Министерство образования Республики Беларусь

Учреждение образования
«Гомельский государственный технический
университет имени П. О. Сухого»

Кафедра «Информационные технологии»

Л. К. Титова

СОВРЕМЕННЫЕ ТЕХНОЛОГИИ FRONT-END РАЗРАБОТКИ

УЧЕБНО-МЕТОДИЧЕСКОЕ ПОСОБИЕ
для студентов специальности
1-40 05 01 «Информационные системы и технологии
(по направлениям)»
дневной и заочной форм обучения

Гомель 2023

УДК 004.4(075.8)
ББК 32.973я73
Т45

Рекомендовано научно-методическим советом
факультета автоматизированных и информационных систем ГГТУ им. П. О. Сухого ГГТУ
(*протокол № 5 от 09.06.2022 г.*)

Рецензент: доц. каф. «Промышленная электроника» ГГТУ им. П. О. Сухого
канд. техн. наук, доц. *А. В. Ковалев*

Титова, Л. К.

Т45 Современные технологии Front-end разработки : учеб.-метод. пособие для студентов специальности 1-40 05 01 «Информационные системы и технологии (по направлениям)» днев. и заоч. форм обучения / Л. К. Титова. – Гомель : ГГТУ им. П. О. Сухого, 2023. – 193 с. – Систем. требования: PC не ниже Intel Celeron 300 МГц ; 32 Mb RAM ; свободное место на HDD 16 Mb ; Windows 98 и выше ; Adobe Acrobat Reader. – Режим доступа: https://elib.gstu.by. – Загл. с титул. экрана.

Рассмотрена архитектура современных приложений корпоративных информационных систем, а также как средства разработки Front-end приложений, так и технологические аспекты проектирования подобных систем, ориентированных на использование в глобальной сети Интернет.

Для студентов специальности 1-40 05 01 «Информационные системы и технологии (по направлениям)» дневной и заочной форм обучения.

УДК 004.4(075.8)
ББК 32.973я73

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ

Современное сайтостроение развивается небывалыми темпами, предлагая для улучшения качества отображения и работы ресурса самые различные технологии, методы и способы.

Верстка сайта выступает одним из этапов разработки веб-сайта и представляет собой «оживление» дизайн-макета сайта, то есть написание программного кода сайта в соответствии с заранее подготовленным шаблоном, продуманной структурой и функционалом, позволяющее страницам открываться и работать в браузерах и на различных устройствах.

Front-end разработка позволяет смотреть еще глубже, обеспечивая точную настройку и оптимизацию всех процессов, работу всех элементов интерфейса.

Frontend – это публичная часть *web*-приложений (вебсайтов), с которой пользователь может взаимодействовать и контактировать напрямую. Во *Frontend* входит отображение функциональных задач, пользовательского интерфейса, выполняемые на стороне клиента, а также обработка пользовательских запросов. По сути, фронтенд – это всё то, что видит пользователь при открытии *web*-страницы.

В свою очередь, *web*-приложение – клиент-серверное приложение, в котором клиентом выступает в основном браузер, а сервером – *web*-сервер. Логика *web*-приложения распределена между сервером и клиентом, хранение данных осуществляется преимущественно на сервере, обмен информацией происходит по сети. Проще говоря, это то, что видит пользователь и какие действия выполняет каждый раз, когда подключается к сети интернет и открывает любой браузер.

Frontend-разработка – это работа по созданию публичной части *web*-приложения, с которой непосредственно контактирует пользователь, и функционала, который обычно выполняется на стороне клиента. То есть, фронтенд разработчик работает над тем, чтобы на сайте каждая кнопочка, иконка, текст и окно не только стояли на своем месте, не перекрывали друг друга и выглядели целостно (это веб-верстка), но и чтобы они выполняли свое прямое предназначение – производили какие-то действие (например, чтобы кнопка «купить» открывала корзину, а «*play*» – запускала воспроизведение игры, фильма или музыки).

С целью создания востребованного и доступного продукта (веб-приложения) фронтенд-разработчику необходимо взаимодействовать с другими программистами, дизайнерами, маркетологами, аналитиками и прочими специалистами.

Компоненты фронтенд разработки:

– ***HTML*** (***HyperText Markup Language***) – это язык разметки элементов и документов на странице, и их взаимодействие в структуре страницы.

– ***CSS*** (***Cascading Style Sheets***) – это язык стилизации внешнего вида документа. С помощью *CSS*-кода браузер понимает, как именно необходимо отображать элементы. *CSS* используется создателями веб-страниц для задания цветов, шрифтов, расположения отдельных блоков и других аспектов представления внешнего вида этих веб-страниц. Основной целью разработки *CSS* является разделение описания логической структуры веб-страницы (которое производится с помощью *HTML* или других языков разметки) от описания внешнего вида этой веб-страницы (которое производится с помощью формального языка *CSS*).

– ***JavaScript*** – язык, созданный оживить веб-страницы. Задача *JavaScript* – откликаться на действия пользователя, обрабатывать нажатия клавиш, перемещения курсора, клики мышкой. *JavaScript* также дает возможность вводить сообщения, посылать запросы на сервер, а также загружает данные без перезагрузки страницы, и так далее.

Вся фронтенд разработка выполняется на стороне пользователя, она не менее важна чем бекенд разработка, так как это то, что пользователь видит и с чем взаимодействует.

Основная задача фронтенд специалиста – это связать представленные дизайнером графические макеты *Web*-приложения с бэкендом и при необходимости реализовать вычислительный функционал на стороне пользователя.

Конечно, при работе в команде нужно знать и разбираться во многих процессах, смежных с работой фронтенд-разработки. Будучи уже опытным фронтенд-разработчиком необходимо быть знакомым с бэкенд-технологиями и понимать принципы взаимодействия пользователя и с приложениями (*UX*).

В последнее время вакансия фронтенд-разработчика довольно востребована и актуальна. Опытный фронтенд разработчик не просто «верстает макеты», он отлично знает *JavaScript*, ориентируется в

фреймворках и библиотеках, имеет представление и понимание того, что размещается на серверной стороне, и нередко знает дополнительные языки, например *PHP* или *C#*. Фронтенд-разработчик смыслит в препроцессорах и сборщиках *GULP*, *LESS*, *SASS*, *GRUNT*, работает с *SVG*-объектами, *DOM*, *API*, *AJAX* и *CORS* и так далее.

Кроме всего прочего присутствует понимание принципов адаптивной и отзывчивой верстки, *UI*/*UX*-проектирования, кросс-браузерности и кросс-платформенности, базового тестирования, а также знание навыков мобильной разработки.

Продвинутый фронтенд разработчик также умеет использовать графические редакторы, работает с контролем версий *Git*, *GitHub*, *CVS*, с шаблонами различных *CMS*. Стоит отметить, что очень важно знание английского языка на уровне свободного общения с заказчиками и чтения документации.

Чтобы стать востребованным и опытным специалистом в области Фронтенд разработки, необходимо освоить следующие технологии:

- *HTML* и *CSS* (в том числе и *CSS*-фреймворки, спецификации *W3C* и *WHATWG*, *HTML5*/*CSS3 Polyfills*);
- свободно работать с *JavaScript*;
- понимать логику работы клиент-серверной архитектуры в контексте написания реальных приложений;
- знать популярные библиотеки и феймворки: *React.js*, *jQuery*, *Angular.JS*, *Redux*, *js*,
- понимать принципы построения современных одностраничных приложений (*Single Page Application*);
- препроцессоры *CSS* (*Sass*, *Less*, *Stylus* и т. д.);
- популярные *CMS* (например, *WordPress*, *Drupal*, *Joomla* и т.д.);
- *ECMAScript* 6;
- *HTML5 API*;
- *SVG*;
- *DOM*;
- понимать принципы построения *backend* (*Node.js*, *PHP*, *Ruby*, *.NET* и т. д.);
- *JavaScript* транспайлеры (*Babel*);
- инструменты дебаггинга (*Chrome Dev Tools*, *Firebug* и так далее);

- графические редакторы (*Photoshop*, *Illustrator* и прочее);
- инструменты контроля версий (*Git*, *GitHub*, *CVS* и так далее);
- базы данных и языки запросов (*SQL*, *MySql*, *NoSQL*, *MongoDB* и так далее).

Также фронтенд разработчик должен разбираться и уметь разрабатывать веб-интерфейсы и веб-приложения, тестировать и масштабировать веб-приложения, читать чужой код с пониманием того, как он работает, уметь презентовать себя и свой продукт.

1 РАЗЛИЧИЯ И ОСОБЕННОСТИ *FRONTEND* И *BACKEND*-РАЗРАБОТКИ

Frontend является клиентской стороной пользовательского интерфейса к программно-аппаратной части проекта, то есть к бекенду. *Backend* – это программно-аппаратная часть проекта, Другими словами бекенд – это все то, что происходит на стороне сервера и что остается невидимым пользователю (сам сервер тоже является частью бэкенда, только аппаратной). Отсюда и название *front* – это видимое спереди, *back* – это то, что скрыто сзади, невидимое. К примеру, вы оплачиваете покупку в интернете: вводите данные карты, кликаете «оплатить» и видите надпись «ваш платеж принят в обработку» – это был фронтенд. То, как двигаются ваши деньги внутри сети и то, как ваш заказ поступает в магазин – это бекенд. Соответственно, когда магазин видит уведомление о том, что поступил заказ, а деньги зачислились на счет – это снова работа фронтенда.

Бекенд-разработчики имеют дело с серверными языками программирования, такими как *Java*, *Python*, *PHP*, *Ruby* и другие. Также бэкенд-разработчики должны знать базы данных, архитектуру, ко всему прочему им пригодятся знания аппаратной части бэкенда, то есть сервера, его возможности и характеристики. Бекенд-разработчики, как правило, не имеют дела ни с чем, что напрямую взаимодействует с пользователем, они не разбираются в пользовательских интерфейсах *UI* и не углубляются в пользовательский опыт взаимодействия *UX* или в верстку страницы, хотя общее понимание всего этого имеют. Они работают, в основном, с точным анализом и вычислениями, где почти нет творческой, гуманитарной составляющей. При этом, им нужно уметь вычислять

все возможные исходы операций и понимать причины ошибок, появившихся на пути клиент-сервер-клиент.

Рассмотрим процесс взаимодействия *frontend* и *backend*:

- фронтенд отправляет пользовательскую информацию в бэкенд;
- информация обрабатывается;
- информация возвращается обратно, приняв целостную форму и выполнив обработанный запрос.

Все эти задачи выполняет несколько специалистов одновременно, это всегда взаимодополняющая командная работа.

Варианты взаимодействия *frontend* и *backend*:

- *HTTP*-запрос отправляется на сервер, сервер в процессе поиска информации, встраивает ее в шаблон и возвращает обратно в виде *HTML*-страницы.
- Случай с применением инструментария *AJAX* (*Asynchronous JavaScript and XML*). В данном случае запрос отправляет *JavaScript*, который загружен в браузер, ответ же приходит в формате *XML* или *JSON*.
- Одностраничные приложения, которые загружают данные без обновления страниц. Это делается с помощью *AJAX* или фреймворков *Angular* и *Ember*. *Ember* или библиотека *React* оказывают помощь в использовании приложения и в клиентской части и на сервере. *Frontend* и *backend* взаимодействуют через *AJAX* и *HTML*-код, который обрабатывается на сервере.

Работа и обязанности *frontend* и *backend* разработчиков чаще всего разделены, но иногда возникает необходимость у программиста решать проблемы как на стороне сервера, так и в клиентской части. Довольно часто можно встретить специалистов, которые могут совмещать *frontend* и *backend*, они абсолютно уверенно себя чувствуют как с одной так и с другой стороны медали.

Чаще всего карьера *frontend*-разработчика начинается с верстальщика и тестировщика. В процессе работы изучается *HTML+CSS*, далее по нарастающей приобретаются знания *JavaScript*, библиотек и фреймворков. Затем изучаются основные понятия построения серверной части, параллельно добавляя инструменты, согласно выбранной специализации. Дополнительно нарабатываются навыки работы с графическими редакторами и пониманием принципов *UI*/*UX* дизайна.

2 АРХИТЕКТУРА ВЕБ-ПРИЛОЖЕНИЯ: КОМПОНЕНТЫ, СЛОИ И МОДЕЛИ

Архитектура веб-приложения в основном представляет отношения и взаимодействия между такими компонентами, как пользовательские интерфейсы, мониторы обработки транзакций, базы данных и другие. Основная цель – убедиться, что все элементы правильно работают вместе.

Логика довольно проста: когда пользователь вводит URL-адрес в браузере и нажимает «ввод», браузер делает запрос к серверу. Сервер отвечает, а затем показывает требуемую веб-страницу. Все эти компоненты создают архитектуру веб-приложения.

Все приложения состоят из двух частей - клиентской (front-end) и серверной (back-end).

Интерфейс – это визуальная часть приложения. Пользователи могут видеть интерфейс и взаимодействовать с ним. Клиентский код реагирует на действия пользователей. Серверная часть не визуальна для пользователей, но заставляет их запросы работать. Он обрабатывает бизнес-логику и отвечает на HTTP-запросы. Поэтому, когда вы вводите свои учетные данные в регистрационную форму, то имеете дело с внешним интерфейсом, но как только нажимаете «ввод» и регистрируетесь – это серверная часть заставляет его работать.

При правильной работе клиентская и серверная стороны составляют архитектуру программного обеспечения веб-приложения.

Веб-приложения разделяют свои основные функции на уровни. Это позволяет заменять или обновлять каждый слой независимо.

Веб-архитектура имеет компоненты пользовательского интерфейса и структурные компоненты. Последние также делятся на клиентские и серверные.

Компоненты пользовательского интерфейса обозначают все элементы интерфейса, такие как журналы активности, информационные панели, уведомления, настройки и многое другое. Они являются частью макета интерфейса веб-приложения.

Структурные компоненты состоят из клиентской и серверной сторон. Существует ***четыре общих уровня веб-приложений***:

- уровень представления (*PL*);
- уровень обслуживания данных (*DSL*);
- уровень бизнес-логики (*BLL*);
- уровень доступа к данным (*DAL*).

Уровень представления (*PL*) отображает пользовательский интерфейс и упрощает взаимодействие с пользователем. Уровень представления имеет компоненты пользовательского интерфейса, которые визуализируют и показывают данные для пользователей. Также существуют компоненты пользовательского процесса, которые задают взаимодействие с пользователем. *PL* предоставляет всю необходимую информацию клиентской стороне. Основная цель уровня представления – получить входные данные, обработать запросы пользователей, отправить их в службу данных и показать результаты.

Слой бизнес-логики *BLL* несет ответственность за надлежащий обмен данными. Этот уровень определяет логику бизнес-операций и правил. Вход на сайт – это пример уровня бизнес-логики.

Уровень службы данных *DSL* передает данные, обработанные уровнем бизнес-логики, на уровень представления. Этот уровень гарантирует безопасность данных, изолируя бизнес-логику со стороны клиента.

Уровень доступа к данным *DAL* предлагает упрощенный доступ к данным, хранящимся в постоянных хранилищах, таких как двоичные файлы и файлы *XML*. Уровень доступа к данным также управляет операциями *CRUD* – создание, чтение, обновление, удаление.

Типы архитектуры веб-приложений

В зависимости от того, как логика приложения распределяется между клиентской и серверной сторонами, можно выделить несколько типов архитектуры веб-приложений. Наиболее распространенные архитектуры веб-приложений:

- одностраничные веб-приложения;
- многостраничные веб-приложения;
- архитектура микросервисов;
- бессерверная архитектура;
- прогрессивные веб-приложения.

Разберемся в деталях каждого вида.

Одностраничное приложение или *SPA* – это веб-сайт или веб-приложение, которое загружает всю необходимую информацию при входе на страницу. Одностраничные приложения имеют одно существенное преимущество – они обеспечивают потрясающий пользовательский интерфейс, поскольку пользователи не испытывают перезагрузки веб-страниц. Одностраничные веб-приложения часто разрабатываются с использованием фреймворков *JavaScript*, таких как *Angular*, *React* и других.

Известные СПА : *Gmail*, *Facebook*, *Twitter*, *Slack*.

Многостраничное приложение или *MPA* приложения более популярны в Интернете, так как в прошлом все веб-сайты были *MPA*. В наши дни компании выбирают *MPA*, если их веб-сайт довольно большой (например, *eBay*). Такие решения перезагружают веб-страницу для загрузки или отправки информации с / на сервер через браузеры пользователей.

Известные *MPA*: *eBay*, *Amazon*.

Одностраничное приложение или многостраничное приложение? У многостраничного и одностраничного приложения есть недостатки и преимущества.

Чтобы понять архитектуру микросервисов, лучше сравнить ее с монолитной моделью.

Традиционная монолитная архитектура веб-приложения состоит из трех частей – базы данных, клиентской и серверной сторон. Это означает, что внутренняя и внешняя логика, как и другие фоновые задачи, генерируются в одной кодовой базе. Чтобы изменить или обновить компонент приложения, разработчики программного обеспечения должны переписать все приложение.

Что касается микросервисов, этот подход позволяет разработчикам создавать веб-приложение из набора небольших сервисов. Разработчики создают и развертывают каждый компонент отдельно.

Архитектура микросервисов выгодна для больших и сложных проектов, поскольку каждый сервис может быть изменен без ущерба для других блоков. Поэтому, если необходимо обновить логику оплаты, не придется на время останавливать работу сайта.

Известные проекты: *Netflix*, *Uber*, *Spotify*, *PayPal*.

Типы архитектуры веб-приложений:

- монолитные и микросервисы;
- бессерверная архитектура.

Этот тип архитектуры веб-приложений заставляет разработчиков использовать облачную инфраструктуру сторонних поставщиков услуг, таких как *Amazon* и *Microsoft*.

Это означает, чтобы сохранить веб-приложение в Интернете, разработчики должны управлять серверной инфраструктурой (виртуальной или физической), операционной системой и другими процессами хостинга, связанными с сервером. Поставщики облачных услуг, такие как *Amazon* или *Microsoft*, предлагают виртуальные серверы, которые динамически управляют распределением машинных ресурсов. Другими словами, если ваше приложение испытывает огромный всплеск трафика, к которому ваши серверы не готовы, приложение не будет отключено.

Одна из основных тенденций в разработке веб-приложений последних лет – это *прогрессивные веб-приложения или PWA*. Это веб-решения, которые работают как собственные приложения на мобильных устройствах. *PWA* предлагают *push*-уведомления, автономный доступ и возможность установить приложение на домашний экран.

Для создания *PWA* разработчики используют «языки веб-программирования», такие как *HTML*, *CSS* и *JavaScript*. Если приложению требуется доступ к функциям устройств, разработчики используют дополнительные *API – NFC API*, *API* геолокации, *Bluetooth API* и другие.

Известные *PWA*: *Uber*, *Starbucks*, *Pinterest*.

Как разработать архитектуру для веб-приложения

Качественная архитектура веб-приложения делает процесс разработки более эффективным и простым. Веб-приложение с продуманной архитектурой легче масштабировать, изменять, тестировать и отлаживать.

Есть несколько общих критериев для хорошо построенной архитектуры веб-приложения:

- эффективность;
- гибкость;
- расширяемость;
- соблюдение принципа открыто-закрыто;

- масштабируемость процесса разработки;
- легко проверить;
- возможность повторного использования;
- хорошо структурированный и читаемый код;
- нижняя граница.

3 СОВРЕМЕННЫЕ ПРИНЦИПЫ И ПОДХОДЫ К *FRONTEND* АРХИТЕКТУРЕ ВЕБ-ПРИЛОЖЕНИЙ

Рассмотрим приложение как систему – то есть набор компонентов, объединенных для выполнения определенной функции.

Архитектура идентифицирует главные компоненты системы и способы их организации и взаимодействия. Также немаловажной характеристикой любой архитектуры является выбор таких методов, которые будут неизменны в будущем и составлять базис системы.

Архитектура – это организация системы, воплощенная в её компонентах, их отношениях между собой и с окружением.

Чтобы быстро понять, какое архитектурное решение является верным, необходимо задать себе вопрос: «Насколько сложно будет в будущем поменять данную архитектуру на другую?». Если ответ на этот вопрос положительный, то, скорее всего, необходимо будет выбрать другую архитектуру.

Также можно выделить несколько критериев хорошей архитектуры:

- гибкость и масштабируемость системы;
- тестируемость и сопровождаемость;
- независимость от бизнес-логики и предметной области;
- эффективность работы.

Рассмотрим понятие архитектуры программного обеспечения (ПО) в рамках *Frontend* части – клиентской части любого клиент-серверного ПО.

Структуру *Frontend*, как и структуру почти любого ПО, условно можно поделить на какое-то определенное количество крупных блоков. Каждый из этих блоков имеет свою реализацию, которая позволяет связываться с другими блоками. Внутри этих блоков можно выделить модули, внутри которых есть методы. И каждый узел этой цепочки вложенности имеет какой-то свой контракт общения, так называемые входные и выходные параметры. Можно сказать, что у

каждого звена есть своя архитектура. И поэтому архитектура, в принципе, обладает свойством иерархичности.

В настоящее время *Web*-приложения больше не привязаны к ПК (персональным компьютерам). В мире появляется все больше и больше гаджетов, мобильных телефонов, планшетов. Стоит отметить, что больше всего интернет трафика проходит через мобильные приложения, а не через ПК.

JavaScript, мощнейший язык для веб-разработки, превратился в полноценный язык программирования и продолжает развиваться стремительными темпами.

Для веб-приложений стали доступны такие *API*(*Application Program Interface*) как:

- *File System*, *Camera*, аппаратные датчики и другие;
- *UX* (*User Experience*) становится решающим фактором при выборе продукта пользователями;
- стек *JavaScript* технологий стал использоваться для кроссплатформенной разработки (одно и то же веб-приложение может быть запущено что на *Android*, что на *iOS*, что и в браузере).

И современная архитектура *Frontend* приложений вынуждена реагировать на все эти изменения новыми технологиями, фреймворками и подходами. Стоит отметить, что, если раньше любые технологии очень часто упирались в ограниченные ресурсные возможности устройств, то сейчас, ввиду прорывных технологий в производстве микросхем, эта проблема исчезает, что также влияет на тенденции архитектуры веб-приложений.

Современная архитектура *Frontend* базируется на трех принципах.

1. Поток данных занимает центральное место в архитектуре.

Учитывая масштабы и сложность среды, в которой может выполняться *frontend* приложения, бизнес-логика кода играет очень важную роль. По статистике большая часть времени разработки обычно тратится на отладку и чтение кода, поэтому необходимым условием любой архитектуры является быстрота нахождения нужного нам кода, ответственного за бизнес логику. Всякий раз, когда требуется внести какую-то новую функциональность в систему, чрезвычайно важно понимать, как она повлияет на будущую работоспособность и эффективность ПО.

Единственный способ осуществления такого рода задачи – это понимание на каждом этапе механизма перемещения данных по

архитектуре ПО. Поэтому сегодня любой *Frontend* фреймворк построен на этом принципе, на четком разделении на модуль *State* (хранение и манипуляция с данными) и на модуль *View* (отображение данных).

К примеру, такой фреймворк как *Angular I*, использовал принцип двунаправленного потока данных, что создавало неконтролируемые процессы передачи данных. После появления паттерна проектирования *Flux*, который использовал более эффективный и надежный принцип однонаправленного потока данных, вышел *Angular II*, который использовал принцип однонаправленности, как *Flux*.

2. Компонентный подход

Компонентный подход является неизбежным следствием первого пункта о потоке данных. Можно выделить три главных аспекта компонентного подхода.

1) Особое внимание потокам данных. Традиционные архитектуры *frontend* приложений ориентированы на горизонтальное распределение функционала. Такой способ можно сравнить с «размазыванием функционала тонким слоем» по всей архитектуре (довольно привычный способ в *MV** паттернах). Но первый принцип о потоке данных требует, чтобы функциональность была ориентирована вертикально. Поэтому компонентой может являться какой-либо блок кода, инкапсулирующий какую-то одну «*pure responsibility*» логику (функцию с единственностью ответственности).

2) Постоянно усложняющие *View* (представления). В настоящее время, в связи с появлением *SPA*-приложений *View* становятся все сложнее и сложнее, что усложняет процесс внедрения новых *feature*. *MV** подобные архитектуры не обладают возможностью решить проблему, в то время как использование компонентного подхода позволяет упрощать *View* классы.

3) Структура компоненты. Любая компонента должна состоять из четырех элементов: графическая структура(*HTML-View*), стилизация(*CSS-View*), поведение(*JavaScript-Component*), бизнес логика(*JavaScript-Component*/*Model*).

Можно сказать, что компонентный подход позволяет переиспользовать и тестировать какие-то блоки веб-приложения, не ломая ее архитектуру. К примеру, блок «Авторизации», состоящий из формы ввода полей и различных действий авторизации («войти», «забыли пароль» и т.д.), может быть вставлен как внутри отдельной

страницы, так в внутри всплывающего диалогового окна, при этом не внося никакой лишней бизнес-логики в родительские элементы. В этом и заключается вся мощь компонентного подхода.

Резюмируя, приведем некоторые критерии компонент. Компоненты должны быть:

- независимые;
- слабосвязанные;
- переиспользуемые.

Компоненты могут быть достаточно сложные внутри, но обязаны быть простыми снаружи. Здесь подразумевается, что интерфейс любой компоненты должен быть настолько простым, чтобы процесс подключения компоненты к родителю проходил без побочных эффектов.

На рисунке 1 приведен пример разбиения типичного *View* на компоненты.

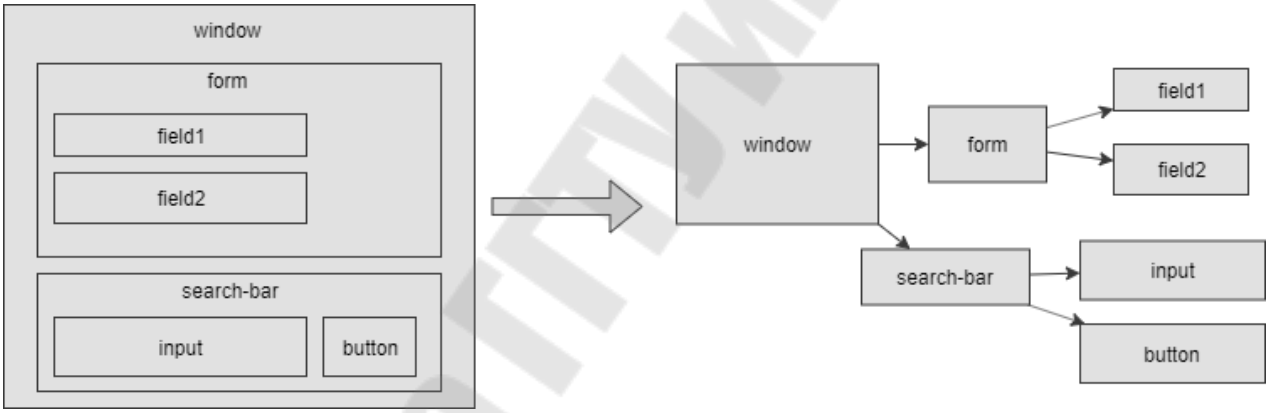


Рисунок 1 – Декомпозиция *View*

Компоненты делят на два типа:

- *Dummy*-компоненты – компоненты, которые либо вообще не содержат никакой логики (чисто визуальные компоненты), либо содержат логику, которая глубоко инкапсулирована внутри компонента. К примеру, компоненты «Форма», «Чекбокс» или «Аватарка»;
- *Smart*-компоненты – компоненты, которые управляют множеством других компонентов, содержат в себе бизнес-логику и хранят какое-то состояние. К примеру «Блок рекомендаций» или «Фильтр».

3. Автоматическое выборочное обновление DOM.

