поддерживать.
* При разработке распределенных систем возникают специфические проблемы.
* Используется модель слоев для снижения сложности системы. Middleware обеспечивает дополнительное удобство и дополнительные сервисы.
* Выбор модели архитектуры зависит от особенностей задачи.
  + Клиент – сервер.
  + Модель предоставления услуг пулом серверов.
  + Модель прокси- и кэш-серверов.
  + Модель равных процессов.

**JDBC**

В 1996 году компания Sun выпустила первую версию интерфейса для организации доступа Java – приложений к базам данных (БД) JDBC (Java DataBase Connectivity). Настоящий интерфейс позволяет программистам соединяться с БД, запрашивать и обновлять данные с помощью языка SQL (Structured Query Language), который фактически стал стандартным средством взаимодействия с реляционными базами данных.

Java и JDBC имеют весомое преимущество по сравнению с другими инструментами для работы с БД. Программы, созданные с помощью Java и JDBC, не зависят от используемой платформы и поставщика программного обеспечения.

Средства JDBC неоднократно обновлялись. В состав JDK 1.2, выпущенной в 1998 году, была включена версия 2 JDBC. В JDK 1.4 и 1.5 была включена версия 3 JDBC и т.д.

С самого начала поставщики программного обеспечения для БД поддерживали идею создания компанией Sun стандартного сетевого протокола доступа к БД, но при условии, что за основу будет принят их собственный протокол. В конечно итоге поставщики СУБД и инcтрументов доступа к базам всё-таки согласились с тем, что сотрудники Sun создали Java API для SQL-доступа к данным и диспетчер драйверов, который позволил бы подключать к базам драйверы независимых производителей. Такой подход позволял производителям СУБД создавать собственный драйверы, которые подключались бы с помощью диспетчера и должны лишь соответствовать требованиям API диспетчера драйверов.

В результате было создано два интерфейса. Разработчики приложений используют JDBC API, а поставщики БД и инструментальных средств – JDBC Driver API. Подход, используемый при создании JDBC, основан на модели ODBC(Open Database Connectivity) — интерфейса доступа к БД, разработанного фирмой [Microsoft](https://ru.wikipedia.org/wiki/Microsoft): программы, соответствующие API, могут взаимодействовать с диспетчером драйверов JDBC, который, в свою очередь, использует присоединенные драйверы для организации взаимодействия с БД.

**Типы JDBC – драйверов:**

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

Большинство поставщиков БД применяют драйверы типа 3 или 4. Кроме того, некоторые компании специализируются на создании драйверов, которые позволяют добиться более полного соответствия принятым стандартам, поддерживают большее количество платформ, обладают более высокой производительностью или надежностью, чем драйверы, предлагаемые производителями СУБД.

**Основные цели интерфейса JDBC можно сформулировать следующим образом:**

1) Разработчики создают программы на языке Java, пользуясь для доступа к БД стандартными средствами языка SQL, при этом они следуют только соглашениям языка Java.

2) Производители СУБД и инструментальных средств поставляют драйверы только низкого уровня.

**Примеры использования JDBC**

Согласно традиционной модели клиент/сервер, графический пользовательский интерфейс реализуется на стороне клиента, а БД располагается на стороне сервера. В этом случае драйвер JDBC находится на стороне клиента(рис.1).

Протокол БД

Сервер БД

Клиент

JDBC

RКлиент

Рисунок 4 – Традиционная структура приложения клиент/сервер

Однако современная тенденция развития программного обеспечения заключается в переходе от архитектуры клиент/сервер к «трехуровневой модели» (рис.2).

HTTP, RMI, … Протокол БД

Клиент (визуальное представление)

Сервер БД

JDBC

Промежуточный уровень  
(бизнес-логика)

Рисунок 5 – Приложение, созданное в соответствии с трехуровневой моделью

В ней клиент не формирует обращений к БД. Вместо этого он обращается к средствам промежуточного уровня на сервере, который, в свою очередь, выполняет запросы к БД. Трехуровневая модель отделяет визуальное представление (на компьютере-клиенте) от бизнес-логики (промежуточный уровень) и данных (в базе данных). Таким образом, становится возможным доступ к тем же данным и тем же бизнес-правилам посредством клиентов различных типов, например, Java-приложений, апплетов и Web-форм.

Взаимодействие между клиентом и промежуточным уровнем может быть реализовано на основе протокола HTTP (при использовании Web-браузера в качестве клиента), средств RMI (при использовании приложений или апплетов) или с помощью какого-либо другого механизма. JDBC используется для управления взаимодействием между промежуточным уровнем и БД.

**Основы программирования JDBC**

Классы, используемые для программирования JDBC, содержатся в пакетах java.sql и javax.sql.

Класс *DriverManager* отвечает за выбор драйверов БД и создание нового соединения с БД. Однако перед тем, как диспетчер активизирует драйвер, его надо зарегистрировать. Имя драйвера можно задать в командной строке:

*java –Djdbc.drivers=org.postgresql.Driver MyClass*

Кроме того, приложение может установить системное свойство с помощью вызова:

*System.setProperty("jdbc.drivers", "org.postgresql.Driver");*

Драйвер также можно зарегистрировать вручную, загрузив его класс:

*Class.forName("org.postgresql.Driver");*

Такой подход применяют, если диспетчер не может загрузить драйвер из-за ограничений конкретного драйвера либо в том случае, когда программа выполняется в среде контейнера, например, сервлета.

После регистрации драйвера можно установить соединение с БД. При работе с JDBC база данных представляется с помощью URL (Uniform Resource Locator). Для PostgreSQL он принимает одну из следующих форм:

* *jdbc:postgresql:database*
* *jdbc:postgresql://host/database*
* *jdbc:postgresql://host:port/database*

Для подключения к базе нужно использовать класс Connection:

*Connection connection = DriverManager.getConnection(url, username, password);*

Например:

*Connection connection = DriverManager.getConnection("jdbc:postgresql://localhost:5432/MyTest", "postgres", "root");*

Для указания драйвера, URL, имени пользователя и пароля удобно применять файл свойств:

*jdbc.drivers =* *org.postgresql.Driver*

*jdbc.url = jdbc:postgresql://localhost:5432/MyTest*

*jdbc.username = postgres*

*jdbc.password = root*

Далее читаем файл свойств и устанавливаем соединение с БД:

*Properties props = new Properties();*

*FileInputStream in = new FileInputStream("database.properties");*

*props.load(in);*

*in.close();*

*String drivers = props.getProperty("jdbc.drivers");*

*if(drivers != null) System.setProperty("jdbc.drivers", drivers);*

*String url = props.getProperty("jdbc.url");*

*String username = props.getProperty("jdbc.username");*

*String password = props.getProperty("jdbc.password");*

*Connection connection = DriverManager.getConnection(url, username, password);*

Метод *getConnection()* возвращает объект *Connection*, с помощью которого создаётся объект *Statement,* который используется для выполнения SQL – команд. Метод *executeQuery()* возвращает объект *ResultSet,* который можно использовать для построчного просмотра результатов.

Извлечение данных:

*Statement statement = connection.createStatement();*

*ResultSet resultSet = statement.executeQuery("SELECT \* FROM item");*

*while (resultSet.next()) {*

*String str = resultSet.getString("name");*

*System.out.println(str);*

*}*

В классе *ResultSet* итератор располагается в начале работы перед первой строкой, поэтому для перемещения его к первой строке нужно вызвать метод *next().* Строки в наборе результатов располагаются совершенно произвольно. Если порядок следования важен, его необходимо установить с помощью оператора ORDER BY.

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

*resultSet.getString(1);*

*resultSet.getDouble("Price");*

Для каждого типа данных языка Java предусмотрен отдельный метод извлечения информации, например, *getString() и getDouble().* Для каждого из них реализовано два варианта, один из которых предполагает числовой, а второй – строковый параметр. При использовании числового параметра метод извлечет данные из столбца с указанным номером, нумерация столбцов начинается с 1. При использовании строкового параметра метод извлечет данные из столбца с указанным именем.

Если указанный тип не соответствует фактическому типу, метод извлечения данных выполняет преобразование. Например, метод *resultSet.getString("Price")* преобразует число с плавающей точкой из столбца Price в строку.

Типы данных SQL и Java не всегда совпадают. В таблице 1 приведен список основных типов данных языка SQL и их эквивалентов в Java.

Таблица 1

|  |  |
| --- | --- |
| Типы данных языка SQL | Типы данных языка Java |
| INTEGER или INT | int |
| SMALLINT | short |
| NUMERIC(m,n), DECIMAL(m,n) или DEC(m,n) | java.math.BigDecimal |
| REAL | float |
| DOUBLE | double |
| CHARACTER(n) или CHAR(n), VARCHAR | String |
| BOOLEAN | boolean |
| DATE | java.sql.Date |
| TIME | java.sql.Time |

Метод *executeUpdate()* возвращает количество строк, полученных в результате выполнения SQL-команды. Он может применяться для выполнения команд INSERT, UPDATE и DELETE, а также команд определения данных, в частности, CREATE TABLE и DROP TABLE.

Закончив работу с классом ResultSet, Statement или Connection нужно вызвать метод *close(),* так как перечисленные объекты используют структуры данных большого объема и не стоит ждать, пока процедура сборки мусора освободит ресурсы. Метод *close()* класса Statement автоматически закрывает связанные с ним наборы результатов( если они открыты). Аналогично, метод *close()* класса Connection закрывает все объекты Statement для этого соединения. Если время существования соединения невелико, нужно поместить вызов метода *close()* в блок finally, гарантируя, что объект соединения не останется открытым.

**Предварительно подготовленные команды**

Вместо создания отдельной SQL-команды для каждого запроса следует предварительно подготовить запрос с подстановочной переменной и многократно использовать его, меняя только значение переменной. Эта возможность существенно повышает эффективность работы программы. Перед каждым выполнением запроса СУБД создает план его эффективного исполнения. Предварительно подготавливая запрос для последующего многократного повторного использования, можно избежать повторного создания плана. Подстановочная переменная в запросе обозначается символом ?. При использовании нескольких подстановочных переменных нужно следить за их расположением, чтобы правильно выполнять подстановку фактических значений.

*String s = "SELECT name, price FROM tovar WHERE price >? AND price <?";*

*PreparedStatement ps = connection.prepareStatement(s);*

*ps.setDouble(1,2000);*

*ps.setDouble(2,6000);*

*ResultSet rs1 = ps.executeQuery();*

Перед выполнением предварительно подготовленного запроса нужно с помощью метода *set()* связать подстановочные переменные с их фактическими значениями. Как и при использовании методов *get()*, класса *ResultSet,* для различных типов предусмотрены разные методы *set().* Например:

*ps.setDouble(1,2000);*

Первый параметр обозначает номер позиции подстановочной переменной, второй – её фактическое значение. При повторном использовании предварительно подготовленного запроса с несколькими подстановочными переменными все связи остаются в силе, если только они не изменены с помощью метода *set().* Это значит, что методы *set()* нужно вызывать только для тех подстановочных переменных, которые изменяются в последующих запросах. Для изменения данных используется метод *executeUpdate().*

**Метаданные**

В JDBC предусмотрены дополнительные возможности для получения информации о структуре таблиц и самой БД. Например, можно получить список всех таблиц БД либо имена всех столбцов с типами данных в них. Эти сведения пригодятся, если вы пишете программный продукт для работы с произвольной БД.

В языке SQL информация и структуре БД и её компонентов называется **метаданными**. Информация о структуре БД хранится в классе DatabaseMetaData. Ниже приведен пример получения набора результатов, содержащего информацию обо всех таблицах БД.

*DatabaseMetaData meta = connection.getMetaData();*

*ResultSet mrs = meta.getTables(null, null, null, new String[] {"TABLE"});*

Метод *getTables(String catalog, String schemaPattern, String tableNamePattern String types[])* представляет описание всех таблиц в каталоге, которые соответствуют шаблонам схемы и имен таблиц, а также заданным типам. **Схемой** называется группа связанных таблиц и полномочий доступа. **Каталогом** называется связанная группа схем. Параметры *catalog* и *schemaPattern* могут быть пустыми строками(" ") для извлечения таблиц без каталога и схемы либо иметь значение null, если нужно возвращать таблицы без учета каталога или схемы. Массив types содержит имена типов таблиц: TABLE, VIEW, ALIAS и т.д. Если для types задано значение null, то будут возвращены таблицы всех типов. Результат выполнения запроса содержит 5 столбцов, которые имеют тип String и показаны в таблице 2.

Таблица 2

|  |  |  |
| --- | --- | --- |
| 1 | TABLE\_CAT | Каталог таблиц (значение может быть равно null) |
| 2 | TABLE\_SCHEM | Схема (значение может быть равно null) |
| 3 | TABLE\_NAME | Имя таблицы |
| 4 | TABLE\_TYPE | Тип таблицы |
| 5 | REMARKS | Комментарии для таблицы |

Если нас интересует имя таблицы, т.е. третий столбец, то нужно вызвать метод *mrs.getString(3).*

Сведения о структуру набора результатов хранятся в классе *ResultSetMetaData.*

*ResultSet rs = statement.executeQuery("SELECT \* FROM item");*

*ResultSetMetaData rsmd= rs.getMetaData();*

После получения результатов выполнения запроса можно определить количество столбцов методом *getColumnCount()*, имена столбцов *getColumnLabel(int column)* и т.д.

**XML**

SGML(Standard Generalized Markup Language)

HTML XML

(Hypertext Markup Language) (Extensible Markup Language)

* + SGML – с 1970-х годов использовался для описания структуры сложных документов в некоторых отраслях промышленности с высокими требованиями к документации.
  + HTML – язык разметки гипертекста.
  + XML – язык для описания иерархических данных.

**Отличия XML от HTML**

* В отличие от HTML, в языке XML учитывается регистр символов.
* В языке HTML некоторые закрывающие дескрипторы могут отсутствовать, в XML это не разрешается.
* Для элементов, тело которых отсутствует, в XML предусмотрена сокращенная запись открывающего дескриптора, совмещённого с закрывающим. В этом случае открывающий дескриптор заканчивается символом /
* В XML значения атрибутов должны быть помещены в кавычки, а в HTML кавычки могут отсутствовать.
* В HTML допускается использование имен атрибутов без указания их значений, а в HML все атрибуты должны иметь значения.

Пример 1

Пример XML:

*<configuration>*

*<title>*

*<font>*

*<name>Helvetica</name>*

*<size>36</size>*

*</font>*

*</title>*

*<body>*

*<font>*

*<name>Times Roman</name>*

*<size>12</size>*

*</font>*

*</body>*

*<window>*

*<width>400</width>*

*<height>200</height>*

*</window>*

*<color>*

*<red>0</red>*

*<green>50</green>*

*<blue>100</blue>*

*</color>*

*<menu>*

*<item>Times Roman</item>*

*<item>Helvetica</item>*

*</menu>*

*</configuration>*

**Структура XML - документа**

* **XML- документ начинается с заголовка (необязательно)***<?xml version="1.0"?>*

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

* **Далее идёт DTD (Document Type Definition) (необязательно)***<!DOCTYPE web-app PUBLIC "-//Sun Microsystems, Inc.*

*//DTD Web Application 2.2//EN“*

*"http://java.sun.com/j2ee/dtds/web-app\_2\_2.dtd">*

* + **Наконец, тело XML-документа содержит корневой  
    элемент,** в состав которого могут входить другие  
    элементы

*<configuration>* – корневой элемент

* Каждый элемент может содержать дочерние элементы, текст либо и то, и другое
* XML-документы рекомендуется формировать так, чтобы элементы содержали либо дочерние элементы, либо текст, т.е. следует избегать создания структур:

*<font>*

*Helvetica*

*<size>36</size>*

*</font>*

Такая структура называется **смешанным содержимым/**

* XML-элементы могут содержать атрибуты

*<size unit="pt">36</size>*

* Элементы и текст являются основными компонентами XML-документов, но возможны и другие инструкции:
  + Ссылки на символы &#d или &#xh, где d – десятичное, а h – шестнадцатеричное Unicode-значение. Например, &#233, &#x2122
  + Ссылки на примитивы имеют вид &name. Например,  
    &lt; &gt; &amp; &quot; &apos;
  + CDATA-разделы ограничены группами символов <![CDATA[ и ]]>, предназначены для включения строк с символами < > &, которые не следует интерпретировать как символы разметки

< ! [CDATA[ < & > are my favorite delimiters ] ] >

* + Инструкции обработки ограничиваются символами <? и ?>, нужны приложению в котором используется данный XML-документ  
    *<?xml version="1.0"?>*
  + Комментарии ограничиваются символами <! -- и -->. В комментариях нельзя использовать строки типа --. Не следует включать в комментарии скрытые команды, для их выполнения предназначены инструкции обработки

*<!-- This is a comment. -->*

**DTD и XML-схема**

* + XML-анализатор может автоматически проверять корректность структуры документа
  + Для указания структуры документа нужно создать DTD либо XML-схему
  + Они указывают допустимые дочерние узлы элементов и атрибутов каждого элемента
  + XML-схема позволяет формулировать сложные правила, которые нельзя выразить средствами DTD
  + В отличие от DTD, правила, соответствующие схеме, выражаются на языке XML, что упрощает их обработку
  + XML-схема сложнее DTD

**Document Type Definition (DTD)**

* + Существует несколько способов связывания DTD с XML-документом:
    - Первый способ  
       *<?xml version="1.0"?>  
      <!DOCTYPE configuration [  
       <!ELEMENT configuration …>*

*другие правила*

*…*

*]>*

*< configuration >*

*…*

*</ configuration>*

* + Второй способ  
    *<!DOCTYPE configuration SYSTEM "config.dtd">*
  + Третий способ  
    *<!DOCTYPE configuration SYSTEM "http://myserver.com/config.dtd">*
  + Рассмотрим правила, которые могут задаваться посредством DTD.
  + Правило ELEMENT указывает дочерние узлы данного элемента в виде регулярного выражения, которое состоит из перечисленных в таблице 3 компонентов.

Таблица 3

|  |  |
| --- | --- |
| **Правило** | **Описание** |
| E\* | 0 или более элементов Е |
| E+ | 1 или более элементов Е |
| E? | 0 или 1 элемент Е |
| E1|E2|…|En | Одно из элементов Е1, Е2, …, Еn |
| E1, E2, …, En | Элемент E1, за которым следуют E2, …, En |
| #PCDATA | Текст |
| (#PCDATA|E1|…|En)\* | 0 или более текстовых элементов и элементы Е1, Е2, …, Еn, расположенные в произвольном порядке (смешанное содержание) |
| ANY | Любой дочерний узел |
| EMPTY | Нет дочерних узлов |

**Примеры правил**

* + *<!ELEMENT menu (item)\*>*
  + *<!ELEMENT font (name,size)>*

*<!ELEMENT name (#PCDATA)>*

*<!ELEMENT size (#PCDATA)>*

* + *<!ELEMENT chapter (intro,(heading,(para|image|table|note)+)+)>*
  + Для описания допустимых атрибутов элементов используется приведенный ниже синтаксис:  
    *<!ATTLIST элемент атрибут тип поведение\_по\_умолчанию>*
  + В таблицах 4 и 5 приведены допустимые типы атрибутов и синтаксис параметра поведение\_по\_умолчанию.

Таблица 4 – Типы атрибутов

|  |  |
| --- | --- |
| **Тип** | **Описание** |
| CDATA | Произвольная символьная строка |
| (A1|A2|…|An) | Один из строковых атрибутов A1| A2| …|An |
| NMTOKEN, NMTOKENS | Одна или несколько лексем, соответствующих имени |
| ID | Уникальный идентификатор |
| IDREF, IDREFS | Одна или несколько ссылка на уникальный идентификатор |
| ENTITY, ENTITIES | Один или несколько непроанализированных примитивов |

Таблица 5 – Поведение по умолчанию

|  |  |
| --- | --- |
| **Тип** | **Описание** |
| #REQUIRED | Атрибут является обязательным |
| #IMPLIED | Атрибут не является обязательным |
| A | Атрибут не является обязательным, анализатор возвращает значение А, если атрибут не указан |
| #FIXED A | Атрибут не должен быть указан, либо должен быть равен А, в любом случае анализатор возвращает значение А |

**Примеры выражений:**

*<!ELEMENT gridbag (row)\*>*

*<!ELEMENT row (cell)\*>*

*<!ATTLIST cell gridwidth CDATA "1">*

*<!ATTLIST cell gridheight CDATA "1">*

*<!ATTLIST cell fill (NONE|BOTH|HORIZONTAL|VERTICAL) "NONE">*

*<!ATTLIST cell anchor (CENTER|NORTH|NORTHEAST|EAST*

*|SOUTHEAST|SOUTH|SOUTHWEST|WEST|NORTHWEST)*

*"CENTER">*

*<!ATTLIST cell ipadx CDATA "0">*

*<!ATTLIST cell ipady CDATA "0">*

**DTD и соответствующий XML-документ**

*<?xml version="1.0"?>*

*<!DOCTYPE employee [*

*<!ELEMENT employee (Name, Dept, Title)>*

*<!ELEMENT Name (#PCDATA)>*

*<!ELEMENT Dept (#PCDATA)>*

*<!ELEMENT Title (#PCDATA)>*

*]>*

*<employee>*

*<Name>John Goodman</Name>*

*<Dept>Manufacturing</Dept>*

*<Title>Supervisor</Title>*

*</employee>*

**XML-схема**

* + Для включения в документ ссылку на файл схемы нужно указать в корневом элементе соответствующие атрибуты, например:  
    *<?xml version="1.0"?>   
    < configuration xmlns: xsi = http://www.w3.org/2001/XMLSchema-instance  
     xsi: noNamespaceSchemaLocation ="config.xsd">*

*…  
 < /configuration >*

* + В данном случае в составе корневого элемента содержится информация о том, что при проверке документа должен использоваться файл схемы config.xsd.
  + Если применяется пространство имен – синтаксис усложняется.
  + Схема определяет тип каждого элемента
  + Это может быть простой или сложный тип
  + Некоторые простые типы предопределены в XML-схеме:
    - *xsd:string*
    - *xsd:int*
    - *xsd:boolean*
  + Можно определять собственные простые типы
  + Вместо префикса xsd (обозначает пространство имен XSL Schema Definition), можно применять xs:
  + Определяя элемент, следует указать его тип:
    - *<xsd:element name="name" type="xsd:string"/>*
    - *<xsd:element name=“size" type="xsd:int"/>*
  + Простые типы можно объединять в сложные:  
    *<xsd:complexType name = "FontType">*

*<xsd:sequence>*

*<xsd:element ref = "name" />*

*<xsd:element ref = “size" />*

*<xsd:element ref = “style" />*

*</xsd:sequence>  
 </xsd:complexType >*

* + FontType – последовательность элементов name, size и style.
  + Атрибут ref используется для ссылки на определения, находящиеся в схеме.
  + Возможны вложенные определения.
  + Для определения атрибутов необходимо включать элементы xsd:attribute.

**Dom и SAX**

* + Для разборки XML-документа необходимо выполнить синтаксический анализ или разбор его содержимого.
  + **Анализатором (parser)** называется программа, которая считывает файл, подтверждает корректность его формата, разбивает данные на составные элементы и предоставляет программисту доступ к ним.
  + Существует два основных типа XML-анализаторов:
    - **DOM-анализатор (Document Object Model)** считывает XML-документ и представляет его в виде древовидной структуры.
    - **SAX-анализатор (Simple API for XML)** генерирует события по мере чтения XML-документа.
  + DOM-анализатор представляет документ как древовидную структуру, SAX-анализатор генерирует события в ходе разбора документа.
  + При обработке больших документов DOM-анализатор неэффективен, так как требуется память для хранения всей древовидной структуры, SAX-анализатор более экономичный.
  + Сформированную древовидную структуру можно многократно и по-разному обходить, при использовании SAX-анализатора события генерируются последовательно в порядке возникновения и повторный обход документа невозможен.

**Dom**

* + Пакет org.w3c.dom содержит определения типов интерфейсов, например, Document и Element.
  + Ряд компаний, например, IBM и Apache, разработали собственные варианты DOM-анализаторов, которые реализуют эти интерфейсы.
  + В API обработки XML-документов(Java API for XML Processing-JAXP) компании Sun предусмотрена возможность подключения данных анализаторов.
  + Кроме того, компания Sun включила в состав пакета JDK собственный анализатор.

Для считывания XML-документа потребуется объект **DocumentBuilder,** который можно получить с помощью класса **DocumentBuilder Factory.**

*DocumentBuilder Factory factory = DocumentBuilder Factory.newInstance();  
DocumentBuilder builder = factory.new DocumentBuilder();*

*File f = … // можно использовать URL или InputStream  
Document doc = builder.parse(f);*

* + Объект **Document** является внутренним представлением древовидной структуры XML-документа.
  + Он состоит из экземпляров классов, реализующих интерфейс **Node** и его дочерние интерфейсы (**Attr, Comment, Document, Element, Text**).
  + Анализ содержимого документа начинается с вызова метода **getDocumentElement()**, который возвращает корневой элемент  
    Element root = doc.getDocumentElement();
  + Метод **getTagName()** возвращает имя дескриптора элемента.
  + Для извлечения элементов, дочерних по отношению к данному (вложенных элементов, текста, комментариев или других узлов) предназначен метод **getChilNodes()**, возвращающий набор данных типа **NodeList.**
  + Метод **item()** возвращает элемент набора с заданным индексом.
  + Метод **getLength()** возвращает общее количество элементов.
  + Методы **getFirstChild()** и **getLastChild()** используются для получения первого и последнего дочернего узла.
  + Метод **getNextSibling()** позволяет получить следующий узел того же уровня.
  + Метод **getAttributes()** возвращает карту **NameNodeMap** с объектами типа Node, описывающими атрибуты.
  + Методы **getNodeName()** и **getNodeValue()** извлекают имя и значение атрибута соответственно.
  + Метод **getAttributes(**"**имя** "**)** извлекает значение атрибута по имени.

Пример 2

* *import org.w3c.dom.\*;*
* *import javax.xml.parsers.\*;*
* *public class MyXml {*
* *public static void main(String[] args) {*
* *try {*
* *DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();*
* *DocumentBuilder builder = factory.newDocumentBuilder();*
* *Document doc = builder.parse("G:\\Haulmont\\java\_SE\\config.xml");*
* *System.out.print("The elements are: ");*
* *printElements(doc);*
* *System.out.println("The attributes of each element are: ");*
* *printElementAttributes(doc);*
* *} catch (Exception e) { System.out.println(e.toString()); }*
* *}*
* *static void printElements(Document doc) {*
* *NodeList nl = doc.getElementsByTagName("\*");*
* *for (int i = 0; i < nl.getLength(); i++) {*
* *System.out.print(nl.item(i).getNodeName() + " ");*
* *}*
* *System.out.println();*
* *}*
* *static void printElementAttributes(Document doc) {*
* *NodeList nl = doc.getElementsByTagName("\*");*
* *for (int j = 0; j < nl.getLength(); j++) {*
* *Element e = (Element) nl.item(j);*
* *System.out.println(e.getTagName() + ":");*
* *NamedNodeMap nnm = e.getAttributes();*
* *if (nnm != null) {*
* *for (int i = 0; i < nnm.getLength(); i++) {*
* *Node n = nnm.item(i);*
* *System.out.print(" " + n.getNodeName() + " = " + n.getNodeValue());*
* *}*
* *}*
* *System.out.println();*
* *} } }*

**SAX**

* + Для использования SAX-анализатора необходимо создать обработчик событий, которые происходят во время разбора XML-документа.
  + В интерфейсе **ContentHandler** определено несколько методов, к которым обращается анализатор:
    - Методы **startElement()** и **endElement()** вызываются при получении открывающего и закрывающего дескрипторов.
    - Метод **characters()** вызывается при получении символьных данных.
    - Методы **startDocument()** и **endDocument()** вызываются в начале и в конце документа.
  + Для получения SAX-анализатора используется следующая команда:  
     *SAXParserFactory factory = SAXParserFactory.newInstance();*

*SAXParser parser = factory.newSAXParser();*

* + Для обработки документа следует использовать команду:

*parser.parse(source, handler);*Здесь source – это источник входных данных, который может быть файлом, URL или входным потоком;  
handler – подкласс класса **DefaultHandler,** в котором определены методы, объявленные в интерфейсах **ContentHandler, DTDHandler, EntityResolver, ErrorHandler.**

* + Эти методы не выполняют никаких действий.

Пример 3

* *import java.io.IOException;*
* *import javax.xml.parsers.\*;*
* *import org.xml.sax.\*;*
* *import org.xml.sax.helpers.\*;*
* *class CustomSAXParser extends DefaultHandler{*
* *public void startDocument() throws SAXException {*
* *System.out.println("Start parse XML...");*
* *}*
* *public void startElement(String namespaceURI, String localName,*
* *String qName, Attributes atts) throws SAXException {*
* *System.out.println("Start element " + qName + ":");*
* *}*
* *public void endElement(String namespaceURI, String localName,*
* *String qName) throws SAXException {*
* *System.out.println("End element " + qName + ":");*
* *}*
* *public void endDocument() {*
* *System.out.println("Stop parse XML...");*
* *}*
* *public static void main(String[] args) {*
* *try {*
* *SAXParserFactory factory = SAXParserFactory.newInstance();*
* *SAXParser parser = factory.newSAXParser();*
* *CustomSAXParser custom = new CustomSAXParser();*
* *parser.parse("G:\\Haulmont\\java\_SE\\config.xml", custom);*
* *} catch (IOException ex) {*
* *ex.printStackTrace();*
* *} catch (ParserConfigurationException ex) {*
* *ex.printStackTrace();*
* *} catch (SAXException ex) {*
* *ex.printStackTrace();*
* *} } }*

**Создание XML-документов**

* + Используется пакет **javax.xml.transform.**
  + Удобнее создать древовидную структуру DOM, представляющую документ, а затем записать содержимое этого дерева в файл.
  + В первую очередь следует сгенерировать пустой документ с помощью метода **newDocument()** класса **DocumentBuilder***Document doc = builder.newDocument();*
  + Затем для создания элементов документа нужно применить метод **createElement()**  класса **Document***Element rootElement = doc.createElement(rootName);  
    Element childElement = doc.createElement(childName);*
  + Далее создаем текстовые узлы с помощью метода **createTextNode().**

*Text textNode = doc.createTextNode(textContents);*

* + Для включения в документ корневого элемента и для связывания дочерних узлов с родительскими используется приведенный ниже код:  
    *doc.appendChild(rootElement);  
    rootElement.appendChild(childElement);  
    childElement.appendChild(textNode);*
  + При формировании дерева DOM атрибуты элементов задаются с помощью метода **setAttribute()** класса **Element***rootElement.setAttribute(name, value);*

Для записи дерева DOM в выходной поток к документу применяется преобразование (XSLT-преобразование), в процессе которого не выполняется никаких действий и результат записывается в выходной поток.

* + Для включения узла DOCTYPE в состав выходных данных нужно указать в качестве свойств идентификаторы SYSTEM и PUBLIC  
    *Transformer t = TransformerFactory.newInstance().newTransformer();  
    t.setOutputProperty(OutputKeys.DOCTYPE\_SYSTEM, systemIdentifier);  
    t.setOutputProperty(OutputKeys.DOCTYPE\_PUBLIC, publicIdentifier);   
    t.transform(new DOMSource(doc), new StreamResult(new FileOutputStream(file)));*

Пример 4

*import org.w3c.dom.\*;*

*import javax.xml.parsers.\*;*

*import javax.xml.transform.\*;*

*import javax.xml.transform.dom.\*;*

*import javax.xml.transform.stream.\*;*

*import java.io.\*;*

*public class WriteXML {*

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

*try {*

*DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();*

*DocumentBuilder builder = factory.newDocumentBuilder();*

*Document doc = builder.parse("G:\\Haulmont\\java\_SE\\config.xml");*

*Transformer t = TransformerFactory.newInstance().newTransformer();*

*t.setOutputProperty(OutputKeys.DOCTYPE\_SYSTEM, doc.getDoctype().getSystemId());*

*FileOutputStream f = new FileOutputStream("G:\\Haulmont\\java\_SE\\config1.xml");*

*t.transform(new DOMSource(doc), new StreamResult (f));*

*} catch (Exception e) {*

*e.printStackTrace();*

*}*

*}*

*}*

**Hibernate**

[ORM](http://ru.wikipedia.org/wiki/ORM) (Object-relational mapping) — это отображение объектов какого-либо объектно-ориентированного языка в структуры реляционных баз данных.

**Преимущества ORM**

* Нет необходимости писать рутинные insert/update/delete/select для CRUD(create, read, update, delete) операций.
* Условия связи между объектами (строками таблиц) указываются декларативно в одном месте.
* Возможность использовать полиморфные запросы для иерархий классов.
* Высокая степень независимости от конкретной СУБД.

**Недостатки ORM**

* Возможны проблемы с производительностью для сложных запросов на объектном SQL.
* Затрудняет использование специфических конструкций языка SQL конкретной СУБД.
* Потеря объектно-реляционного соответствия.

ORM-решением для языка [Java](http://ru.wikipedia.org/wiki/Java) является технология [Hibernate](http://ru.wikipedia.org/wiki/Hibernate_(%D0%B1%D0%B8%D0%B1%D0%BB%D0%B8%D0%BE%D1%82%D0%B5%D0%BA%D0%B0)), которая не только связывает Java классы с таблицами базы данных, но также предоставляет средства для автоматического построения запросов и извлечения данных и может значительно уменьшить время разработки, которое обычно тратится на ручное написание SQL и [JDBC](http://ru.wikipedia.org/wiki/Jdbc) кода. Hibernate генерирует SQL вызовы и освобождает разработчика от ручной обработки результирующего набора данных и конвертации объектов.

Mapping (сопоставление, проецирование) Java-классов с таблицами БД осуществляется с помощью конфигурационных [XML](https://ru.wikipedia.org/wiki/XML) файлов или [Java-аннотаций](https://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%BD%D0%BE%D1%82%D0%B0%D1%86%D0%B8%D1%8F_(Java)). Обеспечиваются возможности по организации отношений между классами «[один-ко-многим](https://ru.wikipedia.org/w/index.php?title=%D0%9E%D0%B4%D0%B8%D0%BD-%D0%BA%D0%BE-%D0%BC%D0%BD%D0%BE%D0%B3%D0%B8%D0%BC&action=edit&redlink=1)» и «[многие-ко-многим](https://ru.wikipedia.org/w/index.php?title=%D0%9C%D0%BD%D0%BE%D0%B3%D0%B8%D0%B5-%D0%BA%D0%BE-%D0%BC%D0%BD%D0%BE%D0%B3%D0%B8%D0%BC&action=edit&redlink=1)».

Для применения [Hibernate](http://ru.wikipedia.org/wiki/Hibernate_(%D0%B1%D0%B8%D0%B1%D0%BB%D0%B8%D0%BE%D1%82%D0%B5%D0%BA%D0%B0)) на практике во-первых, необходимо описать [XML](https://ru.wikipedia.org/wiki/XML) файл, содержащий информацию о настройках связи с БД и о способе Mapping. Располагаться он должен в самом корне дерева классов.

Пример 5

Файл hibernate.cfg.xml.

*<hibernate-configuration>*

*<session-factory>*

*<!-- PostgreSQL connection settings -->*

*<property name="connection.driver\_class">org.postgresql.Driver</property>*

*<property name="connection.url">jdbc:postgresql://localhost:5433/orm</property>*

*<property name="connection.username">postgres</property>*

*<property name="connection.password">sa</property>*

*<property name="dialect">org.hibernate.dialect.PostgreSQLDialect</property>*

*<property name="hbm2ddl.auto">update</property>*

*<!-- Other hibernate settings -->*

*<property name="show\_sql">true</property>*

*<!-- Annotated classes (Mapping from Annotations) -->*

*<mapping package="org.haulmont.catalog.model"/>*

*<mapping class="org.haulmont.catalog.model.Shop"/>*

*<mapping class="org.haulmont.catalog.model.Book"/>*

*<mapping class="org.haulmont.catalog.model.Magazine"/>*

*<mapping class="org.haulmont.catalog.model.Author"/>*

*<!-- Mapping from configuration file (as an alternative to annotations) -->*

*<!--<mapping resource="mapping.hbm.xml"/>-->*

*</session-factory>*

*</hibernate-configuration>*

* connection.driver\_class Показывает класс драйвера, который используется для соединения с базой данных.
* connection.url –URL для соединения с базой данных.
* connection.username – логин к базе данных.
* connection.password – пароль к базе данных.
* dialect – т.к. Hibernate может работать с разными базами данных, и каждая имеет какие-то особенности (генерация первичного ключа, страничный вывод, функции), нам надо указать, с какой базой мы работаем. В данном случае у нас PostgreSQL, что мы и указываем.
* hbm2ddl.auto - свойство, которое указывается что нужно сделать со схемой БД при инициализации. Может принимать такие значения:  
  **update** - сверяет схему БД с имеющимися конфигурациями классов, если мы внесли какие-то изменения, они автоматически занесуться в БД. При этом данные, которые были занесены в базу не изменятся - даже, если мы решили удалить некоторые поля из таблицы, они все одно останутся;

**create** - каждый раз при запуске приложения, схема БД будет создаваться наново. Все данные, которые были занесены раньше, будут удалены;

**create-drop** - каждый раз при запуске приложения, схема БД будет создаваться наново, а при завершении - удаляться. Все данные, которые были занесены во время работы приложения, будут удалены по завершению приложения;

**validate** - при инициализации будет совершена проверка соответствуют ли конфигурации классов и схема БД. Если мы внесли изменение в конфигурацию какого-то класса, а схема в БД не была изменена, выбросится исключение;

* show\_sql – данное свойство указывает, будут ли выводится SQL-запросы, которые генерит Hibernate на консоль. В процессе отладки это бывает очень удобно.
* тэг – mapping. Он используется для перечисления всех классов, которые имеют связь с базой данных. Т.е. если вы хотите использовать какой-то класс для связи с какой-то таблицей, вы должны его здесь указать.