Все операции с *DOM* являются довольно дорогостоящими по ресурсам и времени, поэтому в какой-то момент *Frontend* сообщество пришло к идее создать технологию, которая смогла бы выполнять с *DOM* минимально возможное количество действий по перерисовке каких-то конкретных *DOM* элементов при изменении данных веб-приложения, так называемый *State*.

В итоге сообщество *Frontend* пришло к решению:

1) *Data-binding* между *Model* и *View*.

В совокупности с компонентным подходом использование дата-биндинга является идеальным решением проблемы взаимосвязи данных(*Model*) и представления(*View*). *View* можно представить как функцию модели, которая знает, какие поля данных нужны ей для отрисовки тех или иных компонент. К примеру, во фреймворке *Angular* существуют шаблоны, в *React JSX*–язык, являющийся смесью *HTML* и *JavaScript* языков.

2) Оповещение об изменении данных (*Change detection*).

Одним из требований также является возможность архитектурного решения каким-либо образом следить за тем, какие поля *State* были обновлены, удалены и добавлены. Данная возможность в *Angular I* реализовалась через метод грязной проверки (*dirty checking*), который проверял в бесконечном цикле через определенный период времени, какие поля были изменены. Во фреймворке *React* существуют неизменяемые структуры данных и посредством событийной шины происходит обновление необходимых полей *State*. В будущем в *Vue3* планируется использование объекта *Proxy* для реализации *data checking* (проверка данных на изменение).

3) Обновление *DOM*.

После того, как были обнаружены изменения *State*, необходимо выполнить перерисовку *DOM* на основе них. В *React* для этого используется технология *VirtualDOM*.

Таким образом, современные принципы построения архитектуры *Frontend*, как и в принципе любой другой области разработки ПО, основываются главным образом на компонентном и модульном подходе, который позволяет легко и быстро масштабировать любой продукт, занимаясь по большей части не проблемами реализации и внедрением в текущую архитектуру, а бизнес-требованиями к ПО. Также стоит отметить, что немаловажным критерием является скорость эффективности того или иного решения, вследствие чего и появились *data-binding* и *VirtualDOM*.

4 ПРЕИМУЩЕСТВА И НОВЫЕ ОСОБЕННОСТИ *HTML5*

HTML5 (***HyperText Markup Language, version 5***) – язык для структурирования и представления содержимого всемирной паутины. Другими словами, *HTML5* является мощным кроссплатформенным инструментом для создания веб-приложений высочайшего качества и функциональности, использующих аудио, видео, графику, анимацию и многое другое.

В совокупности с другими современными технологиями, такими как *CSS*3, *HTML5* становится лучшим средством для создания современных веб-приложений любой сложности. В распоряжение разработчика передается много новых инструментов для улучшения пользовательского интерфейса: от более содержательных тегов и улучшенных средств межсайтовых и межоконных коммуникаций до анимации и улучшенной мультимедийной поддержки.

В *HTML5* реализовано множество новых синтаксических особенностей. К основным возможностям, которые привнес с собой *HTML5* в разработку веб-приложений и интерфейсов, относятся:

- ***Новая улучшенная разметка документов.***

Благодаря новым элементам разметки создание документов на основе *HTML5* становится более быстрым и качественным, что приводит к меньшим затратам на создание макетов для веб-страниц и приложений. Повышается семантика страницы – поисковые системы автоматически распознают где на странице навигация, а где содержание.

- ***Рисование на странице.***

HTML5 определяет тег *canvas* как «холст для растровой графики, который может использоваться для отображения диаграмм, компьютерных игр или вывода других изображений на лету». Сам холст представляет собой прямоугольник на странице, в котором с помощью *JavaScript* рисуется, что вы пожелаете. *HTML5* определяет набор функций называемых «*Canvas API*» для рисования фигур, контуров, создания градиентов и трансформации.

- ***Новые элементы форм.***

Благодаря *HTML5* у разработчиков появились новые возможности, ранее доступные только при использовании сложных *JavaScript*

библиотек. Проверка правильности введенных данных происходит теперь «на лету» прямо в браузере, что упрощает заполнение формы.

- ***Поддержка аудио и видео потоков.***

Теперь с легкостью можно вставлять медиафайлы прямо в веб-страницу без использования «тяжелых» технологий, таких как *Adobe Flash* или *Microsoft Silverlight*. Более того, спецификации *HTML5* позволяют осуществлять непосредственный контроль над воспроизведением этих файлов, что может пригодиться, например, при синхронизации видео и субтитров к нему.

- ***Кроссплатформенная поддержка.***

Спецификации *HTML5* подходят для различных юзер-агентов, которыми могут быть не только компьютерные браузеры, но и различные портативные устройства. На *HTML5* создаются различные приложения для смартфонов, мобильных телефонов, домашних игровых консолей текущего поколения.

- ***Обработка ошибок.***

Документы могут не всегда содержать корректный синтаксис, но *HTML5*-совместимые браузеры, так же, как и их предшественники, применяют алгоритмы разбора ошибок разметки в документах для построения правильной объектной модели (*DOM*). Чёткое определение требований к юзер-агентам делается с целью достижения совместимости между браузерами разных производителей. Так же, как и требования к синтаксису разметки документов с целью корректного отображения их в различных браузерах.

- ***Геолокация.***

Некоторым веб-приложениям с разрешения пользователя могут быть переданы данные о его местонахождении. Существует несколько способов определить ваше положение — по *IP*-адресу, подключению к беспроводной сети, сотовому оператору или через *GPS* оборудование.

Кроме этих возможностей в *HTML5* включены новые теги для разметки документа, выброшены устаревшие теги и модифицированы некоторые другие.

Рассмотрим наиболее важные *HTML5*-тега и атрибуты, которые необходимо знать и использовать на практике.

4.1 Структурная разметка *HTML5*

В *HTML5* появилось несколько новых тегов, которые призваны заменить некоторые существующие блоки *div*. Внешне, так сказать, ничего не изменилось, но в сущности новые теги несут в себе смысловую (семантическую) нагрузку, и строго определяют для каждого блока его место и роль:

- ***<!DOCTYPE html>*** – данное объявление переводит все браузеры в нормальный режим. Браузеры, не поддерживающие *HTML5* в данном режиме, будут интерпретировать старые теги и игнорировать новые, которые они не поддерживают;
- ***<header>*** – определяет верхний колонтитул сайта;
- ***<footer>*** – определяет колонтитул сайта;
- ***<nav>*** – определяет навигационные функции страницы.
- ***<article>*** – создает автономный фрагмент.
- ***<section>*** – объединяет в группу различные статьи, или помечает различные разделы одной статьи.
- ***<time>*** – служит для разметки времени и даты.
- ***<aside>*** – используется для дополнительного блока контента, который связан с основным контентом.
- ***<hgroup>*** – создает оболочку, скрывающую дополнительные заголовки в структуре заголовков страницы, если требуется, чтобы учитывался только один заголовок.
- ***<figure>*** и ***<figcaption>*** – служит для создания рисунков и подписи для рисунка.
- ***<mark>*** – служит для выделения важных частей контента.

Пример общей структуры *html*-документа в *HTML4* и *HTML5* с использованием новых тэгов разметки приведен на рисунке 2.

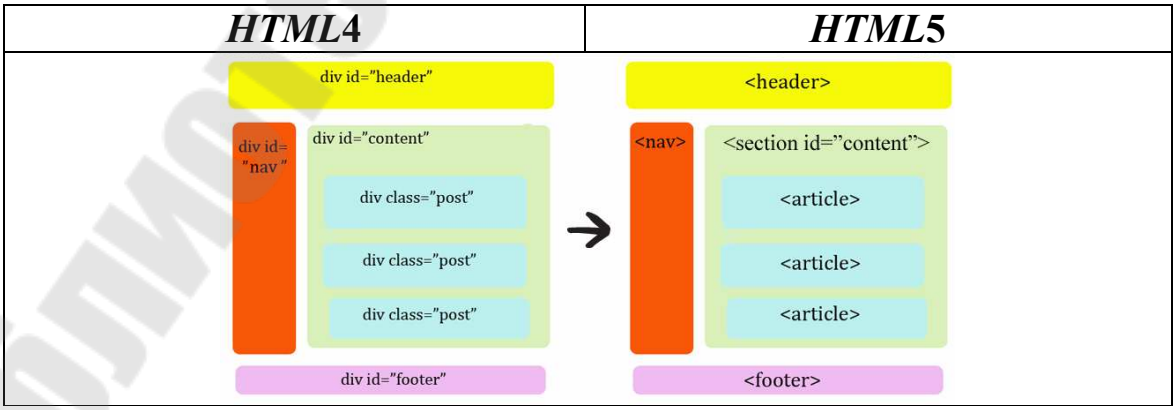


Рисунок 2 – Общая структура документа в ***HTML4*** и ***HTML5***

Пример семантическая вёрстка страницы:

```
<!DOCTYPE html>
<html>
   <head>
      <meta charset="utf-8">
      <title>
         (Это title) Пример страницы на HTML5
      </title>
   </head>
   <body>
      <header>
         <hgroup>
            <h1>
               Заголовок "h1" из hgroup
            </h1>
            <h2>
               Заголовок "h2" из hgroup
            </h2>
         </hgroup>
      </header>
      <nav>
         <menu>
            <li>
               <a href="link1.html">
                  Первая ссылка из блока "nav"
               </a>
            </li>
            <li>
               <a href="link2.html">
                  Вторая ссылка из блока "nav"
               </a>
            </li>
         </menu>
      </nav>
      <section>
         <article>
            <h3>
               Заголовок статьи из блока "article"
            </h3>
```

```
            <p>
               Текст абзаца статьи из блока "article"
            </p>
            <details>
               <summary>
                  Блок "details", текст тега "summary"
               </summary>
               <p>
                  Абзац из блока "details"
               </p>
            </details>
         </article>
      </section>
      <footer>
         <time>
            Содержимое тега "time" блока "footer"
         </time>
         <p>
            Содержимое абзаца из блока "footer"
         </p>
      </footer>
   </body>
</html>
```

Вид вышеприведенного *html*-документа в браузере представлен на рисунке 3.

Рисунок 3 – Вид ***html***-документа в браузере

4.2 Расширение возможности форм в *HTML5*

Новые свойства форм

В *HTML5* введены новые типы *input* полей. Они позволяют писать более семантически правильный код, адаптированный для мобильных устройствах. Например, при использовании типа *email* происходит автоматическая валидация введенного текста, на предмет идентичности адресу электронной почты и т.д.

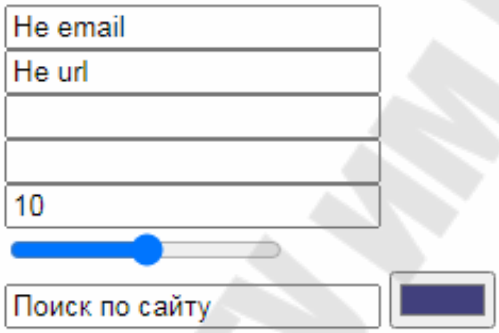
Новые типы *input* полей:

- ***input type=email*** – определяет поле, которые должно содержать *email* адрес. Значение введенное в поле автоматически проверяется перед отправкой на сервер;
- ***input type=url*** – пределяет поле, которое должно содержать *url* адрес. Значение введенное в поле автоматически проверяется перед отправкой на сервер;
- ***input type=tel*** – определяет поле для ввода телефонного номера. С помощью атрибута *pattern* можно установить формат принимаемого телефонного номера. Формат задается с помощью регулярных выражений;
- ***input type=number*** – определяет поле, которое должно содержать числа. Ограничивать диапазон принимаемых чисел можно с помощью атрибутов *min* (минимальное допустимое число) и *max* (максимальное допустимое число). С помощью атрибута *step* Вы можете задать шаг допустимых чисел (к примеру если шаг равен 2, то в поле могут вводиться числа 0,2,4,6 и т.д.) ;
- ***input type=range*** – определяет поле, которые может содержать значения в определенном интервале. Отображается как ползунок, который можно перетаскивать мышкой. Вы можете ограничивать диапазон принимаемых чисел с помощью атрибутов *min* (минимальное допустимое число) и *max* (максимальное допустимое число). С помощью атрибута *step* Вы можете задать шаг допустимых чисел (к примеру если шаг равен 2, то в поле могут вводиться числа 0,2,4,6 и т.д.) ;
- ***input type=search*** – определяет поле поиска (может использоваться, например, для создания поиска по сайту).

Пример создания полей форм *html*-документа в блокноте и вид в браузере:

```
<input name='email' type='email' value='Не email' />
<input name='url' type='url' value='Не url' />
<input name='tel1' type='tel' pattern='8[0-9]{10}' />
<input name='tel2' type='tel' pattern='[0-9]{2,3}-[0-9]{2}-[0-
9]{2}' />
<input name='number' type='number' value='10' />
<input name='range' type='range' min='1' max='5' />
<input name='search' type='search' value='Поиск по сайту' />
<input name='color' type='color' />
```

Результат работы в баузере:



Новые элементы в *HTML5* формах:

- ***datalist*** – позволяет привязать список к полям формы. Значения списка будут выводиться как поисковые подсказки во время ввода информации в поле связанное с ним;
- ***keygen*** – позволяет генерировать открытые и закрытые ключи, которые используются для безопасной связи с сервером;
- ***output*** – может использоваться для вывода различной информации. С помощью атрибута ***for*** можно указать связанные поля.

Вместе с полями ввода нередко используются метки, которые представлены элементом ***label***. Метки создают аннотацию или заголовок к полю ввода, указывают, для чего это поле предназначено. Для связи с полем ввода метка имеет атрибут ***for***, который указывает на ***id*** поля ввода.

Новые атрибуты в *HTML5* формах:

- ***autofocus*** – делает поле активным после загрузки страницы (может использоваться со всеми типами *input*);

– ***form*** – указывает форму, которой принадлежит данное поле (может использоваться со всеми типами *input*).

– ***multiple*** – указывает, что данное поле может принимать несколько значений одновременно (может использоваться с *input* типов *email* и *file*);

– ***novalidate*** – указывает, что данное поле не должно проверяться перед отправкой (может использоваться с *form* и *input*);

– ***placeholder*** – отображает текст-подсказку в поле (может использоваться с *input* следующих типов: *text*, *search*, *url*, *tel*, *email* и *password*.);

– ***required*** – указывает, что данное поле должно быть обязательно заполнено перед отправкой.

Выбор даты

В *HTML5* были добавлены новые элементы ввода, позволяющие удобно выбирать дату и время:

– ***date*** – позволяет выбрать дату в формате год-месяц-день_месяца;

– ***time*** – позволяет выбрать время;

– ***datetime*** – позволяет выбрать дату в формате год-месяц-день_месяца-время (отчет ведется по глобальному времени);

– ***datetime-local*** – позволяет выбрать дату в формате год-месяц-день_месяца-время (отчет ведется по местному времени);

– ***month*** – позволяет выбрать дату в формате год-месяц;

– ***week*** – позволяет выбрать дату в формате год-неделя.

Регулярные выражения для валидации

До появления ***HTML5*** при использовании формы на сайте, необходимо было пропускать введенный текст через *JavaScript* для проверки. Теперь с *HTML5* и атрибутом *pattern*, можно определить шаблон регулярного выражения для проверки данных. Пример:

```
<!-- для проверки адресов электронной почты -->
<input type="text" title="электронный адрес" required pattern=
"[^@]+@[^@]+\.[a-zA-Z]{2,6}" />
<!-- для паролей -->
<input type="text" title="по крайней мере восемь символов, содержащих
хотя бы одну цифру, один символ нижнего и верхнего регистра" required
pattern="(?=.*\d)(?=.*[a-z])(?=.*[A-Z]).{8,}" />
```

```
<!-- для проверки телефонного номера -->
<input type="text" required pattern="(\+?\d[- .]*){7,13}"
title="интернациональный, национальный или местный номер телефона"/>
```

Автозаполнение с *HTML5 datalist*

Использование *dataList* элемента *HTML5* позволяет создавать список данных для автозаполнения полей ввода. Пример:

```
<input name="frameworks" list="frameworks" />

<datalist id="frameworks">
   <option value="MooTools">
   <option value="Moobile">
   <option value="Dojo Toolkit">
   <option value="jQuery">
   <option value="YUI">
</datalist>
```

Автофокус полей формы

Атрибут ***autofocus*** позволит вам установить фокус на любой элемент формы (в том числе кнопку).

```
<input type="text" autofocus="autofocus" />
```

Замещающий текст поля

Это тот самый (серенький) текст, который находится, например, в поле поиска, призывает ввести искомое слово и нажать ***Enter***. В ***HTML***5 замещающий текст реализуется вот-так:

```
<input type="search" placeholder="Поиск на сайте" />
```

Скрытые элементы с помощью *HTML5*

В *HTML5* введен атрибут *hidden*, который позволяет скрыть определенный элемент, на подобии того как это применяется в *CSS* с использованием *dispaly*: *none*.

```
<p hidden>Здесь находится текст, который скрыт с помощью
HTML5! </p>
```

***HTML5 prefetching* — предварительная загрузка**

С помощью этого механизма можно заранее кэшировать указанный ресурс (картинку, страницу или файл), например, сразу подгрузить стили, которые будут использованы на сайте:

```
<link rel="prefetch" href="style_dark.css">
```

HTML5 <details> и <summary> теги

С помощью этих тегов пользователь может манипулировать отображением (раскрывать и скрывать снова), какой либо дополнительной информации.

```
<details>
   <summary>Заголовок 1</summary>
      <p>Pellentesque habitant morbi tristique senectus et
netus ....</p>
</details>
<details>
   <summary>Заголовок 2</summary>
      <p>Pellentesque habitant morbi tristique senectus et
netus et malesuada ...</p>
</details>
```

Атрибут *download*

В *HTML5* появился новый атрибут для ссылок. Теперь, чтобы указать браузеру, что ссылку надо загружать, а не открывать, достаточно использовать данный атрибут.

```
<!-- имя в диалоге будет muzika.mp3-->
<a href="sound.mp3" download="muzika">Скачать песню</a>
```

4.3 Новые *API*

В дополнение к определению разметки, *HTML5* устанавливает *API*, который может быть использован с *JavaScript*. Возможности *DOM* расширены и фактически используемые свойства задокументированы.

Веб хранилища представляют собой более функциональную альтернативу *cookie*.

Преимущества веб хранилищ:

- можно хранить неограниченные объемы информации;
- информация, сохраненная в хранилищах, доступна даже без подключения к интернету;
- данные, находящиеся в хранилище, не отсылаются при каждом запросе страниц;
- информацию более удобно сохранять и извлекать;
- хранилища более безопасны чем ***cookie***.

Сохранене данных происходит с использованием двух специальных объектов: ***sessionStorage*** и ***localStorage***.

Обращаться к данным объектам можно с помощью ***JavaScript*** и других клиентских скриптов.

Обратите внимание, что каждый веб-сайт имеет доступ только к своим данным в хранилище и поэтому не может обращаться к данным, которые были сохранены другими сайтами.

Использование *sessionStorage*

Объект *sessionStorage* сохраняет данные в течении пользовательской сессии. Когда браузер пользователя будет закрыт, данные, сохраненные в объекте, будут удалены.

Данные в хранилище доступны со всех страниц сайта, а не только с той, с которой они были сохранены.

Использование *localStorage*

Объект *localStorage* сохраняет данные на неограниченный период времени.

Данные, сохраненные в данном объекте, будут доступны даже без подключения к интернету.

4.4 Технология *Drag and Drop*

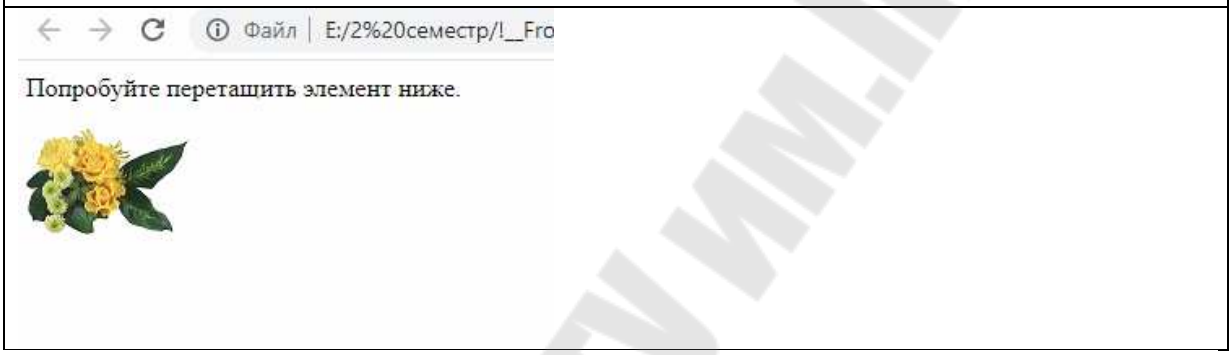
Технология *drag and drop* (*drag'n'drop*) позволяет сделать элементы на веб-страницах перетаскиваемыми (как окна с программами в операционной системе).

Ранее это было возможным только с помощью реализаций сложных *JavaScript* функций или подключением *jQuery*. Сейчас технология *drag and drop* является частью спецификации *HTML5* и ее поддержка встроена во все современные веб-браузеры.

Практически любые элементы на веб-страницах можно сделать перетаскиваемыми, включая картинки, ссылки и обычные текстовые элементы.

Для того, чтобы сделать элемент перетаскиваемым необходимо добавить к нему *HTML5* атрибут *draggable* со значением *true*. Напрмер:

```
<body>
<p>Попробуйте перетащить элемент ниже.</p>
<img src="picture_1.jpg" draggable="true">
</body>
```



Чтобы извлечь из этого какую-либо пользу, нужно дополнительно обработать перетаскиваемый элемент с помощью специальных событий перетаскивания, которые были добавлены в *HTML5*:

- ***dragstart*** – выполнится в начале операции перетаскивания;
- ***ondrag*** – выполнится во время перетаскивания элемента;
- ***ondragenter*** – выполнится, когда перетаскиваемый элемент будет наведен на элемент, который может его принять;
- ***ondragleave*** – выполнится, когда перетаскиваемый элемент будет выведен за пределы границ элемента, который может его принять;
- ***ondragover*** – выполнится, когда перетаскиваемый элемент будет перемещаться в пределах границ элемента, который может его принять;
- ***ondrop*** – выполнится, когда перетаскиваемый элемент будет перемещен в принимающий его элемент;
- ***ondropend*** – выполнится в конце операции перетаскивания.

Обработка элемента во время перетаскивания

Всю обработку перетаскивания элемента можно условно разбить на два шага:

1. Сохранение данных перетаскиваемого элемента в начале операции перетаскивания.

2. Извлечение ранее сохраненных данных, после того, как перетаскиваемый элемент будет перемещен в принимающий его элемент.

Для того, чтобы сохранить данные, необходимо воспользоваться методом *setData* («тип_данных», «сохраняемые_данные») *HTML5* объекта *DataTransfer*. Данные перетаскиваемого элемента необходимо сохранять в самом начале перетаскивания, то есть во время события *ondragstart*.

```
<body>
<style>
#dragel {
   display:inline;
   padding:10px;
   border:4px solid;
}
</style>
<script>
function dragImg(e) {
//С помощью e.target мы можем обращаться к перетаскиваемому
//элементу внутри обработчика события
alert(e.target.id);
//Данная команда позволяет нам сохранить значение атрибута id.
//В первом ее атрибуте мы указываем
//тип данных, которых мы хотим сохранить. В данном случае это
//обычный текст (text/plain), вторым
//атрибутом должны передаваться сами данные.
e.dataTransfer.setData("text/plain",e.target.id);
}
function dragEl(e) {
alert(e.target.innerHTML);
//Так как в содержимом могут присутстововать HTML тэги мы должны
//указать text/html в первом атрибуте
e.dataTransfer.setData("text/html",e.target.innerHTML);
```

```
}
</script>
<p>Начните перетаскивать элементы ниже, чтобы запустить код
обработчиков.</p>
<img src=" picture_1.jpg" draggable="true" width="120px"
ondragstart="dragImg(event)" id="draggable-element">
<br><br><br />
<div id="dragel" draggable="true"
ondragstart="dragEl(event)">Меня можно перетаскивать!</div>
</body>
```

Для того, чтобы извлечь ранее сохраненные данные, необходимо воспользоваться методом *getData*(«тип_данных») объекта *DataTransfer*. Данные перетаскиваемого элемента необходимо извлекать после того, как перетаскиваемый элемент будет перемещен в область принимающего его элемента, то есть во время события *ondrop*.

Обратите внимание: тип данных во время сохранения и извлечения должен совпадать, то есть если сохраняли данные с типом «*text/plain*», вы должны использовать «*text/plain*» и для их извлечения.

По умолчанию элементы не могут принимать другие элементы, поэтому если хотите создать принимающий элемент, то должны привязать к нему обработчик события *ondragover* и передать ему *event.preventDefault()*.

```
 <body>
<style>
#dropel {
   width:120px;
   height:120px;
   border:4px solid #BBCC25;
   padding:10px;
}
</style>
<script>
function makeDroppable(e) {
e.preventDefault();
}
function dragImg(e) {
e.dataTransfer.setData("text/plain",e.target.id);
}
function dropImg(e) {
e.preventDefault();
//При извлечении данных Вы должны использовать такой же тип
//как и при их сохранение
var rdata = e.dataTransfer.getData("text/plain");
//Очистим содержимое перетаскиваемого элемента
e.target.innerHTML="";
//Сделаем перетаскиваемый элемент элементом потомком
//принимающей области т.е. фактически мы
//переносим его в содержимое принимающего его элемента
e.target.appendChild(document.getElementById(rdata))
}
</script>
<p>Перетащите элемент в принимающую его область.</p>
<img src=" picture_1.jpg" draggable="true" width="120px"
ondragstart="dragImg(event)" id="draggable-element">
<br><br><br />
<div id="dropel" ondrop="dropImg(event)"
ondragover="makeDroppable(event)">Перетащите элемент
сюда!</div>
</body>
```

4.5 Видео и аудио в *HTML5*

Видео

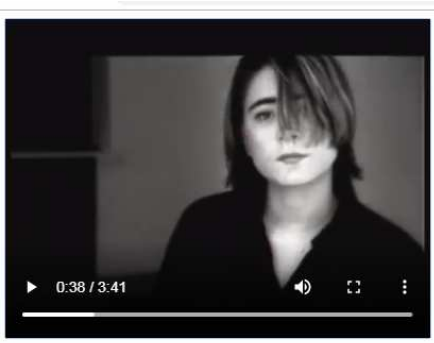
Для воспроизведения видео в *HTML5* используется элемент ***video***. Чтобы настроить данный элемент, необходимо использовать следующие его атрибуты:

- ***src*** – источник видео, это может быть какой-то видеофайл;
- ***width*** – ширина элемента;
- ***height*** – высота элемента
- ***controls*** – добавляет элементы управления воспроизведением;
- ***autoplay*** – устанавливает автовоспроизведение;
- ***loop*** – задает повторение видео;
- ***muted*** – отключает звук по умолчанию.