Во-вторых, нужно описать классы для работы с данными.

Пример 6



* Класс **Shop** связан с таблицей **SHOP.**
* Первичный ключ – **ID**, связан со свойством **id** (функции **getId()**/**setId()**).
* Для генерации значений первичного ключа **ID** используется sequence **SEQ\_ID.**
* Атрибут **NAME** связан со свойством **name** (функции **getName()**/**setName()**).
* Объекты **Shop** содержат список **Item**:
  + связь “один-ко-многим” (**@OneToMany**)
  + обязательная (**nullable=false**)
  + Внешний ключ, определяющий связь – **SHOP\_ID.**
* Список **Item** загружаются по требованию (**FetchType.LAZY**).
* При сохранении объекта **Shop** автоматически сохраняются все его **items** (**CascadeType.ALL**).

Рассмотрим подробнее используемые аннотации:

* @Entity – говорит о том, что данный класс представляет собой сущность, которую надо сохранять в базе данных.
* @Table – показывает, какая таблица используется для хранения.
* @Id – говорит о том, что данное поле является идентификатором.
* @GeneratedValue – аннотация дает указание, что данная величина будет генериться.
* @Column –аннотация для указания имени столбца, с которым связано поле класса.
* @JoinColumn – данная анотация по сути похожа на @Column. Только она указывает, что колонка является связующей.
* @ManyToOne – аннотация указывает, что поле указывает на другой класс, который связан с текущим классом связью многие-к-одному. Здесь также указывается правило каскада cascade= CascadeType.REFRESH – в упрощенном виде оно гласит “что сделали с тобой, то сделай и со связью. CascadeType.ALL обозначает выполнение каскадных операций при save-update и delete, по умолчанию Hibernate будет игнорировать ассоциации, поэтому “разруливать” зависимости придется самостоятельно. Запись fetch=FetchType.LAZY указывает, что загрузка данного поля будет только в случае обращения к данному полю. Пока программа этого не делает, поле будет пустым.
* @OneToMany – такая запись говорит о том, что поле служить для связи один-ко-многим (потому и список).
* @ManyToMany – эта аннотация говорит о том, что поле будет использовано для связи много-ко-многим.
* @JoinTable – указывает имя таблицы, которая используется для организации связи много-ко-многим. Параметр joinColumns указывает название столбца, который является ссылкой на текущий класс, параметр inverseJoinColumns указывает на колонку, в которой находится ссылка на класс с другой стороны.

В качестве альтернативы аннотациям возможно использование XML-файла с настройками (файл mapping.cfg.xml.). Во-третьих, нужно описать класс для работы с Hibernate API.

* Работая с данными в объектно-ориентированном языке, мы работаем с объектами, заполняя и считывая значения полей, создавая новые или изменяя существующие объекты, определяя зависимости между объектами.
* При операции **save()** мы передаем объект типа **Shop**, который сохраняется в базу данных по описанным правилам отображения. В том числе сохраняются и все зависимые объекты (**Item**).
* Составляя запросы к базе данных, мы уже указываем не столбцы таблицы, а свойства объектов.

Обратите внимание на работу с транзакциями: начало транзакции( метод beginTransaction()), подтверждение транзакции (метод commit()), откат транзакции (метод rollback()).

Пример 7, 8





**Servlets**

**Servlet (сервлет)** – это класс на Java, расширяющий возможности сервера, к которому происходят обращения в рамках модели «запрос-отклик». Теоретически может поддерживать любой тип запросов, но чаще всего используются HTTP-сервлеты.

**Servlet container (контейнер сервлета)** - контейнер, обеспечивающий сетевые службы, при помощи которых посылаются запросы и ответы, декодируются запросы и форматируются ответы. Все контейнеры сервлетов должны поддерживать HTTP протокол, но могут также поддерживать дополнительные протоколы, например, HTTPS.

**Distributed servlet container (распределенный контейнер сервлета)** - контейнер сервлета, запускающий Web-приложения, которые помечены как распределенные и выполняются на нескольких виртуальных машинах Java. При этом виртуальные машины могут быть запущены, как на одном, так и на разных компьютерах.

**Servlet context (контекст сервлета)** - объект, содержащий представление (вид) Web-приложения, в котором запущен сервлет. Используя контекст, сервлет может вести журнал событий, получать URL-ссылки на ресурсы, а также устанавливать и хранить атрибуты, которые могут использоваться другими сервлетами в приложении.

**Servlet mapping (отображение сервлета)** - определяет связь между структурой URL и сервлетом. Используется для отображения запросов в сервлеты. Если контейнер, обрабатывающий запрос, является JSP-контейнером, то неявно отображается URL, содержащий расширение .jsp.

**Жизненный цикл сервлета**

* Если экземпляр сервлета не существует, то контейнер:
  1. загружает класс сервлета;
  2. создает его экземпляр;
  3. вызывает метод **init().**
* При получении запроса вызывается метод **service(),** которому передаются объекты запроса и отклика.
* Если контейнеру требуется удалить объект сервлета, то вызывается метод **destroy().**

**Особенности сервлетов**

* Существуют механизмы реагирования на события, связанные с деятельностью сервлетов в приложении:
  + **javax.servlet.*Событие*Listener**
  + **javax.servlet.*Событие*Event**
* Для обработки события требуется:
  + написать класс, реализующий интерфейс.
  + снабдить этот класс аннотацией **@WebListener** (пакет **javax.servlet.annotation**).
* Требуется учитывать то, что сервлет является разделяемым ресурсом и работает с разделяемыми ресурсами.
* Не следует использовать поля класса для хранения состояния, а особенно для передачи состояния между вызовами.

**Объект запроса**

* Реализует интерфейс **javax.servlet.ServletRequest.**
* Позволяет узнать характеристики запроса, в том числе значения параметров:
  + **String getParameter(String name)**
  + **Enumeration getParameterNames()**
  + и ряд других методов
* Позволяет получить доступ к потоку запроса:
  + **ServletInputStream getInputStream()**
  + **BufferedReader getReader()**
  + Можно вызвать только один из этих двух методов
* Позволяет получить параметры запроса
  + **int getContentLength()**
  + **String getContentType()**
  + **String getProtocol()**
  + и т.д.

**Объект отклика**

* Реализует интерфейс **javax.servlet.ServletResponse**
* Основные методы:
  + **ServletOutputStream getOutputStream()**Далее для вывода используются обычные средства бинарного вывода
  + **PrintWriter getWriter()**Далее для вывода используются обычные средства символьного вывода
  + Можно вызвать только один из этих двух методов.
* Позволяет установить параметры отклика
  + Для установки типа отклика используется метод **setContentType()**
  + Для установки кодировки используется метод **setCharacterEncoding()**
  + Сбрасывает текущий буфер клиенту **flushBuffer()**
  + Очищает буфер и сбрасывает статус **reset()**

**HTTP Servlet**

* **javax.servlet.http.HttpServlet** - абстрактный класс, позволяющий создавать сервлеты, удобные для Web.
* Наследные классы должны переопределять хотя бы один из методов обработки запроса.
* Обычно это один из методов:
  + **void doGet(HttpServletRequest req, HttpServletResponse resp)**
  + **void doPost(HttpServletRequest req, HttpServletResponse resp)**
* Классы снабжаются аннотацией @WebServlet.

**Пример 9. Класс сервлета**

*import java.io.\*;*

*import javax.servlet.\*;*

*import javax.servlet.http.\*;*

*import javax.servlet.annotation.\*;*

*@WebServlet("/MyServlet.html")*

*public class MyServlet extends HttpServlet {*

*protected void doGet(HttpServletRequest request, HttpServletResponse response) throws IOException {*

*PrintWriter out = response.getWriter();*

*try {*

*out.print("<html><body>This is text document, which has "*

*+ "<a href=\"other.html\">link</a>"*

*+ " to another document</body></html>");*

*}*

*finally {*

*out.close(); }*

*}*

*}*

**Поддержка сессий**

* В HTTP-сервлетах в объектах запроса и отклика добавляются дополнительные методы.
* Среди них появляются методы работы с сессиями, включая работу с **Cookies** (англ. печенье) - небольшой фрагмент данных, отправленный [веб-сервером](https://ru.wikipedia.org/wiki/%D0%92%D0%B5%D0%B1-%D1%81%D0%B5%D1%80%D0%B2%D0%B5%D1%80) и хранимый на [компьютере](https://ru.wikipedia.org/wiki/%D0%9A%D0%BE%D0%BC%D0%BF%D1%8C%D1%8E%D1%82%D0%B5%D1%80) пользователя. Веб-клиент (обычно [веб-браузер](https://ru.wikipedia.org/wiki/%D0%91%D1%80%D0%B0%D1%83%D0%B7%D0%B5%D1%80)) всякий раз при попытке открыть страницу соответствующего сайта пересылает этот фрагмент данных веб-серверу в составе [HTTP-запроса](https://ru.wikipedia.org/w/index.php?title=HTTP-%D0%B7%D0%B0%D0%BF%D1%80%D0%BE%D1%81&action=edit&redlink=1). Применяется для сохранения данных на стороне пользователя, на практике обычно используется для [аутентификации](https://ru.wikipedia.org/wiki/%D0%90%D1%83%D1%82%D0%B5%D0%BD%D1%82%D0%B8%D1%84%D0%B8%D0%BA%D0%B0%D1%86%D0%B8%D1%8F) пользователя, хранения персональных предпочтений и настроек пользователя, отслеживания состояния [сеанса](https://ru.wikipedia.org/w/index.php?title=%D0%A1%D0%B5%D0%B0%D0%BD%D1%81_(%D0%B8%D0%BD%D1%84%D0%BE%D1%80%D0%BC%D0%B0%D1%82%D0%B8%D0%BA%D0%B0)&action=edit&redlink=1) доступа пользователя, ведения статистики о [пользователях](https://ru.wikipedia.org/wiki/%D0%9F%D0%BE%D0%BB%D1%8C%D0%B7%D0%BE%D0%B2%D0%B0%D1%82%D0%B5%D0%BB%D1%8C).
* Для передачи значений между запросами можно использовать атрибуты сессии.
  + **HttpSession.getAttribute(String name)**
  + **HttpSession.setAttribute(String name, Object value)**
  + И ряд других методов

**Пример 10.**

*public class CashierServlet extends HttpServlet {*

*public void doGet (HttpServletRequest request,*

*HttpServletResponse response)*

*throws ServletException, IOException {*

*// Get the user's session and shopping cart*

*HttpSession session = request.getSession();*

*ShoppingCart cart = (ShoppingCart)session.getAttribute("cart");*

*//...*

*}*

*}*

*out.println("<a href=\"" +*

*response.encodeURL(request.getContextPath() +*

*"/bookcatalog") + "\">");*

**Общий дескриптор развёртывания web.xml**

* Определяет параметры развёртывания web-приложения в контейнере.
* Некоторые из этих параметров даже не могут быть указаны с помощью аннотаций классов, т.к. относятся к web-приложению в целом.
* Указанные в дескрипторе параметры перекрывают параметры, указанные с помощью аннотаций классов.

**Пример 11. Дескриптор развертывания web.xml**

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

*<web-app version="3.0" xmlns="http://java.sun.com/xml/ns/javaee" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-app\_3\_0.xsd">*

*<listener>*

*<listener-class>org.tempuri.MyListner</listener-class>*

*</listener>*

*<servlet>*

*<servlet-name>MyServlet</servlet-name>*

*<servlet-class>org.tempuri.MyServlet</servlet-class>*

*</servlet>*

*<servlet-mapping>*

*<servlet-name>MyServlet</servlet-name>*

*<url-pattern>/MyServlet.html</url-pattern>*

*</servlet-mapping>*

*<welcome-file-list>*

*<welcome-file>/MyServlet.html</welcome-file>*

*</welcome-file-list>*

*</web-app>*

**Фильтрация**

* Получаемые запросы и формируемые отклики могут быть подвергнуты фильтрации.
* API для классов фильтрации определен интерфейсами **Filter, FilterConfig, FilterChain** пакета **javax.servlet.**
* Включение фильтров осуществляется с помощью аннотации **@WebFilter.**
* Фильтры используются в основном для перехвата и изменения запросов и ответов от сервера. Рассмотрим случай, в котором проверяется сессия для каждого запроса пользователя, и если она верна, то только тогда пользователь может получить доступ к странице. Этого можно добиться путем проверки сессий на всех сервлет страницах (страницы JSP), которые запрашивают пользователи, или можно сделать это с помощью фильтра.

**Пример 12. Фильтр**

*public class LogFilter implements Filter {*

*public void doFilter(ServletRequest req, ServletResponse res, FilterChain chain) throws IOException, ServletException {*

*HttpServletRequest request = (HttpServletRequest) req;*

*String ipAddress = request.getRemoteAddr();*

*System.out.println("IP " + ipAddress + ", Time " + new Date());*

*chain.doFilter(req, res);*

*}*

*public void init(FilterConfig config) throws ServletException {*

*//Get init parameter*

*String testParam = config.getInitParameter("test-param");*

*//Print the init parameter*

*System.out.println("Test Param: " + testParam);*

*}*

*public void destroy() {*

*//add code to release any resource*

*}*

*}*

* Метод **init()** используется для инициализации любого кода, который используется фильтром. Также отметим, что **init()** получит объект FilterConfig, который содержит различную информация уровня фильтров, а также параметры инициализации, который передаются из web.xml (дескриптора развертывания).
* **doFilter()** фактически выведет информацию. Можно изменить этот метод и добавить код, который может изменить запрос / состояние сессии / ответ, добавить атрибут в запрос и т.д.
* **destroy()** вызывается контейнером, когда он хочет убрать мусор за фильтром. Это обычно делается, когда фильтр не используется в течение длительного времени и сервер хочет выделить память для других приложений.

**Пример 13. Маппинг фильтра в web.xml**

*<filter>*

*<filter-name>LogFilter</filter-name>*

*<filter-class>sample.LogFilter</filter-class>*

*<init-param>*

*<param-name>test-param</param-name>*

*<param-value>This parameter is for testing.</param-value>*

*</init-param>*

*</filter>*

*<filter-mapping>*

*<filter-name>LogFilter</filter-name>*

*<url-pattern>/\*</url-pattern>*

*</filter-mapping>*

**JSP (Java Server Pages)**

**JSP** – это технология, которая упрощает создание web страниц с динамически изменяющимся (во время генерации страницы) содержимым.

*<html>*

*<body>*

*<p>HI<%= (request.getParameter("name")!=null) ? ", " + request.getParameter("name"): "" %>!!!</p>*

*</body>*

*</html>*

**JSP файл –** это текстовый документ, содержимое которого можно разбить на две части:

* + статический текст (каркас) – создается при помощи специальных редакторов.
  + динамический (генерируемый во время выполнения страницы) – обычно пишется в среде разработки программирования.

Принцип работы (рис. 6):

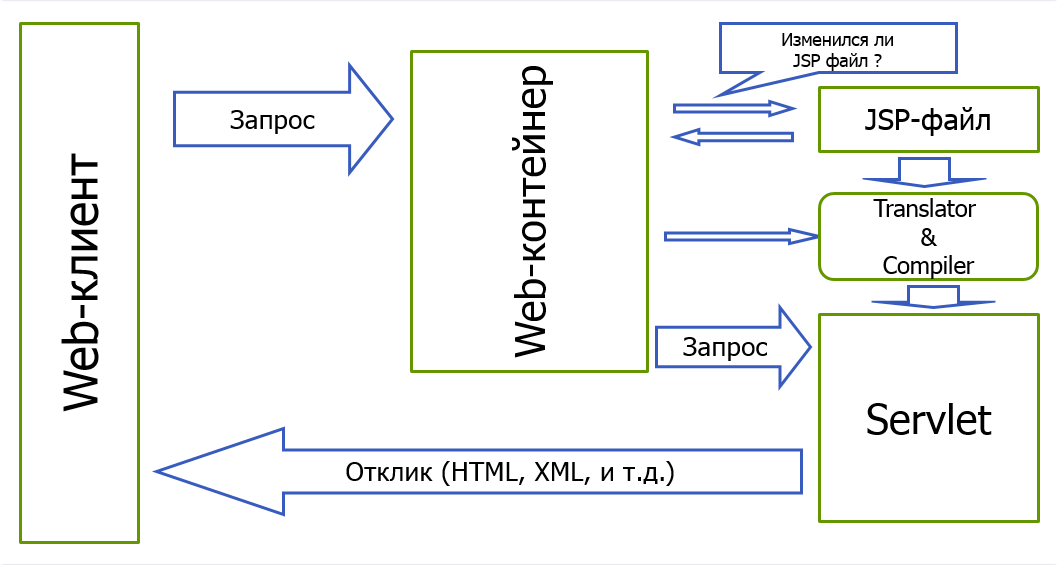


Рисунок 6 – Принцип работы

* + JSP-файл транслируется в файл java.
  + Процесс трансляции управляется директивами (<%@ page […] %>).
  + Полученный файл компилируется в класс сервлета.
  + Далее с ним идет работа как с сервлетом.
  + Конфигурация сервера может включать ряд настроек по работе с jsp-страницами.

**Обработка ошибок**

* Во время трансляции и компиляции
  + поведение зависит от используемого сервера;
  + класс сервлета создан не будет.
* Во время выполнения
  + если для данной JSP страницы (или приложения) определена «страница на случай ошибки» (errorPage), то будет выведена указанная страница;
  + иначе – в зависимости от сервера и его настроек.

**Виды JSP:**

* JSP-страница
  + Это страница из статических и динамических элементов.
  + Статическая часть имеет произвольный формат.
* JSP-документ
  + XML-файл, содержащий тэги особого вида.
  + Имеет ту же функциональность, что и JSP-страница, но требует представления в виде тэгов ряда элементов.

**Создание страницы**

* Создание статического содержимого. Для этого удобно использовать редакторы HTML, XML и пр.
* Вставка динамических объектов и элементов сценариев JSP. В созданный файл вставляются элементы **<%? […] %>**

**Элементы JSP-страницы**

* Статический текст
* Элементы сценариев
  + Комментарии **<%-- […] --%>**
  + Объявления **<%! […] %>**
  + Скриптлеты **<% […] %>**
  + Выражения **<%= […] %>**
  + Директивы **<%@ ? […]%>**
* Стандартные объекты.

**Комментарии**

* <%-- […] --%>
* Содержат текст, который не будет включен в отклик страницы.
* Содержимое комментария игнорируется транслятором.
* Не путайте с комментариями HTML  
  <!-- […] -->

**Пример 14. JSP-страница с комментариями**

JSP-страница

*<html><body>*

*Hello World!*

*<%-- This is a JSP comment --%>*

*<!-- This is a HTML comment -->*

*</body></html>*

Фрагмент примерного кода сервлета

*out.write("<html><body>\n Hello World!\n ");*

*out.write("\n <!-- This is a HTML comment -->"*

*+ "\n</body></html>\n");*

**Объявления**

* <%! […] %>
* Обычно содержат объявления переменных, методов, классов и интерфейсов.
* При трансляции преобразуются в элементы класса сервлета (поля, методы и вложенные типы).
* Нельзя использовать поля для хранения значений, использующихся в рамках одного запроса.
* При работе с внутренними классами есть особенности.

**Пример 15. JSP-страница с объявлениями**

JSP-страница

*<%!*

*private String str = "Message: ";*

*public String createMessage(String arg) {*

*return str + arg;*

*}*

*%>*

Фрагмент примерного кода сервлета

*public final class index\_jsp extends org.apache.jasper.runtime.HttpJspBase*

*implements org.apache.jasper.runtime.JspSourceDependent {*

*private String str = "Message: ";*

*public String createMessage(String arg) {*

*return str + arg;*

*}*

**Скриптлеты**

* <% […] %>
* Содержат динамически выполняемый код.
* Преобразуются во фрагменты методов сервлета, формирующих отклик.
* Синтаксически являются фрагментами обычных Java-методов.
* Каждая завершенная инструкция заканчивается знаком точки с запятой.

**Пример 16. JSP-страница со скриптлетами**

JSP-страница

*<html><body>*

*<% String server\_name = System.getProperty("os.name");*

*if (server\_name == null)*

*server\_name = "Undefined."; %>*

*</body></html>*

Фрагмент примерного кода сервлета

*out.write("<html><body>\n");*

*String server\_name = System.getProperty("os.name");*

*if (server\_name == null)*

*server\_name = "Undefined.";*

*out.write("\n");*

*out.write("</body></html>\n");*

**Выражения**

* <%= […] %>
* Содержит выражения, значения которых будут выведены в отклик.
* Синтаксически являются вычислимыми выражениями Java, для которых может быть получено текстовое представление.
* В конце выражений точки с запятой нет!

**Пример 17. JSP-страница с выражениями**

JSP-страница

*<html><body>*

*OS Name: <%=server\_name%>*

*<br><%=createMessage("It works!")%>*

*</body></html>*

Фрагмент примерного кода сервлета

*out.write("<html><body>\n");*

*out.write("OS Name: ");*

*out.print(server\_name);*

*out.write("\n");*

*out.write("<br>");*

*out.print(createMessage("It works!"));*

*out.write("\n");*

*out.write("</body></html>\n");*

**Директива page**

* <%@ page […] %>
* Позволяет управлять процессом трансляции JSP-страницы в сервлет.
* Часто используемые параметры:
  + Тип отклика и кодировка.
  + Автоматическая инициализация сессии.
  + Импортируемые типы.
  + Страница для обработки ошибок:
* Является ли данная страница обработчиком.
* Какая страница используется как обработчик.

**Пример 18. Тип отклика, кодировка и импорт типов**

JSP-страница

*<%@page contentType="text/html; charset=UTF-8"%>*

*<%@page session="true"%>*

*<%@page import="java.util.\*, java.lang.reflect.\*" %>*

Фрагмент примерного кода сервлета

*import java.util.\*;*

*import java.lang.reflect.\*;*

*...*

*response.setContentType("text/html; charset=UTF-8");*

*session = pageContext.getSession();*

*...*

**Возникновение и обработка ошибок**

**Пример 19. Возникновение и обработка ошибок. JSP-страницы**

Основная страница

*<%@page contentType="text/html; charset=UTF-8"%>*

*<%@page errorPage="errorMessage.jsp"%>*

*<html><body>*

*<% if(true) throw new Exception("Something happened!");%>*

*</body></html>*

Страница-обработчик

*<%@page contentType="text/html" pageEncoding="UTF-8"%>*

*<%@page isErrorPage="true" %>*

*<html><body>*

*An error has occurred! Please, contact administrator.*

*<%someLogMethod(exception);%>*

*</body></html>*

**Пример 20. Возникновение и обработка ошибок. Примерный код сервлетов**

Основная страница

*try {*

*pageContext = \_jspxFactory.getPageContext(this, request,*

*response, "errorMessage.jsp", true, 8192, true);*

*if(true) throw new Exception("Something happened!");*

*} catch (Throwable t) {*

*\_jspx\_page\_context.handlePageException(t);*

*}*

Страница-обработчик

*Throwable exception =*

*org.apache.jasper.runtime.JspRuntimeLibrary.getThrowable(request);*

*...*

*out.write("<html><body>\n");*

*out.write("An error has occurred! Please, contact administrator.\n");*

*someLogMethod(exception);*

*out.write("</body></html>\n");*

**Директива include**

* Предназначена для включения в формируемый отклик содержимого другого файла.
* Включение статическое: происходит на этапе трансляции.
* Включаться могут как статические, так и динамические ресурсы.
* Существует специальный вид JSP-страниц – JSP Fragments (JSPF).

**Пример 21. Директива include. JSP-страницы**

Основная страница

*<%@page contentType="text/html; charset=UTF-8"%>*

*<html><body>*

*<%@include file="WEB-INF/jspf/fragment.jspf" %>*

*</body></html>*

Включаемая страница

*<%@ page pageEncoding="UTF-8" %>*

*Some fragment text*

**Пример 22. Директива include.Примерный код сервлетов**

Основная страница

*public final class index\_jsp ... {*

*private static java.util.Vector \_jspx\_dependants;*

*static {*

*\_jspx\_dependants = new java.util.Vector(1);*

*\_jspx\_dependants.add("/WEB-INF/jspf/fragment.jspf");*

*}*

*public void \_jspService(HttpServletRequest request,  
 HttpServletResponse response)  
 throws java.io.IOException, ServletException {*

*...*

*out.write("<html><body>\n");*

*out.write("Some fragment text\n");*

*out.write("</body></html>\n");*

*...*

*}*

**Стандартные объекты**

* **page**Ссылка на текущий объект (по сути – объект сервлета).
* **config**Имеет тип javax.servlet.ServletConfig, позволяет получить контекст сервлета и параметры сервлета.
* **application**Имеет тип javax.servlet.ServletContext, определяет контекст сервлета.
* **pageContext**Имеет тип javax.servlet.jsp.PageContext, определяет контекст jsp-страницы.
* **session**Имеет тип javax.servlet.http.HttpSession, представляет текущую сессию.
* **request**Имеет тип javax.servlet.http.HttpServletRequest, представляет текущий запрос.
* **response**Имеет тип javax.servlet.http.HttpServletResponse, представляет текущий отклик.
* **out**Имеет тип javax.servlet.jsp.JspWriter, представляет поток для вывода, по функциональности схож с классом PrintWriter.

**Пример 23. Стандартные объекты. Примерный код сервлета**

*PageContext pageContext = null;*

*HttpSession session = null;*

*ServletContext application = null;*

*ServletConfig config = null;*

*JspWriter out = null;*

*Object page = this;*

*...*

*pageContext = \_jspxFactory.getPageContext(this, request,  
 response, null, true, 8192, true);*

*application = pageContext.getServletContext();*

*config = pageContext.getServletConfig();*

*session = pageContext.getSession();*

*out = pageContext.getOut();*

**Тэги <jsp:…>**

* Являются еще одним средством управления ходом трансляции страницы.
* В основном предназначены для формата   
  JSP-документов.
* Позволяют:
  + Описывать теги и их составляющие.
  + Перенаправлять обработку запросов.
  + Работать с компонентами JavaBeans.
  + Включать в код страницы обращения к апплетам.

**Тэги <jsp:include>**

* <jsp:include page="inclPage">
* Включает в страницу статический или динамический ресурс.
* Включение происходит динамически, во время выполнения.

**Пример 24.**

*<jsp:include page="{relativeURL | <%= expression %>}"*

*flush="true | false" >*

*<jsp:param name="parameterName"*

*value="{parameterValue|<%=expr%>}|${…}” />*

*</jsp:include>*

JSP-страница

*<html><body>*

*Before*

*<jsp:include page="simple.html"></jsp:include>*

*After*

*</body></html>*

Примерный код сервлета

*out.write("<html><body>\n");*

*out.write("Before\n"); org.apache.jasper.runtime.JspRuntimeLibrary.include(request,*

*response, "simple.html", out, false);*

*out.write("\n");*

*out.write("After\n");*

*out.write("</body></html>\n");*

**Тэги <jsp:forward>**

* <jsp:forward page="inclPage">
* Передает обработку запроса другому ресурсу.
* Могут быть указаны дополнительные параметры.

**Пример 25.**

*<jsp:forward page="{relativeURL | <%= expression %>}">*

*<jsp:param name="parameterName"*

*value="{parameterValue|<%=expr%>}|%{…}” />*

*</jsp:forward>*

JSP-страница

*<html><body>*

*Before*

*<jsp:forward page="simple.html"></jsp:forward>>*

*After*

*</body></html>*

Примерный код сервлета

*out.write("<html><body>\n");*

*out.write("Before\n");*

*if (true) {*

*\_jspx\_page\_context.forward("simple.html");*

*return;*

*}*

*out.write("\n");*

*out.write("After\n");*

*out.write("</body></html>\n");*

**Взаимодействие с HTML-формами**

* Элементы HTML-управления должны находиться в форме.
* Для формы указывается адрес для перехода.
* Значения элементов ввода записываются в параметры запроса.
* В новой странице значения элементов ввода автоматически не заполняются.
* Элемент управления, активация которого привела к смене страницы, вносится в параметры запроса.

**Пример 26. Взаимодействие с HTML-формами.**

*<%@page contentType="text/html" pageEncoding="UTF-8"%>*

*<%@page import="java.util.\*" %>*

*<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"*

*"http://www.w3.org/TR/html4/loose.dtd">*

*<html>*

*<form name="SimpleForm" action="index.jsp">*

*<input type="text" name="NameEditor" value="" size="30" />*

*<input type="checkbox" name="SomeCheckBox" value="ON" />*

*<br>*

*<input type="submit" value="Action 1" name="Action1Button" />*

*<input type="submit" value="Action 2" name="Action2Button" />*

*</form>*

*<% Enumeration names = request.getParameterNames();*

*if (names.hasMoreElements()) {*

*String name = null;*

*String value = null;%>*

*<table border="1">*

*<thead>*

*<tr>*

*<th>Parameter name</th>*

*<th>Parameter value</th>*

*</tr>*

*</thead>*

*<tbody>*

*<% while (names.hasMoreElements()) {*

*name = (String) names.nextElement();*

*value = request.getParameter(name);%>*

*<tr>*

*<td><%=name%></td>*

*<td><%=value%></td>*

*</tr>*

*<% }%>*

*</tbody>*

*</table>*

*<% } %>*

*</html>*

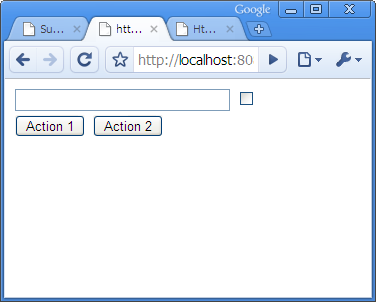
**

Рисунок 7 – Форма при первом запуске.

Адрес: http://host:8080/WebApplication1/index.jsp

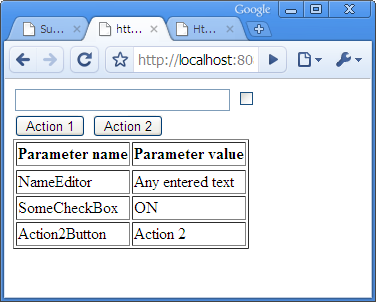


Рисунок 8 – Форма после нажатия кнопки.

Адрес: http://localhost:8080/WebApplication1/index.jsp?NameEditor=  
Any+entered+text&SomeCheckBox=ON&Action2Button=Action+2

**Недостатки раннего JSP:**

* Смешение Java-кода и тегов страницы.
* Много «типовых» действий.
* «Некомфортность» работы на Java в ходе обработки запросов из-за:
  + Громоздкости кода.
  + Необходимости знания особенностей преобразования в сервлет. **Основные идеи смены парадигмы:**
* Альтернативный язык Expression Language.
* Использование библиотек тегов, особенно JSTL – JSP Standard Tag Library.
* Ориентация на компоненты в смысле JavaBeans и использование понятия свойств.
* Общие принципы остаются прежними.

**Пример 27. Как писали раньше**

*<%@page contentType="text/html" pageEncoding="UTF-8"%>*

*<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"*

*"http://www.w3.org/TR/html4/loose.dtd">*

*<html> <head>*

*<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">*

*<title>JSP Page</title></head> <body>*

*<h1>Hello page</h1><br/>*

***<% if (request.getParameter("name") != null) {***

***out.print("<b>Hello, ");***

***out.print(request.getParameter("name"));***

***out.println("!</b><br/>");***

***} %>***

*<form name="oldform" action="old.jsp">*

*What's your name?<br>*

*<input type="text" name="name" value="" size="50" />*

*<input type="submit" value="Tell name" name="tell" />*

*</form> </body> </html>*

**Пример 28. Как пишут теперь**

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

*<%@page contentType="text/html" pageEncoding="UTF-8"%>*

*<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"*

*"http://www.w3.org/TR/html4/loose.dtd">*

*<html> <head>*

*<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">*

*<title>JSP Page</title></head> <body>*

*<h1>Hello page</h1><br/>*

***<c:if test="${!empty param.name}" var="val" scope="request">***

***<b>Hello, ${param.name}!</b><br/>***

***</c:if>***

*<form name="oldform" action="old.jsp">*

*What's your name?<br>*

*<input type="text" name="name" value="" size="50" />*

*<input type="submit" value="Tell name" name="tell" />*

*</form>*

*</body></html>*

**Expression Language**

* Является альтернативой выражениям и скриптлетам.
* Динамическое чтение данных из:
  + JavaBeans-компонентов.
  + Неявных объектов.
  + Различных структур данных.
* Динамическая запись данных в:
  + Формы.
  + JavaBeans-компоненты.
* Вызов статических и публичных методов.
* Динамическое выполнение простых операций.

**Виды выражений EL**

* По моменту вычисления:
  + Немедленное вычисление.
  + Отложенное вычисление.
* По цели выражения:
  + Обращение к данным.
  + Вызов метода.
* По режиму доступа:
  + Только чтение.
  + Чтение и запись данных.

**Немедленное и отложенное вычисление**

* Немедленное вычисление (JSP и JSF):
  + ${ *выражение* }
  + Позволяет лишь получить значение выражения.
* Отложенное вычисление (JSF):
  + #{ *выражение* }
  + Выражение может использоваться различным образом в разные моменты жизненного цикла.
  + Позволяет не только получать значения, но и записывать их (в т.ч. для свойств бинов).

**Применение выражений с немедленным вычислением**

* Выражения EL могут применяться:
  + в статическом тексте
  + в качестве значений атрибутов тегов

**Пример 29.**

*<some:tag>*

*some text ${expr} some text*

*</some:tag>*

*<some:tag value="${expr}"/>*

*<some:tag value="some${expr}${expr}text${expr}"/>*

**Обращение к данным**

* Компоненты JavaBeans
  + ${beanName.propertyName}
  + ${beanName["propertyName"]}
* Коллекции
  + ${beanName.listName[i]}
  + ${beanName.mapName["string key"]}
* Перечислимые типы JavaSE
  + ${reference == "enumConstant"}
* Неявные объекты
  + ${param["paramName"]}
* Вычисление выражений
  + ${customer.age + 20}

**Вызов методов**

* Без параметров
  + ${object.method}
  + ${object["method"]}
* С параметрами
  + ${object.method(param1, ..., paramN}
  + ${object["method"](param1, ..., paramN)}

**Литералы в выражениях**

* Логические true, false
* Целочисленные 136
* С плавающей точкой 3.14, 1.0e10
* Строковые “string”
* Ссылочные null

**Операторы в выражениях**

* Доступ к элементам []
* Операторные скобки ()
* Унарные - not ! Empty (Empty А возвращает true, если А – ноль)
* Арифметические \* / div % mod + -
* Сравнение < > <= >= lt gt le ge == != eq ne
* Логические && and || or ? :

**Неявные объекты**

* Объекты доступа к другим объектам
  + pageScope
  + requestScope
  + sessionScope
  + applicationScope
* Часто используемые объекты
  + pageContext
  + session
  + request
  + response
  + param
  + cookie
  + ...

**Примеры выражений EL**

Таблица 6 – Примеры выражений EL

|  |  |
| --- | --- |
| **Выражение** | **Результат** |
| **${(10\*10) ne 100}** | false |
| **${'a' < 'b'}** | true |
| **${1.2E4 + 1.4}** | 12001.4 |
| **${!empty param.Add}** | true, если параметр Add не null и не пустая строка |
| **${pageContext.request.contextPath}** | Путь контекста |
| **${param['mycom.productId']}** | Значение параметра mycom.productId |
| **${departments[deptName]}** | Значение элемента deptName в departments |
| **${header["host"]}** | Имя хоста |

**JSP-страница с применением EL**

**Пример 30. JSP-страница с применением EL**

JSP-страница

*${20 + 335}<br/>*

*${!empty param.Add}<br/>*

Фрагмент примерного кода сервлета

*out.write((java.lang.String)*

*org.apache.jasper.runtime.PageContextImpl.evaluateExpression(*

*"${20 + 335}", java.lang.String.class,*

*(PageContext)\_jspx\_page\_context, null));*

*out.write("<br/>\n");*

*out.write((java.lang.String)*

*org.apache.jasper.runtime.PageContextImpl.evaluateExpression(*

*"${!empty param.Add}", java.lang.String.class,*

*(PageContext)\_jspx\_page\_context, null));*

*out.write("<br/>\n");*

**Custom tags**

* Определяются пользователем и описывают часто используемые операции.
* Описываются и распространяются в виде библиотек тегов, определяющих семейства тегов и содержащих объекты, реализующие теги.
* Для использования тегов необходимо:
  + Объявить библиотеку тегов
  + Сделать ее доступной для web-приложения

*<prefix:tag attr1="value" ... attrN="value" />*

*<prefix:tag attr1="value" ... attrN="value" >*

*body*

*</prefix:tag>*

**Библиотеки тегов**

* Библиотеки тегов описываются в tld-файлах
* Библиотеки могут быть
  + публичными и храниться в сети
  + локальными и храниться в каталоге WEB-INF
* Для подключения библиотеки необходимо использовать директиву **taglib**

*<%@ taglib prefix="tlt" uri="/WEB-INF/iterator.tld"%>*

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

**Обработка custom tags**

* В процессе компиляции jsp-страницы теги будут преобразовываться в java-код в соответствии с описанием в библиотеке тегов.
* Если соответствующая библиотека не была подключена, относящиеся к ней теги будут трактоваться как статическая часть jsp-страницы и просто выводиться в отклик.