Хотя можно установить ширину и высоту, но они не окажут никакого влияния на аспектное отношения ширины и высоты самого видео. Например, если видео имеет формат 375×240, то, к примеру, при настройках *width*="375" *height*="280" видео будет центрироваться на 280-пиксельном пространстве в *HTML*, что позволяет избавить видео от искажений, которые были бы при растягивании.

```
<!DOCTYPE html>
<html>
    <head>
        <title>Видео в HTML5</title>
    </head>
    <body>
      <video src="clip.mp4" width="400" height="300" controls
>
      </video>
    </body>
```

```
</html>
```



Применим атрибуты *autoplay* и *loop*:

```
<video src="clip.mp4" width="400" height="300" controls
autoplay loop >
</video>
```

Теперь видео будет автоматически проигрываться бесконечное число раз. Если при воспроизведении необходимо отключить звук, то можем воспользоваться атрибутом ***muted***:

```
<video src="clip.mp4" width="400" height="300" controls muted >
</video>
```

Атрибут ***preload***

Еще один атрибут – *preload* призван управлять загрузкой видео. Он принимает следующие значения:

- ***auto*** – видео и связанные с ним метаданные будут загружаться до того, как видео начнет воспроизводиться;
- ***none*** – видео не будет загружаться в фоне, пока пользователь не нажмет на кнопку начала проигрывания;
- ***metadata*** – в фоне до воспроизведения будут загружаться только метаданные (данные о формате, длительности и т.д), само видео не загружается.

```
<video src="clip.mp4" width="400" height="300" controls
preload="auto"></video>
```

Атрибут ***poster***

Атрибут *poster* позволяет установить изображение, которое будет отображаться до запуска видео. Этому атрибуту в качестве значения передается путь к изображению:

```
<video src="clip.mp4" width="400" height="300" controls
poster="mycat.jpg"></video>
```

Поддержка форматов видео

Главной проблемой при использовании элемента *video* является поддержка различными веб-браузерами определенных форматов.

С помощью вложенных элементов *source* можно задать несколько источников видео, один из которых будет использоваться:

```
<video width="400" height="300" controls>
    <source src="clip.mp4" type="video/mp4">
    <source src="clip.webm" type="video/webm">
    <source src="clip.ogv" type="video/ogg">
</video>
```

Элемент *source* использует два атрибута для установки источника видео:

- ***src*** – путь к видеофайлу;
- ***type*** – тип видео (*MIME*-тип).

Если браузер не поддерживает первый тип видео, то он пытается загрузить второй видеофайл. Если же и тип второго видеофайла не поддерживается, то браузер обращается к третьему видеофайлу.

Аудио

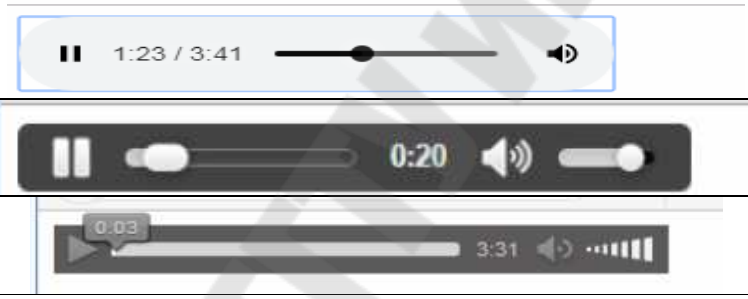
Для воспроизведения звука без видео в *HTML5* применяется элемент ***audio***. Он во многом похож на элемент ***video***. Для настройки элемента *audio* мы можем использовать следующие его атрибуты:

- ***src*** – путь к аудиофайлу;
- ***controls*** – добавляет элементы управления воспроизведением;
- ***autoplay*** – устанавливает автовоспроизведение;
- ***loop*** – задает повторение аудиофайла;

- ***muted*** – отключает звук по умолчанию;
- ***preload*** – устанавливает режим загрузки файла.

Действие всех этих атрибутов будет аналогично их действию в элементе *video*. Опять же, в зависимости от браузера внешний вид элементов управления может отличаться.

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="utf-8">
        <title>Аудио в HTML5</title>
    </head>
    <body>
        <audio src="audio.mp3" controls></audio>
    </body>
</html>
```



Поддержка форматов аудио

Ключевым моментом для работы с аудио является поддержка браузером тех или иных форматов. На данный момент подавляющее большинство браузеров поддерживают ***mp***3. Однако если у нас есть неуверенность, что наше аудио в определенном формате будет поддерживаться браузером пользователя, то можно использовать вложенный элемент *source* и указать аудио в иных форматах:

```
<audio width="400" height="300" controls>
    <source src="audio.mp3" type="audio/mpeg">
    <source src="audio.m4a" type="audio/aac">
    <source src="audio.ogg" type="audio/ogg">
</audio>
```

Как и в случае с элементом *video*, у элемента *source* устанавливается атрибут *src* с ссылкой на файл и атрибут *type* – тип файла.

4.6 *CANVAS* – ЭЛЕМЕНТ *HTML5*, ПОЗВОЛЯЮЩИЙ РИСОВАТЬ ГРАФИКУ НА *JAVASCRIPT*

Разработчики или дизайнеры могут создавать интерфейсы с помощью технологий, основанных на стандартах *HTML5*. Это позволяет существенно улучшить взаимодействие с пользователем, поскольку установка подключаемых модулей не требуется. В настоящее время за графику, как правило, отвечает браузер.

Одной из интересных возможностей нового стандарта *HTML5* является элемент *canvas*, или холст. *HTML5 Canvas* – веб-технология, которая позволяет использовать высококачественную графику в браузерах.

Холст предназначен для создания растровых изображений на странице средствами графического движка браузера. *Canvas* способен изображать не только статические, но и динамические изображения (анимацию).

Главным фактором появления анимации и графики на основе ‹*canvas*› является высокий рост мобильного сегмента Интернет, особенно устройств под управлением операционных систем *Android* и *iOS*. Установить плагин на них нет возможности, а *i*-устройства не поддерживают *Flash*, а начиная с версии 4.0, его также не поддерживают устройства под управлением *Android*. Поэтому для поддержки огромной доли рынка мобильных устройств с возможностью подключения к сети Интернет анимацию начали создавать при помощи *JavaScript*. Для этого обычно используют библиотеки, как например *jQuery*. С введением в действие стандарта *CSS3* часть анимаций возможно создавать с помощью каскадных таблиц стилей. Однако самыми широкими возможностями по созданию изображений и анимаций пользуется стандарт *HTML5* и его новый элемент *‹canvas›*.

Элемент *‹canvas›* – это новый элемент *HTML5*, который позволяет создавать изображения на сайте с помощью *JavaScript*. Область использования холстов довольно широкая. Чаще всего его можно увидеть при создании деловой графики (чарты, диаграмы, графики), а также для рендеринга браузерных игр (чаще всего

встречаются в социальных сетях). У *‹canvas›* есть только два атрибута – ширина и высота.

Элемент *‹canvas›* создает контекст отрисовки, на котором в будущем можно создавать и манипулировать объектами *JavaScript*. Другими словами, *‹canvas›* представляет собой прямоугольную область, в которой с помощью *javascript* можно «рисовать».

На сегодняшний день стандарт полностью описывает работу двумерных контекстов (для плоской графики, 2*D*). Элемент *‹canvas›* также используется технологией *WebGL* для отрисовки аппаратно-ускоренной *3D*-графики на вебстраницах.

Для размещения элемента на странице HTML достаточно указать:

```
‹canvas width=600 height=250› ‹/canvas›
```

После помещения на страницу элементом *‹canvas›* можно манипулировать различными способами: помещать на него текст, рисовать графические элементы и линии, выполнять заливку, добавлять анимацию. Данные манипуляции совершаются при помощи команд *javascript*. Чтобы использовать холст программным путем, необходимо получить доступ к его контексту. После этого выполняются все необходимые действия с контекстом и только тогда результат подтверждается и выводится на холст. Сначала изображение создается программно, а потом результат выводится визуально.

Холст представлен двумерной сеткой. Начало координат (0,0) находится в левом верхнем углу холста.

Как правило, для элемента *canvas* задаются ширина и высота с помощью атрибутов `width` и `height`. Если не указать эти атрибуты, то по умолчанию ширина будет составлять *300*, а высота – *150* пикселей.

Все рисование на *canvas* производится с помощью кода *JavaScript*. Чтобы начать рисовать на *canvas*, необходимо получить его контекст, то есть экземпляр объекта *CanvasRenderingContext2D*. Сделать это можно следующим образом:

```
var canvas = document.getElementById("Canvas");
var context = canvas.getContext("2d");
```

В первой строке получается сам холст, а во второй, с помощью вызова единственного метода объекта холста *getContext(),* получается контекст этого холста. Параметр *2D* указывает на то, что получаемый контекст будет создавать плоское изображение (экземпляр объекта *CanvasRenderingContext2D*).

Рисование прямоугольников

Для рисования простейших фигур – прямоугольников, могут понадобиться три метода:

– *clearRect(x,y,w,h)* – очищает определенную прямоугольную область, верхний левый угол которой имеет координаты *x* и *y*, ширина равна *w*, а высота равна *h;*

– *fillRect(x, y, w,h)* – заливает цветом прямоугольник, верхний леый угол которого имеет координаты *x* и *y*, ширина равна *w*, а высота равна *h;*

– *strokeRect(x,y,w,h)* – рисует контур прямоугольника без заливки его каким-то определенным цветом.

Настройка рисования

Контекст элемента *canvas* предоставляет ряд свойств, с помощью которых можно настроить отрисовку на *canvas*. К подобным свойствам относятся следующие:

– *strokeStyle* – устанавливает цвет линий или цвет контура. По умолчанию установлен черный цвет;

– *fillStyle* – устанавливает цвет заполнения фигур. По умолчанию установлен черный цвет;

– *lineWidth* – устанавливает толщину линий. По умолчанию равно *1.0;*

– *lineJoin* – устанавливает стиль соединения линий;

– *globalAlpha* – устанавливает прозрачность отрисовки на *canvas;*

– *setLineDash* – создает линию из коротких черточек.

Фоновые изображения

Вместо конкретного цвета для заливки фигур, например, прямоугольников, можно использовать изображения. Для этого у кконтекста canvas имеется функция *createPattern*(), которая принимает два параметра: изображение, которое будет использоваться в качестве фона, и принцип повторения изображения.

Последний параметр играет роль в том случае, если размер изображения у нас меньше, чем размер фигуры на *canvas*. Этот параметр может принимать следующие значения:

– *repeat* – изображение повторяется для заполнения всего пространства фигуры;
– *repeat-x* – изображение повторяется только по горизонтали;
– *repeat-y* – изображение повторяется только по вертикали;
– *no-repeat* – изображение не повторяется.

Создание градиента

Элемент *Canvas* позволяет использовать градиент в качестве фона. Для этого применяется объект *CanvasGradient*, который можно создать либо с помощью метода *createLinearGradient()* (линейный градиент), либо с помощью метода *createRadialGradient()* (радиальный градиент).

Линейный градиент. Метод *createLinearGradient(x0, y0, x1, y1),* где *x0* и *y0* – это начальные координаты градиента относительно верхнего левого угла canvas, а *x1* и *y1* – координаты конечной точки градиента.

```
var gradient =context.createLinearGradient(50,30,150,150);
```

Также для создания градиента необходимо задать опорные точки, которые определяют цвет. Для этого у объекта *CanvasGradient* применяется метод *addColorStop(offset, color)*, где *offset* – это смещение точки градиента, а *color* – ее цвет. Например:

```
gradient.addColorStop(0, "blue");
```

Смещение представляет значение в диапазоне от 0 до 1. Смещение 0 представляет начало градиента, а 1 – его конец.

Цвет задается либо в виде строки, либо в виде шестнадцатиричного значения, либо в виде значения *rgb/rgba*.

Радиальный градиент создается с помощью метода *createRadialGradient(x0, y0, r0, x1, y1, r1)*, который принимает следующие параметры:

– *x0* и *y0*: координаты центра первой окружности;

– *r0*: радиус первой окружности;
– *x1* и *y1*: координаты центра второй окружности;
– *r1*: радиус второй окружности.
Например:

```
var gradient =
context.createRadialGradient(120,100,100,120,100,30);
```

Рисование текста

Наряду с геометрическими фигурами и изображениями, *canvas* позволяет выводить текст. Для этого необходимо установить у контекста *canvas* свойство *font*:

```
var canvas = document.getElementById("Canvas"),
 context = canvas.getContext("2d");
 context.font = "22px Verdana";
```

Свойство *font* в качестве значения принимает опредление шрифта. В данном случае это шрифт *Verdana* высотой 22 пикселя. В качестве шрифтов используются стандартные шрифты.

Далее выводится некоторый текст с помощью метода *fillText()*.

Метод *fillText(text, x, y)* принимает три параметра: выводимый текст, *x* и *y* координаты точки, с которой выводится текст.

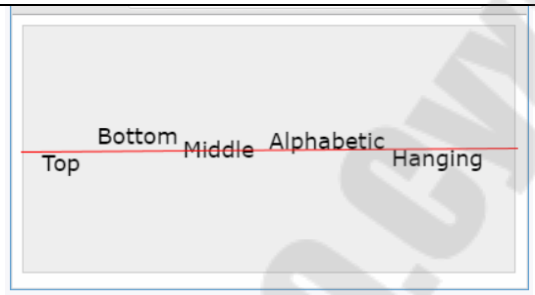
Для вывода текста можно также применять метод *strokeText()*, который создает границу для выводимых символов.

Свойство *textAlign* позволяет выровнить текст относительно одной из сторон. Это свойство может принимать следующие значения:

– *left* – текст начинается с указанной позиции;
– *right* – текст завершается до указанной позиции;
– *center* – текст располагается по центру относительно указанной позиции;
– *start* – значение по умолчанию, текст начинается с указанной позиции;
– *end* – текст завершается до указанной позиции.

Свойство *lineWidth* задает ширину линии текста.

Свойство *textBaseline* задает выравнивание текста по базовой линии. Оно может принимать следующие значения:

– *top;* – *middle;* – *bottom;* – *alphabetic;* – *hanging;* – *ideographic.*	

С помощью метода *measureText()* можно определить ширину текста на canvas.

Рисование фигур

Кроме прямоугольников *canvas* позволяет рисовать и более сложные фигуры. Для оформления сложных фигур используется концепция геометрических путей, которые представляют набор линий, окружностей, прямоугольников и других более мелких деталей, необходимых для построения сложной фигуры.

Для создания нового пути надо вызвать метод ***beginPath()***, а после завершения пути вызывается метод ***closePath().***

```
var canvas = document.getElementById("Canvas"),
    context = canvas.getContext("2d");
            context.beginPath();
            // здесь инструкции по созданию фигур
            context.closePath();
```

Между вызовами методов `beginPath()` и `closePath()` находятся методы, непосредственно создающие различные участки пути. Это методы *moveTo()* и *lineTo()*.

moveTo(x, y) – перемещает нас на точку с координатами *x* и *y*.

lineTo(x, y) – рисует линию от текущей позиции до точки с координатами *x* и *y*.

Метод stroke() служит для отображения пути.

Метод *rect(x, y, width, height)* создает прямоугольник, где *x* и *y* – это координаты верхнего левого угла прямоугольника относительно canvas, а width и height – соответственно ширина и высота прямоугольника.

Метод *fill()* заполняет цветом все внутреннее пространство нарисованного пути.

С помощью свойства `fillStyle` опять же можно задать цвет заполнения фигуры.

Метод *clip()* позволяет вырезать из *canvas* определенную область, а все, что вне этой области, будет игнорироваться при последующей отрисовке.

Метод *arc()* добавляет к пути участок окружности или арку. Его синтаксис:

```
arc(x, y, radius, startAngle, endAngle, anticlockwise)
```

Здесь используются следующие параметры:

x и *y*	*x* и *y* – координаты, в которых начинается арка
`radius`	радиус окружности, по которой создается арка
`startAngle` `endAngle`	начальный и конечный угол, которые усекают окржность до арки.
`anticlockwise`	направление движения по окружности при отсечении ее части, ограниченной начальным и конечным углом. При значении `true` направление против часовой стрелки, а при значении `false` – по часовой стрелке.

В качестве единицы измерения для углов применяются радианы.

Например, полная окружность – это *2π* радиан. Если, к примеру, необходимо нарисовать полный круг, то для параметра *endAngle* можно указать значение *2π*. В *JavaScript* эту величину можно получить с помощью выражения `Math.PI * 2`.

Параметр *anticlockwise* играет важную роль, так как определяет движение по окружности, и в случае изменения *true* на *false* и наоборот, можно получить совершенно разные фигуры:

Метод *arcTo(x1, y1, x2, y2, radius)* также рисует дугу. Здесь *x1* и *y1* – координаты первой контрольной точки, *x2* и *y2* – координаты второй контрольной точки, а *radius* – радиус дуги.

Метод *quadraticCurveTo(x1, y1, x2, y2)* создает квадратичную кривую, где *x1* и *y1* – координаты первой опорной точки, а *x2* и *y2* – координаты второй опорной точки.

Метод *bezierCurveTo(x1, y1, x2, y2, x3, y3)* рисует кривую Безье. Здесь *x1* и *y1* – координаты первой опорной точки,

x2 и *y2* – координаты второй опорной точки,
x3 и *y3* – координаты третьей опорной точки.

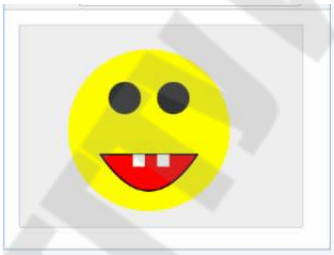
Комплексные фигуры

Объединим несколько фигур вместе и нарисуем более сложную двухмерную сцену:

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="utf-8">
        <title>Canvas </title>
    </head>
    <body>
        <canvas id="Canvas" width="350" height="250"
 style="background-color:#eee; border: 1px solid #ccc;
margin:10px;">
            Ваш браузер не поддерживает Canvas
        </canvas>

    <script>
     var canvas = document.getElementById("Canvas");
          if (canvas.getContext) {
            var ctx = canvas.getContext("2d");
            ctx.beginPath();
            ctx.fill();
            ctx.fillStyle = "yellow";
            ctx.beginPath();
            ctx.arc(160, 130, 100, 0, 2 * Math.PI);
            ctx.fill();

            ctx.beginPath();
            ctx.moveTo(100, 160);
            ctx.quadraticCurveTo(160, 250, 220, 160);
            ctx.closePath();

            ctx.fillStyle = "red";
             ctx.fill();
             ctx.lineWidth = 2;
             ctx.strokeStyle = "black";
             ctx.stroke();

                ctx.fillStyle = "#FFFFFF";
                ctx.fillRect(140, 160, 15, 15);
                ctx.fillRect(170, 160, 15, 15);
```

```
                //
                ctx.beginPath();
                ctx.arc(130, 90, 20, 0, 2 * Math.PI);
                ctx.fillStyle = "#333333";
                ctx.fill();
                ctx.closePath();

                ctx.beginPath();
                ctx.arc(190, 90, 20, 0, 2 * Math.PI);
                ctx.fillStyle = "#333333";
                ctx.fill();
                ctx.closePath();
            }
        </script>
    </body>
</html>
```

Результат работы в браузере:



Изображения на *Canvas*

Для вывода изображения на *canvas* применяется метод:

```
drawImage(image, sx, sy, sWidth, sHeight, dx, dy, dWidth,
dHeight)
```

где *sx* и *sy* – координаты на изображении, с которого начиется обрезка изображения;

sWidth u sHeight – ширину и высоту выреза относительно координат *sx* и *sy;*

dx u dy – координаты отрисовки обрезанного изображения;

dWidth и *dHeight* – ширина и высота изображения на *canvas.*

Одной из замечательных функциональностей элемента *canvas* является возможность ***захвата изображения с другого элемента***, например, элемента *video* или другого элемента canvas.

Добавление теней

Элемент *canvas* поддерживает добавление теней к нарисованным объектам. Для создания теней применяются следующие свойства:

– *shadowOffsetX*: горизонтальное смещение в пикселях справа (или слева при отрицательном значении);

– *shadowOffsetY*: вертикальное смещение в пикселях снизу (или сверху при отрицательном значении);

– *shadowBlur*: число пикселей ля установки размыти тени;

– *shadowColor*: цвет тени.

Редактирование пикселей

HTML5 предоставляет встроенную функциональность для редактирования изображения и установки значения конкретных пикселей на *canvas*.

В частности, можно изменить цветовые значения пикселя, его прозрачность. Для этого предназначены такие методы, как:

– *getImageData(),*

– *putImageData(),*

– *createImageData()*.

–

Метод *getImageData(sx, sy, sw, sh)* позволяет извлечь из *canvas* какую-либо часть изображеня, где *sx* и *sy* – координаты верхнего левого угла области, из которой извлекаются данные на *canvas*, *sw* и *sh* – соотвественно ширина и высота этой области.

Метод *putImageData(imageData, dx, dy)* устанавливает на canvas новые данные.

Параметры *dx* и *dy* указывают координаты верхнего левого угла условного прямоугольника *imageData*, в который размещается на *canvas*.

Метод *createImageData()* создает новый объект *ImageData*, который затем может использоваться на *canvas*.

Метод *createImageData()* имеет две формы:

- *createImageData(width, height);*
- *createImageData(imagedata).*

Первая форма принимает параметры *width* и *height*, которые устанавливают соотвественно ширину и высоту создаваемого объекта *ImageData*.

Вторая форма принимает в качестве параметра другой объект *ImageData*, по которому будет создан новый объект *ImageData*.

Трансформации

Элемент *canvas* поддерживает трансформации – перемещение, вращение, масштабирование.

Метод *translate(x, y)* – перемещение. Первый параметр указывает на смещение по оси *X*, а второй параметр – по оси *Y*.

Метод *rotate(angle)* – вращение.

Метод *scale(xScale, yScale)*– масштабирование. Параметр *xScale* указывает на масштабирование по оси *X,* а *yScale* – по оси *Y*.

При необходимости будем применять последовательно несколько преобразований:

```
ctx.scale(1.5, 1.3);
ctx.translate(100, 150);
ctx.rotate(0.34);
```

Контекст элемента *canvas* также предоставляет метод *transform(a,b,c,d,e,f)*, который позволяет задать матрицу преобразования. Все параметры этого метода последовательно представляют элементы матрицы преобразования:

- *a:* масштабирование по оси *X;*
- *b:* поворот вокруг оси *X;*
- *c:* поворот вокруг оси *Y;*
- *d:* масштабирование по оси *Y;*
- *e:* горизонтальное смещение;

– *f:* вертикальное смещение.

При последовательном применении разных трансформаций они просто последовательно применяются к фигурам. Однако может возникнуть ситуация, когда надо применить трансформацию не вместе с другими, а вместо других, то есть ***заменить трансформацию***.

Для этого применяется метод *setTransform(a, b, c, d, e, f)*.

Его параметры представляют матрицу преобразования, и в целом его применение аналогино применению метода *transform()*.

При применении трансформаций вся последующая отрисовка фигур подвергается данным трансформациям. Но возможна ситуация, когда после одиночного применения трансформации нам больше не нужно ее применение. И для всей последующей отрисовки можно сбросить трансформации с помощью метода *resetTransform()*.

Рисование мышью

Стоит отметить, что можно создавать фигуры динамически, просто рисуя указателем мыши.

Для этого определим следующую страницу:

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="utf-8">
        <title>Canvas </title>
    </head>
    <body>
        <canvas id="Canvas" width="350" height="250"
         style="background-color:#eee; border: 1px
                              solid #ccc; margin:10px;">
            Ваш браузер не поддерживает Canvas
        </canvas>
    <script>
     var canvas = document.getElementById("Canvas"),
                context = canvas.getContext("2d"),
                w = canvas.width,
                h = canvas.height;
            var mouse = { x: 0, y: 0 };
            var draw = false;
```

```
    canvas.addEventListener("mousedown", function (e)
            {
                mouse.x = e.pageX - this.offsetLeft;
                mouse.y = e.pageY - this.offsetTop;
                draw = true;
                context.beginPath();
                context.moveTo(mouse.x, mouse.y);
            });
     canvas.addEventListener("mousemove", function (e) {
                if (draw == true) {
                 mouse.x = e.pageX - this.offsetLeft;
                 mouse.y = e.pageY - this.offsetTop;
                 context.lineTo(mouse.x, mouse.y);
                 context.stroke();
                }
            });
        canvas.addEventListener("mouseup", function (e) {
                mouse.x = e.pageX - this.offsetLeft;
                mouse.y = e.pageY - this.offsetTop;
                context.lineTo(mouse.x, mouse.y);
                context.stroke();
                context.closePath();
                draw = false;
            });
         </script>
    </body>
</html>
```

Для обработки движения мыши для элемента *canvas* определены три обработчика – нажатия мыши, перемещения и отпускания мыши.

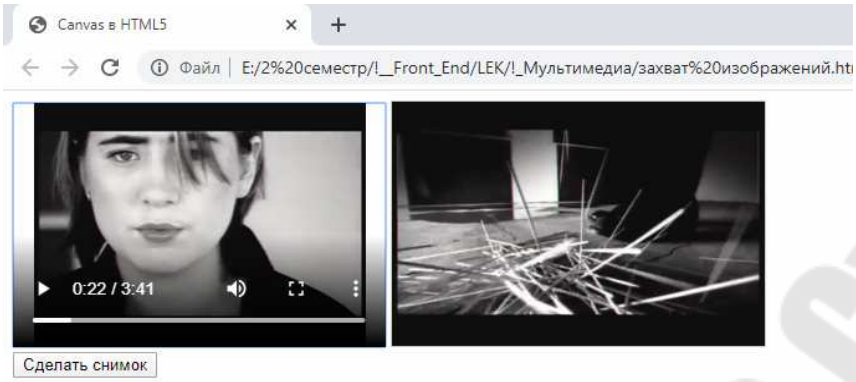
При нажатии мыши устанавливаем переменную *draw* равным *true*. То есть идет рисование. Также при нажатии фиксируем точку, с которой будет идти рисование.

При перемещении мыши получаем точку, на которую переместился указатель, и рисуем линию. При отпускании указателя закрываем графический путь методом `context.closePath()` и сбрасываем переменную *draw* в *false*.

Захват изображений с других элементов

Одной из замечательных функциональностей элемента *canvas* является возможность захвата изображения с другого элемента, например, элемента *video* или другого элемента *canvas*. Например:

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="utf-8">
        <title>Canvas в HTML5</title>
    </head>
    <body>
   <video id="myVideo" src="video.mp4" width="300"
height="200" controls >
   </video>
        <canvas id="myCanvas" width="300" height="200"
         style="background-color:#eee; border:1px solid
#ccc;">
            Ваш браузер не поддерживает Canvas
        </canvas>
        <div>
         <button id="snap">Сделать снимок</button>
        </div>
        <script>
            var canvas = document.getElementById("myCanvas"),
                  context = canvas.getContext("2d");
            var video = document.getElementById("myVideo");
              document.getElementById("snap").onclick =
function (e) {
                context.drawImage(video, 0, 0, 300, 200);
            }
</script>
</body>
</html>
```



По нажатии на кнопку *canvas* будет получать текущий кадр воспроизводимого видео и фиксировать его в качестве изображения. При этом в метод drawImage в качестве первого параметра передается сам элемент, используемый как источник изображения.

5 ПРЕИМУЩЕСТВА ИСПОЛЬЗОВАНИЯ *CSS3*

*CSS*3 – третья версия таблицы каскадных стилей, которая значительно расширила возможности предыдущих поколений. Особенность *CSS*3 состоит в том, что с ее помощью можно создавать анимированные элементы без помощи *Javascript*. Она включает поддержку различных градиентов и теней, использует новые формы сглаживания и т. д.

Использование набора технологий *HTML*5 + *CSS*3 открывает разработчикам огромный арсенал программных средств, с которыми значительно легче стало создавать удобные, стильные, современные и функциональные веб-ресурсы.

Если для разработки использовать эти средства, то они ускорят загрузку страниц и привнесут ряд привлекательных новаций. *CSS*3 позволит создать уникальный дизайн сайта, благодаря использованию большого количества визуальных эффектов.

Возможности *CSS*3:

- создавать элементы со сглаженными углами;
- создавать линейные и сферические градиенты;
- более легко оформить фоновую картинку;
- добавить к элементу и к тексту элементов теней;
- использовать небезопасные шрифты (не боясь при этом, что они не поддерживаются браузером пользователя);
- создавать анимацию и различные эффекты переходов;

- задавать множество разнообразных способов и многое другое.