**Пример 31. JSP-страница с применением custom tags**

JSP-страница

*<c:if test="${!empty param.name}" var="nameExists" scope="request">*

*<b>Hello, ${param.name}!</b><br/>*

*</c:if>*

Фрагмент примерного кода сервлета

*org.apache.taglibs.standard.tag.rt.core.IfTag \_jspx\_th\_c\_if\_0 =*

*(org.apache.taglibs.standard.tag.rt.core.IfTag)*

*\_jspx\_tagPool\_c\_if\_var\_test\_scope.get(*

*org.apache.taglibs.standard.tag.rt.core.IfTag.class);*

*\_jspx\_th\_c\_if\_0.setPageContext(\_jspx\_page\_context);*

*\_jspx\_th\_c\_if\_0.setParent(null);*

*\_jspx\_th\_c\_if\_0.setTest(((java.lang.Boolean)*

*org.apache.jasper.runtime.PageContextImpl.evaluateExpression(*

*"${!empty param.name}", java.lang.Boolean.class,*

*(PageContext)\_jspx\_page\_context, null)).booleanValue());*

*\_jspx\_th\_c\_if\_0.setVar("nameExists");*

*\_jspx\_th\_c\_if\_0.setScope("request");*

*int \_jspx\_eval\_c\_if\_0 = \_jspx\_th\_c\_if\_0.doStartTag();*

*if (\_jspx\_eval\_c\_if\_0 != javax.servlet.jsp.tagext.Tag.SKIP\_BODY) {*

*do {*

*out.write("<b>Hello, ");*

*out.write((java.lang.String)*

*org.apache.jasper.runtime.PageContextImpl.evaluateExpression(*

*"${param.name}", java.lang.String.class,*

*(PageContext)\_jspx\_page\_context, null));*

*out.write("!</b><br/>\n");*

*int evalDoAfterBody = \_jspx\_th\_c\_if\_0.doAfterBody();*

*if (evalDoAfterBody !=*

*javax.servlet.jsp.tagext.BodyTag.EVAL\_BODY\_AGAIN)*

*break;*

*} while (true);*

*}*

**JSP Standard Tag Library**

* Библиотека включает в себя набор тегов, функциональность которых часто используется в web-приложениях.
* Эти теги должны обрабатываться единообразно в рамках всех контейнеров.
* Это позволяет:
  + избежать использования различных библиотек.
  + единообразно организовывать JSP-страницы .

**Библиотеки JSTL**

* JSTL поставляется в виде набора библиотек:
  + Различные направления функциональности
  + Различные пространства имен
* Библиотеки и соответствующие пути
  + Core *http://java.sun.com/jsp/jstl/core*
  + XML *http://java.sun.com/jsp/jstl/xml*
  + Internationalization *http://java.sun.com/jsp/jstl/fmt*
  + Database *http://java.sun.com/jsp/jstl/sql*
  + Functions *http://java.sun.com/jsp/jstl/functions*

Таблица 7 – Библиотеки JSTL

|  |  |  |
| --- | --- | --- |
| **Библиотека** | **Функциональность** | **Префикс** |
| Core | Работа с переменными | **c** |
| Управление ходом выполнения |
| Работа с URL |
| Разное |
| XML | Базовые | **x** |
| Управление выполнением |
| Преобразования |
| I18n | Локализация | **fmt** |
| Форматирование сообщений |
| Форматирование чисел и дат |
| Database | SQL | **sql** |
| Functions | Возможности коллекций | **fn** |
| Работа со строками |

Таблица 8 – Библиотека Core

|  |  |
| --- | --- |
| **Функциональность** | **Тэги** |
| Работа с переменными | **remove set** |
| Управление ходом выполнения | **choose (when, otherwise) forEach**  **forTokens**  **if** |
| Работа с URL | **import (param) redirect (param)**  **url (param)** |
| Разное | **catch**  **out** |

Таблица 9 – Библиотека XML

|  |  |
| --- | --- |
| **Функциональность** | **Тэги** |
| Базовая | **out**  **parse**  **set** |
| Управление ходом выполнения | **choose (when, otherwise) forEach**  **if** |
| Преобразования | **transform (param)** |

**Пример 32. Пример использования**

*<c:if test="${!empty param.Add}">*

*<c:set var="bid" value="${param.Add}"/>*

*<jsp:useBean id="bid" type="java.lang.String" />*

*<sql:query var="books"*

*dataSource="${applicationScope.bookDS}">*

*select \* from PUBLIC.books where id = ?*

*<sql:param value="${bid}" />*

*</sql:query>*

*<c:forEach var="bookRow" begin="0" items="${books.rows}">*

*<jsp:useBean id="bookRow" type="java.util.Map" />*

*<jsp:useBean id="addedBook" class="database.Book"   
 scope="page" />*

*...*

*<% cart.add(bid, addedBook); %>*

*...*

*</c:if>*

**Язык JavaScript**

JavaScript – это язык программирования

* + Интерпретируемый
  + Объектно-ориентированный
  + Со слабой типизацией
  + С динамической типизацией
  + С автоматическим управлением памятью
  + Вместо наследования классов используются прототипы объектов
  + Функции являются объектами

**Области применения**

* Скриптовые фрагменты серверных приложений.
* Виджеты (примитив [графического интерфейса пользователя](https://ru.wikipedia.org/wiki/%D0%93%D1%80%D0%B0%D1%84%D0%B8%D1%87%D0%B5%D1%81%D0%BA%D0%B8%D0%B9_%D0%B8%D0%BD%D1%82%D0%B5%D1%80%D1%84%D0%B5%D0%B9%D1%81_%D0%BF%D0%BE%D0%BB%D1%8C%D0%B7%D0%BE%D0%B2%D0%B0%D1%82%D0%B5%D0%BB%D1%8F), имеющий стандартный внешний вид и выполняющий стандартные действия).
* Прикладное программное обеспечение.
* Букмарклеты (небольшие приложения, размещаемые в закладках браузера).
* Пользовательские скрипты в браузерах.
* Программы на стороне клиента в web-приложениях.
  + Код встраивается в HTML-страницу.
  + Код выполняется браузером.
  + Возможно изменение структуры страницы, её элементов и их параметров.
  + Изменения могут происходить без перезагрузки страницы.
  + Могут выполняться дополнительные действия.
    - Взаимодействие с браузером
    - Взаимодействие с сервером
    - Бизнес-логика в целом

**Пример 33. Здравствуй, мир!!!**

*<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">*

*<html>*

*<head>*

*<title>Проверка JS</title>*

*</head>*

*<body>*

*<h1>*

***<script>***

***document.write("Hello, world!!!");***

***</script>***

*</h1>*

*</body>*

*</html>*

**Размещение в HTML-документе**

* Тег <script>

*<html>*

*<body>*

*<p id="demo">This is a paragraph.</p>*

*<script type="text/javascript">*

*document.getElementById("demo").innerHTML=Date();*

*</script>*

*</body>*

*</html>*

* Атрибуты тегов (обработка событий)

*<a href="delete.jsp" onclick="return confirm('Вы верены?');">Удалить</a>*

* Гипертекстовые ссылки

*<a href="javascript:document.mainform.submit();">Отправить</a>*

**Тег <script>**

* Указание языка
  + По умолчанию – именно JavaScript
  + В явном виде – с помощью атрибута
    - Устаревший вариант  
      **language="JavaScript"**
    - Современный вариант  
      **type="text/javascript"**
* Две формы кода
  + Код размещается внутри тега
  + Код размещается в отдельном файле (обычно с расширением **js**), файл указывается с помощью атрибута **src**
  + Второй способ «перекрывает» первый
* Типичное использование
  + В заголовке документа – для описания функций и предварительных действий
  + В теле документа – для условной генерации документа

**Обработка браузером**

* Обычно код интерпретируется
  + Возможны Just-In-Time решения
* Код выполняется по мере обнаружения
  + Обращаться к элементам документа, описанным в документе позднее – не лучшая мысль
  + Однако обращаться к позднее описанным сущностям из функций можно, если сущности уже будут существовать на момент выполнения функции
* Возможно возникновение ошибок в ходе выполнения
  + Блок скрипта не завершит работу
  + В консоли браузера появится сообщение об ошибке

**Замечания**

* JavaScript чувствителен к регистру! (HTML – нечувствителен к регистру, хотя есть рекомендации).
* Соседство атрибутов HTML и событий/методов JavaScript может приводить к неожиданным результатам (например, onClick и onclick).
* Функции выполняются при явном вызове.
* Просто код в теге <script> выполняется по мере обнаружения.
* Код, указанный в атрибутах тегов как обработка событий, выполняется при возникновении событий.
* Код, указанный в ссылках, выполняется при срабатывании ссылки.

**Комментарии**

**Пример 34. Комментарии**

*<html>*

*<body>*

*<script>*

*document.write("Это будет выведено.");*

*// document.write("А это не будет выведено...");*

*/\**

*document.write("И это не будет выведено!");*

*document.write("И это тоже не будет, надо же...");*

*\*/*

*<!-- document.write("И это не будет?!!");*

*</script>*

*</body>*

*</html>*

* Однострочные
  + От символов **//** до конца строки
  + От символов <!-- до конца строки

*<script type="text/javascript">*

*<!--*

*document.write("Вы не увидите никаких следов кода,");*

*document.write("если ваш браузер не поддерживает JavaScript!!!");*

*//-->*

*</script>*

* Многострочные
  + Начало **/\***
  + Конец **\*/**
  + Не могут быть вложенными

**Литералы и переменные**

* Литералы
  + Числовые  
    *10 010 0x10 1.1 .1 1. 1e1*
  + Строковые  
    *"Строка" 'Строка' "I'm a string" '\'' "\""*
* Переменные
  + Могут объявляться с помощью ключевого слова **var**
    - *var a = 5, b = " лет";*
    - Если объявление вне функции – переменная глобальная, иначе – локальная
  + Могут объявляться и без ключевого слова **var**
    - *c = "И так тоже можно"*
    - Переменная всегда глобальная…
  + Объявление переменных «на лету» + их глобальность + чувствительность к регистру = источник ошибок
  + Тип переменной определяется её содержимым
  + Приведение типов автоматическое

Таблица 10 – Операторы

|  |  |  |
| --- | --- | --- |
| **Приоритет** | **Оператор** | **Запись** |
| 1 | Доступ к элементу | . [] |
| Создание объекта | new |
| 2 | Вызов функции | () |
| 3 | Инкремент (две формы) | ++ |
| Декремент (две формы) | -- |
| 4 | Логическое отрицание | ! |
| Побитовое отрицание | ~ |
| Унарный плюс | + |
| Унарный минус | - |
| Тип объекта | typeof |
| Вычисление выражения без возврата значения | void |
| Удаление свойства объекта | delete |
| 5 | Умножение | \* |
| Деление | / |
| Остаток от деления | % |
| 6 | Сложение, конкатенация | + |
| Вычитание | - |
| 7 | Побитовые сдвиги | << >> >>> |
| 8 | Сравнение | < <= > >= |
| Проверка наличия свойства в объекте | in |
| Проверка «типа» объекта | instanceof |
| 9 | Равенство и строгое (с типом) равенство | == != === !=== |
| 10 | Побитовое И | & |
| 11 | Побитовое исключающее ИЛИ | ^ |
| 12 | Побитовое ИЛИ | | |
| 13 | Логическое И | && |
| 14 | Логическое ИЛИ | || |
| 15 | Тернарный условный оператор | ?: |
| 16 | Присваивания | = += -= \*= /= %= <<= >>= >>>= &= ^= |= |
| 17 | Разделитель последовательности выражений | , |

**Управление порядком выполнения**

* Операторные скобки
  + {...}
* Ветвление
  + if (условие) инструкция;
  + if (условие) инструкция; else инструкция;
  + Нелогические значения 0, "", null, undefined и NaN считаются ложью, всё остальное – истиной
* Цикл с предусловием
  + while(условие) инструкция;
* Цикл с постусловием
  + do { ... } while(условие);
* Итерационный цикл
  + for(инициализация; условие; действие) инструкция;
* Цикл по элементам
  + for(переменная in объект) инструкция;
* Прерывание блока
  + **break;**
  + **break имяМетки;**
* Переход на следующую итерацию
  + **continue;**
  + **continue имяМетки;**
* Возврат из метода или обработчика
  + **return;**
  + **return значение;**
* Блок переключателей
  + **switch (выражение) {  
     case значение: инструкции;  
     ...  
     default: инструкции}**
  + Для сравнения используется строгое равенство
  + Можно работать со строками
  + Предложений **case** может быть много
  + Для прерывания используется **break**
  + Есть предложение **default**

**Функции**

* Объявление функции (function declaration)  
  **function имя(парамр1, ... параметрN) {  
   тело функции  
  }**
* Формальные и фактические параметры различаются
* Внутри функции объявление переменных с var даёт локальные переменные
* Функции создаются предварительно, до выполнения кода

*writeLine("Итого:");*

*writeLine(745);*

*function writeLine(line) {*

*document.write(line);*

*document.write("<br/>");*

*}*

* Функции – это объекты, у которых есть имя и значение, и которые можно передавать в функции

*writeLine(writeLine);*

*/\* Будет выведено следующее:*

*function writeLine(line) { document.write(line); document.write("<br/>"); }*

*\*/*

* Функция-выражение (function expression)  
  **function(параметр1, ..., параметрN) {  
   тело функции  
  }** 
  + Полученную функцию можно присвоить в переменную.
  + Функции создаются не заранее, а когда до них доходит выполнение.
  + Условное объявление функций (переменная объявляется заранее, реализация – потом).
  + Лучше такой формой не злоупотреблять.
* Можно вызывать функции, полученные в качестве объектов, и передавать им параметры

**Пример 35.**

*function showResult(a, b, calculator) {*

*document.write("Операнд 1: " + a + "<br/>");*

*document.write("Операнд 2: " + b + "<br/>");*

*document.write("Результат: " + calculator(a, b) + "<br/>");*

*}*

*function sum(a, b) {*

*return a + b;*

*}*

*showResult(5, 7, sum); //*

*showResult(8, 3, function (a, b) { return a - b; });*

**Объекты**

* Являются по сути ассоциированными массивами (картами, map)
  + ключи – имена свойств (только строки)
  + значения – значения свойств (любые типы, включая функции)
* Доступ к свойствам
  + ссылка.имяСвойства
  + ссылка["имяСвойства"]
* Свойства добавляются и исключаются динамически.
* Переменные хранят ссылки на объекты.
* Свойства объектов являются переменными.

**Пример 36. Операции со свойствами**

*var employee = {}; // Создание пустого объекта*

*employee.name = "King"; // Добавление свойств*

*employee.salary = 5000;*

*document.write(employee.name + ": " + employee.salary); // Чтение свойств*

*delete employee.salary // Удаление свойства*

*for (p in employee) { // Ещё одно чтение свойств*

*document.write(typeof(p) + " " + p + " " + employee[p]);*

*}*

*/\* Будет выведено*

*King: 5000*

*string name King*

*/\**

*// Литеральное создание объекта*

*var rectangle = {*

*width: 200,*

*height: 300,*

*getArea: function() { return this.width \* this.height; }*

*}; // this в методах при обращении к свойствам обязателен!!!*

*// Изменение свойств*

*rectangle.width = rectangle.height = 5;*

*rectangle.getPerimeter = function() {*

*return 2 \* (this.width + this.height);*

*};*

*// Проверка работы*

*document.write(rectangle.getArea() + "<br/>");*

*document.write(rectangle.getPerimeter() + "<br/>");*

**Функции-конструкторы**

* Конструкторами объектов являются функции
  + Их принято называть с большой буквы
  + Вызываются с помощью new
  + Могут иметь параметры
  + Формируют объект, используя слово this
  + Возвращают ссылку на созданный объект

**Пример 37.**

*function Employee(name, salary) {*

*this.name = name;*

*this.salary = salary;*

*this.fired = false;*

*}*

*var king = new Employee("King", "5000");*

**Встроенные объекты**

* Конструкторы
  + Math
  + Date
  + RegExp
  + Function
  + Array
  + ...
* «Обёртки»
  + String
  + Number
  + Boolean

**Наследование**

* Базовым является не наследование классов, а наследование объектов.
* Родительский объект называется прототипом.
* Механизмы наследования заметно отличаются от классического ООП с классами.
* Можно эмулировать почти что классические классы со всеми их возможностями и проблемами.

**Обработка исключений**

* Обработка исключений

*try {  
 // код, потенциально выбрасывающий исключения  
}  
catch (e) {  
 // код обработки ошибки  
}  
finally {  
 // код, всегда выполняющийся в конце  
}*

* Блока **catch** или **finally** может не быть.
* Для непользовательских методов ошибка обычно «имеет тип» **Error,** свойства которого могут отличаться в зависимости от браузера.

**Выбрасывание исключений**

* Метод вправе выбросить своё исключение  
  *throw ссылкаНаОбъектИсключения;*
* Можно выбросить вообще любой объект  
  *throw 12345;*
* Но лучше выбрасывать что-нибудь вразумительное  
  *throw {name: "OhShit!", message: "Ohhhh"}  
  throw new Error("Oooops!");*

**Пример 38. Смена контекста**

**Смена текущего объекта**

*var employee = {name: "King", salary: 5000};*

*with (employee) {*

*document.write(name + ": " + salary);*

*}*

**Смена контекста this у функции/метода**

*function raiseSalary(addition, factor) {*

*this.salary = addition + this.salary \* factor;*

*}*

*var petrov = {position: "Salesman", salary: 1000};*

*var president = {name: "King", salary: 5000};*

*raiseSalary.call(petrov, 100, 1.1);*

*raiseSalary.apply(president, [0, 1.2]);*

**Пользовательские массивы**

* Одномерные
  + Но можно создать массив массивов, в т.ч. непрямоугольный.
  + Обращение по индексу с помощью оператора **[].**
  + Нумерация элементов с 0.
  + Есть поле **length**, хранящее количество элементов.
* Способы создания
  + Пустой массив  
    **var a1 = new Array();**
  + Массив с заданным количеством элементов  
    **var a2 = new Array(10);**
  + Массив с заданными элементами  
    **var a3 = new Array(10, "и это не длина", 5.5, '!');**
  + Литеральная форма с заданными элементами  
    **var a4 = [10, "и это не длина", 5.5, '!'];**
* Динамические
  + Изменение значения **length**
  + Добавление новых элементов
  + Явно не указанные элементы получают значение **undefined**

**Пример 39.**

*<script>*

*var a = [1, 2];*

*a[5] = 5;*

*document.write(a[4] + "<br/>");*

*document.write(a[5] + "<br/>");*

*a.length = 2;*

*a.length = 5;*

*document.write(a[4] + "<br/>");*

*document.write(a[5] + "<br/>");*

*</script>*

**Сортировка пользовательских массивов**

* Выполняется с помощью метода **sort()** массива.
* По умолчанию – это сортировка в лексикографическом порядке.
* Если требуется другой порядок, то следует задать свой критерий сравнения в виде функции.

**Пример 40**

*<script>*

*var a = [1, 2, 15, 23];*

*a.sort();*

*document.write(a + "<br/>"); // 1,15,2,23*

*function compareNumeric(a, b) {*

*return a - b;*

*}*

*a.sort(compareNumeric);*

*document.write(a + "<br/>"); // 1,2,15,23*

*</script>*

**Дополнительные методы пользовательских массивов**

|  |  |
| --- | --- |
| Инвертирование порядка элементов | **reverse()** |
| Добавление элемента в конец | **push(element)** |
| Удаление элемента из конца | **pop()** |
| Добавление элемента в начало | **unshift(element)** |
| Удаление элемента из начала | **shift()** |
| Объединение в строку с указанием разделителя | **join(separator)** |
| Удаление и вставка элементов | **splice(start, [deleteCount, add1, ..., addN])** |
| Копирование части массива | **slice(begin, end)** |

**Пример 41**

*<html>*

*<body>*

*<script>*

*var a = [1, 2, 15, 23];*

*a.reverse();*

*document.write(a + "<br/>"); // 23,15,2,1*

*a.push("добавка 1");*

*document.write(a + "<br/>"); // 23,15,2,1,добавка 1*

*a.pop();*

*document.write(a + "<br/>"); // 23,15,2,1*

*a.unshift("добавка 2");*

*document.write(a + "<br/>"); // добавка 2,23,15,2,1*

*a.shift();*

*document.write(a + "<br/>"); // 23,15,2,1*

*//...*

*// ...*

*var str = a.join("; ");*

*document.write(str + "<br/>"); // 23; 15; 2; 1*

*var b = str.split("; ");*

*document.write(b + "<br/>"); // 23,15,2,1*

*a.splice(2, 1, "Замена");*

*document.write(a + "<br/>"); // 23,15,Замена,1*

*var c = a.slice(1, a.length);*

*c[0] = "И что будет?";*

*document.write(a + "<br/>"); // 23,15,Замена,1*

*document.write(c + "<br/>"); // И что будет?,Замена,1*

*</script>*

*</body>*

*</html>*

**Виды объектов**

* Встроенные
  + По сути – библиотеки и базовые объекты
* Пользовательские
  + Всё, что создаёт пользователь-программист
* Серверные
  + Определяют и предоставляют взаимодействие с сервером
* Клиентские
  + Определяют и предоставляют взаимодействие с браузером и документом

**BOM и DOM**

* Browser Object Model
  + Объектная модель для взаимодействия с браузером
  + В настоящий момент не стандартизована
  + Базовый объект – **window**
* Document Object Model
  + Объектная модель для взаимодействия с документом
  + В целом стандартизирована
  + Базовый объект – **document**

**Объект window**

* Глобальный объект.
* Все объявляемые переменные и объекты становятся его свойствами.
* Содержит либо напрямую информацию о документе, либо ассоциированный массив (карт, map) фреймов (frames).
* Имеет свои свойства, методы и события.

**Пример 42. Взаимодействие с пользователем**

*// Окно с сообщением*

*alert("Случилось страшное!!!");*

*// Окно с запросом на подтверждение*

*// Возвращает true или false*

*confirm("Хотите поговорить об этом?..");*

*// Окно со вводом строки*

*// Возвращает введённую строку*

*prompt("Вам слово!", "говорите сюда");*

**Создание и закрытие новых окон**

* Создание
  + Параметры задаются строкой
  + Поведение зависит от браузера
  + Возвращается ссылка на объект окна

*var ncwin = window.open('http://www.netcracker.com',*

*'Сайт компании', 'directories=no,height=300,location=no,' +*

*'menubar=no,resizable=yes,scrollbars=yes,' +*

*'status=no,toolbar=no,width=300');*

* Закрытие

*ncwin.close();*

**Запуск новых потоков**

* Запуск нового потока
  + *setTimeout("код", времяВМиллисекундах)*
  + Возвращает ссылку на поток
* Остановка запущенного разового потока
  + *clearTimeOut(ссылкаНаПоток)*
* Периодический запуск нового потока
  + *setInterval("код", времяВМиллисекундах)*
  + Возвращает ссылку на поток
* Остановка запущенного периодического потока
  + *clearInterval(ссылкаНаПоток)*

**Свойства window**

* **location** – объект «типа» URL, текущий адрес
  + **href** – собственно URL
  + Набор свойств, соответствующих фрагментам URL
  + Методы
  + **replace()**
  + **reload()**
* **history** – объект, хранящий историю переходов по страницам
  + **forward()**
  + **back()**
  + **go(количествоСтраниц)**
* **navigator** – объект, хранящий данные о браузере
* **opener** – ссылка на родительское окно

**Объект document**

* В нём и живёт DOM.
* Это тоже ассоциированный массив специфического вида.
* Все теги получают в соответствие объект.
  + Имя объекта определяется значением атрибута **name** тега.
  + Атрибуты тегов становятся свойствами соответствующих объектов.
  + Не все свойства объектов соответствуют атрибутам тегов.
* Объекты формируются по ходу чтения документа браузером.
* Некоторые виды объектов объединяются в дополнительные массивы (формы, ссылки и т.д.).
* У объектов могут быть события.
* У объектов могут быть дополнительные методы.

**Прямая запись в документ**

* Методы
  + **document.write(значение)**
  + **document.writeln(значение)**
* Используются для генерирования (в т.ч. условного) содержимого документа.
* Можно использовать только если документ ещё «открыт».
  + Для загружаемых документов – документ ещё не до конца прочитан браузером.
  + Для документов в динамически открытых окнах – после вызова **document.open()** и до вызова **document.close().**
* Если документ уже закрыт, то его можно изменять только через объекты и управление ими.

**События объектов**

* Обычно именуются **onчто-нибудь.**
* Обработчик можно указать прямо в HTML-коде.
* Обработчик можно задать программно.
* Возврат из некоторых обработчиков null означает прекращение обработки.

**Пример 43.**

*<html>*

*<body>*

*<p*

*onMouseOver="document.getElementById('addition').innerHTML = 'А он есть...'"*

*onMouseOut="document.getElementById('addition').innerHTML = ''">*

*Видишь суслика?..</p>*

*<p id="addition"></p>*

*<img id="picture" src="http://turbo-suslik.ru/images/suslik.jpg"*

*width="150px" height="150px"/>*

*<script>*

*document.getElementById("picture").onclick = function() {*

*alert("Вот же он, суслик!")};*

*</script>*

*</body>*

*</html>*

**Пример 44. Динамическое изменение DOM**

*<html>*

*<body>*

*<table border="1px" id="table">*

*<head id="head"> <td>Фамилия</td> <td>Имя</td> </head>*

*<tr> <td>Иванов</td> <td>Иван</td> </tr>*

*</table>*

*<form name="mainform">*

*Фамилия:*

*<input type="text" name="sname"/><br/>*

*Имя:*

*<input type="text" name="fname"/><br/>*

*<input type="submit" name="addition" value="Добавить"*

*onClick="appendToTable(document.mainform.fname.value, document.mainform.sname.value); return false;"/>*

*</form>*

*<script>*

*function appendToTable(fname, sname) {*

*var table = document.getElementById("table");*

*var row = document.createElement("tr");*

*var surname = document.createElement("td");*

*var firstname = document.createElement("td");*

*surname.innerHTML = sname;*

*firstname.innerHTML = fname;*

*row.appendChild(surname);*

*row.appendChild(firstname);*

*table.appendChild(row);*

*}*

*</script>*

*</body>*

*</html>*

**Работа с формами**

* Доступ к форме:
  + По имени как к свойству документа
  + В массиве **document.forms**
* Доступ к элементам:
  + По имени как к свойству формы
  + В массиве **elements** формы
* Атрибуты тегов элементов формы и самой формы – свойства соответствующих объектов.
* События формы:
  + **onsubmit**
  + **onreset**
* Методы формы
  + **submit()**
  + **reset()**

**Программирование гиперссылок**

* Доступ к гиперссылкам:
  + По имени/идентификатору
  + В массиве **document.links[]**
  + Объект имеет «тип» URL
* Объект ссылки имеет свойства и события
* Ссылка может иметь программный вид
  + **javascript:код**
  + результат работы кода может быть показан в браузере
  + Результатом считается результат последнего выражения
  + Приём **void(0);**
  + Так можно не только в ссылке, но и в **action** у форм

**Пример 45. Использование ссылок**

*<html>*

*<body>*

*<table border="1px" id="table">*

*<head id="head"> <td>Фамилия</td> <td>Имя</td> </head>*

*<tr> <td>Иванов</td> <td>Иван</td> </tr>*

*</table>*

*<form name="mainform">*

*Фамилия: <input type="text" name="sname"/><br/>*

*Имя: <input type="text" name="fname"/><br/>*

*</form>*

*<a href="javascript:appendToTable(document.mainform.fname.value,*

*document.mainform.sname.value);*

*void(0);">Добавить</a>*

*<a href="javascript:document.mainform.submit();">*

*Отправить данные</a>*

**Некоторые замечания**

* Часто JavaScript воспринимается как набор приёмов по программированию клиентской части.
* Это не так.
* JavaScript – интересный и богатый язык с широкими возможностями.
* И детальное его изучение требует значительного времени…

**HTML (самостоятельно)**

**HTML (HyperText Markup Language)** – язык разметки гипертекста. Его предназначение - указывать браузеру, как разместить элементы на странице при ее отображении на экране монитора.

Например,

*<b>Этот текст будет отображаться жирным</b>*

*<i>А этот - курсивом</i>*

Бывают также теги одиночные, т.е. не требующие закрытия. Такие теги просто совершают определенное действие, например, перевод строки или вставку рисунка.

Теги могут иметь параметры (или атрибуты), например,

*<body bgcolor="red">Здесь содержимое</body>*

**Структура HTML**

**Пример 30.**

*<html>*

*<head>*

*<title>Тег body в html</title>*

*<base href="http://www.my\_site.ru/">*

*<link rel="stylesheet" type="text/css" href="style.css">*

*<meta http-equiv="Content-Type" content="text/html;charset=windows-1251">*

*<meta http-equiv="KEYWORDS" content="тег, html, заголовок страницы">*

*<meta http-equiv="DESCRIPTION" content="Описание элементов заголовка">*

*<script type="text/javascript" src="function.js"></script>*

*</head>*

*<body bgcolor="yellow" leftmargin="100" topmargin="50" rightmargin="50" bottommargin="50" text="gray" alink="red" vlink="green">*

*Просто текст*

*<br/>*

*<a href="index.html">Ссылка на страницу index1.html</a>*

*<br/>*

*<a href="map.html">Ссылка на страницу index2.html</a>*

*</body>*

*</html>*

Теги *<html> </html>* являются контейнером для всех остальных, т.е. в них помещаются все остальные. Таким образом, ваш документ должен начинаться с тега *<html>*, а заканчиваться тегом *</html>*.

Сам документ условно разделен на две части - заголовок документа (теги *<head> </head>*) и тело документа (теги *<body> </body>*). Заголовок документа содержит служебную информацию и не влияет на внешний вид документа. Его задачей является предоставление браузеру пользователя или серверу информации о том, как отобразить ваш документ.

Единственным обязательным элементом заголовка документа являются теги *<title> </title>*. Они необходимы, чтобы дать документу название, оно отражается в заголовке окна браузера.

Одиночный тег *<base>* служит для указания полного URL-адреса документа. Зачем это нужно? Представьте, что вы сохранили какую-нибудь html-страницу себе на компьютер, с тем, чтобы просмотреть ее позже. Все картинки на этой страницы превратятся в красные крестики. Но если вы не отключены от сети, а на странице присутствует тег *<base>*, то браузер будет знать, где искать необходимый файл, найдет его и загрузит картинки.

Одиночный тег *<link>* необходим для подключения внешних файлов. Например, если вы будете использовать каскадную таблицу стилей, то ее удобнее хранить в отдельном файле и подключать этот файл ко всем страницам сайта.

У тега *<link>* несколько атрибутов:

*href* - указывает URL-адрес подключаемого файла.

*rel* - указывает на тип отношения данного документа к внешнему (например: rel="stylesheet" указывает, что внешний файл определяет стиль текущего документа).

*type* - указывает тип и параметры присоединенной таблицы стилей.

Информация в этом теге не имеет никакого отношения к HTML, однако ее использование очень удобно для решения ряда задач:

- указание кодировки страницы, например, для русского текста в кодировке Windows;

- указание ключевых слов страницы (используется при оптимизации страниц);

* указание краткого описания страницы.

Возможны и другие варианты. Рассмотрим параметры тега *<meta>*:

*http-equiv* - определяет свойство тега (тип контента, ключевые слова, описание и т.д.).

*name* - используется для дополнительного описания тега, если отсутствует, то считаеся эквивалентным параметру *http-equiv*.

*content* - значение параметра *http-equiv*.

Теги *<script></script>* используются для подключения внешних файлов скриптов. Это позволяет оптимизировать код страниц. Например, если вы используете функции java script для большинства своих страниц, то поместив эти функции на отдельную страницу - function.js, с помощью тега *<script>* можно указать путь к этой странице.

Это повышает читабельность кода и ускоряет загрузку страниц. У этого тега несколько параметров:

*language* - указывает язык написания скрипта, в спецификации HTML 4.0 данный параметр не рекомендуется к употреблению. Вместо него следует указывать параметр *type*.

*type* - указывает тип MIME для языка.

*src* - указывает путь к внешнему файлу со скриптами.

Все, что отображается на web-странице, находится в тегах *<body></body>* . Это текст, картинки и исполняющиеся скрипты, а также теги для оформления всего этого.

Обязательных параметров у тега *<body>* нет, да и применение необязательных параметров тоже не приветствуется. Тем не менее, большинство параметров до сих пор поддерживается разными браузерами. Рассмотрим те, которые пока поддерживаются всеми браузерами:

*alink* - устанавливает цвет активной ссылки. Текущий цвет ссылки меняется на активный при нажатии на нее.

*vlink* - устанавливает цвет посещенной ссылки, т.е. той, по которой уже щелкали.

*background* - указывает на изображение, которое будет использоваться в качестве фонового рисунка. Этот рисунок заполняет собой все видимое пространство окна. Если рисунок меньше окна браузера, то он повторяется, образуя мозаику из одинаковых картинок. На стыке этих картинок возникают видимые переходы. Поэтому к подбору фоновых рисунков следует подходить с большим вниманием.

*bgcolor* - указывает фоновый цвет документа.

*leftmargin* - определяет отступ от левого края окна браузера до контента страницы.

*rightmargin* - определяет отступ от правого края окна браузера до контента страницы.

*topmargin* - определяет отступ от верхнего края окна браузера до контента страницы.

*bottommargin* - определяет отступ от нижнего края окна браузера до контента страницы.

*text* - устанавливает цвет текста для всего документа.

**Теги, делающие текст заголовками**

*<h1></h1>*

*<h2></h2>*

*<h3></h3>*

*<h4></h4>*

*<h5></h5>*

*<h6></h6>*

Эти теги выделяют текст в виде заголовков. Т.е. каждый заголовок начинается с новой строки, выделен полужирным шрифтом и имеет свой размер (заголовок первого уровня самый большой, шестого - самый маленький).

Эти теги могут использоваться с параметром горизонтального выравнивания *align*. Возможные значения этого параметра:

*left* - слева,

*right* - справа,

*center* - по центру,

*jastify* - по ширине.

**Абзацы и перенос строки**

Тег *<br>* - тег принудительного перевода строки. Текст, после этого тега начинается с новой строки.

Теги *<p></p>* разделяют текст на абзацы. Перед началом каждого абзаца следует поместить тег *<p>*, закрывающий тег не обязателен. В отличии от тега *<br>* абзацы отделяются друг от друга пустой строкой.

У тега *<p>* есть параметр *align*, который задает способ выравнивания текста внутри параграфа. Возможные значения этого параметра:

*left* - слева,

*right* - справа,

*center* - по центру,

*jastify* - по ширине.

**Курсив, полужирный, подчеркивание**

*<cite></cite>*

*<dfn></dfn>*

*<em></em>*

*<i></i>*

Эти теги выделяют текст курсивом, но делают они это по разным причинам. Теги *<cite></cite>* используются для логического выделения названий книг, статей и цитат. Теги *<dfn></dfn>* используются для выделения определений. Тегами *<em></em>* и *<i></i>* выделяют важные фрагменты текста. Последний не рекомендуется к употреблению.

*<strong></strong>*

*<b></b>*

Оба используются для выделения важных фрагментов текста, но предпочтительнее использовать первый.

*<ins></ins>*  
*<u></u>*

Оба используются для подчеркивания важных фрагментов текста, но предпочтительнее использовать первый.

**Тег font и его параметры**

Теги *<font></font>* указывают параметры шрифта текста:

*face* - название шрифта. Названий шрифтов можно привести несколько, через запятую. В этом случае, если первый указанный шрифт не будет найден (вы же не знаете, какие шрифты установлены на компьютере пользователя), браузер станет использовать следующий по списку.

*size* - размер шрифта в условных единицах от 1 до 7. По умолчанию размер шрифта равен 3.

*color* - цвет текста (по умолчанию - черный).   
Существуют два способа задания цвета: по имени и указанием шестнадцатеричного кода цвета.

**Форматирование текста**

Теги *<abbr></abbr>* указывают на то, что текст является аббревиатурой. По умолчанию текст подчеркивается пунктиром, а при наведении на него курсора мыши появляется всплывающая подсказка с расшифровкой аббревиатуры. Это достигается добавлением в тег параметра *title*, в качестве значения которого и выступает расшифровка. Теги *<acronym></acronym>* ведут себя аналогично тегам *<abbr></abbr>* , но поддерживаются всеми браузерами.

Теги *<address></address>* применяются для указания адреса. Отображается курсивом с новой строки.

Теги *<code></code>* используются при необходимости вывести на экран фрагмент программного кода.

Теги *<del></del>* делают текст перечеркнутым. Аналогично ведут себя теги *<strike></strike>* и теги *<s></s>* , но они не рекомендуются к использованию спецификацией HTML 4.0.  
 Теги *<q></q>* выделяют короткие цитаты.

Теги *<big></big>* выводят текст большего размера, чем остальная часть текста. Теги *<small></small>* выводят текст меньшего размера, чем остальная часть текста.

Тег *<hr>* рисует горизонтальную линию, отделенную сверху и снизу пустыми строками.

Теги *<pre></pre>* используются тогда, когда необходимо отобразить текст так, как он набран.