Размер фонового изображения

в *CSS*3 можно устанавливать размер фоновых изображений с помощью свойства ***background-size***.

Размер фоновых изображений может быть указан в пикселях или в процентах.

```
#wrap1 {
background-image:url("spider2.gif");
background-size:150px 250px;
}
#wrap2 {
background-image:url("spider2.gif");
background-size:70% 70%;
}
```

Несколько фоновых изображений в *CSS*3

*CSS*3 расширяет возможности свойства ***background-image.*** Теперь один элемент может иметь несколько фоновых изображений одновременно.

```
#wrap1 {
background-image:url(wislink.gif),url(mountimg3.jpg);
background-position:bottom right, center;
background-size:150px 40px,100% 100%;
background-repeat:no-repeat,no-repeat;
}
```

Свойство *background-origin*

С помощью нового *CSS*3 свойства *background-origin* можно установить, как должно вычисляться положение элемента относительно границ его родительского элемента.

Данное свойство может иметь три различных значения:

- ***border-box*** положение элемента вычисляется относительно верхнего левого угла границы элемента;
- ***padding-box*** положение элемента вычисляется относительно верхнего левого угла блока *padding*;

- ***content-box*** положение элемента вычисляется относительно верхнего левого угла содержимого.

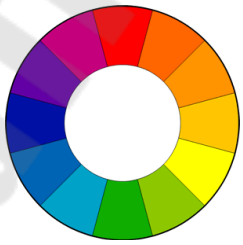
```
#wrap1 {
background-origin:border-box;
background-image:url("border-box.png");
}
#wrap2 {
background-origin:padding-box;
background-image:url("padding-box.png");
}
#wrap3 {
background-origin:content-box;
background-image:url("content-box.png");
}
```

Задание цвета с помощью *HSL*

В *CSS*3 цвет может задаваться с помощью *HSL*, то есть оттенка, насыщенности и яркости.

Для того, чтобы задать цвет этим способом вы должны указать:

1. оттенок цвета указывается в градусах поворота цветового круга (*0 градусов - красный, 120 градусов - зеленый, 240 градусов - голубой и т.д.*);



2. насыщенность цвета указывается в процентах (*по мере понижения процентов цвет будет блекнуть*);
3. яркость цвета также указывается в процентах (*0% - темный, 100% - светлый*).

```
#wrap1 {
background-color:hsl(0,30%,50%);
}
#wrap2 {
background-color:hsl(120,100%,80%);
}
```

```
#wrap3 {
background-color:hsl(240,100%,50%);
}
```

Задание цвета с помощью *RGBA*

Данный способ позволяет определять цвет и прозрачность одновременно. Вначале необходимо указать значения *RGB*, а затем значение прозрачности (*0 - максимальная прозрачность, 1 - минимальная прозрачность*).

Обратите внимание: задание прозрачности с помощью *RGBA* отличается от действия свойства *opacity* тем, что *opacity* делает прозрачным сам элемент и все его элементы потомки, а *RGBA* делает прозрачным только сам элемент.

```
#wrap1
{
background-color:rgba(0,0,0,0.6);
}
```

Задание цвета с помощью *HSLA*

Подобно *RGBA* цвет сразу вместе с прозрачностью можно задавать используя *HSLA*.

```
#wrap1
{
background-color:hsla(0,100%,0%,0.6);
}
```

Создание элементов со сглаженными углами

С помощью нового *CSS*3 свойства *border-radius* можно делать углы элементов сглаженными.

```
#el1 {
border-radius:5px;
}
#el2 {
border-radius:10px;
}
#el3 {
border-radius:20px;
}
```

```
#el4 {
border-radius:15px;
}
```

Данное свойство может применяться не ко всем углам элемента, а только к определенным:

- ***border-top-left-radius*** делает сглаженным только верхний левый угол элемента;
- ***border-top-right-radius*** делает сглаженным только верхний правый угол элемента;
- ***border-bottom-left-radius*** делает сглаженным только нижний левый угол элемента;
- ***border-bottom-right-radius*** делает сглаженным только нижний правый угол элемента.

```
#el1 {
border-top-left-radius:20px;
}
#el2 {
border-top-right-radius:20px;
}
#el3 {
border-bottom-left-radius:20px;
}
#el4 {
border-bottom-right-radius:20px;
}
```

Добавление к элементам тени

С помощью свойства *box-shadow* можно добавлять к элементам страницы тени.

Добавляя тени к элементам Вы делаете дизайн страницы более «естественным» (*то есть имитирующим реальный мир так как объекты в нем отбрасывают тени*).

Тень может быть внешней и внутренней. Внешние тени создают эффект приподнятости элемента над остальным содержимым, а внутренние создают эффект вдавленности элемента.

```
#el1 {
box-shadow:4px 4px black;
```

```
}
#el2 {
box-shadow:6px 6px 6px 2px black;
}
#el3 {
box-shadow:0px 0px 6px 2px black inset;
}
```

Установка цвета границы

С помощью нового *CSS*3 свойства *border-colors* можно регулировать цвет каждого пикселя границы.

```
#el1
{
border:8px solid;
-moz-border-top-colors: #FF0000 #EB1010 #D22E2E #B03E3E;
-moz-border-right-colors: #FF0000 #EB1010 #D22E2E #B03E3E;
-moz-border-bottom-colors: #FF0000 #EB1010 #D22E2E #B03E3E;
-moz-border-left-colors: #FF0000 #EB1010 #D22E2E #B03E3E;
}
```

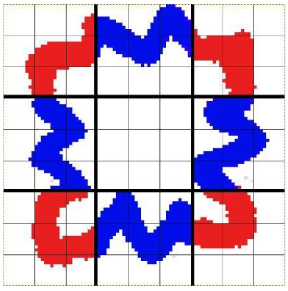
Вставка изображений в качестве границы

В *CSS*3 было добавлено новое свойство *border-image* позволяющее вставлять произвольные изображения в качестве границы элемента. Синтаксис:

```
border-image:путь(1) отступ(2) ширина(3) повторение(4);
```

Для того, чтобы сделать это необходимо:

1. указать путь к изображению-границе;
2. указать величину отступа от каждого края изображения для того, чтобы разрезать изображения на 9 частей (верхний левый угол, верхняя сторона, верхний правый угол, левая сторона и т.д.).



В данном примере величина отступа установлена равной 30*px* с каждой стороны изображения (размер клетки равен 10x10 пикселей). Черными линиями проиллюстрировано как будет в итоге разрезано изображение. Углы и стороны специально выделены разными цветами, чтобы Вы легко могли опознать их в примере далее.

3. Указать ширину границы в пикселях.

4. Указать должно ли изображение повторяться (*repeat*), округляться (*round*) или растягиваться (*stretch*), чтобы заполнить границу элемента.

```
#el1
{
border-image:url("imgborder.jpg") 30 30 round;
-webkit-border-image:url("imgborder.jpg") 30 30 round;
-moz-border-image:url("imgborder.jpg") 30 30 repeat;
-o-border-image:url("imgborder.jpg") 30 30 round;
}
#el2
{
border-image:url("imgborder.jpg") 30 30 stretch;
-webkit-border-image:url("imgborder.jpg") 30 30 stretch;
-moz-border-image:url("imgborder.jpg") 30 30 stretch;
-o-border-image:url("imgborder.jpg") 30 30 stretch;
}
```

*CSS*3 Свойство *@font-face*

В предыдущих версиях *CSS* разработчики были вынуждены использовать только те шрифты, которые гарантированно установлены на компьютере пользователя, в *CSS*3 разработчики могут использовать любые шрифты, которые они захотят.

Когда найдете необходимый шрифт просто разместите его на веб-сервере и подключите его с помощью нового *CSS*3 свойства *@font-face*.

Подключенный шрифт будет загружен и отображен автоматически при посещении страницы пользователем. Синтаксис:

```
@font-face {
font-family:opensans; /* Задаем имя шрифта */
src:url('opensans.woff')
```

```
      /* Указываем местонахождение нашего шрифта */
}
```

Для того, чтобы затем использовать подключенный шрифт, необходимо просто добавить к желаемому элементу свойство *font-family* содержащее имя этого шрифта.

```
div
{
font-family:opensans;
}
```

Также можно добавить курсивное или жирное начертание к подключенному шрифту.

Для этого необходимо дополнительно добавить в *@font-face* свойство *font-style*:*italic* или *font-weight*:*bold* и указать путь к шрифту в соответствующем начертании.

```
/* Подключим курсивную версию шрифта */
@font-face {
font-family:"opensans";
font-style:italic;
src:url('opensans-italic.woff'),;
}
/* Подключим жирную версию шрифта */
@font-face {
font-family:"opensans";
font-style:bold;
src:url('opensans-bold.woff'),;
}
```

Добавление теней к тексту

С помощью нового *CSS*3 свойства *text-shadow* можно добавлять к тексту элементов тени (*к тексту одного элемента может быть добавлено одновременно несколько теней*).

При задании тени для текста необходимо указать: величину смещения тени от текста по горизонтали и вертикали (*может быть отрицательной*), а также радиус размытия и цвет тени.

```
#shadow1 {
```

```
text-shadow:3px 2px #FFAE00;
}

#shadow2 {
text-shadow:1px 1px 10px #FFAE00;
}

#shadow3 {
text-shadow:2px 2px 2px #FFAE00, 2px 2px 15px #1435AD;
}
```

Свойство *text-overflow*

В *CSS*3 было добавлено новое свойство *text-overflow*, которое позволяет указать, что должно случиться с текстом вышедшем за пределы границ элемента.

```
#wrap1 {
text-overflow:ellipsis;
overflow:hidden;
}
#wrap2 {
text-overflow:clip;
overflow:hidden;
}
```

Свойство *word-wrap*

С помощью нового *CSS*3 свойства *word-wrap* можно указать, что длинные слова, выходящие за пределы границ элемента, должны разделяться и переносится на новую строку.

```
#wrap2 {
word-wrap:break-word;
}
```

Создание прозрачных элементов и картинок

С помощью *CSS* можно создавать прозрачные элементы и картинки. Для создания прозрачных элементов во всех браузерах кроме *Internet Explorer* используется свойство *opacity:x*, где *x* – значение которое может изменяться от 0.0 (полностью прозрачный элемент) до 1.0 (полностью непрозрачный элемент).

Для создания прозрачных элементов в *Internet Explorer* используется свойство *filter*:*alpha*(*opacity*=*x*), где *x* значение которое может изменяться от 0 (полностью прозрачный элемент) до 100 (полностью непрозрачный элемент).

```
.op1
{
opacity:0.8;
filter:alpha(opacity=80);
}
```

Отображение картинок с разными уровнями прозрачности

```
.op1
{
opacity:0.8;
filter:alpha(opacity=80);
}
.op2
{
opacity:0.2;
filter:alpha(opacity=20);
}
.op3
{
opacity:0.5;
filter:alpha(opacity=50);
}
```

***CSS*3 трансформирование**

С помощью *CSS*3 свойства ***transform*** можно трансформировать элементы. В качестве значения данного свойства должна указываться одна из функций трансформирования.

На данный момент в современных браузерах поддерживаются только 2*D* трансформации, но в будущем будут также доступны и 3*D* трансформации.

С помощью функции *translate*(*x*,*y*) можно сместить элемент на указанное количество пикселей по горизонтали и вертикали.

```
#el2 {
transform: translate(180px,180px);
}
```

С помощью функции ***rotate***(градусы) можно повернуть элемент на указанное количество градусов по часовой стрелке.

```
#el3 {
transform: rotate(45deg);
}
#el4 {
transform: rotate(120deg);
}
```

С помощью метода ***scale(x,y)*** можно растянуть элемент в ширину или высоту.

```
#el6 {
transform:scale(1.3,1);
}
```

С помощью метода ***skew(x,y)*** можно скосить элемент на указанное количество градусов по горизонтали и вертикали.

```
#el7 {
transform:skew(40deg,20deg);
}
```

*CSS*3 функции трансформирования

Функция	Описание
translate(x,y)	смещает элемент от изначальной позиции по горизонтали и вертикали.
translateX(x)	смещает элемент по горизонтали.
translateY(y)	смещает элемент по вертикали.
scale(x,y)	растягивает элемент по вертикали и горизонтали.

***scaleX*(*x*)**	растягивает элемент по горизонтали.
***scaleY*(*y*)**	растягивает элемент по вертикали.
***rotate*(градусы)**	поворачивает элемент по часовой стрелке.
***skew*(*x*,*y*)**	скашивает элемент по горизонтали и вертикали.
***skewX*(*x*)**	скашивает элемент по горизонтали.
***skewY*(*y*)**	скашивает элемент по вертикали.
***matrix*(*x*,*x*,*x*,*x*,*x*,*x*)**	совмещает все перечисленные выше методы в один.

***CSS*3 линейные градиенты**

В *CSS*2.1 градиенты реализовывались в виде отдельных картинок вставляющихся как фоновые, в *CSS*3 имеются встроенные свойства для создания градиентов. Так как в *CSS*3 сам браузер отрисовывает градиенты необходимость дополнительных запросов градиентных картинок у сервера отпадает и это позволяет увеличить скорость загрузки страниц.

Линейные градиенты создаются с помощью *CSS*3 метода *linear-gradient*, который должен указываться в значение свойства *background*.

Для того, чтобы создать линейный градиент, необходимо указать его направление (*может задаваться с помощью ключевых слов или градусов*) и цвета перехода.

```
#wrap1 {
background:linear-gradient(top,white,black);
}
#wrap2 {
background:linear-gradient(left,white,black);
}
#wrap3 {
background:linear-gradient(0deg,white,black);
}
#wrap4 {
background:linear-gradient(270deg,white,black);
}
```

Цвета перехода – это цвета, которые принимает градиент в определенных его точках, например градиент, который плавно изменяет цвет с белого на черный имеет белый цвет перехода в начальной точке и черный в конечной.

Линейные градиенты могут иметь неограниченное количество цветов перехода.

Можно указывать координаты местоположения цветов с помощью % (0% подразумевает начало градиента, 100% конец).

```
#wrap1 {
background:linear-gradient(top,white 0%,green 50%,black 100%);
}
#wrap2 {
background:linear-gradient(left,#8F04A8 0%,#7CE700 60%,#FFE100
100%);
}
```

Сферические градиенты

С помощью метода ***radial-gradient*** можно создавать сферические градиенты.

Синтаксис определения сферических градиентов очень похож на синтаксис линейных, но требует также задания формы градиента (*может быть сферической или эллипсоидной*).

```
#wrap1 {
background:radial-gradient(white 20%,black 40%);
}
#wrap2 {
background:radial-gradient(circle,#8F04A8 0%,#5D016D 40%,black
60%);
}
```

Повторяющиеся градиенты

Повторяющиеся градиенты задаются с помощью ***CSS***3 методов ***repeating-linear-gradient*** (*создает повторяющийся линейный градиент*) и ***repeating-radial-gradient*** (*создает повторяющийся сферический градиент*).

Для того, чтобы создать повторяющийся градиент Вы должны указать направление градиента, а также цвета перехода и расстояние, которое они должны занимать.

```
#wrap1 {
background:repeating-linear-gradient(50deg,white,white 5px,
black 5px,black 10px);
}

#wrap2 {
background:repeating-radial-gradient(circle,#8F04A8 0%,#5D016D
40%, black 60%);
}
```

*CSS*3 Переходы

С помощью нового ***CSS***3 свойства ***transition*** можно создавать эффекты перехода.

Для создания переходов необходимо указать какое ***CSS*** свойство будет изменятся и скорость выполнения этих изменений в секундах.

```
#wrap1
{
width:200px;
transition: width 4s;
}
#wrap1:hover
{
width:500px;
}
```

Для того, чтобы добавить эффект перехода к нескольким свойствам просто перечислите их названия через запятую.

```
#wrap1
{
background-color:#E869AA;
color:#000;
width:200px;
transition: color 4s, width 4s, background-color 4s;
}

#wrap1:hover
{
```

```
color:#FFFFFF;
width:400px;
background-color:#880045;
}
```

Функции смягчения

Плавность выполнения переходов контролируется с помощью функций смягчения. В *CSS*3 существуют несколько видов таких функций:

- ***linear***;
- ***ease*** (*функция смягчения по умолчанию*);
- ***ease-in***;
- ***ease-out***;
- ***cubic-bezier***(***x***,***x***,***x***,***x***) (*поведение функции контролируется переданными параметрами*).

```
div {
width:230px;
transition:width 4s;
}

div:hover {
width:600px;
}

#el1 {
transition-timing-function:linear;
}
#el2 {
transition-timing-function:ease;
}
#el3 {
transition-timing-function:ease-in;
}
#el4 {
transition-timing-function:ease-out;
}
#el5 {
transition-timing-function:ease-in-out;
}
```

```
#el6 {
transition-timing-function:cubic-bezier(0.6,0.2,0.5,0.6);
}
```

***CSS*3 свойства переходов**

Свойство	Описание
transition	позволяет задать значения четырех различных свойств перехода в одном определении.
transition-property	позволяет указать имя ***CSS*** свойства, к которому будет применен эффект перехода.
transition-duration	позволяет указать время выполнения перехода (*по умолчанию имеет значение 0*).
transition-timing-function	позволяет задать функцию смягчения отвечающую за плавность выполнения перехода (*по умолчанию имеет значение **'ease'***).
transition-delay	позволяет задать задержку перед началом выполнения перехода (*по умолчанию имеет значение 0*).

***CSS*3 Анимация**

Для создания анимации в *CSS*3 используется свойство ***@keyframes***.

Данное свойство представляет собой контейнер, в который должны помещаться различные свойства оформления. Синтаксис:

```
@keyframes имяАнимации
{
from {CSS свойства}  // Оформление элемента перед началом
анимации
to {CSS свойства}   // Оформление элемента после завершения
анимации
}
```

После того, как анимация была создана, необходимо добавить к элементу, который Вы хотите анимировать, *CSS*3 свойство *animation* и указать в нем имя анимации (*1 значение*) и время (*2 значение*), в течении которого она будет выполняться.

Также можно устанавливать количество повторов анимации (*3 значение*).

```
@keyframes anim {
from {margin-left:3px;}
to {margin-left:500px;}
}

#wrap1 {
animation:anim 4s 3;
}
```

Можно определять **ход выполнения анимации** не только с помощью ключевых слов *from* и *to* (*которые использовались в предыдущем примере*), но и с помощью %.

С помощью % можно более точно контролировать ход выполнения анимации, например можно указать, что определенный элемент в начале анимации (0%) должен быть белым к середине (50%) должен окрашиваться в оранжевый цвет, а к концу (100%) становиться черным.

```
@keyframes anim {
0% {margin-left:3px;margin-top:3px;background-color:#7F0055;}
30% {margin-left:3px;margin-top:250px;background-
color:#7F0055;}
60% {margin-left:500px;margin-top:250px;background-
color:black;}
100% {margin-left:3px;margin-top:3px;background-color:#7F0055;}
}
#wrap1 {
animation:anim 6s 3;  }
```

***CSS*3 свойства анимации**

Свойство	Описание
@keyframes	Контейнер для определения анимации.

animation	Позволяет задать все значения для настройки выполнения анимации за одно определение.
animation-name	Позволяет указать имя анимации.
animation-duration	Позволяет задать скорость выполнения анимации в секундах (*по умолчанию имеет значение 0*).
animation-timing-function	Позволяет задать функцию смягчения отвечающую за плавность выполнения анимации (*по умолчанию имеет значение* ***ease***).
animation-delay	Позволяет задать задержку перед началом выполнения анимации (*по умолчанию имеет значение 0*).
animation-iteration-count	Позволяет задать количество повторов анимации (*по умолчанию имеет значение 1*).
animation-direction	При значении ***alternate*** в нечетные разы (*1,3,5 ...*) анимация будет проигрываться в нормальном, а в четные (*2,4,6 ...*) в обратном порядке. По умолчанию данное свойство имеет значение ***normal***, при данном значении анимация всегда проигрывается в нормальном порядке.

Разбиение текста на столбцы

С помощью *CSS*3 свойства *column-count* можно указать количество столбцов, на которые необходимо разбить текст выбранного элемента.

```
div {
column-count:3;
}
```

Оформление столбцов текста

С помощью *CSS*3 свойство *column-gap* можно установить величину отступа между столбцами текста.

```
div {
column-count:4;
```

```
column-gap:50px;
}
```

С помощью свойства *column-rule* можно задать ширину, цвет и стиль оформления пространства между столбцами.

```
div {
column-count:4;
column-rule:2px dotted #7F0055;
}
```

*CSS*3 свойство *column-width* позволяет указывать ширину столбцов текста.

```
div {
column-width:150px;
}
```

6 *CSS* МОДУЛЬ РАСКЛАДКИ *FLEXBOX.* *FLEX CONTAINER*

CSS* модуль раскладки *Flexible Box был разработан как модель одномерного-направленного макета и как один из методов распределения пространства между элементами в интерфейсе, с мощными возможностями выравнивания.

Flexbox – это общее название для модуля *Flexible Box Layout*, который имеется в *CSS*3. *Flexbox* предоставляет инструменты для быстрого создания сложных, гибких макетов и функций, которые были сложными в различных методах *CSS*.

Данный модуль определяет особый режим компоновки/верстки пользовательского интерфейса, который называется *flex layout*.

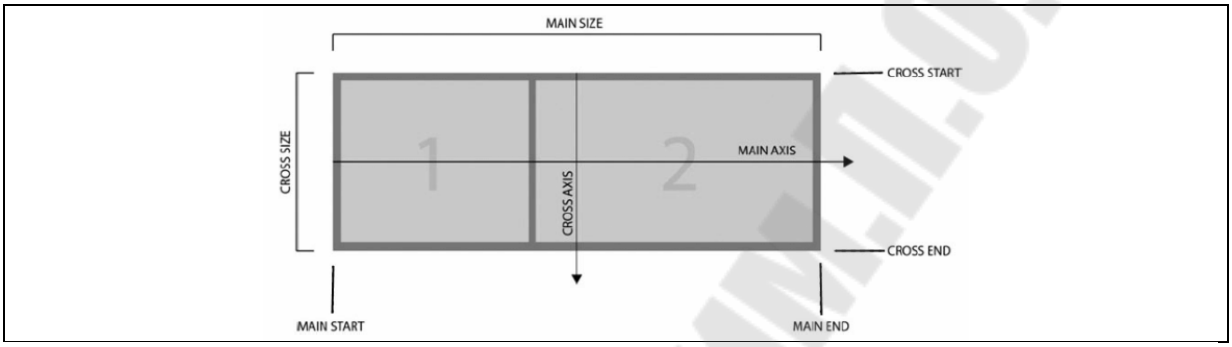
В этом плане *Flexbox* предоставляет иной подход к созданию пользовательского интерфейса, который отличается от табличной или блочной верстки.

Благодаря *Flexbox* проще создавать сложные, комплексные интерфейсы, где с легкостью можем переопределять направление и выравнивание элементов, создавать адаптивные табличные представления.

Основными составляющими компоновки *flexbox* являются *flex*-контейнер (*flex container*) и *flex*-элементы (*flex items*).

Flex container представляет некоторый элемент, внутри которого размещены *flex*-элементы.

Одно из ключевых понятий представляет *main axis* или центральная ось. Это условная ось во *flex*-контейнере, вдоль которой позиционируются *flex*-элементы.



Элементы в контейнере могут располагаться по горизонтали в виде строки и по вертикали в виде столбца. В зависимости от типа расположения будет меняться и центральная ось.

Если расположение в виде строки, то центральная ось направлена горизонтально слева направо. Если расположение в виде столбца, то центральная ось направлена вертикально сверху вниз.

Термины *main start* и *main end* описывают соответственно начало и конец центральной оси, а расстояние между между ними обозначается как *main size*.

Кроме основной оси существует также поперечная ось или *cross axis*. Она перпендикулярна основной. При расположении элементов в виде строки *cross axis* направлена сверху вниз, а при расположении в виде столбца она направлена слева направо.

Начало поперечной оси обозначается как *cross start*, а ее конец - как *cross end*. Расстояние между ними описывается термином *cross size*. То есть, если элементы располагаются в строку, то *main size* будет представлять ширину контейнера или элементов, а *cross size* – их высоту.

Если же элементы располагаются в столбик, то, наоборот, *main size* представляет высоту контейнера и элементов, а *cross size* – их ширину.

Создание *flex*-контейнера

Для создания *flex*-контейнера необходимо присвоить его стилевому свойству *display* одно из двух значений: *flex* или *inline-flex*.

Создадим простейшую веб-страницу, которая применяет *flexbox*:

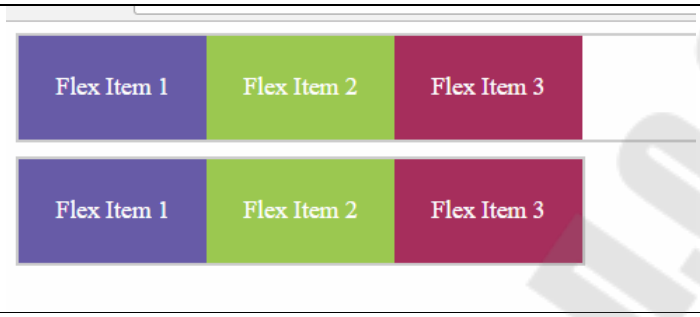
```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="utf-8">
        <title>Flexbox в CSS3</title>
        <style>
            .flex-container {
                display: flex;
            }
            .flex-item {
                text-align:center;
                font-size: 1.1em;
                padding: 1.5em;
                color: white;
            }
            .color1 {background-color: #675BA7;}
            .color2 {background-color: #9BC850;}
            .color3 {background-color: #A62E5C;}
        </style>
    </head>
    <body>
        <div class="flex-container">
            <div class="flex-item color1">Flex Item 1</div>
            <div class="flex-item color2">Flex Item 2</div>
            <div class="flex-item color3">Flex Item 3</div>
        </div>
    </body>
</html>
```

Если значение *flex* определяет контейнер как блочный элемент, то значение *inline-flex* определяет элемент как строчный (*inline*).