**Специальные символы**

*&lt;* - знак "меньше"

*&gt;* - знак "больше"

*&nbsp;* - непрерывный пробел

*&copy;* - знак "copyright"

*&amp;* - знак "амперсанд"

*&laquo;* - левая открывающаяся кавычка

*&raquo;* - правая закрывающаяся кавычка

*&uarr;* - стрелка вверх

*&rarr;* - стрелка вправо

*&darr;* - стрелка вниз

*&larr;* - стрелка влево

*&harr;* - двунаправленная стрелка влево-вправо

**Ссылки**

Ссылка - одно из важнейших понятий для html-документов. За переходы отвечает тег-контейнер *<a></a>*, а за то, куда направить пользователя по щелчку - его атрибут *href*.

При абсолютной адресации используется полный путь, т.е. с указанием протокола, сервера, каталога и страницы. В нашем примере абсолютный адрес страницы index.html следующий:

*<a href="http://www.mysite.ru/public/index.html">*

При относительной адресации путь указывается от того каталога, в котором хранится исходный файл. Например, если мы хотим разместить ссылку со страницы index.html (исходный файл) на страницу articles.html, то, учитывая, что обе страницы лежат в одном каталоге (т.е имеют один уровень), она будет выглядеть следующим образом:

*<a href="articles.html">*

Если же мы захотим сделать ссылку со страницы index.html на страницу price.html, то, учитывая, что страницы лежат в разных каталогах, она будет выглядеть следующим образом:

*<a href="../private/price.html">*

По умолчанию ссылки загружаются в то же окно, но если указать параметр *target="\_blank"*, то ссылка загрузится в новое окно.

Иногда, когда документ слишком длинный, полезно сделать ссылки на его разделы. Для таких переходов следует создать пустую ссылку с параметром *name* там, куда следует перейти. И ссылку с указанием этого имени (после знака #) в качестве адреса параметра *href* там, откуда следует перейти.

У ссылок есть параметр *title*, в качестве значения которого можно указать текст, который будет выводиться в качестве подсказки, если подвести курсор мыши к ссылке.

**Нумерованные списки**

Такой список представляет собой перечень "пронумерованный" каким-либо образом. Это могут быть арабские цифры, цифры римского алфавита или буквы. Чтобы указать браузеру, что список будет нумерованным, используются теги *<ol></ol>*. Весь список располагается между этими тегами и каждый его пункт задается тегами *<li></li>*.

*<ol>*

*<li>кошки</li>*

*<li>собаки</li>*

*<li>лошади</li>*

*</ol>*

По умолчанию нумерация ведется арабскими цифрами. Это можно изменить с помощью параметра *type*:

- *type="A"* - нумерация заглавными латинскими буквами (A, B, C).

- *type="a"* - нумерация прописными латинскими буквами (a, b, c).

- *type="I"* - нумерация большими римскими цифрами (I, II, III).

- *type="i"* - нумерация маленькими римскими цифрами (i, ii, iii).

Если нам нужно начать нумерацию не с единицы, то следует использовать параметр *start*, в качестве значения которого указывается число, с которого будет начинаться список (если нумерация задается с помощью букв, то это число будет указывать на позицию в алфавите).

**Маркированные списки**

В маркированном списке вместо букв и цифр используются маркеры. Чтобы указать браузеру, что список будет маркированным, используются теги *<ul></ul>*. Весь список располагается между этими тегами и каждый его пункт задается тегами *<li></li>*.

*<ul>*

*<li>кошки</li>*

*<li>собаки</li>*

*<li>лошади</li>*

*</ul>*

По умолчанию маркер отображается в виде черного кружочка. Это можно изменить с помощью параметра *type*:

- *type="disc"* - закрашенный кружочек.

- *type="circle"* - пустой кружочек.

- *type="square"* - закрашенный квадратик.

Если нужно начать нумерацию не с единицы, то следует использовать параметр *start*, в качестве значения которого указывается число, с которого будет начинаться список (если нумерация задается с помощью букв, то это число будет указывать на позицию в алфавите).

**Работа с изображениями**

Фоновое изображение заполняет собой все пространство элемента для которого оно задано. Так, указав в теге *<body>* атрибут *background="fon.gif"*, рисунком fon.gif будет залито все окно браузера.

Для размещения на странице изображений используется тег *<img>*, имеющий единственный обязательный параметр *src*, определяющий URL-адрес изображения.

Для выравнивания изображений используется параметр *align*. Он имеет следующие значения: *"left", "right", "top", "texttop", "middle", "absmiddle", "bottom", "absbottom".*

Любое изображение имеет размер, заданный в пикселах. Для этого у тега *<img>* существуют параметры *width* - ширина и *height* - высота. Они задаются в пикселах или в процентах.

Вы, наверно, обратили внимание, что текст очень близко прилипает к картинкам. Это не всегда красиво. Для решения проблемы прилипания текста к картинкам имеются параметры *hspace* - горизонтальный отступ и *vspace* - вертикальный отступ. Отступы задаются в пикселах.

Если у пользователя отключен режим загрузки изображений, то вместо картинки он увидит красный крестик (в Internet Explorer) или другую иконку (в других браузерах). Чтобы дать ему подсказку, что изображено на рисунке, используется параметр *alt*.

Чтобы изображение сделать ссылкой, достаточно поместить его внутрь тега-контейнера *<a>*. В примере наша картинка будет ссылаться на страницу html\_example1.html, которая будет открываться в отдельном окне.

*<a href="html\_example1.html" target="\_blank">*

*<img src="../images/fish.gif" alt="рыбка">*

*</a>*

**Работа с таблицами**

Таблица - один из основных инструментов для создания web-страниц. Без использования CSS, только с помощью таблиц можно создавать страницы со сложным дизайном.

*<table>*

*<caption>Название таблицы</caption>*

*<tr><th>1</th><th>2</th><th>3</th></tr>*

*<tr><td>11</td><td>12</td><td>13</td></tr>*

*<tr><td>21</td><td>22</td><td>23</td></tr>*

*<tr><td>31</td><td>32</td><td>33</td></tr>*

*</table>*

Для оформления у тега *<table>* существует ряд параметров:

*width* - задает ширину таблицы (в пикселах или % от ширины экрана),

*bgcolor* - задает цвет фона ячеек таблицы,

*background* - заливает фон таблицы рисунком,

*border* - задает рамку указанной ширины (в пикселах) вокруг всей таблицы,

*align* - задает выравнивание таблицы: слева (right), справа (left), по центру (center),

*cellspacing* - задает расстояние между ячейками таблицы (в пикселах),

*cellpadding* - задает расстояние между текстом и внутренней границей ячейки таблицы (в пикселах).

За границы отвечают два параметра:   
*frame* - задает вид рамки вокруг таблицы и может принимать следующие значения: *void*, *above*, *below*, *hsides*, *vsides*, *lhs*, *rhs*, *box,*  
*rules* - задает вид внутренних границ таблицы и может принимать следующие значения: *none*, *groups*, *rows*, *cols*, *all*.

У ячеек можно указать размер: *width* - задает ширину столбца (в пикселах или в процентном соотношении, где за 100% принимается ширина таблицы), *height* - задает высоту ячейки, причем все ячейки в этой же строке станут этой высоты. Параметр *colspan* указывает, сколько столбцов занимает ячейка. Аналогично *rowspan* – для строк.

**HTML формы**

Формы дают возможность пользователям вводить информацию. В html форма задается тегами *<form></form>*. Все остальные элементы формы располагаются между этими тегами.

У тега *<form>* есть несколько параметров:

*name* - имя формы. Необходимо, если на странице несколько форм,

*action* - определяет URL-адрес, по которому будет отправлена информация введеная пользователем,

*method* - определяет способ отправки информации,

*target* - указывает имя окна, в котором будут отображаться результаты обработки отправленной формы.

*<form name="forma"></form>*

**Текстовое поле**

Простое однострочное текстовое поле, в которое можно вводить и редактировать текст. Задается тегом *<input>*

*<form name="forma">*

*<input type="text" name="text1" size="20" maxlength="50" value="Введите текст">*

*</form>*

Параметры:

*name* - имя элемента,

*type* - тип элемента (в данном случае - text),

*size* - размер текстового поля в символах, которые одновременно будут видны, при вводе большего количества символов, они будут прокручиваться,

*maxlength* - максимальное количество символов, которое можно ввести в поле, если опустить этот параметр, то число символов будет неограниченным,

*value* - текст, который будет отображаться (его можно стереть), при отсутствии этого параметра поле будет пустым,

*disabled* - блокирует поле от любых изменений,

*readonly* - делает поле доступным только для чтения.

Для ввода паролей используется текстовое поле с параметром *type="password".*

**Чекбокс**

*<form name="forma">*

*Какими языками вы владеете:<br>*

*<input type="checkbox" name="lan1" value="english" checked>английский*

*<input type="checkbox" name="lan2" value="german"> немецкий*

*<input type="checkbox" name="lan3" value="spanish"> испанский*

*<input type="checkbox" name="lan4" value="french"> французский*

*</form>*

Рассмотрим параметры:

*type* - тип элемента (в данном случае - checkbox),

*name* - имя элемента, указывает программе-обработчику формы, какой пункт выбрал пользователь,

*value* - значение элемента, указывает программе-обработчику формы значение пункта, который выбрал пользователь. В нашем примере выбран пункт английский, следовательно, программа-обработчик получит: lan1="english",

*checked* - им обычно помечают наиболее вероятные для выбора пункты, пользователь щелчком мыши может выбрать другие пункты.

**Радиокнопки**

В отличие от чекбоксов, можно выбрать только один пункт. В связи с этим значения параметра *name* должны быть одинаковы для всех элементов группы. Параметр *type="radio"*, все остальные аналогичны.

*<form name="forma">*

*Укажите ваш пол:<br>*

*<input type="radio" name="sex" value="man" checked>мужской*

*<input type="radio" name="sex" value="woman"> женский*

*</form>*

**Кнопки**

Существует четыре вида кнопок:

*submit* - кнопка отправки содержимого формы web-серверу. Ее параметры: *type="submit"* - тип кнопки, *name* - имя кнопки, *value* - надпись на кнопке.

*image* - графическая кнопка отправки содержимого формы web-серверу. Для ее использования необходимо подготовить картинку кнопки, а потом использовать ее в виде кнопки. Ее параметры: *type="image"* - тип графической кнопки, *name* - имя кнопки, *src* - адрес картинки для кнопки.

*reset* - кнопка, позволяющая восстановить все значения по умолчанию в форме. Ее параметры: *type="reset"* - тип кнопки очищения, *name* - имя кнопки, *value* - надпись на кнопке.

*button* - произвольная кнопка, ее действия назначаются вами, т.е. сама она делать ничего не умеет. Ее параметры: *type="button"* - тип произвольной кнопки, *name* - имя кнопки, *value* - надпись на кнопке, *onclick* - указывает, что делать при щелчке по кнопке. Также у этого типа кнопок есть и другие события (например, двойной щелчок).

*<form name="forma">*

*<input type="submit" name="submit" value="Отправить">*

*<input type="image" name="but\_img" src="but.gif">*

*<input type="reset" name="reset" value="Очистить">*

*<input type="button" name="button" value="Отправить">*

*</form>*

**Файлы и многострочный ввод**

Поле для ввода имени файла, сопровождаемое кнопкой Browse (Обзор), при щелчке по которой открывается окно просмотра дерева папок компьютера, где можно выбрать нужный файл. Выбранный файл присоединяется к содержимому формы при отправке на сервер.

*<form name="forma">*

*<input type="file" name="load" size="50">*

*</form>*

Для объемных текстов, например, для почтовых сообщений, удобно использовать именно этот элемент. Он создается тегами *<textarea> </textarea>* и имеет следующие параметры:

*name* - имя поля,

*cols* - ширина поля в символах,

*rows* - количество строк текста, видимых на экране,

*wrap* - способ переноса слов: *off* - переноса не происходит, *virtual* - перенос отображается, но на сервер поступает неделимая строка, *physical* - перенос и на экране и при поступлении на сервер,

*disabled* - неактивное поле,

*readonly* - разрешено только чтение.

*<form name="forma">*

*<textarea cols="20" rows="3" wrap="off">*

*</form>*

**Раскрывающиеся списки**

Списки бывают с возможностью выбора одного элемента и с множественным выбором. Задаются и те, и другие с помощью тегов *<select> </select>*, внутри которых располагаются элементы значений, заданных тегом *<option>*. Рассмотрим параметры этих тегов:

*<select>*:

*name* - имя списка. Каждый выбранный элемент списка при передаче на сервер будет иметь вид: name.value, где значение (value) берется из тега option,

*size* - определяет количество видимых элементов в списке: 1 - простой раскрывающийся список, больше 1 - список с полосой прокрутки,

*multiple* - разрешает выбор нескольких элементов списка.

*<option>*:

*selected* - им помечают наиболее вероятный для выбора элемент списка, если список со множественным выбором, то можно пометить несколько пунктов,

*value* - значение, которое будет отправлено серверу, если пункт выбран.

*<form name="forma">*

*Какой язык вы хотите изучать:*

*<select name="language" size="1">*

*<option selected value="html">html*

*<option value="php">php*

*<option value="java">java*

*</select>*

*</form>*

**Навигационные карты**

Многие html-страницы используют для организации ссылок так называемые карты-изображения. При таком подходе берется изображение и к различным его областям привязываются ссылки.

Для описания таких областей используются теги *<map> </map>* с единственным параметром *name*, который задает имя карты ссылок и используется потом для ссылки на эту карту. Карту надо подключить к картинке, для этого в тег *<img>* надо добавить параметр *usemap*, в качестве значения которого выступает имя карты после значка #:

*<body>*

*<img src="map.gif" usemap="#map" border="0">*

*<map name="map">*

*<area shape="rect" coords="550,135,590,192" href="#" alt="area1">*

*<area shape="rect" coords="591,90,660,192" href="#" alt="area2">*

*<area shape="rect" coords="661,0,730,192" href="#" alt="area3">*

*</map>*

*</body>*

Для описания конкретных областей внутри тегов *<map> </map>* используются теги *<area>*. Этот тег имеет следующие параметры:

*shape* - определяет форму области, может принимать следующие значения: *rect* - область в виде прямоугольника, *circle* - область в виде круга, *poly* - область в виде многоугольника, *default* - вся область.

*coords* - задает координаты отдельной области: для *rect* задаются координаты верхнего левого и правого нижнего углов прямоугольника, для *circle* задаются координаты центра круга и радиус, для *poly* задаются координаты вершин многоугольника в нужном порядке.

*href* - определяет адрес ссылки.

*target* - применяется при использовании фреймов и указывает фрейм в который нужно загрузить страницу.

*alt* - задает альтернативный текст для области.

**CSS**

HTML - это язык разметки, но его создатели так хотели облегчить жизнь разработчиков сайтов, что добавляли в него элементы и параметры, отвечающие за внешний вид страницы: теги *<font>*, *<b>*, *<i>*, *<bgcolor>* и так далее. На какой-то момент код страниц стал слишком громоздким и нечитабельным. Тогда было принято решение разделить разметку страницы (HTML) и ее визуальное оформление (CSS).

CSS (Cascading Style Sheets) - каскадные таблицы стилей. *Стиль* - набор параметров, задающий внешнее представление объекта.

Каждый элемент на странице может иметь свой стиль (параграфы, заголовки, линии, текст). Набор стилей всех элементов называют *таблицей стилей*. Если для одного элемента задано несколько стилей (как в примере с заголовками), то применяется *каскадирование*, которое определяет приоритет того или иного стиля.

**Преимущества CSS**

* CSS позволяет значительно сократить размер кода и сделать его читабельным.
* CSS позволяет задавать такие параметры, которые нельзя задать только языком HTML. Например, убрать подчеркивание у ссылок.
* CSS позволяет легко изменять внешний вид страниц. Представьте, вы сделали сайт из 50 страниц, на которых все заголовки синего цвета. Через какое-то время, вы захотели поменять синий цвет на зеленый. Вам придется пройтись по всем 50 страницам и поменять цвет в соответствующем атрибуте. С CSS вам придется сделать это лишь один раз, в таблице стилей.

С CSS связана так называемая блочная верстка сайта.

**Подключение CSS к HTML**

*Внешние таблицы стилей*

Нам надо подключить страницу style.css к html-странице. Для этого в html существует тег *<link>*, который и отвечает за подключение внешних файлов.

*<head>*

*<title>Подключение CSS к HTML</title>*

*<link rel="stylesheet" type="text/css" href="style.css">*

*</head>*

*Внутренние таблицы стилей*

Так называют таблицу стилей, заданную внутри элемента HTML, при помощи атрибута *style*.

*<h1 style="color:red">Это заголовок красного цвета</h1>*

Недостаток этого способа очевиден: параметр *style* придется задавать каждому заголовку. Таким образом, теряется преимущество использования CSS.

*Встроенные таблицы стилей*

При этом способе таблица стилей встраивается в заголовок html-страницы. Для этого в HTML существуют теги *<style></style>*, с параметром *type*, который указывает, что подключается именно таблица стилей CSS (существуют и другие).

*<head>*

*<title>Подключение CSS к HTML</title>*

*<style type="text/css"> h1 { color:red } </style>*

*</head>*

*<body>*

*<h1>Этот заголовок будет красного цвета</h1>*

*<h1 style="color:blue">Этот заголовок будет синего цвета</h1>*

*</body>*

**Правила и селекторы CSS**

CSS, как и любой язык, имеет свой синтаксис. В нем нет ни элементов, ни параметров, ни тегов. В нем есть правила. Правило состоит из селектора и блока объявления стилей, заключенного в фигурные скобки. Сам блок объявления стилей состоит из свойств и их значений, разделенных точкой с запятой.  
*body {*

*background: blue;*

*color: white;*

*}*

*h1 {*

*color: red;*

*}*

В HTML идентификатор элемента задается при помощи параметра *id*, в качестве значения которого указывается уникальное имя.

*p#pink { color:pink; }*

*<p id="pink">Текст параграфа с идентификатором (id).</p>*

Если нужно, чтобы розовый цвет текста был у двух или трех параграфов, то для этого в HTML существует параметр *class*, в качестве значения которого указывается имя стиля.

*p.pink { color:pink; }*

Контекстный селектор

*p i { color:green; }*

*p > i { color:green; }*

Группировка селекторов

*h1, h2, h3 { color:green; }*

**Псевдоэлементы и псевдоклассы**

К псевдоэлементам относят:

*first-letter* - задает стиль первой букве слова.

*first-line* - задает стиль первой строке абзаца.

В CSS существуют четыре псевдокласса, они позволяют работать со ссылками. У ссылок есть четыре состояния: простая, активная, посещенная и та, на которую наведен курсор. Состояние ссылок зависит от действия пользователя, и браузер, в зависимости от этих действий может применять разные стили. Для описания этих стилей и существуют псевдоклассы.

*a:link* - задает стиль обычной ссылки.