Рассмотрим оба способа на примере:

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="utf-8">
        <title>Flexbox в CSS3</title>
        <style>
            .flex-container {
                display: flex;
                border:2px solid #ccc;
            }
            .inline-flex-container {
                display: inline-flex;
                border:2px solid #ccc;
                margin-top:10px;
            }
            .flex-item {
                text-align:center;
                font-size: 1.1em;
                padding: 1.5em;
                color: white;
            }
            .color1 {background-color: #675BA7;}
            .color2 {background-color: #9BC850;}
            .color3 {background-color: #A62E5C;}
        </style>
    </head>
    <body>
        <div class="flex-container">
            <div class="flex-item color1">Flex Item 1</div>
            <div class="flex-item color2">Flex Item 2</div>
            <div class="flex-item color3">Flex Item 3</div>
        </div>

        <div class="inline-flex-container">
            <div class="flex-item color1">Flex Item 1</div>
```

```
            <div class="flex-item color2">Flex Item 2</div>
            <div class="flex-item color3">Flex Item 3</div>
        </div>
    </body>
</html>
```



В частности, в первом случае *flex*-контейнер растягивается по ширине страницы, а во втором случае занимает именно столько места, сколько необходимо для *flex*-элементов.

Направление *flex-direction*

Flex-элементы во *flex*-контейнере могут иметь определенное направление, а именно они могут располагаться в виде строк или в виде столбцов. Для управления направлением элементов *CSS*3 предоставляет свойство *flex-direction*. Оно определяет направление элементов и может принимать следующие значения:

- ***row***: значение по умолчанию, при котором элементы располагаются в виде строки слева направо;
- ***row-reverse***: элементы также располагаются в виде стоки только в обратном порядке справа налево;
- ***column***: элементы располагаются в столбик сверху вниз;
- ***column-reverse***: элементы располагаются в столбик в обратном порядке снизу вверх.

Например, расположение в виде строки:

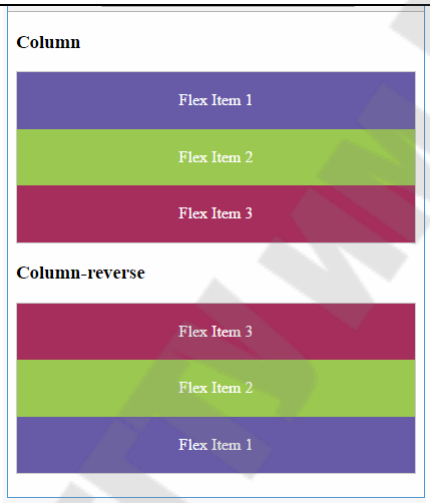
```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="utf-8">
        <title>Flexbox в CSS3</title>
        <style>
            .flex-container {
```

```
                display: flex;
                border:1px solid #ccc;
            }
            .row {
                flex-direction: row;
            }
            .row-reverse {
                flex-direction: row-reverse;
            }
            .flex-item {
                text-align:center;
                font-size: 1.1em;
                padding: 1.5em;
                color: white;
            }
            .color1 {background-color: #675BA7;}
            .color2 {background-color: #9BC850;}
            .color3 {background-color: #A62E5C;}
        </style>
    </head>
    <body>
        <h3>Row</h3>
        <div class="flex-container row">
            <div class="flex-item color1">Flex Item 1</div>
            <div class="flex-item color2">Flex Item 2</div>
            <div class="flex-item color3">Flex Item 3</div>
        </div>

        <h3>Row-reverse</h3>
        <div class="flex-container row-reverse">
            <div class="flex-item color1">Flex Item 1</div>
            <div class="flex-item color2">Flex Item 2</div>
            <div class="flex-item color3">Flex Item 3</div>
        </div>
    </body>
</html>
```

Аналогично работает расположение в виде столбца:

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="utf-8">
        <title>Flexbox в CSS3</title>
        <style>
            .flex-container {
                display: flex;
                border:1px solid #ccc;
            }
            .column {
                flex-direction: column;
            }
            .column-reverse {
                flex-direction: column-reverse;
            }
            .flex-item {
                text-align:center;
                font-size: 1em;
                padding: 1.2em;
                color: white;
            }
            .color1 {background-color: #675BA7;}
            .color2 {background-color: #9BC850;}
            .color3 {background-color: #A62E5C;}
        </style>
    </head>
    <body>
        <h3>Column</h3>
        <div class="flex-container column">
```

```
            <div class="flex-item color1">Flex Item 1</div>
            <div class="flex-item color2">Flex Item 2</div>
            <div class="flex-item color3">Flex Item 3</div>
        </div>
        <h3>Column-reverse</h3>
        <div class="flex-container column-reverse">
            <div class="flex-item color1">Flex Item 1</div>
            <div class="flex-item color2">Flex Item 2</div>
            <div class="flex-item color3">Flex Item 3</div>
        </div>
    </body>
</html>
```



flex-wrap

Свойство *flex-wrap* определяет, будет ли *flex*-контейнер несколько рядов элементов (строк или столбцов) в случае если его размеры недостаточны, чтобы вместить в один ряд все элементы. Это свойство может принимать следующие значения:

– ***nowrap***: значение по умолчанию, которое определяет *flex*-контейнер, где все элементы раполагаются в одну строку (при расположении в виде строк) или один столбец (при расположении в столбик);

– ***wrap***: если элементы не помещаются во *flex*-контейнер, то создает дополнительные ряды в контейнере для размещения элементов.

При расположении в виде строки создаются дополнительные строки, а при расположении в виде столбца добавляются дополнительные столбцы:

– ***wrap-reverse***: то же самое, что и значение *wrap*, только элементы располагаются в обратном порядке.

Например, возьмем значение по умолчанию *nowrap*:

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="utf-8">
        <title>Flexbox в CSS3</title>
        <style>
            .flex-container {
                display: flex;
                border: solid 0.25em #000;
                width: 60%;
                height:8.25em;
                flex-wrap:nowrap;
            }
            .row {
                flex-direction: row;
            }
            .row-reverse {
                flex-direction: row-reverse;
            }
            .flex-item {
                text-align:center;
                font-size: 1em;
                padding: 1.5em;
                color: white;
                opacity: 0.8;
            }
            .color1 {background-color: #675BA7;}
            .color2 {background-color: #9BC850;}
            .color3 {background-color: #A62E5C;}
            .color4 {background-color: #2A9FBC;}
            .color5 {background-color: #F15B2A;}
        </style>
    </head>
    <body>
        <h3>Row</h3>
        <div class="flex-container row">
            <div class="flex-item color1">Flex Item 1</div>
```

```
            <div class="flex-item color2">Flex Item 2</div>
            <div class="flex-item color3">Flex Item 3</div>
            <div class="flex-item color4">Flex Item 4</div>
            <div class="flex-item color5">Flex Item 5</div>
        </div>

        <h3>Row-reverse</h3>
        <div class="flex-container row-reverse">
            <div class="flex-item color1">Flex Item 1</div>
            <div class="flex-item color2">Flex Item 2</div>
            <div class="flex-item color3">Flex Item 3</div>
            <div class="flex-item color4">Flex Item 4</div>
            <div class="flex-item color5">Flex Item 5</div>
        </div>
    </body>
</html>
```

Здесь в каждом из *flex*-контейнеров по пять элементов, однако ширина контейнера может вмещать не все элементы, тогда они уходят за границу контейнера:

При установке значения *wrap* во *flex*-контейнере добавляются дополнительные ряды для помещения всех элементов в контейнере. Так, изменим значение свойства *flex-wrap* в контейнере:

```
.flex-container {
    display: flex;
    border: solid 0.25em #000;
    width: 60%;
    height:8.25em;

    flex-wrap:wrap;
```

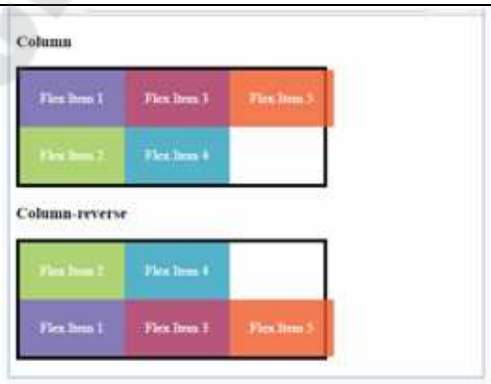
```
}
```

В этом случае появится дополнительная строка:

При расположении в виде столбца контейнер будет создавать дополнительные столбцы:

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="utf-8">
        <title>Flexbox в CSS3</title>
        <style>

            .flex-container {
                display: flex;
                border: solid 0.25em #000;
                width: 60%;
                height:8.25em;
                flex-wrap:wrap;
            }
            .column {
                flex-direction: column;
            }
            .column-reverse {
                flex-direction: column-reverse;
            }
            .flex-item {
                text-align:center;
                font-size: 1em;
                padding: 1.5em;
```

```
                color: white;
                opacity: 0.8;
            }
            .color1 {background-color: #675BA7;}
            .color2 {background-color: #9BC850;}
            .color3 {background-color: #A62E5C;}
            .color4 {background-color: #2A9FBC;}
            .color5 {background-color: #F15B2A;}
        </style>
    </head>
    <body>
        <h3>Column</h3>
        <div class="flex-container column">
            <div class="flex-item color1">Flex Item 1</div>
            <div class="flex-item color2">Flex Item 2</div>
            <div class="flex-item color3">Flex Item 3</div>
            <div class="flex-item color4">Flex Item 4</div>
            <div class="flex-item color5">Flex Item 5</div>
        </div>

        <h3>Column-reverse</h3>
        <div class="flex-container column-reverse">
            <div class="flex-item color1">Flex Item 1</div>
            <div class="flex-item color2">Flex Item 2</div>
            <div class="flex-item color3">Flex Item 3</div>
            <div class="flex-item color4">Flex Item 4</div>
            <div class="flex-item color5">Flex Item 5</div>
        </div>
    </body>
</html>
```



***flex-flow*. Порядок элементов**

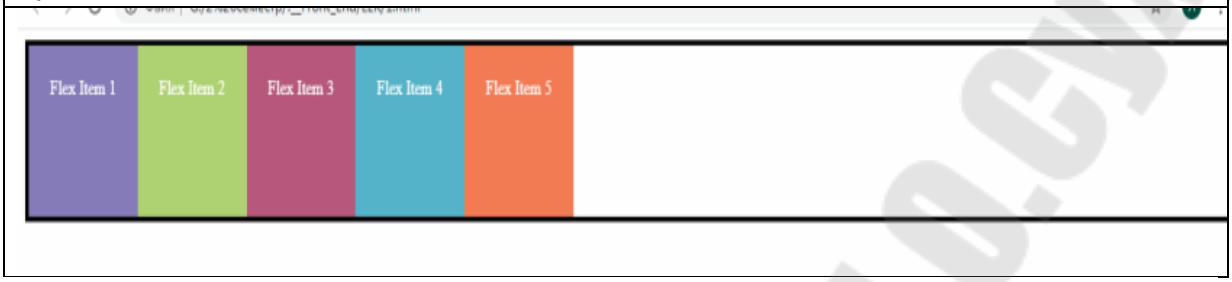
Свойство *flex-flow* позволяет установить значения сразу для обоих свойств *flex-direction* и *flex-wrap*. Оно имеет следующий формальный синтаксис:

```
flex-flow: [flex-direction] [flex-wrap]
```

Причем второе свойство – *flex-wrap* можно, в принципе, опустить, тогда для него будет использоваться значение по умолчанию – *nowrap*.

```
<html>
    <head>
        <meta charset="utf-8">
        <title>Flexbox в CSS3</title>
        <style>

            .flex-container {
                display: flex;
                border: solid 0.25em #000;
                height:8.25em;
                flex-flow: row wrap;
            }
            .flex-item {
                text-align:center;
                font-size: 1em;
                padding: 1.5em;
                color: white;
                opacity: 0.8;
            }
            .color1 {background-color: #675BA7;}
            .color2 {background-color: #9BC850;}
            .color3 {background-color: #A62E5C;}
            .color4 {background-color: #2A9FBC;}
            .color5 {background-color: #F15B2A;}
        </style>
    </head>
    <body>
        <div class="flex-container">
            <div class="flex-item color1">Flex Item 1</div>
            <div class="flex-item color2">Flex Item 2</div>
            <div class="flex-item color3">Flex Item 3</div>
```

```
            <div class="flex-item color4">Flex Item 4</div>
            <div class="flex-item color5">Flex Item 5</div>
        </div>
    </body>
</html>
```



Свойство *order*

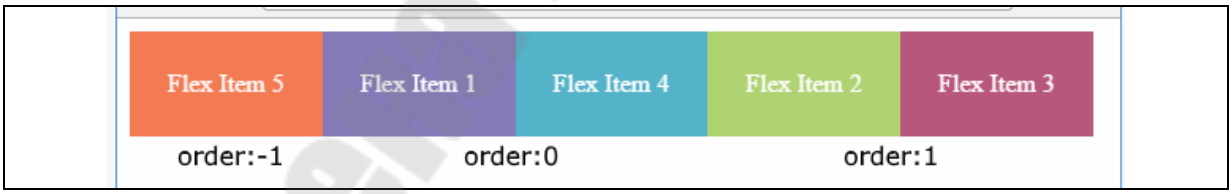
Свойство *order* позволяет установить группу для *flex*-элемента, позволяя тем самым переопределить его позицию внутри *flex*-контейнера. В качестве значения свойство принимает числовой порядок группы. К одной группе может принадлежать несколько элементов.

Например, элементы в группе 0 располагаются перед элементами с группой 1, а элементы с группой 1 располагаются перед элементами с группой 2 и так далее.

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="utf-8">
        <title>Flexbox в CSS3</title>
        <style>

            .flex-container {
                display: flex;
                flex-flow: row wrap;
            }
            .flex-item {
                text-align:center;
                font-size: 1em;
                padding: 1.5em;
                color: white;
                opacity: 0.8;
            }
```

```
            .group1{
                order:-1;
            }
            .group2{
                order:1;
            }
            .color1 {background-color: #675BA7;}
            .color2 {background-color: #9BC850;}
            .color3 {background-color: #A62E5C;}
            .color4 {background-color: #2A9FBC;}
            .color5 {background-color: #F15B2A;}
        </style>
    </head>
    <body>
        <div class="flex-container">
            <div class="flex-item color1">Flex Item 1</div>
            <div class="flex-item color2 group2">Flex Item 2</div>
            <div class="flex-item color3 group2">Flex Item 3</div>
            <div class="flex-item color4">Flex Item 4</div>
            <div class="flex-item color5 group1">Flex Item 5</div>
        </div>
</html>
```

В данном случае определены 3 группы. Первый отображается последний элемент, так как он имеет группу -1:



По умолчанию если у элементов явным образом не указано свойство *order*, то оно имеет значение 0. И последними в данном случае отображаются второй и третий элемент, так как у них свойство *order* равно 1.

Выравнивание элементов. *justify-content*

Иногда можно сталкнуться с тем, что пространство *flex*-контейнеров по размеру отличается от пространства, необходимого для *flex*-элементов. Например:

- *flex*-элементы не используют все пространство *flex*-контейнера;
- *flex*-элементам требуется большее пространство, чем доступно во *flex*-контейнере. В этом случае элементы выходят за пределы контейнера.

Для управления этими ситуациями мы можем применять свойство *justify-content*. Оно выравнивает элементы вдоль основной оси - *main axis* (при расположении в виде строки по горизонтали, при расположении в виде столбца - по вертикали) и принимает следующие значения:

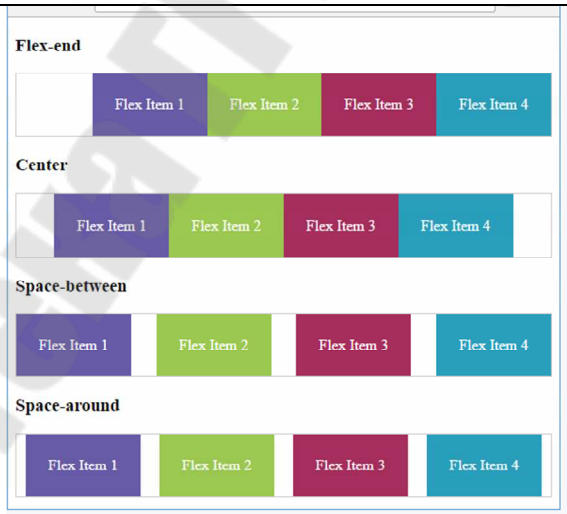
- ***flex-start***: значение по умолчанию, при котором первый элемент выравнивается по левому краю контейнера(при расположении в виде строки) или по верху (при расположении в виде столбца), за ним располагается второй элемент и так далее;
- ***flex-end***: последний элемент выравнивается по правому краю (при расположении в виде строки) или по низу (при расположении в виде столбца) контейнера, за ним выравнивается предпоследний элемент и так далее;
- ***center***: элементы выравниваются по центру;
- ***space-between***: если в стоке только один элемент или элементы выходят за границы *flex*-контейнера, то данное значение аналогично *flex-start*. В остальных случаях первый элемент выравнивается по левому краю (при расположении в виде строки) или по верху (при расположении в виде столбца), а последний элемент - по правому краю контейнера (при расположении в виде строки) или по низу (при расположении в виде столбца). Все оставшееся пространство между ними равным образом распределяется между остальными элементами;
- ***space-around***: если в строке только один элемент или элементы выходят за пределы контейнера, то его действие аналогично значению ***center***. В ином случае элементы равным образом распределяют пространство между левым и правым краем контейнера, а расстояние между первым и последним элементом и границами контейнера составляет половину расстояния между элементами.

Выравнивание для расположения элементов в виде строки:

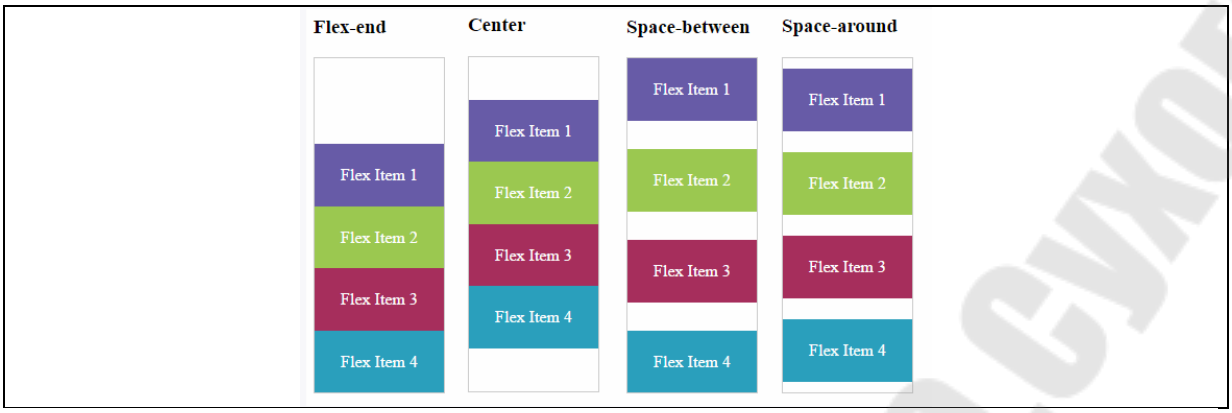
```
<!DOCTYPE html>
<html>
```

```
<head>
    <meta charset="utf-8">
    <title>Flexbox в CSS3</title>
    <style>

        .flex-container {
            display: flex;
            border:1px #ccc solid;
        }
        .flex-end{
            justify-content: flex-end;
        }
        .center{
            justify-content: center;
        }
        .space-between{
            justify-content: space-between;
        }
        .space-around{
            justify-content: space-around;
        }
        .flex-item {
            text-align:center;
            font-size: 1em;
            padding: 1.5em;
            color: white;
        }
        .color1 {background-color: #675BA7;}
        .color2 {background-color: #9BC850;}
        .color3 {background-color: #A62E5C;}
        .color4 {background-color: #2A9FBC;}
        .color5 {background-color: #F15B2A;}
    </style>
</head>
<body>
    <h3>Flex-end</h3>
    <div class="flex-container flex-end">
        <div class="flex-item color1">Flex Item 1</div>
        <div class="flex-item color2">Flex Item 2</div>
        <div class="flex-item color3">Flex Item 3</div>
        <div class="flex-item color4">Flex Item 4</div>
    </div>
```

```
        <h3>Center</h3>
        <div class="flex-container center">
            <div class="flex-item color1">Flex Item 1</div>
            <div class="flex-item color2">Flex Item 2</div>
            <div class="flex-item color3">Flex Item 3</div>
            <div class="flex-item color4">Flex Item 4</div>
        </div>
        <h3>Space-between</h3>
        <div class="flex-container space-between">
            <div class="flex-item color1">Flex Item 1</div>
            <div class="flex-item color2">Flex Item 2</div>
            <div class="flex-item color3">Flex Item 3</div>
            <div class="flex-item color4">Flex Item 4</div>
        </div>
        <h3>Space-around</h3>
        <div class="flex-container space-around">
            <div class="flex-item color1">Flex Item 1</div>
            <div class="flex-item color2">Flex Item 2</div>
            <div class="flex-item color3">Flex Item 3</div>
            <div class="flex-item color4">Flex Item 4</div>
        </div>
    </body>
</html>
```



Выравнивание при расположении в виде столбцов:



Выравнивание элементов. *align-items* и *align-self*

Свойство *align-items* также выравнивает элементы, но уже по поперечной оси (*cross axis*) (при расположении в виде строки по вертикали, при расположении в виде столбца - по горизонтали). Это свойство может принимать следующие значения:

- ***stretch***: значение по умолчанию, при котором *flex*-элементы растягиваются по всей высоте (при расположении в строку) или по всей ширине (при расположении в столбик) *flex*-контейнера;
- ***flex-start***: элементы выравниваются по верхнему краю (при расположении в строку) или по левому краю (при расположении в столбик) ***flex***-контейнера;
- ***flex-end***: элементы выравниваются по нижнему краю (при расположении в строку) или по правому краю (при расположении в столбик) ***flex***-контейнера;
- ***center***: элементы выравниваются по центру *flex*-контейнера;
- ***baseline***: элементы выравниваются в соответствии со своей базовой линией.

Выравнивание при расположении в строку:

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="utf-8">
        <title>Flexbox в CSS3</title>
        <style>

            .flex-container {
                display: flex;
                border:1px #ccc solid;
                height:5em;
```

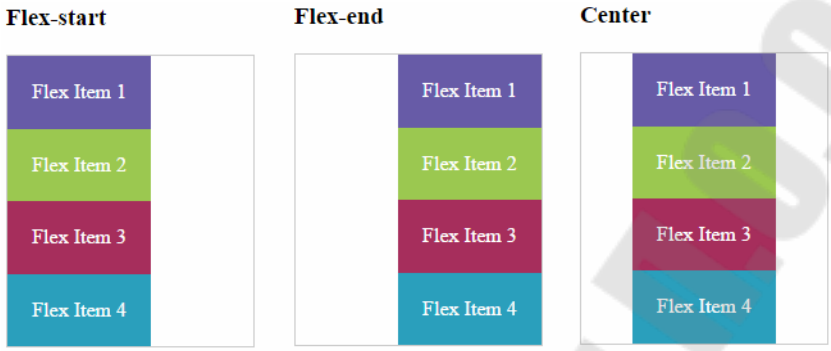
```
            }
            .flex-start{
                align-items: flex-start;
            }
            .flex-end{
                align-items: flex-end;
            }
            .center{
                align-items: center;
            }
            .baseline{
                align-items: baseline;
            }
            .flex-item {
                text-align:center;
                font-size: 1em;
                padding: 1.2em;
                color: white;

            }
            .largest-item{
                padding-top:2em;
            }
            .color1 {background-color: #675BA7;}
            .color2 {background-color: #9BC850;}
            .color3 {background-color: #A62E5C;}
            .color4 {background-color: #2A9FBC;}
        </style>
    </head>

    <body>
        <h3>Flex-start</h3>
        <div class="flex-container flex-start">
            <div class="flex-item color1">Flex Item 1</div>
            <div class="flex-item color2">Flex Item 2</div>
            <div class="flex-item color3">Flex Item 3</div>
            <div class="flex-item color4">Flex Item 4</div>
        </div>
        <h3>Flex-end</h3>
        <div class="flex-container flex-end">
            <div class="flex-item color1">Flex Item 1</div>
            <div class="flex-item color2">Flex Item 2</div>
```

```
            <div class="flex-item color3">Flex Item 3</div>
            <div class="flex-item color4">Flex Item 4</div>
        </div>
        <h3>Center</h3>
        <div class="flex-container center">
            <div class="flex-item color1">Flex Item 1</div>
            <div class="flex-item color2">Flex Item 2</div>
            <div class="flex-item color3">Flex Item 3</div>
            <div class="flex-item color4">Flex Item 4</div>
        </div>
        <h3>Baseline</h3>
        <div class="flex-container baseline">
            <div class="flex-item color1">Flex Item 1</div>
            <div class="flex-item color2 largest-item">Flex
Item 2</div>
            <div class="flex-item color3">Flex Item 3</div>
            <div class="flex-item color4">Flex Item 4</div>
        </div>
</html>
```

Аналогично свойство работает при расположении в столбик. Например, изменим стили *flex*-контейнера следующим образом:

```
.flex-container {
    display: flex;
```

```
    border:1px #ccc solid;
    flex-direction:column;
    width:12em;
}
```



Свойство *align-self*

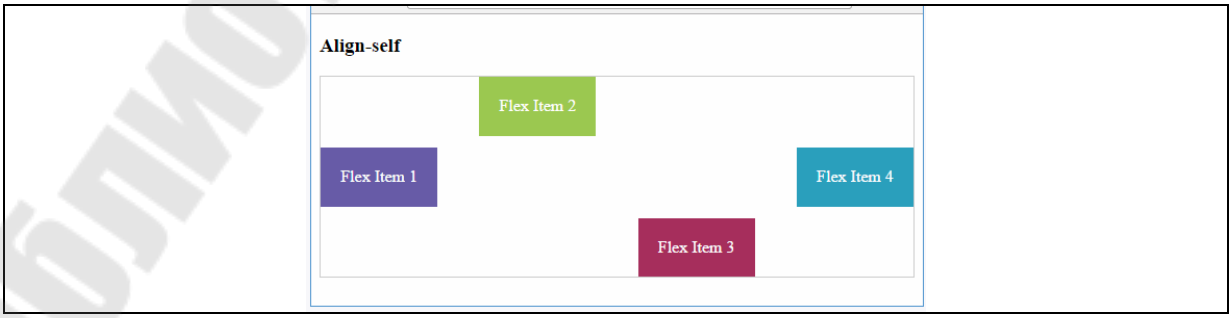
Свойство *align-self* позволяет переопределить значение свойства `align-items` для одного элемента. Оно может принимать те же значения плюс зн-ие "*auto*":

- ***auto***: значение по умолчанию, при котором элемент получает значение от свойства `align-items`, которое определено в *flex*-контейнере. Если в контейнере такой стиль не определен, то применяется значение *stretch*.
- ***stretch*;**
- ***flex-start*;**
- ***flex-end*;**
- ***center*;**
- ***baseline*.**

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="utf-8">
        <title>Flexbox в CSS3</title>
        <style>
            .flex-container {
                display: flex;
                border:1px #ccc solid;
                justify-content: space-between;
```

```
                align-items: stretch;
                height:12em;
            }
            .flex-item {
                text-align:center;
                font-size: 1em;
                padding: 1.2em;
                color: white;
            }
            .item1 {background-color: #675BA7; align-self:
center; }
            .item2 {background-color: #9BC850; align-self:
flex-start;}
            .item3 {background-color: #A62E5C; align-self:
flex-end;}
            .item4 {background-color: #2A9FBC; align-self:
center;}
        </style>
    </head>
    <body>
        <h3>Align-self</h3>
        <div class="flex-container">
            <div class="flex-item item1">Flex Item 1</div>
            <div class="flex-item item2">Flex Item 2</div>
            <div class="flex-item item3">Flex Item 3</div>
            <div class="flex-item item4">Flex Item 4</div>
        </div>
</html>
```

Здесь для ***flex***-контейнера задано растяжение по высоте с помощью значения ***align-items***: ***stretch***;. Однако каждый из элементов переопределяет это поведение:



Выравнивание строк и столбцов. *align-content*

Свойство *align-content* управляет выравниванием рядов (строк и столбцов) во *flex*-контейнере и поэтому применяется, если свойство *flex-wrap* имеет значение *wrap* или *wrap-reverse*. Свойство *align-content* может иметь следующие значения:

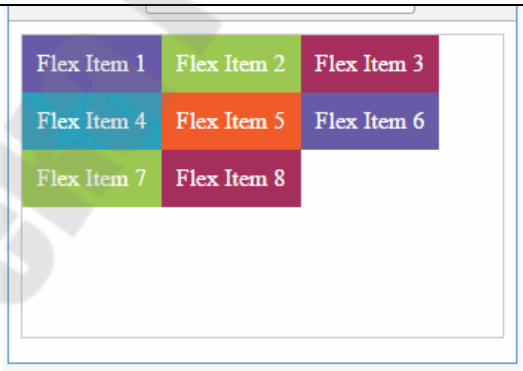
- ***stretch***: значение по умолчанию, при котором строки (столбцы) растягиваются, занимая все свободное место;
- ***flex-start***: строки (столбцы) выравниваются по началу контейнера (для строк - это верхний край, для столбцов - это левый край контейнера);
- ***flex-end***: строки (столбцы) выравниваются по концу контейнера (строки - по нижнему краю, столбцы - по правому краю);
- ***center***: строки (столбцы) позиционируются по центру контейнера;
- ***space-between***: строки (столбцы) равномерно распределяются по контейнеру, а между ними образуются одинаковые отступы. Если же имеющегося в контейнере места недостаточно, то действует аналогично значению *flex-start*;
- ***space-around***: строки (столбцы) равным образом распределяют пространство контейнера, а растояние между первой и последней строкой (столбцом) и границами контейнера составляет половину расстояния между соседними строками (столбцами).

Стоит учитывать, что это свойство имеет смысл, если в контейнере две и больше строки (столбца).

Например, расположение строк в начале контейнера:

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="utf-8">
        <title>Flexbox в CSS3</title>
        <style>
            .flex-container {
                display: flex;
                border:1px #ccc solid;
                flex-wrap: wrap;
                height:200px;
                align-content: flex-start;
            }
```

```
            .flex-item {
                text-align:center;
                font-size: 16px;
                padding: 10px;
                color: white;
            }
            .item1 {background-color: #675BA7;}
            .item2 {background-color: #9BC850;}
            .item3 {background-color: #A62E5C;}
            .item4 {background-color: #2A9FBC;}
            .item5 {background-color: #F15B2A;}
        </style>
    </head>
    <body>
        <div class="flex-container">
            <div class="flex-item item1">Flex Item 1</div>
            <div class="flex-item item2">Flex Item 2</div>
            <div class="flex-item item3">Flex Item 3</div>
            <div class="flex-item item4">Flex Item 4</div>
            <div class="flex-item item5">Flex Item 5</div>
            <div class="flex-item item1">Flex Item 6</div>
            <div class="flex-item item2">Flex Item 7</div>
            <div class="flex-item item3">Flex Item 8</div>
        </div>
</html>
```

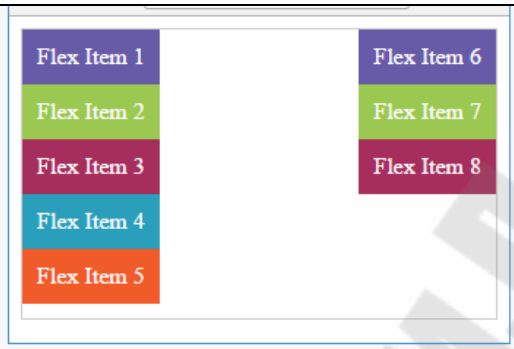


Изменим стиль контейнера:

```
.flex-container {
    display: flex;
    border:1px #ccc solid;
    flex-wrap: wrap;
    height:200px;
```

```
    align-content: space-between;
    flex-direction: column;
}
```

И в этом случае получим ряд столбцов, разделенных отступами:



Управление элементами. *flex-basis*, *flex-shrink* и *flex-grow*

Кроме свойств, устанавливающих выравнивание элементов относительно границ *flex*-контейнера, есть еще три свойства, которые позволяют управлять элементами:

- ***flex-basis***: определяет начальный размер *flex*-элемента;
- ***flex-shrink***: определяет, как *flex*-элемент будет уменьшаться относительно других *flex*-элементов во *flex*-контейнере;
- ***flex-grow***: определяет, как *flex*-элемент будет увеличиваться относительно других *flex*-элементов во *flex*-контейнере.

flex-basis

Flex-контейнер может увеличиваться или уменьшаться вдоль своей центральной оси, например, при изменении размеров браузера, если контейнер имеет нефиксированные размеры. И вместе с контейнером также могут увеличиваться и уменьшаться его *flex*-элементы. Свойство *flex-basis* определяет начальный размер *flex*-элемента до того, как он начнет изменять размер, подстраиваясь под размеры *flex*-контейнера.

Это свойство может принимать следующие значения:

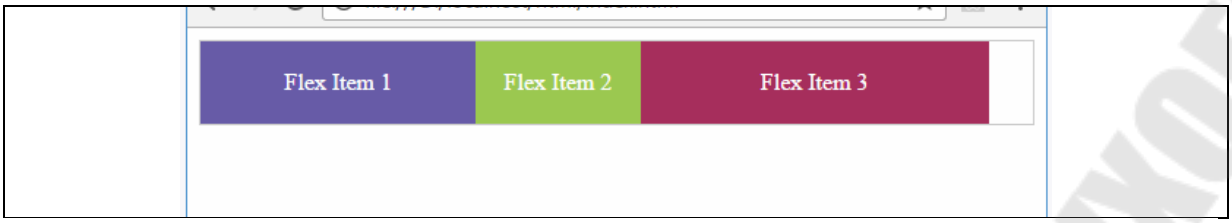
- ***auto***: начальный размер *flex*-элемента устанавливается автоматически;

- ***content***: размер *flex*-элемента определяется по его содержимому, в то же время это значение поддерживается не всеми современными браузерами, поэтому его пока стоит избегать;

- ***числовое значение***: можем установить конкретное числовое значение для размеров элемента.

Например:

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="utf-8">
        <title>Flexbox в CSS3</title>
        <style>
            .flex-container {
                display: flex;
                border:1px #ccc solid;
            }
            .flex-item {
                text-align:center;
                font-size: 1em;
                padding: 1.2em;
                color: white;
            }
            .item1 {background-color: #675BA7; flex-basis:
auto; width:150px;}
            .item2 {background-color: #9BC850; flex-basis:
auto; width:auto;}
            .item3 {background-color: #A62E5C; flex-basis:
200px;width:150px;}
        </style>
    </head>
    <body>
        <div class="flex-container">
            <div class="flex-item item1">Flex Item 1</div>
            <div class="flex-item item2">Flex Item 2</div>
            <div class="flex-item item3">Flex Item 3</div>
        </div>
</html>
```



У первого элемента у свойства *flex-basis* установлено значение *auto*. Поэтому первый элемент в качестве реального значения для ширины будет использовать значение свойства *width*.

У второго элемента у свойства *flex-basis* установлено значение *auto*, однако и свойство *width* имеет значение *auto*. Поэтому реальная ширина элемента будет устанавливаться по его содержимому.

У третьего элемента свойство *flex-basis* имеет конкретное значение, которое и используется. А свойство *width* в этом случае уже не играет никакой роли.

flex-shrink

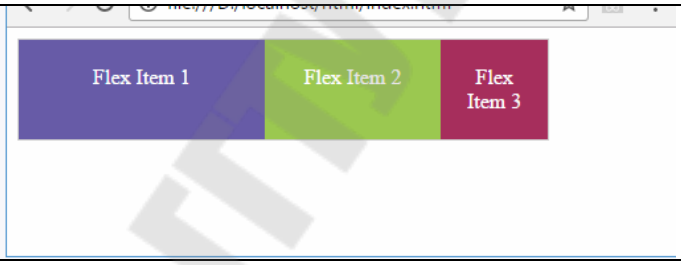
Если *flex*-контейнер имеет недостаточно места для размещения элемента, то дальнейшее поведение этого элемента мы можем определить с помощью свойства *flex-shrink*. Оно указывает, как элемент будет усекаться относительно других элементов.

В качестве значения свойство принимает число. По умолчанию его значение 1.

Рассмотрим действие этого свойства на примере:

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="utf-8">
        <title>Flexbox в CSS3</title>
        <style>
            .flex-container {
                display: flex;
                border:1px #ccc solid;
                width:400px;
            }
            .flex-item {
                text-align:center;
                font-size: 1em;
```

```
                padding: 1.2em;
                color: white;
            }
            .item1 {background-color: #675BA7; flex-basis: 200px;
flex-shrink:1;}
            .item2 {background-color: #9BC850; flex-basis: 200px;
flex-shrink:2;}
            .item3 {background-color: #A62E5C; flex-basis: 200px;
flex-shrink:3;}
        </style>
    </head>
    <body>
        <div class="flex-container">
            <div class="flex-item item1">Flex Item 1</div>
            <div class="flex-item item2">Flex Item 2</div>
            <div class="flex-item item3">Flex Item 3</div>
        </div>
</html>
```



В данном случае начальная ширина каждого элемента равна 200*px*, то есть совокупная ширина составляет 600*px*. Однако ширина *flex*-контейнера составляет всего 400*px*. То есть размер контейнера недостаточен для вмещения в него элементов, поэтому в действие вступает свойство `flex-shrink`, которое определено у элементов.

Для усечения элементов браузер вычисляет коэффициент усечения (*shrinkage factor*). Он вычисляется путем перемножения значения свойства `flex-basis` на `flex-shrink`. Таким образом, для трех элементов получим следующие вычисления:

```
// первый элемент
200px * 1 = 200
// второй элемент
200px * 2 = 400
// третий элемент
```

```
200px * 3 = 600
```

Таким образом, получаем, что для второго элемента коэффициент усечения в два раза больше, чем коэффициент для первого элемента. А для третьего элемента коэффициент больше в три раза, чем у первого элемента. Поэтому в итоге первый элемент при усечении будет в три раза больше, чем третий и в два раза больше, чем второй.

flex-grow

Свойство *flex-grow* управляет расширением элементов, если во *flex*-контейнере есть дополнительное место. Данное свойство во многом похоже на свойство *flex-shrink* за тем исключением, что работает в сторону увеличения элементов.

В качестве значения свойство *flex-grow* принимает положительное число, которое указывает, во сколько раз элемент будет увеличиваться относительно других элементов при увеличении размеров *flex*-контейнера. По умолчанию свойство *flex-grow* равно 0.

Итак, рассмотрим свойство *flex-grow*:

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="utf-8">
        <title>Flexbox в CSS3</title>
        <style>
            .flex-container {
                display: flex;
                border:1px #ccc solid;
            }
            .flex-item {
                text-align:center;
                font-size: 1em;
                padding: 1.3em;
                color: white;
            }
            .item1 {background-color: #675BA7; flex-grow:0;}
            .item2 {background-color: #9BC850; flex-grow:1;}
            .item3 {background-color: #A62E5C; flex-grow:2;}
```

```
        </style>
    </head>
    <body>
        <div class="flex-container">
            <div class="flex-item item1">Flex Item 1</div>
            <div class="flex-item item2">Flex Item 2</div>
            <div class="flex-item item3">Flex Item 3</div>
        </div>
</html>
```

Для каждого элемента есть базовые начальные размеры. Здесь явным образом размеры для элементов не указаны, поэтому размер каждого элемента в данном случае будет складываться из размеров внутреннего содержимого, к которым добавляются внутренние отступы.

По мере растягивания контейнера будут увеличиваться элементы в соответствии со свойством *flex-grow*, которое указано для каждого элемента. Пространство, на которое растягивается контейнер, считается дополнительным пространством.

Так как у первого элемента свойство *flex-grow* равно 0, то первый элемент будет иметь константные постоянные размеры. У второго элемента *flex-grow* равно 1, а третьего – 2.

Таким образом, в сумме они дадут 0 + 1 + 2 = 3. Поэтому второй элемент будет увеличиваться на 1/3 дополнительного пространства,

на которое растягивается контейнер, а третий элемент будет получать 2/3 дополнительного пространства.

Свойство *flex*

Свойство *flex* является объединением свойств *flex-basis*, *flex-shrink* и *flex-grow* и имеет следующий формальный синтаксис:

```
flex: [flex-grow] [flex-shrink] [flex-basis]
```

По умолчанию свойство *flex* имеет значение `0 1 auto`.

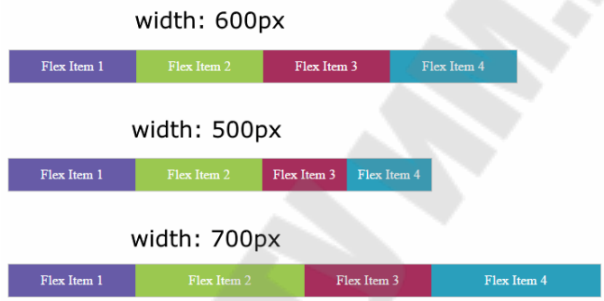
Кроме конкретных значений для каждого из подсвойств мы можем задать для свойства *flex* одно из трех общих значений:

- ***flex: none***: эквивалентно значению `0 0 auto`, при котором *flex*-элемент не растягивается и не усекается при увеличении и уменьшении контейнера;
- ***flex: auto***: эквивалентно значению `1 1 auto;`
- ***flex: initial***: эквивалентно значению `0 1 auto`.

Так, применим свойство *flex*:

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="utf-8">
        <title>Flexbox в CSS3</title>
        <style>
            .flex-container {
                display: flex;
                border:1px #ccc solid;
                width: 600px;
            }
            .flex-item {
                text-align:center;
                font-size: 16px;
                padding: 10px 0;
                color: white;
            }
            .item1 {background-color: #675BA7; width: 150px;
flex: 0 0 auto }
            .item2 {background-color: #9BC850; width: 150px;
flex: 1 0 auto;}
```

```
            .item3 {background-color: #A62E5C; width: 150px;
flex: 0 1 auto;}
            .item4 {background-color: #2A9FBC; width: 150px;
flex: 1 1 auto;}
        </style>
    </head>
    <body>
        <div class="flex-container">
            <div class="flex-item item1">Flex Item 1</div>
            <div class="flex-item item2">Flex Item 2</div>
            <div class="flex-item item3">Flex Item 3</div>
            <div class="flex-item item4">Flex Item 4</div>
        </div>
</html>
```



Здесь каждый элемент имеет начальную ширину в 150 пикселей, так как у всех элементов свойство *flex-basis* имеет значение 0, что в целом для всех элементов будет составлять 600 пикселей.

При сжатии контейнера будут уменьшаться 3-й и 4-й элементы, так как у них свойство *flex-shrink* больше нуля. И так как у обоих элементов это свойство равно 1, то оба элемента будут уменьшаться в равных долях.

При растяжении контейнера будут увеличиваться 2-й и 4-й элементы, так как у этих элементов свойство *flex-grow* больше нуля. И также, так как это свойство равно 1, то эти элементы будут увеличиваться в равных долях.

Многоколоночный дизайн на *Flexbox*

Двуколоночный дизайн

```
<!DOCTYPE html>
<html>
```

```
<head>
    <meta charset="utf-8">
    <title>Flexbox в CSS3</title>
    <style>
        *{
            box-sizing: border-box;
        }
        html,
        body {
            padding: 0;
            margin: 0;
            font-family: verdana, arial, sans-serif;
        }

        body {
            display: flex;
            padding: 1em;
            flex-direction: column;
        }

        .item {
            background-color: #455a64;
            color: #fff;
            font-size: 1.1em;
            padding: 1em;
        }

        .item:nth-child(even) {
            background-color: #607d8b;
        }
        @media screen and (min-width: 600px) {

            body {
                flex-direction: row;
            }
        }
    </style>
</head>
<body>
    <div class="item">
        <h2>Что такое Lorem Ipsum?</h2>
        <p>Lorem Ipsum - это текст-"рыба", часто
```

```
используемый в печати и вэб-дизайне. Lorem Ipsum является
            стандартной "рыбой" для текстов на латинице с
начала XVI века...</p>
        </div>

        <div class="item">
            <h2>Откуда он появился?</h2>
            <p>Многие думают, что Lorem Ipsum - взятый с
потолка псевдо-латинский набор слов, но это не совсем так.
            Его корни уходят в один фрагмент классической
латыни 45 года н.э...</p>
        </div>
    </body>
</html>
```

Здесь *flex*-контейнером является элемент *body*. Так как на мобильных устройствах (особенно смартфонах) размер экрана не такой большой, поэтому по умолчанию устанавливаем расположение элементов в столбик. Однако для устройств с экраном от 600px и выше действует правило *media-query*, которое устанавливает расположение в виде строки.

Трехколоночный режим

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="utf-8">
        <title>Flexbox в CSS3</title>
        <style>
            *{
                box-sizing: border-box;
            }
            html,
            body {
                padding: 0;
                margin: 0;
                font-family: verdana, arial, sans-serif;
            }

            body {
                display: flex;
                padding: 1em;
```

```
                flex-direction: column;
            }

            .item {
                background-color: #455a64;
                color: #fff;
                font-size: 1.1em;
                padding: 1em;
                flex: 1;
            }

            .item:nth-child(1) {
                background-color: #607d8b;
            }
            @media screen and (min-width: 600px) {

                body {
                    flex-direction: row;
                }
                .item:nth-child(2) {
                    order: -1;
                }
            }
        </style>
    </head>
    <body>

        <div class="item">
            <h1>Что такое Lorem Ipsum?</h1>
            <p>Lorem Ipsum - это текст-"рыба", часто
используемый в печати и вэб-дизайне. Lorem Ipsum является
стандартной "рыбой"
            для текстов на латинице с начала XVI века. В то
время некий безымянный печатник создал ...</p>
        </div>

        <div class="item">
            <h3>Классический текст Lorem Ipsum, используемый с
XVI века</h3>
            <p>"Lorem ipsum dolor sit amet, consectetur
adipiscing elit, sed do eiusmod tempor incididunt ut labore et
dolore
```

```
            magna aliqua. Ut enim ad minim veniam, quis
nostrud exercitation ullamco</p>
        </div>
        <div class="item">
            <h3>Где его взять?</h3>
            <p>Есть много вариантов Lorem Ipsum, но
большинство из них имеет не всегда приемлемые модификации,
например,
            юмористические вставки или слова, которые даже
отдалённо не напоминают латынь.</p>
        </div>
    </body>
</html>
```

В отличие от предыдущего примера, здесь добавлен еще один элемент. Особенностью этого примера является то, что столбцы имеют одинаковые размеры. Для этого у них установлено свойство `flex: 1`, то есть при растяжении или уменьшении границ контейнера все элементы будут масштабироваться на равную величину.

И кроме того, при ширине экрана больше 600*px* у второго элемента устанавливается свойство `order: -1`, благодаря чему этот элемент помещается первым.

Подобным образом можем добавить и большее количество столбцов. Но в данном случае по умолчанию столбцы имеют одинаковую ширину.

Но что делать, если один из столбцов (как правило, центральный) должен иметь ширину больше, чем у остальных? Для этого добавим в стили страницы следующее правило:

```
.item:first-child {
    flex: 0 0 50%;
}
```

В этом случае первый элемент всегда будет занимать 50% пространства контейнера.

Макет страницы на *Flexbox*

Теперь рассмотрим создание стандартного макета страницы, который состоит их шапки, футера и центральной части, в которой есть три столбца: основное содержимое и два сайдбара.

<table>
<tr><td colspan="3">Шапка – Header</td></tr>
<tr><td>Navigation</td><td>Основное содержимое</td><td>Sidebar</td></tr>
<tr><td colspan="3">Footer</td></tr>
</table>

Для этого определим следующую веб-страницу:

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="utf-8">
        <meta name="viewport" content="width=device-width" />
        <title>Flexbox в CSS3</title>
        <style>
            *{
                box-sizing: border-box;
            }
            html, body {
                padding: 0;
                margin: 0;
                font-family: verdana, arial, sans-serif;
            }

            body {
                color: #fff;
                font-size: 1.1em;
                padding: 1em;
                display: flex;
                flex-direction: column;
            }
```

```
            main {
                display: flex;
                flex-direction: column;
            }

            article {
                background-color: #546e7a;
                flex: 2 2 12em;
                padding: 1em;
            }

            nav, aside {
                flex: 1;
                background-color: #455a64;
            }

            nav {
                order: -1;
            }

            header, footer {
                flex: 0 0 5em;
                background-color: #37474f;
            }

            @media screen and (min-width: 600px) {

                body{
                    min-height: 100vh;
                }
                main {
                    flex-direction: row;
                    min-height: 100%;
                    flex: 1 1 auto;
                }
            }
        </style>
    </head>
    <body>
        <header>
            <p>Header</p>
        </header>
```

```
        <main>
            <article>
                <h1>Что такое Lorem Ipsum?</h1>
                <p>Lorem Ipsum - это текст-"рыба", часто
используемый в печати и вэб-дизайне. Lorem Ipsum является
стандартной
                "рыбой" для текстов на латинице с начала XVI
века. В то время некий безымянный печатник создал большую
коллекцию
                размеров и форм шрифтов, используя Lorem Ipsum
для распечатки образцов. Lorem Ipsum не только успешно пережил
без
                заметных изменений пять веков, но и перешагнул
в электронный дизайн. Его популяризации в новое время</p>
            </article>
            <nav>
                <p>Navigation</p>
            </nav>
            <aside>
                <p>Sidebar</p>
            </aside>
        </main>
        <footer>
            <p>Footer</p>
        </footer>
    </body>
</html>
```

Итак, *flex*-контейнером верхнего уровня здесь является элемент *body*. Его *flex*-элементами являются *header*, *main* и *footer*. *Body* располагает все свои элементы сверху вниз в столбик. Здесь также стоит отметить, что при ширине от 600*px* и выше для заполнения всего пространства браузера у *body* устанавливается стиль `height: 100vh;`.

Элементы *header* и *footer* аналогичны. Их свойство `flex: 0 0 5em;` указывают, что при любом изменении контейнера эти элементы будут иметь размер в 5*em*. То есть они имеют статический размер.

Более сложным является элемент *main*, который определяет основное содержимое. При этом будучи *flex*-элементом, он также является *flex*-контейнером для вложенных элементов и управляет их позиционированием. При ширине браузера до 600*px* он располагает элементы в столбик, что очень удобно на мобильных устройствах.

При ширине от 600*px* вложенные элементы *nav*, *article* и *aside* располагаются в виде строки. И поскольку при такой ширине браузера родительский элемент *body* заполняет по высоте все пространство браузера, то для заполнения всей высоты контейнера *body* при его изменении у элемента *main* устанавливается свойство `flex: 1 1 auto;`.

У вложенных в *main flex*-элементов стоит отметить, что элемент навигации *nav* и элемент сайдбара *aside* будут иметь одинаковые размеры при масштабировании контейнера. А элемент *article*, содержащий основное содержимое, будет соответственно больше. При этом хотя *nav* определен после элемента *article*, но благодаря установке свойства `order: -1` блок навигации будет стоять до блока *article*.

7 ЯЗЫК ПРОГРАММИРОВАНИЯ *JAVASCRIPT*

7.1 Общие сведения о *JavaScript*

С помощью *JavaScript* создаются интерактивные веб-страницы.

Интерактивные страницы могут взаимодействовать с пользователем (выводить сообщения, изменять содержимое после определенных действий и т.д.)

JavaScript встраивается прямо в веб-страницы и исполняется браузером во время их загрузки.

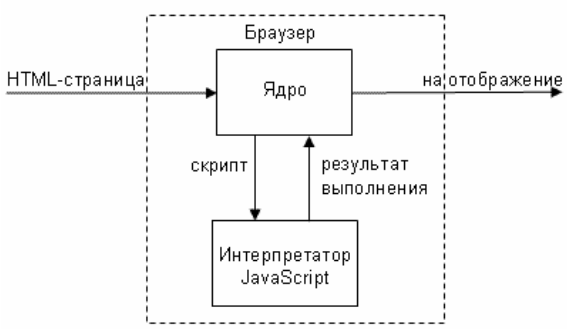
Так как браузер является клиентом в клиент-серверной технологии, то *JavaScript* называют также *браузерным* или *клиентским языком программирования*.

Роль *JavaScript* заключается в расширении возможностей пользователя, облегчая для него получение и передачу информации.

Фрагменты кода, написанные на языке *JavaScript*, отделяются от остальной части *HTML*-документа тегами <*script*> и </*script*>.

Рассмотрим работу браузера со скриптом.

На рисунке показано, что в браузер входят программы ядра и интерпретатора *JavaScript*.



Ядро выполняет основную функцию браузера, то есть отображает *HTML*-страницу на экране компьютера. Интерпретатор (англ. *Interpreter* – истолкователь, устный переводчик) переводит и выполняет скрипт строка за строкой.

Ядро загружает *HTML*-страницу и анализирует её код.

Если встречаются теги *<script>...</script>*, то содержащийся в них *JavaScript*-код не обрабатывается ядром, а отправляется в интерпретатор.

Интерпретатор выполняет скрипт и возвращает результат. Страница изменяется в зависимости от результата и далее ядро посылает её на отображение.

Теги *<script>...</script>* являются сигналом о том, что заключённый в них код нужно обрабатывать не как обычные теги, а как скрипт.

Таким образом, теги *<script>...</script>* на странице заменяются результатом выполнения скрипта.

Возможности *JavaScript*:

- динамически изменять содержимое веб-страниц;
- привязывать к элементам обработчики событий (функции которые выполнят свой код только после того, как совершатся определенные действия);
- выполнять код через заданные промежутки времени;
- управлять поведением браузера (открывать новые окна, загружать указанные документы и т.д.);
- создавать и считывать *cookies*;
- определять, какой браузер использует пользователь (также можно определить ос*m*, разрешение экрана, предыдущие страницы, которые посещал пользователь и т.д.);

– проверять данные форм перед отправкой их на сервер и многое другое.

Включение *JavaScript*-сценариев в *html*-документы осуществляется тремя способами.

1. Включение в секцию *<head> </head>*.
2. Включение в секцию *<body> </body>*.

Пример:

```
<!DOCTYPE html>
<html>
<head>
      <meta charset="utf-8" />
      <title> Первая программа</title>
    <script>
            document.write("Начальный текст");
    </script>
</head>
<body>
     <h2>Заголовок</h2>
    <script>
             document.write("Текст…   Текст");
    </script>
</body>
</html>
```

3. В секции *<head> </head>* размещается ссылка на отдаленный *.js* файл. Ссылка на внешний файл, который находится в папке *js*, выглядит так:

```
<script src="js/myscript.js""></script>
```

Вынесение кода во внешние файлы имеет ряд преимуществ:

– можно повторно использовать один и тот же код на нескольких веб-страницах;

– внешние файлы *javascript* бразуер может кэшировать, за счет этого при следующем обращении к странице браузер снижает нагрузка на сервер, а браузеру надо загрузить меньший объем информации;

– код веб-страницы становится "чище". Он содержит только *html*-разметку, а код поведения хранится во внешних файлах. В итоге можно отделить работу по созданию кода *html*-страницы от написания кода *javascript*.

Поэтому, как правило, предпочтительнее использовать код *javascript* во внешних файлах, а не в прямых вставках на веб-страницу с помощью элемента *script*.

Ввод и вывод данных

В JavaScript предусмотрены довольно скудные средства для ввода и вывода данных. Это вполне оправданно, поскольку *JavaScript* создавался в первую очередь как язык сценариев для веб-страниц.

Можно воспользоваться тремя стандартными методами для ввода и вывода данных: *alert*(), *prompt*(), *confirm*().

alert

Данный метод позволяет выводить диалоговое окно с заданным сообщением и кнопкой ОК. Синтаксис применения метода *alert*:

```
alert(“сообщение”);
```

Сообщение представляет собой данные любого типа: последовательность символов, заключенную в кавычки, число (в кавычках или без них), переменную или выражение. Пример работы метода *alert* представлен на рисунке.

confirm

Метод *confirm* позволяет вывести диалоговое окно с сообщением и двумя кнопками – ОК и Отмена (*Cancel*).

В отличие от метода *alert,* этот метод возвращает логическую величину, значение которой зависит от того, на какой из двух кнопок щелкнул пользователь. Если он щелкнул на кнопке ОК, то

возвращается значение *true* (истина), если же он щелкнул на кнопке Отмена, то возвращается значение *false* (ложь).