*a:active* - задает стиль активной ссылки.

*a:visited* - задает стиль посещенной ссылки.

*a:hover* - задает стиль ссылки, на которую наведен курсор.

**color и background**

Задать цвет текста для элемента можно тремя способами:

*body { color:green; } h1 { color:#FF0000; } h2 { color:rgb(255,0,0); }*

*background – н*а самом деле это группа свойств, так или иначе связанная с фоном. При помощи CSS фон можно задать не только странице, но и заголовку, и абзацу, и любому другому элементу. Рассмотрим группу свойств *background*:

*background-color* - задает цвет фона. По умолчанию не наследуется, но его можно сделать наследуемым, если в качестве значения указать значение *inherit*.

*background-image* - задает фоновое изображение. Значением свойства является URL графического файла. Формат задания следующий: сначала идет обозначение функции url, а затем в круглых скобках указывается путь к файлу. Путь к файлу указывается относительно таблицы стилей.

*background-repeat* - задает возможность повторения фонового изображения. В качестве фонового изображения может выступать как цельное изображение (например, шапка сайта), так и маленькое изображение, которое должно замостить собой все пространство элемента. Данное свойство как раз и указывает, повторять ли изображение и, если да, то как именно повторять. Возможны 4 варианта: *repeat* - повторять изображение по горизонтали и вертикали, *repeat-x* - повторять изображение только по горизонтали, *repeat-y* - повторять изображение только по вертикали, *no-repeat* - не повторять изображение.

*background-attachment* - указывает, должно ли фоновое изображение прокручиваться вместе с текстом элемента или оно должно оставаться неподвижным. Может принимать два значения: *scroll* - фон прокручивается вместе с текстом. Это значение используется по умолчанию, *fixed* - фоновое изображение фиксируется относительно окна браузера, и во время прокрутки текста оно остается неподвижным.

*background-position* - задает расположение элемента относительно окна браузера. Значения можно задавать в процентах, в единицах длины и при помощи ключевых слов.

*body { background-image: url(picture.gif); background-repeat: no-repeat; background-color:#243CED; background-position:50px 50px; color: yellow; }*

*body { background: url(picture.gif) no-repeat #33CCFF center top; color: yellow; }*

**Шрифты**

Для задания параметров шрифтов в CSS используется свойство *font*.

*font-family* задает гарнитуру шрифта. Все шрифты можно условно разделить на несколько групп: *Serif* - шрифты с засечками, например, Times New Roman, *Sans-serif* - шрифты рубленные, без засечек, например, Arial, *Monospace* - моноширинные шрифты, например, Courier New, *Cursive* - курсивные шрифты, например, Calisto MT, *Fantasy* - декоративные шрифты, например, Torhok.

*font-variant* задает вариант написания букв из двух возможных: *normal* (обычный) и *small-caps* (малые прописные буквы).

*font-weight* задает толщину букв шрифта. В качестве значений выступают числа: 100, 200, 300, 400, 500, 600, 700, 800 и 900. А также ключевые слова: *normal* (нормальный), *bold* (полужирный), *bolder* (более жирный по отношению к базовому, унаследованному от предка) и *lighter* (менее жирный по отношению к базовому, унаследованному от предка).

*font-size* задает размер шрифта. Задавать размер в css можно тремя способами: с помощью ключевых слов (xx-small, x-small, small, medium, large, x-large, xx-large, smaller, large), с помощью относительных единиц (% и em), с помощью единиц измерения длины (пикселы, пункты, сантиметры и миллиметры).

*p { font-family: Verdana, sans-serif; font-size: 12px; font-weight: bold; font-variant: small-caps; font-style: italic; }*

*p { font: italic 12px Verdana, sans-serif; }*

**Текст**

*text-decoration* отвечает за оформление текста. Может принимать следующие значения: *none* - у текста нет оформления, *underline* - текст подчеркивается, *overline* - текст надчеркивается линией, расположенной над текстом, *line-through* - текст отображается зачеркнутым, *blink* - текст становится мигающим (не работает в IE).

*text-align* отвечает за горизонтальное выравнивание текста. Может принимать следующие значения: *left* - выравнивание по левому краю, *center* - выравнивание по центру, *right* - выравнивание по правому краю, *justify* - выравнивание по ширине.

*text-indent* отвечает за отступы в абзацах. Отступы задаются в единицах измерения и %, могут быть как положительными (красная строка), так и отрицательными (висячая строка).

*text-transform* отвечает за видоизменение текста, точнее за смену регистра. Используются следующие значения: *capitalize* - меняет первую букву каждого слова на заглавную, *uppercase* - меняет все буквы на заглавные, *lowercase* - меняет все буквы на строчные, *none* - изменений не происходит.

В CSS есть несколько свойств, регулирующих расстояния между словами, буквами в словах и строками: *word-spacing* - задает интервал между словами, *letter-spacing* - задает интервал между буквами, *line-height* - задает интервал между строками.

*body { background: #0033CC; color: #FFFFFF; font: 12px Verdana, sans-serif; }*

*h1 { font-size: 1.6em; color: #FFCC00; letter-spacing: 5px; text-align: center; }*

*h2 { font-size: 1.3em; color: #FFCC00; text-decoration: underline; text-align: center; }*

*p:first-letter { color: #FFCC00; }*

*p { text-indent: 1.5em; word-spacing: 10px; font-style: italic; }*

**Блоки**

В HTML все элементы можно разделить на два типа: блочные и строчные.

Блочные элементы визуально создают самостоятельную структурную единицу - блок. К ним можно отнести, например, элементы H1-H6, P, DIV. Такие элементы отделяются от других абзацными отступами.

Строчные элементы выводятся линейной строкой. К ним можно отнести, например, элементы I, B, U, S и другие. В CSS модель документа представляется блоком. Каждый элемент в дереве элементов документа представляет собой самостоятельный блок. Причем, есть блоки, структурно отделенные от остальных, а есть строчные блоки, которые могут находиться внутри структурных блоков.

У блока есть *содержимое*, например, для элемента *p* - это текст. Вокруг содержимого есть *отступы (padding)*, они служат для того, чтобы текст не примыкал вплотную к границе блока. Фон отступов такой же, как и у содержимого. Затем идет *граница блока (border)*, которая может быть как видимой, так и невидимой. Также блок имеет *поля (margin)*, которые задают дополнительное свободное пространство вокруг блока. Фон полей прозрачный, т.е. сквозь него просвечивает фон родительского элемента. Размер блока, т.е. его *ширина (width)* и *высота (height)*, определяются содержимым. И это надо запомнить: поля и отступы не учитываются в размере блока.

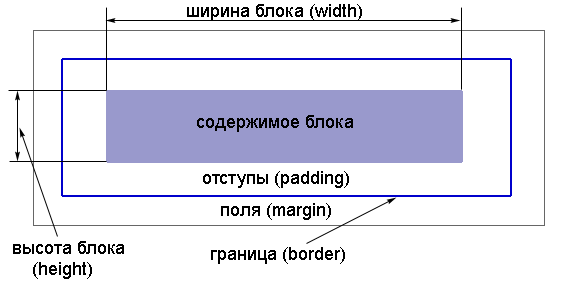


Рисунок 9 –Блок.

**Свойства – border**

*div* является контейнером для остальных. Элемент *div* отделяется от остальных элементов абзацными отступами, содержимое тегов div располагается с абзацным отступом, т.е. одно под другим. Элемент *span*, в отличии от *div*, создает строчный блок.

Границы в css можно задавать отдельно для каждой стороны: *border-top* - верхняя граница, *border-right* - правая граница, *border-bottom* - нижняя граница, *border-left* - левая граница.

Каждый сегмент границы может иметь свои характеристики: цвет, толщину и тип линии. Для этого к свойству границы через дефис необходимо дописать ключевые слова: color (для цвета), width (для толщины) и style (для типа линии). Например, *border-top-color* определяет цвет верхней границы, а *border-left-style* - тип линии для левой границы. Если все четыре границы будут иметь одинаковые значения, то следует воспользоваться сокращенной записью: *border-color* - цвет всех границ, *border-width* - толщина всех границ, *border-style* - стиль всех границ.

Значениями свойства *color* могут быть именные цвета (red, blue и т.д.), шестнадцатеричные коды цветов (типа #FFCCFF) и десятичные коды в модели RGB (типа rgb(255, 0, 0)).

Значениями свойства *width* могут быть ключевые слова: thin (тонкая граница), medium (средняя граница) и thick (толстая граница). А также любая единица измерения.

Значениями свойства *style* могут быть следующие ключевые слова: *none* - граница отсутствует, *dotted* - граница состоит из точек, *dashed* - граница в виде пунктирной линии, *solid* - граница отображается сплошной линией, *double* - граница отображается двойной сплошной линией, *groove* - граница отображается вдавленной объемной линией, *ridge* - граница отображается выпуклой объемной линией, *inset* - граница отображается так, что весь блок выглядит вдавленным, *outset* - граница отображается так, что весь блок выглядит выпуклым.

*#first { border-bottom-color: red; border-top-style: solid; border-left-width: 2px; }*

*#second { border: 1px solid red; }*

**Свойства – margin, padding**

Поля задают свободное пространство вокруг элемента. Как и границы, поля в css можно определять отдельно для верхней, правой, нижней и левой сторон. Для этого используются свойства: *margin-top* - ширина верхнего поля, *margin-right* - ширина правого поля, *margin-bottom* - ширина нижнего поля, *margin-left* - ширина левого поля.

Чаще используется сокращенная запись *margin*, где через пробел указываются ширина верхнего, правого, нижнего и левого полей. Причем, именно в таком порядке.

Отступы позволяют отделить содержимое блока от границы. Параметры отступов в css можно задать для каждой стороны отдельно, используя следующие свойства: *padding-top* - ширина верхнего отступа, *padding-right* - ширина правого отступа, *padding-bottom* - ширина нижнего отступа, *padding-left* - ширина левого отступа.

Значения свойств могут задаваться в различных единицах длины или в процентах. Проценты вычисляются относительно ширины блока. В качестве значения может выступать только положительная величина.

*#third { border: 1px solid red; margin: 10px; padding: 10px 5px; }*

**Позиционирование блоков**

В HTML для позиционирования элементов на странице используются таблицы. У таблиц есть как преимущества (легкость использования, одинаковое отображение браузерами), так и недостатки (объемный, нечитабельный код, нелогичность верстки и т.д.).

В CSS для позиционирования элементов используются блоки (div-ы). Код при этом становится компактным, логичным и легко изменяемым. К недостаткам блочной верстки можно отнести неодинаковую поддержку браузерами, поэтому приходится писать кроссбраузерный код (т.е. код, который отображается разными браузерами почти одинаково).

*Позиционированием в нормальном потоке* означает, что все элементы отображаются в окне браузера сверху вниз, по вертикали, в том порядке, в каком они следуют друг за другом в html-коде. По своей сути нормальный поток ничем не отличается от позиционирования элементов в HTML. И для верстки такой страницы без CSS, нам пришлось бы использовать таблицу, за неимением других вариантов.

В CSS же предоставляются и другие схемы позиционирования: абсолютное позиционирование, относительное позиционирование, плавающая блоковая модель.

Для определения схемы позиционирования используется свойство *position*, оно может принимать четыре значения, соответствующие выбранной схеме позиционирования:

*static* - блок позиционируется в нормальном потоке. Это значение по умолчанию.

*relative* - относительное позиционирование (относительно нормального потока).

*absolute* - абсолютное позиционирование.

*fixed* - фиксированное позиционирование (фиксируется относительно области просмотра).

**Абсолютное позиционирование**

При этой схеме позиционирования расположение блока на странице не зависит от того, в каком месте html-кода расположен этот блок. Расположение каждого блока задается указанием, в каком месте экрана отобразить данный блок. Для этого существуют четыре свойства:

*left* – указывает, на сколько надо сместить блок относительно левого края окна.

*right* – указывает, на сколько надо сместить блок относительно правого края окна.

*top* – указывает, на сколько надо сместить блок относительно верхнего края окна.

*bottom* – указывает, на сколько надо сместить блок относительно нижнего края окна.

Для блока *content* укажем свойство *position:absolute* и зададим смещение: от левого края окна на 200 пикселов, а от верхнего края окна на 100 пикселов.

*#content {*

*background: oldlace;*

*width: 500px;*

*height: 300px;*

*position: absolute;*

*left: 200px;*

*top:100px;*

*}*

**Относительное позиционирование**

При относительном позиционировании блока надо задать свойство *position:relative* и свойства смещения. Смещение в этом случае будет происходить не относительно "родительского" элемента (как при абсолютном позиционировании), а относительно самого блока в нормальном потоке.

*Плавающие блоки* нельзя позиционировать с точностью до пиксела, как в предыдущих схемах, но именно эта схема позиционирования очень распространенна. Без плавающих блоков обходится редкий сайт, а уж сделать "резиновую" верстку сайта без них и вовсе невозможно. Такие блоки могут свободно перемещаться по странице, подобным образом ведут себя картинки в HTML, выровненные с помощью параметра *align*.   
Плавающие блоки определяются свойством *float*, которое определяет, будет ли блок плавающим и в какую сторону он будет перемещаться. Возможны три варианта: *left* - блок прижимается к левому краю, остальные элементы обтекают его с правой стороны, *right* - блок прижимается к правому краю, остальные элементы обтекают его с левой стороны, *none* - блок не перемещается и позиционируется согласно свойству *position*.

Иногда нужно, чтобы блоки были прижаты к правому краю, но располагались один под другим. Для этого существует свойство *clear*, которое определяет, какие стороны плавающего блока не могут соседствовать с другими плавающими блоками. У этого свойства может быть задано одно из четырех значений: *left* - блок должен располагаться ниже всех левосторонних блоков, *right* - блок должен располагаться ниже всех правосторонних блоков, *both* - блок должен располагаться ниже всех плавающих блоков, *none* - никаких ограничений нет, это значение по умолчанию.

*#blok1 { border:1px solid red; width: 150px; height: 50px; float: right; }*

*#blok2 { border:1px solid red; width: 150px; height: 50px; float: right; clear: right; }*

**Виды и свойства блоков**

Элементы могут быть блочными и строчными. По умолчанию для каждого элемента его вид определен, так элементы *div* и *p* являются блочными, а *span* и *a* - строчными. Но иногда это необходимо изменить, для этого используется свойство *display*. Это свойство может принимать одно из четырех значений: *display:block* делает элемент блочным, *display:inline* делает элемент строчным, *display:list-item* делает любой элемент частью списка, *display:none* убирает элемент со страницы.

В CSS есть еще одно свойство на первый взгляд похожее на *display:none*. Это свойство, отвечающее за видимость блока - *visibility*. Оно может принимать два значения: *visible* (отображать) и *hidden* (сделать невидимым). Различия здесь следующее: *display:none* скрывает элемент, как будто его и не было, а *visibility:hidden* делает элемент прозрачным.

Свойство *overflow* может принимать четыре значения: *visible* - если содержимое превышает размеры блока, оно все равно останется на экране, *hidden* - браузер отрежет содержимое, которое превышает размер блока, *scroll* - блок будет снабжен полосами прокрутки, причем как горизонтальной, так и вертикальной, *auto* - блок будет снабжен только теми полосами прокрутки, которые необходимы.

**Слои**

В CSS можно создать несколько слоев, на каждом разместить необходимые элементы и пронумеровать слои с помощью свойства *z-index*. Чем больше номер, тем выше слой находится в стопке слоев. Например, если сделать 6 слоев, то пользователь сначала увидит именно слой *z-index:6*.

Слои часто используются при создании раскрывающихся меню.

**Списки**

Как всегда CSS предоставляет большие возможности при создании списков, чем только HTML. Собственно свойств для списков существует всего три: *list-style-type* - определяет внешний вид маркера или нумератора, *list-style-image* - определяет пользовательское изображение маркера, *list-style-position* - определяет положение маркеров относительно блока.

Для маркированных списков *list-style-type не содержит* ничего нового, все теже значения, что и в HTML: *disk* - закрашенный кружок, *circle* - незакрашенный кружок, *square* - закрашенный квадрат. Для нумерованных списков перечень более широк, чем в HTML, но не все значения поддерживаются всеми браузерами. Поэтому пока рекомендуется использовать только следующие: *decimal* - десятичные числа, *lower-roman* - строчные римские цифры, *upper-roman* - прописные римские цифры, *lower-alpha* - строчные латинские буквы.

Свойство *list-style-image* позволяет задать свой вид маркера. Для этого надо создать картинку с маркером и указать ее адрес в стиле.

Свойство *list-style-position* определяет положение маркера: внутри блока - *inside* или снаружи – *outside*.

*#spisok1 { list-style: circle url(marker.gif) inside; }*

**Литература**

1. Хорстманн Кей С., Корнелл Гари. Java`2. Библиотека HTMпрофессионала, том II. Тонкости программирования. 7-е издание.: Пер. с англ. – М.: Издательский дом «Вильямс», 2008. – 1168 с.: ил. ISBN 978-5-8459-1033-2 (рус.).
2. [http://docs.oracle.com/cd/B10501\_01/appdev.920/a96621/adx04paj.htm#1003830](http://docs.oracle.com/cd/B10501_01/appdev.920/a96621/adx04paj.htm)
3. <http://www.quizful.net/post/sax-parser-java>
4. <http://en.wikipedia.org/wiki/Object-relational_mapping>
5. <http://www.hibernate.org/hib_docs/reference/ru/html_single/>
6. <http://java-course.ru/student/book2/hibernate-annotation/>
7. Дейтел, Х.М. Технологии программирования на Java 2. Книга 3. Корпоративные системы, сервлеты, JSP, Web-сервисы [Текст] / Х.М. Дейтел, П.Дж. Дейтел, С.И. Сантари. – М. : Бином-пресс, 2003. – 672 с.
8. Перри, Б.У. Java сервлеты и JSP. Сборник рецептов [Текст] / Брюс У. Перри. – М. : Кудиц-пресс, 2009. – 768 с.
9. Соломон, М.К. Oracle. Программирование на языке Java [Текст] / Мартин К. Соломон, Нирва Мориссо-Леруа, Джули Басу. – М. : Лори, 2010. – 512 с.
10. Курванян, Б. Программирование web-приложений на языке Java [Текст] / Буди Курванян. – М. : Лори, 2009. – 880 с.
11. JavaEEDocumentation [Электронный ресурс]. – Режим доступа: <http://www.oracle.com/technetwork/java/javaee/documentation/index.html>, дата доступа: 13.12.2013.
12. JavaEETutorials [Электронный ресурс]. – Режим доступа: <http://www.oracle.com/technetwork/java/javaee/documentation/tutorials-137605.html>, дата доступа: 13.12.2013.
13. <http://orlova.rsue.ru/content/sst/j2ee1.htm>
14. https://ru.wikipedia.org/wiki/HTTP\_cookie
15. http://learn.javascript.ru/#book-toc-137