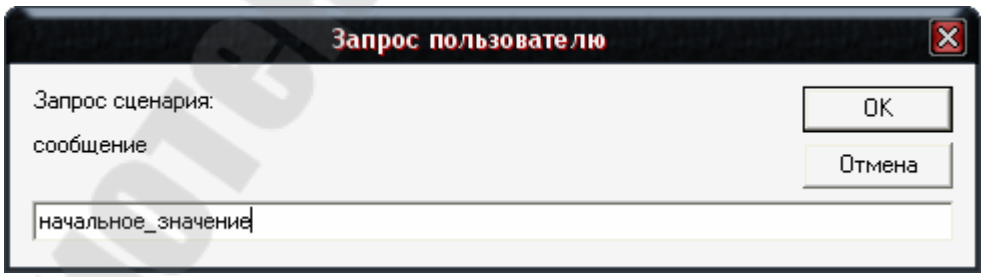
Синтаксис применения метода *confirm*:

```
confirm(“сообщение”);
```

prompt

Метод *prompt* позволяет вывести на экран диалоговое окно с сообщением, а также с текстовым полем, в которое пользователь может ввести данные. Кроме того, в этом окне предусмотрены две кнопки: ОК и Отмена (*Cancel*). Данный метод принимает два параметра: сообщение и значение, которое должно появиться в текстовом поле ввода данных по умолчанию. Если пользователь щелкнет на кнопке ОК, то метод вернет содержимое поля ввода данных, если он щелкнет на кнопке Отмена, то возвращается логическое значение *false* (ложь). Синтаксис:

```
prompt(“сообщение”,”начальное_значение”);
```



JavaScript использует Си-подобный синтаксис, который требует использовать точку с запятой для разграничения операторов.

В *JavaScript* установка точки с запятой в конце каждого оператора не обязательна, вместо нее можно выполнить перевод

строки, и такой код будет считаться рабочим для веб-барузера, но такой подход может повлечь за собой ошибки.

Чтобы улучшить читабельность кода и снизить число возможных ошибок, рекомендуется определять каждую инструкцию *javascript* на отдельной строке и завершать ее точкой с запятой, исключения могут быть только для операторов *for*, *function*, *if*, *switch*, *try* и *while*.

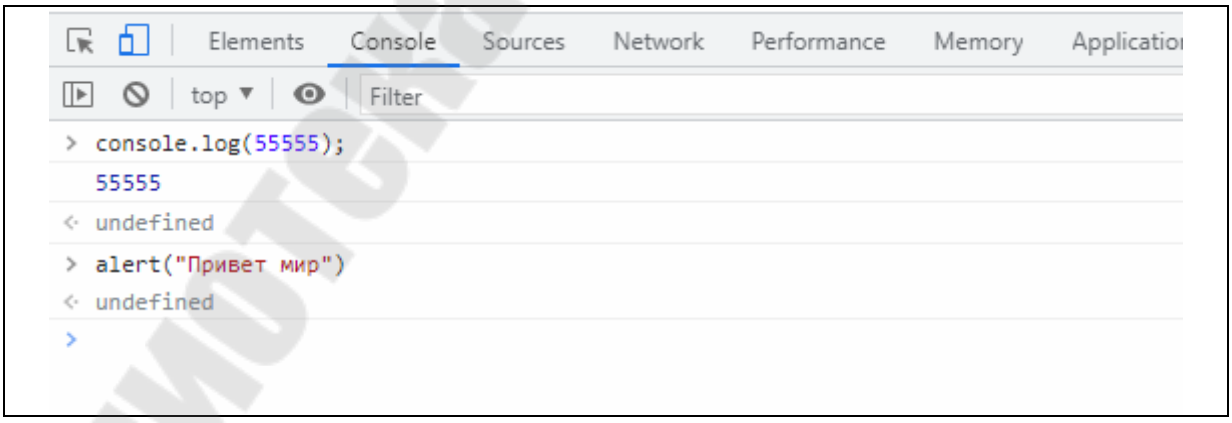
Для пояснения программного кода используются комментарии. В *JavaScript* комментарии могут находиться в любом месте скрипта и могут быть однострочными и многострочными.

```
// Этот комментарий занимает всю строку
/*
Это -
       многострочный комментарий.
*/
```

Консоль браузера и *console.log*

Незаменимым инструментом при работе с *JavaScript* является консоль браузера, которая позволяет производить отладку программы. Во многих современных браузерах есть подобная консоль. Пользователь может напрямую вводить в консоль браузера выражения *JavaScript*, и они будут выполняться.

Для вывода различного рода информации в консоли браузера используется специальная функция ***console.log*()**.



Для хранения данных в *JavaScript* используются **переменные**. Переменные предназначены для хранения каких-нибудь временных данных или таких данных, которые в процессе работы могут менять свое значение.

При названии переменных надо учитывать, что *JavaScript* является регистрозависимым языком, то есть в следующем коде объявлены две разные переменные:

```
var UserName;
var Username;
```

Для создания переменной в *JavaScript* используется ключевое слово *let*:

```
let  message;
message = 'Hello';
```

В старых скриптах можно найти другое ключевое слово: ***var*** вместо ***let***. Ключевое слово ***var*** – почти то же самое, что и ***let***. Оно объявляет переменную, но немного по-другому, «устаревшим» способом.

Для объявления констант, то есть, неизменяемой переменной, используется ключевое слово ***const***:

```
const PI = 3.1415;
```

Устаревшее ключевое слово *var*

Существует два основных отличия *var* от *let*/*const*:

Переменные *var* не имеют блочной области видимости, они ограничены, как минимум, телом функции.

Объявления (инициализация) переменных *var* производится в начале исполнения функции (или скрипта для глобальных переменных).

Блочная область видимости – это удобно.

Поэтому *let* и *const* были введены в стандарт и сейчас являются основным способом объявления переменных.

Область действия (или видимости) переменных (scope) – это участки внутри программы, в которых возможна ссылка на переменную.

Переменная, объявляемая внутри функции, является **локальной**.

К значениям этой переменной может иметь доступ только данная функция. При каждом вызове функции все локальные переменные создаются, а после возврата функции – разрушаются.

Если в другой функции присутствует объявление переменной с таким же именем, она рассматривается как совершенно другая переменная. Каждая из переменных адресует свой блок памяти.

Если требуется, чтобы какая-то переменная совместно использовалась в более чем одной функции, следует объявить ее вне функций (но, конечно же, внутри пары дескрипторов <*script*>). В этом случае данная переменная будет доступна в любой части всего приложения, в том числе и во всех функциях. Все переменные, которые объявлены вне функций, являются **глобальными**. Рекомендуется объявлять глобальные переменные в разделе <*head*> *HTML*-страницы, тем самым гарантируя, что они загрузятся раньше любой другой части приложения.

Сокрытие переменных

Предположим, у нас есть две переменных – одна глобальная, а другая локальная, которые имеют одинаковое имя. В этом случае в функции будет использоваться та переменная, которая определена непосредственно в функции. То есть локальная переменная скроет глобальную.

var* или *let

При использовании оператора *let* каждый блок кода определяет новую область видимости, в которой существует переменная. Например, можно одновременно определить переменную на уровне блока и на уровне функции.

С помощью оператора *var* нельзя определить одновременно переменную с одним и тем же именем и в функции, и в блоке кода в этой функции. Т.е. с помощью *var* можно определить переменную с одним именем либо на уровне функции, либо на уровне блока кода.

Все, что относится к оператору *let*, относится и к оператору *const*, который позволяет определить константы. Блоки кода задают область видимости констант, а константы, определенные на вложенных блоках кода, скрывают внешние константы с тем же именем:

Необъявленные переменные

Если не используем никакое ключевое слово при определении переменной в функции, то такая переменная будет глобальной.

Все используемые данные в *javascript* имеют определенный тип.

В *JavaScript* имеется восемь типов данных:

- *String* – представляет строку;
- *Number* – представляет числовое значение;
- *BigInt* – предназначен для представления очень больших целых чисел;
- *Boolean* – представляет логическое значение *true* или *false;*
- *Undefined* – представляет одно специальное значение – *undefined* и указывает, что значение не установлено;
- *Null* – представляет одно специальное значение – *null* и указывает на отсутствие значения;
- *Symbol* – представляет уникальное значение, которое часто применяется для обращения к свойствам сложных объектов;
- *Object* – представляет комплексный объект.

Первые семь типов представляют примитивные типы данных. Последний тип – *Object* представляет сложный, комплексный тип данных, который состоит из значений примитивных типов или других объектов.

Интерполяция

Использование косых кавычек позволяет применять такой прием как интерполяция – встраивать данные в строку. Например:

```
let user = "Tom";
let age = 37;
let isMarried = false;
let text = `Name: ${user} Age: ${age} IsMarried: ${isMarried}`;

console.log(text);    // Name: Tom   Age: 37   IsMarried: false
```

Для встраивания значений выражений (например, значений других переменных и констант) в строку перед выражением ставится знак доллара $, после которого в фигурных скобках указывается выражение. Так, в примере выше, ${*user*} означает, что в этом месте строки надо встроить значение переменной *user*.

Подобным образом можно встраивать и больше количество данных.

Оператор ***typeof*** позволяет увидеть, какой тип данных сохранён в переменной:

- имеет две формы: ***typeof x*** или ***typeof*(*x*)**.
- Возвращает строку с именем типа. Например, "***string***".
- Для ***null*** возвращается "***object***" – это ошибка в языке, на самом деле это не объект.

Часто значения, которые заносятся в переменную, являются результатом вычисления выражения. Выражение представляет собой набор операндов и операторов, выполняющих действия над операндами.

Следует, однако, отметить, что язык *JavaScript* является интерпретируемым и слаботипизированным, в отличии от компилируемых. Тип переменной определяется по введенному значению или результату вычисления выражения, поэтому в процессе выполнения сценария тип переменной может меняться. Следует внимательно следить за преобразованием типов.

Для преобразования строк в числа в *JavaScript* предусмотрены встроенные функции *parseInt*() и *parseFloat*().

Функция ***parseInt***(строка, основание) преобразует указанную в параметре строку в целое число в системе счисления по указанному основанию (8, 10 или 16). Если основание не указано, то предполагается 10, то есть десятеричная система счисления.

Функция ***parseFloat***(строка) преобразует указанную строку в число с плавающей разделительной (десятичной, основание) точкой:

```
parseInt("3.14")          // результат= 3
parseInt("Вася")          // результат = NaN, то есть
                          //не является числом
parseInt("0xFF", 16)      // результат = 255
parseFloat("3.l4")        // результат= 3.14
parseFloat("-7.875")      // результат = -7.875
```

7.2 Основные конструкции языка *JavaScript*

Условные конструкции

Условные конструкции позволяют выполнить те или иные действия в зависимости от определенных условий.

Тернарная операция состоит из трех операндов и имеет следующее определение:

```
[первый операнд -условие] ? [второй операнд] : [третий
операнд]
```

В зависимости от условия тернарная операция возвращает второй или третий операнд: если условие равно ***true***, то возвращается второй операнд; если условие равно ***false***, то третий.

Эта операция используется для создания двухшагового процесса.

Оператор *if*

Оператор ***if*** относится к числу наиболее популярных. Оператор ***if*** применяется следующим образом:

```
if (условие) {
          [операторы]
}
```

В качестве условия может указываться любое логическое выражение. Если результат условия равен ***true***, выполняются операторы, продолжается выполнение остального программного кода. Если условие возвращает ***false***, операторы игнорируются.

Конструкция *if..else*

Иногда одной конструкции ***if*** оказывается недостаточно. Часто требуется зарезервировать набор операторов, которые будут выполняться в случае, когда условное выражение возвращает ***false***. Это можно сделать, добавив еще один блок непосредственно после блока ***if***:

```
if (условие) {
    [операторы]
}
  else
   {
     [операторы]
   }
```

Конструкция *switch..case*

Конструкция ***switch..case*** является альтернативой использованию конструкции ***if..else if..else*** и позволяет обработать сразу несколько условий.

После ключевого слова ***switch*** в скобках идет сравниваемое выражение. Значение этого выражения последовательно сравнивается со значениями, помещенными после оператора с***ase***. И если совпадение будет найдено, то будет выполняться определенный блок с***ase***.

В конце каждого блока с***ase*** ставится оператор ***break***, чтобы избежать выполнения других блоков. Если необходимо обработать ситуацию, когда совпадения не будет найдено, то можно добавить блок ***default***.

```
const nom = 300;
switch(nom){

    case 100 :
        console.log("Доход равен 100");
        break;
    case 200 :
        console.log("Доход равен 200");
        break;
    case 300 :
        console.log("Доход равен 300");
        break;
    default:
        console.log("Доход неизвестен");
        break;
}
```

try..catch

Оператор *try..catch* используется для замены заданного по умолчанию способа обработки ошибок.

Конструкция *try ... catch* позволяет проверять блок кода на наличие ошибок. Блок *try* содержит код, который будет исполняться и проверяться, а блок *catch* содержит код, который будет выполнен при возникновении ошибок. Синтаксис:

```
try
    {   //Запустить некоторый код здесь   }
catch(err)
```

```
      {   //Обработка ошибок здесь   }
```

Циклы

Циклы позволяют в зависимости от определенных условий выполнять некоторое действие множество раз. В *JavaScript* имеются следующие виды циклов:

- *for*;
- *for..in*;
- *for..of*;
- *while*;
- *do..while*.

Цикл *for*

Цикл *for* имеет следующее определение:

```
for ([инициализация счетчика]; [условие];[изменение
счетчика]){
        // действия
  }
```

Цикл *for..in*

Цикл *for..in* предназначен для перебора массивов и объектов.

Синтаксис:

```
for (индекс in объект) {
    // действия
}
```

Цикл *for...of*

Цикл *for...of* похож на цикл *for...in* и предназначен для перебора коллекций, например, массивов:

```
for (индекс of массив) {
       // действия
}
```

Цикл *while*

Цикл ***while*** выполняется до тех пор, пока некоторое условие истинно. Его определение:

```
while (условие) {
      // действия
}
```

Цикл *do..while*

В цикле ***do*** сначала выполняется код цикла, а потом происходит проверка условия в инструкции ***while***. И пока это условие истинно, цикл повторяется. Его определение:

```
do {
    // действия
}
while(условие)
```

Операторы *continue* и *break*

Иногда бывает необходимо выйти из цикла до его завершения. В этом случае необходимо воспользоваться оператором ***break***. Если необходимо просто пропустить итерацию, но не выходить из цикла, можем применять оператор ***continue***.

7.3 Использование функций в *JavaScript*

Функции представляют собой набор инструкций, выполняющих определенное действие или вычисляющих определенное значение.

Синтаксис определения функции:

```
function имя_функции([параметр [, ...]]){
// Инструкции
}
```

Определение функции начинается с ключевого слова ***function***, после которого следует имя функции. Наименование функции подчиняется тем же правилам, что и наименование переменной: оно может содержать только цифры, буквы, символы подчеркивания и доллара ($) и должно начинаться с буквы, символа подчеркивания или доллара.

После имени функции в скобках идет перечисление параметров. Даже если параметров у функции нет, то просто идут пустые скобки. Затем в фигурных скобках идет тело функции, содержащее набор инструкций.

Чтобы выполнить тело функции, необходимо вызвать функцию с передачей параметров или без параметров:

```
function sum(a, b, c){
      let d = a + b + c;
      console.log(d);

sum(1, 2, 3);
sum();
```

В отличии от компилируемых языков программирование, где количество фактических параметров, передаваемых при вызове функции, и количество формальных параметров, описанных при объявлении, должны совпадать, *JavaScript* позволяет не соблюдать это правило. Если для параметров не передается значение, то по умолчанию они имеют значение "*undefined*".

Для определения количества переданных параметров и доступа к их значениям при вызове функции создается массив *arguments*, доступный внутри тела функции.

Есть способ определения значения для параметров по умолчанию:

```
function sum(a = 1, b = 2, c = 3){
      let d = a + b + c;
      console.log(d);
}
```

С помощью ***spread***-оператора можем указать, что с помощью параметра можно передать переменное количество значений:

```
function display(season, ...temps){
     console.log(season);
for(index in temps){
     console.log(temps[index]);
}
     }
display("Весна", -2, -3, 4, 2, 5);
display("Лето", 20, 23, 31);
```

Функция может возвращать результат. Для этого используется оператор ***return***:

```
function sum(a, b){
     return a + b;
}

console.log(sum(1, 2);
```

Функции могут выступать в качестве параметров других функций.

JavaScript позволяет определять функции внутри функций и возвращать результат в виде функции.

Среди функций отдельно можно выделить рекурсивные функции. Их суть состоит в том, что функция вызывает саму себя.

Рассмотрим, например, функцию, определяющую факториал числа:

```
function getFactorial(n){
    if (n === 1){
  return 1;
}
     else{
  return n * getFactorial(n - 1);
}
}
let result = getFactorial(4);
console.log(result);                  // 24
```

7.4 Работа с массивами в *JavaScript*

Для работы с наборами данных предназначены массивы. Для создания массива применяется литерал массива или конструкция ***new Array***():

```
let array_name1 = [item1, item2, ...];
let array_name2 = new Array([item1, item2, ...]);
```

Для повышения производительности и читабельности программного кода рекомендуется использовать литерал массива.

Для получения доступа к элементам массива используется индекс. Индексация элементов начинается с нуля:

```
let cars = ["Saab", "Volvo", "BMW"];
console.log(cars[0]);        // Saab
```

Индекс используется как для чтения, так и для записи элемента массива. Причем в отличие от других языков, таких как ***C#*** или ***Java***, можно установить элемент, который изначально не установлен:

```
let cars = ["Saab", "Volvo", "BMW"];
cars[10] = "Toyota"
console.log(cars[10]);      // Toyota
console.log(cars[3]);       // undefined
```

Также стоит отметить, что в отличие от ряда языков программирования в *JavaScript* массивы не являются строго типизированными, один массив может хранить данные разных типов:

```
var objects = ["Tom", 12, true, 3.14, false];
console.log(objects.toString());
```

Массивы могут быть одномерными и многомерными. Каждый элемент в многомерном массиве может представлять собой отдельный массив.

```
const students = [
    ["Иванов", 20, 5.5],
    ["Петров", 18, 8.2],
    ["Сидоров", 21, 7.8]
];
students[0][1] = 19;         // присваиваем отдельное значение
console.log(students[0][1]);  // 56
```

В языке *JavaScript* все свойства и методы обработки массивов собраны в глобальном объекте *Array.prototype*, от которого автоматически наследуются все создаваемые массивы.

Все массивы обладают свойством *length*, которой устанавливает или возвращает количество элементов в массиве:

```
let cars = ["Saab", "Volvo", "BMW"];
```

```
console.log(cars.length);   // 3
cars.length = 5;
console.log(cars[4]);      // undefined
```

Методы массивов

Для добавления и удаления элементов массива используются следующие методы:

- ***push***(...***items***) – добавляет элементы в конец,
- ***pop***() – извлекает элемент из конца,
- ***shift***() – извлекает элемент из начала,
- ***unshift***(...***items***) – добавляет элементы в начало.

Для удаления элемента массива можно использовать оператор ***delete***. Однако этот оператор удаляет только значение элемента с заданным ключом без переиндексации:

```
let cars = ["Saab", "Volvo", "BMW"];
delete cars[1];
console.log(cars.length);   // 3
console.log(cars[1]);      // undefined
```

Универсальный метод ***splice***() используется для добавления, удаления и замены элементов массива:

```
splice(index[, deleteCount, elem1, ..., elemN])
```

Он начинает с позиции *index* удалять *deleteCount* элементов и вставлять *elem*1, ..., *elemN* на их место. Возвращает массив из удалённых элементов.

Метод ***slice*()** возвращает новый массив, в который копирует элементы, начиная с индекса *start* и до *end* (не включая *end*). Оба индекса *start* и *end* могут быть отрицательными. В таком случае отсчёт будет осуществляться с конца массива:

```
slice([start], [end])
```

Метод ***forEach***() позволяет запускать функцию для каждого элемента массива. Его синтаксис:

```
forEach(function(item, index, array) {
```

```
// ... делать что-то с item
});
```

Функция обратного вызова (***callback***) вызывается по очереди для каждого элемента массива и принимает следующие парметра:

- ***item*** – очередной элемент;
- ***index*** – его индекс;
- ***array*** – сам массив.

Для поиска элементов в массиве используются следующие методы:

- ***indexOf***(***item***, ***from***) ищет *item*, начиная с индекса *from*, и возвращает индекс, на котором был найден искомый элемент, в противном случае -1.
- ***lastIndexOf***(***item***, ***from***) – то же самое, но ищет справа налево.
- ***includes***(***item***, ***from***) – ищет *item*, начиная с индекса *from*, и возвращает *true*, если поиск успешен.

Методы ***find***, ***findIndex*** и ***filter*** в качестве условия поиска используют функцию-предикат:

```
let result = arr.find(function(item, index, array) {
   // если true - возвращается текущий элемент и
   //перебор прерывается
   // если все итерации оказались ложными, возвращается
   //undefined
});

let result = arr.findIndex(function(item, index, array) {
/* если true - возвращается индекс, на котором был найден
элемент, и перебор прерывается  если все итерации оказались
ложными, возвращается -1   */
});

let results = arr.filter(function(item, index, array) {
/* если true - элемент добавляется к результату, и перебор
продолжается,  возвращается пустой массив в случае, если
ничего не найдено  */
});
```

Метод ***map***() является одним из наиболее полезных и часто используемых. Он вызывает функцию для каждого элемента массива и возвращает массив результатов выполнения этой функции:

```
let result = arr.map(function(item, index, array) {
    // возвращается новое значение вместо элемента
});
```

Метод ***sort***(***fn***) сортирует массив на месте, меняя в нём порядок элементов. Он возвращает отсортированный массив, но обычно возвращаемое значение игнорируется, так как изменяется сам массив.

Полный список методов есть в справочнике ***MDN***.

Работа со строками в *JavaScript*

В *JavaScript* любые текстовые данные являются строками. Не существует отдельного типа «символ», который есть в ряде других языков.

Внутренний формат для строк – всегда *UTF*-16, вне зависимости от кодировки страницы.

Для задания строк в *JavaScript* есть разные типы кавычек. Строку можно создать с помощью одинарных, двойных либо обратных кавычек:

```
let single = 'single-quoted';
let double = "double-quoted";
let backticks = `backticks`;
```

Одинарные и двойные кавычки работают, по сути, одинаково, а если использовать обратные кавычки, то в такую строку сможем вставлять произвольные выражения, обернув их в ${…}:

```
function sum(a, b) {
    return a + b;
}
console.log(`1 + 2 = ${sum(1, 2)}.`);      // 1 + 2 = 3
```

Для определения длины строки используется свойство *length*.

Чтобы получить символ, который занимает позицию *pos*, можно использовать квадратные скобки: *[pos]*, а также можно использовать метод *charAt*(*pos*). Первый символ занимает нулевую позицию:

```
let str = `Hello`;
console.log(str.length);      //5

// получаем первый символ
console.log(str[0]);            // H
console.log(str.charAt(0));     // H
```

Следует обратить внимание на то, что содержимое строки в *JavaScript* нельзя изменить.

```
let str = `Hello`;
console.log(str);       // Hello
str[0] = "J";
console.log(str);       // Hello
```

Методы ***toLowerCase***() и ***toUpperCase***() меняют регистр символов:

```
alert( 'Interface'.toUpperCase() );    // INTERFACE
alert( 'Interface'.toLowerCase() );     // interface
```

Для поиска подстроки используются следующие методы:

- ***indexOf***(***substr***, ***pos***) ищет подстроку *substr* в строке *str*, начиная с позиции ***pos***, и возвращает позицию, на которой располагается совпадение, либо -1 при отсутствии совпадений;
- ***lastIndexOf***(***substr***, ***position***) ищет подстроку с конца строки к её началу;
- ***includes***(***substr***, ***pos***) возвращает *true*, если в строке *str* есть подстрока *substr*, либо *false*, если нет;
- ***startsWith***(***substr***), ***endsWith***(***substr***) проверяют, соответственно, начинается ли и заканчивается ли строка определённой строкой.

Для получения подстроки в *JavaScript* есть 3 метода:

- ***slice***(***start [***, ***end]***) возвращает часть строки от *start* до (не включая) *end*, если *end* отсутствует, *slice* возвращает символы до конца строки;
- ***substring***(***start [***, ***end]***) возвращает часть строки между *start* и *end*;

– ***substr***(***start [***, ***length]***) возвращает часть строки от *start* длины *length*.

Для сравнение строк используются стандартные операторы сравнения, строки сравниваются посимвольно в алфавитном порядке.

7.5 Работа с объектами

Объектно-ориентированное программирование (ООП) на сегодняшний день является одной из господствующих парадигм в разработке приложений. *JavaScript* предоставляет возможности ООП, но имеет некоторые особенности.

Основным понятием ООП является объект – сложная комплексная структура, объединяющая в себе данные и методы их обработки.

Для работы с подобными структурами в *JavaScript* используются объекты. Каждый объект может хранить свойства, которые описывают его состояние, и методы, которые описывают его поведение.

Создание нового объекта

Есть несколько способов создания нового объекта.

Первый способ заключается в использовании конструктора ***Object***:

```
const user = new Object();
```

В данном случае объект называется *user*. Он определяется также, как и любая обычная переменная. Выражение *new Object*() представляет вызов конструктора – функции, создающей новый объект. Для вызова конструктора применяется оператор *new*. Вызов конструктора фактически напоминает вызов обычной функции.

Второй способ создания объекта представляет использование фигурных скобок – объектного литерала:

```
const user = {};
```

На сегодняшний день более распространенным является второй способ.

Свойства и методы объекта

После создания объекта можем определить в нем свойства. Чтобы определить свойство, надо после названия объекта через точку указать имя свойства и присвоить ему значение, или использовать альтернативный способ с помощью синтаксиса массивов:

```
const user = {};
user.name = "Alex";
user.age = 21;
user["group"] = "IT-41"
```

В данном случае объявляются три свойства *name*, *age* и *group*, которым присваиваются соответствующие значения. После этого мы можем использовать эти свойства, например, вывести их значения в консоли.

Методы объекта определяют его поведение или действия, которые он производит. Методы представляют собой функции. Например, определим метод, который бы выводил имя и возраст человека:

```
user.info = function(){
   console.log(user.name);
   console.log(user.age);
}
```

Также свойства и методы могут определяться непосредственно при определении объекта:

```
var user = {
   name: "Alex", age: 21,
   group: "IT-41",
   info: function(){
     console.log(this.name); console.log(this.age);
     }
};
```

Чтобы обратиться к свойствам или методам объекта внутри этого объекта, используется ключевое слово ***this***. Оно означает ссылку на текущий объект.

Конструктор объектов

Кроме создания новых объектов *JavaScript* предоставляет нам возможность создавать новые типы объектов с помощью конструкторов.

Конструктор позволяет определить новый тип объекта. Тип представляет собой абстрактное описание или шаблон объекта. Определение типа может состоять из функции конструктора, методов и свойств.

Для начала определим конструктор:

```
function User(pName, pAge) {
   this.name = pName;
   this.age = pAge; this.info = function(){
   document.write("Имя: " + this.name + "; возраст: " +
this.age +"<br/>");
     };
   }
```

Конструктор – это обычная функция за тем исключением, что в ней мы можем установить свойства и методы. Для установки свойств и методов используется ключевое слово *this*. В данном случае устанавливаются два свойства *name* и *age* и один метод *info*. Как правило, названия конструкторы в отличие от названий обычных функций начинаются с большой буквы.

После этого в программе можем определить объект типа *User* и использовать его свойства и методы:

```
const alex = new User("Alex", 21);
console.log(alex.name);            // Alex alex.info();
```

Оператор ***instanceof*** позволяет проверить, с помощью какого конструктора создан объект. Если объект создан с помощью определенного конструктора, то оператор возвращает *true*:

```
const alex = new User("Alex", 21);
console.log(alex instanceof User);     // true
```

Расширение объектов. *Prototype*

Кроме непосредственного определения свойств и методов в конструкторе также можно использовать свойство ***prototype***. Каждая функция имеет свойство *prototype*, представляющее прототип функции. То есть свойство *User.prototype* представляет прототип

объектов *User*. И любые свойства и методы, которые будут определены в *User.prototype*, будут общими для всех объектов *User*.

```
function User(pName, pAge) {
  this.name = pName;
  this.age = pAge;

  this.info = function() {
  document.write("Имя: " + this.name + "; возраст: " +
this.age + "<br/>");
  };
}

User.prototype.hello = function() {
  document.write(this.name + " говорит: 'Привет!'<br/>");
};

User.prototype.group = "IT-41";
const alex = new User("Alex", 21);
 alex.hello();
const max = new User("Max", 21);
max.hello();
console.log(max.group);
```

7.6 Классы

С внедреием стандарта *ES*2015 (*ES*6) в *JavaScript* появился новый способ определения объектов – с помощью классов. Класс представляет описание объекта, его состояния и поведения, а объект является конкретным воплощением или экземпляром класса.

Для определения класса используется ключевое слово *class*.

После этого можем создать объекты класса с помощью конструктора:

```
class Person{}
const alex = new Person();
const max = new Person();
```

По умолчанию классы имеют один конструктор без параметров.

Поэтому в данном случае при вызове конструктора в него не передается никаких аргументов. Однако есть возможность определить в классе свои конструкторы. Также класс может содержать свойства и методы:

```
class Person{
   constructor(name, age){
      this.name = name;
      this.age = age;
}
info(){
  console.log(this.name, this.age);
}
}
const tom = new Person("Alex", 21);
alex.info();                        // Tom 34
console.log(alex.name);           // Tom
```

Конструктор определяется с помощью метода с именем *constructor*. По сути это обычный метод, который может принимать параметры. Основная цель конструктора – инициализировать объект начальными данными. И в данном случае в конструктор передаются два значения – для имени и возраста пользователя.

Для хранения состояния в классе определяются свойства. Для их определения используется ключевое слово *this*. В данном случае в классе два свойства: *name* и *age*.

Поведение класса определяют методы. В данном случае определен метод *info*(), который выводит значения свойств на консоль.

Наследование

Одни классы могут наследоваться от других. Наследование позволяет сократить объем кода в классах-наследниках.

Например, определим следующие классы:

```
class Person{
   constructor(name, age){
      this.name = name;
      this.age = age;
}
 info(){
```

```
   console.log(this.name, this.age);
}
}

class Employee extends Person{
   constructor(name, age, company){
      super(name, age);
      this.company = company;
}
 info(){
    super.info();
    console.log("Employee in", this.company);
 }
 work(){
    console.log(this.name, "is hard working");
}
}

const bob = new Person("Bob", 34);
const bill = new Employee("Bill", 64, "Microsoft");
bob.info();

bill.info();
bill.work();
```

Для наследования одного класса от другого в определении класса применяется оператор *extends*, после которого идет название базового класса. То есть в данном случае класс *Employee* наследуется от класса *Person*. Класс *Person* еще называется базовым классом, классом-родителем, суперклассом, а класс *Employee* – классом-наследником, подклассом, производным классом.

Производный класс, как и базовый, может определять конструкторы, свойства, методы. Вместе с тем с помощью слова *super* производный класс может ссылаться на функционал, определенный в базовом. Например, в конструкторе *Employee* можно вручную не устанавливать свойства *name* и *age*, а с помощью выражения *super*(*name*, *age*); вызвать конструктор базового класса и тем самым передать работу по установке этих свойств базовому классу.

Статические методы

Статические методы вызываются для всего класса в целом, а не для отдельного объекта. Для их определения применяется оператор ***static***. Например:

```
class Person{
   constructor(name, age){
     this.name = name;
     this.age = age;
}

static nameToUpper(person){
return person.name.toUpperCase();
}
info(){
console.log(this.name, this.age);
}

}
const tom = new Person("Tom Soyer", 34);
let personName = Person.nameToUpper(tom);
 console.log(personName);        // TOM SOYER
```

В данном случае определен статический метод ***nameToUpper***().

В качестве параметра он принимает объект ***Person*** и переводит его имя в верхний регистр. Поскольку статический метод относится классу в целом, а не к объекту, то мы НЕ можем использовать в нем ключевое слово ***this*** и через него обращаться к свойствам объекта.

7.7 Объектная модель документа (*DOM*)

Одой из ключевых задач *JavaScript* является взаимодействие с пользователем и манипуляция элементами веб-страницы. Для *JavaScript* веб-страница доступна в виде объектной модели документа (*document object model*) или сокращенно *DOM*.

В соответствии с объектной моделью документа, каждый *HTML*-тег является объектом. Вложенные теги являются дочерними объектами родительского элемента. Текст, который находится внутри тега, также является объектом.

Все эти объекты представляют собой узлы в иерархической структуре *DOM* и доступны при помощи *JavaScript* для изменения страницы.

Существует 12 типов узлов. Но на практике в основном используются 4 из них:

- ***Element***: *html*-элемент;
- ***Document***: корневой узел *html*-документа;
- ***Comment***: элемент комментария;
- ***Text***: текст элемента.

У *DOM*-узлов есть свойства и методы, которые позволяют выбирать любой из элементов, изменять, перемещать их на странице и многое другое.

Объект *document*

Для работы со структурой *DOM* в *JavaScript* предназначен объект *document*, который определен в глобальном объекте *window*. Объект *document* предоставляет ряд свойств и методов для управления элементами страницы.

Для поиска элементов на странице применяются следующие методы:

- ***getElementById***(***value***): выбирает элемент, у которого атрибут ***id*** равен ***value***;
- ***getElementsByTagName***(***value***): выбирает все элементы, у которых тег равен ***value***;
- ***getElementsByClassName***(***value***): выбирает все элементы, которые имеют класс ***value***;
- ***querySelector***(***value***): выбирает первый элемент, который соответствует ***css***-селектору ***value***;
- ***querySelectorAll***(***value***): выбирает все элементы, которые соответствуют ***css***-селектору ***value***.

Кроме методов объект ***document*** позволяет обратиться к определенным элементам веб-страницы через свойства:

- ***documentElement***: предоставляет доступ к корневому элементу <***html***>;
- ***body***: предоставляет доступ к элементу <***body***> на веб-странице;

– ***images***: содержит коллекцию всех объектов изображений (элементов ***img***);

– ***links***: содержит коллекцию ссылок - элементов <***a***> и <***area***>, у которых определен атрибут ***href***;

– ***anchors***: предоставляет доступ к коллекции элементов <***a***>, у которых определен атрибут ***name***;

forms: содержит коллекцию всех форм на веб-странице.

Эти свойства не предоставляют доступ ко всем элементам, однако позволяют получить наиболее часто используемые элементы на веб-странице.

Объект *Node*. Навигация по *DOM*

Каждый отдельный узел, будь то ***html***-элемент, его атрибут или текст, в структуре ***DOM*** представлен объектом ***Node***. Этот объект предоставляет ряд свойств, с помощью которых мы можем получить информацию о данном узле:

– ***childNodes***: содержит коллекцию дочерних узлов;

– ***firstChild***: возвращает первый дочерний узел текущего узла;

– ***lastChild***: возвращает последний дочерний узел текущего узла;

– ***previousSibling***: возвращает предыдущий элемент, который находится на одном уровне с текущим;

– ***nextSibling***: возвращает следующий элемент, который находится на одном уровне с текущим;

– ***ownerDocument***: возвращает корневой узел документа;

– ***parentNode***: возвращает элемент, который содержит текущий узел;

– ***nodeName***: возвращает имя узла;

– ***nodeType***: возвращает тип узла в виде числа;

– ***nodeValue***: возвращает или устанавливает значение узла в виде простого текста;

Создание, добавление и удаление элементов веб-станицы

Для создания элементов объект ***document*** имеет следующие методы:

– ***createElement***(***elementName***): создает элемент ***html***, тег которого передается в качестве параметра. Возвращает созданный элемент.

– ***createTextNode***(***text***): создает и возвращает текстовый узел. В качестве параметра передается текст узла.

Для добавления элементов можно использовать один из методов объекта ***Node***:

– ***appendChild***(***newNode***): добавляет новый узел ***newNode*** в конец коллекции дочерних узлов;

– ***insertBefore***(***newNode***, ***referenceNode***): добавляет новый узел ***newNode*** перед узлом ***referenceNode***.

Иногда элементы бывают довольно сложными по составу, и гораздо проще их скопировать, чем с помощью отдельных вызовов создавать из содержимое. Для копирования уже имеющихся узлов у объекта *Node* можно использовать метод *cloneNode*(). В метод *cloneNode*() в качестве параметра передается логическое значение: если передается *true*, то элемент будет копироваться со всеми дочерними узлами; если передается *false* – то копируется без дочерних узлов.

Для удаления элемента вызывается метод *removeChild*() объекта *Node*. Этот метод удаляет один из дочерних узлов.

Для замены элемента применяется метод *replaceChild*(*newNode*, *oldNode*) объекта *Node*. Этот метод в качестве первого параметра принимает новый элемент, который заменяет старый элемент *oldNode*, передаваемый в качестве второго параметра.

Объект *Element*. Управление элементами

Кроме методов и свойств объекта *Node* в *JavaScript* мы можем использовать свойства и методы объектов *Element*. Важно не путать эти два объекта: *Node* и *Element*. *Node* представляет все узлы веб-станицы, в то время как объект *Element* представляет непосредственно только *html*-элементы. То есть объекты *Element* - это фактически те же самые узлы – объекты *Node*, у которых тип узла (свойство *nodeType*) равно 1.

Основные свойства объекта *Element*:

– ***nodeType*** позволяет узнать тип ***DOM***-узла. Его значение –

– числовое: 1 для элементов, 3 для текстовых узлов, и т.д.

– ***tagName*** возвращает название тега (записывается в верхнем регистре, за исключением ***XML***-режима).

– ***innerHTML*** устанавливает или получает внутреннее ***HTML***-

- содержимое элемента.
- ***outerHTML*** устанавливает или получает полный ***HTML***-код элемента. Запись в ***elem***.***outerHTML*** не меняет ***elem***. Вместо этого она заменяет его во внешнем контексте.
- ***innerText***, ***textContent*** устанавливает или получает текст внутри элемента за вычетом всех <тегов>.
- ***hidden*** скрывает или отображает элемент. Когда значение установлено в ***true***, делает то же самое, что и ***CSS display***:***none***.
- ***offsetWidth*** определяет ширину элемента в пикселях. В ширину включается граница элемента.
- ***offsetHeight*** определяет высоту элемента в пикселях. В высоту включается граница элемента.
- ***clientWidth*** определяет ширину элемента в пикселях без учета границы.
- с***lientHeight*** определяют высоту элемента в пикселях без учета границы.

Среди методов объекта ***Element*** можно отметить методы управления атрибутами:
- ***hasAttribute***(***name***) проверяет на наличие атрибута.
- ***getAttribute***(***attr***) возвращает значение атрибута ***attr***.
- ***setAttribute***(***attr***, ***value***) устанавливает для атрибута ***attr*** значение ***value***. Если атрибута нет, то он добавляется.
- ***removeAttribute***(***attr***) удаляет атрибут ***attr*** и его значение.

Изменение стиля элементов

Для работы со стилевыми свойствами элементов в ***JavaScript*** применяются, главным образом, два подхода:
- изменение свойства ***style***;
- изменение значения атрибута ***class***.

Свойство ***style*** представляет сложный объект для управления стилем и напрямую сопоставляется с атрибутом ***style html***-элемента. Этот объект содержит набор свойств ***CSS***.

Свойства объекта ***style*** совпадает со свойством ***css***. Однако ряд свойств ***css*** в названиях имеют дефис, например, ***font-family***. В ***JavaScript*** для этих свойств дефис не употребляется. Только первая буква, которая идет после дефиса, переводится в верхний регистр: ***fontFamily***.

С помощью свойства ***className*** можно установить атрибут ***class*** элемента ***html***. Благодаря использованию классов не придется настраивать каждое отдельное свойство ***css*** с помощью свойства ***style***.

Но при этом надо учитывать, что прежнее значение атрибута ***class*** удаляется. Поэтому, если нам надо добавить класс, надо объединить его название со старым классом в виде конкатенации строк.

Для управления множеством классов гораздо удобнее использовать свойство ***classList***. Это свойство представляет объект, реализующий следующие методы:

- ***add***(***className***): добавляет класс ***className***.
- ***remove***(***className***): удаляет класс ***className***.
- ***toggle***(***className***): переключает у элемента класс на ***className***. Если класса нет, то он добавляется, если есть, то удаляется.

7.8 События

Для взаимодействия с пользователем в *JavaScript* определен механизм событий. Например, когда пользователь нажимает кнопку, то возникает событие нажатия кнопки.

В *JavaScript* есть следующие типы событий:

- события мыши (перемещение курсора, нажатие мыши и т.д.);
- события клавиатуры (нажатие или отпускание клавиши клавиатуры);
- события жизненного цикла элементов (например, событие загрузки веб-станицы);
- события элементов форм (нажатие кнопки на форме, выбор элемента в выпадающем списке и т.д.);
- события, возникающие при изменении элементов *DOM*;
- события, возникающие при касании на сенсорных экранах;
- события, возникающие при возникновении ошибок.

Событию можно назначить обработчик, то есть функцию, которая сработает, как только событие произошло.

Есть несколько способов назначить событию обработчик. Обработчик может быть назначен прямо в разметке, в атрибуте,

который называется *on*<событие>.

Например, чтобы назначить обработчик события *click* на элементе *input*, можно использовать атрибут *onclick*, вот так:

```
<input value="Нажми меня" onclick="alert('Клик!')" type="button">
```

При клике мышкой на кнопке выполнится код, указанный в атрибуте *onclick*.

Атрибут *HTML*-тега – не самое удобное место для написания большого количества кода, поэтому лучше создать отдельную *JavaScript*-функцию и вызвать в обработчике.

Можно назначать обработчик, используя свойство *DOM*-элемента *on*<событие>. К примеру, *elem.onclick*:

```
<input id="elem" type="button" value="Нажми меня!">

<script>
  elem.onclick = function() {
  alert('Спасибо');
  };
</script>
```

Фундаментальный недостаток описанных выше способов назначения обработчика – невозможность повесить несколько обработчиков на одно событие.

Разработчики стандартов предложили альтернативный способ назначения обработчиков при помощи специальных методов ***addEventListener*** и ***removeEventListener***.

Синтаксис добавления обработчика:

```
element.addEventListener(event, handler[, options])
```

где *event* – имя события, например "*click*", *handler* – ссылка на функцию-обработчик, *options* – дополнительный объект со свойствами:

- ***once***: если ***true***, тогда обработчик будет автоматически удалён после выполнения.
- ***capture***: фаза, на которой должен сработать обработчик, подробнее об этом будет рассказано в главе Всплытие и погру- жение. Так исторически сложилось, что ***options*** может быть ***false***/***true***, это тоже самое, что {***capture***: ***false***/***true***}.
- ***passive***: если ***true***, то указывает, что обработчик никогда не вызовет ***preventDefault***().

Для удаления обработчика следует использовать ***removeEventListener*** с аналогичным синтаксисом.

Объект *Event*

При обработке события браузер автоматически передает в функцию обработчика в качестве параметра объект ***Event***, который инкапсулирует всю информацию о событии. И с помощью его свойств мы можем получить эту информацию:

- ***bubbles***: возвращает ***true***, если событие является восходящим;
- ***cancelable***: возвращает ***true***, если можно отменить стандартную обработку события;
- ***currentTarget***: определяет элемент, к которому прикреплен обработчик события;
-
- ***defaultPrevented***: возвращает ***true***, если был вызван у объекта ***Event*** метод ***preventDefault***();
- ***eventPhase***: определяет стадию обработки события;
- ***target***: указывает на элемент, на котором было вызвано событие;
- ***timeStamp***: хранит время возникновения события;
- ***type***: указывает на имя события.

С помощью метода ***preventDefault***() объекта ***Event*** мы можем остановить дальнейшее выполнение события.

События мыши

Одну из наиболее часто используемых событий составляют события мыши:

- ***click***: возникает при нажатии указателем мыши на элемент;
- ***mousedown***: возникает при нахождении указателя мыши на элементе, когда кнопка мыши находится в нажатом состоянии;
- ***mouseup***: возникает при нахождении указателя мыши на элементе во время отпускания кнопки мыши;
- ***mouseover***: возникает при вхождении указателя мыши в границы элемента;
- ***mousemove***: возникает при прохождении указателя мыши над элементом;
- ***mouseout***: возникает, когда указатель мыши выходит за пределы элемента.

Объект ***Event*** является общим для всех событий. Однако для разных типов событий существуют также свои объекты событий, которые добавляют ряд своих свойств. Так, для работы с событиями указателя мыши определен объект ***MouseEvent***, который добавляет следующие свойства:

- ***altKey***: возвращает ***true***, если была нажата клавиша ***Alt*** во время генерации события;
- ***button***: указывает, какая кнопка мыши была нажата;
- ***clientX***: определяет координату X окна браузера, на которой находился указатель мыши во время генерации события;
- ***clientY***: определяет координату ***Y*** окна браузера, на которой находился указатель мыши во время генерации события;
- ***ctrlKey***: возвращает ***true***, если была нажата клавиша ***Ctrl*** во время генерации события;
- ***metaKey***: возвращает ***true***, если была нажата во время генерации события метаклавиша клавиатуры;
- ***relatedTarget***: определяет вторичный источник возникновения события;
- ***screenX***: определяет координату X относительно верхнего левого угла экрана монитора, на которой находился указа- тель мыши во время генерации события;
- ***screenY***: определяет координату ***Y*** относительно верхнего левого угла экрана монитора, на которой находился указатель мыши во время генерации события;
- ***shiftKey***: возвращает ***true***, если была нажата клавиша ***Shift*** во время генерации события

События клавиатуры

Другим распространенным типом событий являются события клавиатуры:

- ***keydown***: возникает при нажатии клавиши клавиатуры и длится, пока нажата клавиша;
- ***keyup***: возникает при отпускании клавиши клавиатуры;
- ***keypress***: возникает при нажатии клавиши клавиатуры, но после события ***keydown*** и до события ***keyup***. Надо учитывать, что данное событие генерируется только для тех клавиш, которые формируют вывод в виде символов, например, при печати символов. Нажатия на остальные клавиши, например, на ***Alt***, не учитываются.

Для работы с событиями клавиатуры определен объект ***KeyboardEvent***, который добавляет к свойствам объекта ***Event*** ряд специфичных для клавиатуры свойств:

- ***altKey***: возвращает ***true***, если была нажата клавиша ***Alt*** во время генерации события;
- ***charCode***: возвращает символ ***Unicode*** для нажатой клавиши (используется для события ***keypress***);
- ***keyCode***: возвращает числовое представление нажатой клавиши клавиатуры;
- ***ctrlKey***: возвращает ***true***, если была нажата клавиша ***Ctrl*** во вре- мя генерации события;
- ***metaKey***: возвращает ***true***, если была нажата во время генерации события метаклавиша клавиатуры;
- ***shiftKey***: возвращает ***true***, если была нажата клавиша ***Shift*** во время генерации события.

7.9 Обработка и валидация форм с использованием *JavaScript*

Один из способов взаимодействия с пользователями представляют ***html***-формы. Например, если нам надо получить от пользователя некоторую информацию, мы можем определить на веб-странице форму, которая будет содержать текстовые поля для ввода информации и кнопку для отправки. После ввода данных пользователем мы можем обработать введенную информацию.

В ***JavaScript*** форма представлена объектом ***HtmlFormElement***.

После создания формы к ней можем обратиться различными способами.

Первый способ заключается в прямом обращении по имени формы:

```
<form name="search">
</form>

<script>
let searchForm = document.search;
</script>
```

Второй способ состоит в обращении к коллекции форм документа и поиске в ней нужной формы:

```
let searchForm;
for (var i = 0; i < document.forms.length; i++) {
      if(document.forms[i].name==="search")
        searchForm = document.forms[i];
}
```

Форма имеет ряд свойств, из которых наиболее важными являются свойство ***name***, а также свойство ***elements***, которое содержит коллекцию элементов формы.

Среди методов формы надо отметить метод ***submit***(), который отправляет данные формы на сервер, и метод ***reset***(), который очищает поля формы.

Элементы форм

Форма может содержать различные элементы ввода ***html***: ***input***, ***textarea***, ***button***, ***select*** и т.д. Но все они имеют ряд общих свойств и методов.

Также как и форма, элементы форм имеют свойство ***name***, с помощью которого можно получить значение атрибута ***name***.

Для получения и установки значения элементов ***input*** и ***textarea*** используется свойство ***input.value*** (строка) или ***input.checked*** (булево значение) для чекбоксов, ***textarea.value***:

```
input.value = "Новое значение";
textarea.value = "Новый текст";
input.checked = true;      // для чекбоксов и переключателей
```

Элемент ***select*** имеет 3 важных свойства:

- ***select.options*** – коллекция из подэлементов ***option***;
- ***select.value*** – значение выбранного в данный момент ***option***;
- ***select.selectedIndex*** – номер выбранного ***option***.

Они дают три разных способа установить значение в ***select***:

- найти соответствующий элемент ***option*** и установить в ***option.selected*** значение ***true***;
- установить в ***select.value*** значение нужного ***option***;
- установить в ***select.selectedIndex*** номер нужного ***option***.

В отличие от большинства других элементов управления, ***select*** позволяет нам выбрать несколько вариантов одновременно, если у него стоит атрибут ***multiple***. Эту возможность используют редко, но в

этом случае для работы со значениями необходимо использовать первый способ, то есть ставить или удалять свойство ***selected*** у подэлементов ***option***.

Валидация формы

HTML5 представил новую концепцию валидации ***HTML***, названную проверкой ограничений, которая основана на:

- атрибутах элементов ввода ***HTML***;
- псевдо-селекторах ***CSS***;
- свойствах и методах ***DOM***.

Для валидации формы используются следующие атрибуты элементов ввода ***HTML***:

- ***disabled***: указывает, что элемент ввода должен быть отключен;
- ***max***: определяет максимальное значение элемента ввода;
- ***min***: определяет минимальное значение элемента ввода;
- ***pattern***: определяет шаблон значения элемента ввода;
- ***required***: указывает, что для поля ввода требуется элемент;
- ***type***: определяет тип элемента ввода.

Проверка ограничений ***CSS*** использует следующие псевдо-селекторы:

- **:*disabled*** – выбирает элементы ввода с указанным атрибутом
- «***disabled***»;
- :***invalid*** – выбирает элементы ввода с недопустимыми значениями;
- :***optional*** – выбирает элементы ввода без указания атрибута
- «***required***»;
- :***required*** – выбирает элементы ввода с указанным атрибутом
- «***required***»
- :***valid*** – выбирает элементы ввода с допустимыми значениями

Методы ***DOM*** включают функции:

- ***checkValidity*** (): возвращает ***true***, если входной элемент содержит допустимые данные;
- ***setCustomValidity*** (): устанавливает свойство
- ***validationMessage*** элемента ввода.

Свойства ***DOM***:

- ***validity***: содержит логические свойства, связанные с достоверностью входного элемента;
- ***validationMessage***: содержит сообщение, которое браузер отобразит, когда допустимость ложна;
- ***willValidate***: указывает, будет ли проверен входной элемент.

В свою очередь свойство ***validity*** является объектом со следующими свойствами:

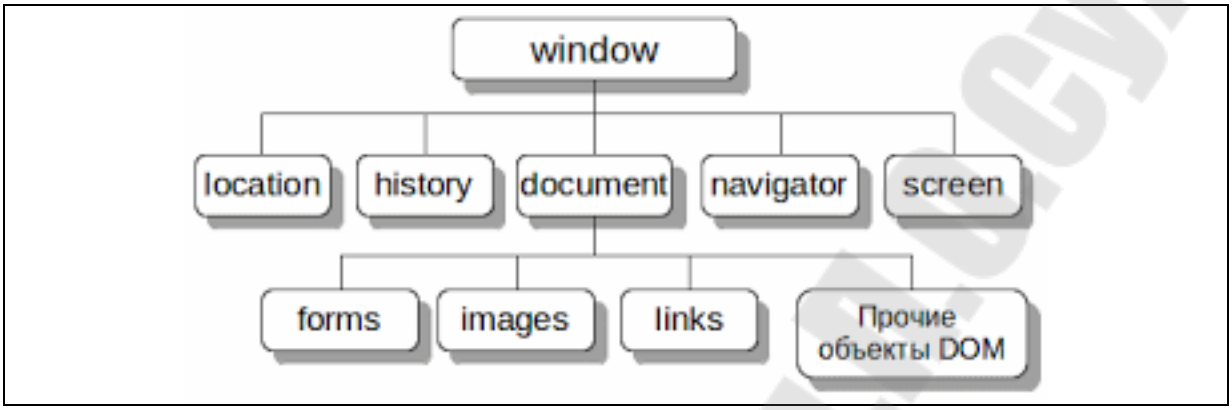
- ***customError***: устанавливается в ***true***, если настроено пользовательское сообщение о достоверности;
- ***patternMismatch***: устанавливается в ***true***, если значение элемента не соответствует его атрибуту ***pattern***;
- ***rangeOverflow***: устанавливается в ***true***, если значение элемента больше, чем его атрибут ***max***;
- ***rangeUnderflow***: устанавливается в ***true***, если значение элемента меньше его атрибута ***min***;
- ***stepMismatch***: устанавливается в ***true***, если значение элемента недопустимо для его атрибута шага;
- ***tooLong***: устанавливается в ***true***, если значение элемента превышает его атрибут ***maxLength***;
- ***typeMismatch***: устанавливается в ***true***, если значение элемента недопустимо для его атрибута типа;
- ***valueMissing***: устанавливается в ***true***, если элемент (с обязательным атрибутом) не имеет значения;
- ***valid***: устанавливается в ***true***, если значение элемента допустимо.

7.10 Объектная модель браузера (*BOM*). Таймеры

Большое значение в *JavaScript* имеет работа с веб-браузером и теми объектами, которые он предоставляет. Например, использование объектов браузера позволяет манипулировать элементами *html*, которые имеются на странице, или взаимодействовать с пользователем.

Все объекты, через которые *JavaScript* взаимодействует с браузером, описываются таким понятием как *Browser Object Model* (Объектная Модель Браузера).

Browser Object Model можно представить в виде следующей схемы:



В вершине находится главный объект – объект *window*, который представляет собой браузер. Этот объект в свою очередь включает ряд других объектов, в частности, объект *document*, который представляет отдельную веб-страницу, отображаемую в браузере.

Объект *window*

Объект *window* представляет собой окно веб-браузера, в котором размещаются веб-страницы. *Window* является глобальным объектом, поэтому при доступе к его свойствам и методам необязательно использовать его имя.

Все объявляемые в программе глобальные переменные или функции автоматически добавляются к объекту *window*.

Ранее были рассмотрены три метода объекта *window* для взаимодействия с пользователем:

- ***alert***(***str***): показывает сообщение в диалоговом окне;
- ***prompt***(***str***, ***default***): показывает сообщение и запрашивает ввод текста от пользователя. Возвращает напечатанный текст в поле ввода или ***null***, если была нажата кнопка «Отмена» или *Esc* с клавиатуры;
- ***confirm***(***str***): показывает сообщение и ждёт, пока пользователь нажмёт *OK* или Отмена. Возвращает *true*, если нажата *OK*, и *false*, если нажата кнопка «Отмена» или *Esc* с клавиатуры.

Все эти методы являются модальными: останавливают выполнение скриптов и не позволяют пользователю взаимодействовать с остальной частью страницы до тех пор, пока окно не будет закрыто.

Объект *window* также предоставляет методы для работы с окном браузера:

- ***window.open*** (): открыть новое окно;
- ***window.close*** (): закрыть текущее окно;
- ***window.moveTo*** (): переместить текущее окно;
- ***window.resizeTo*** (): изменить размер текущего окна.

Объект *history*

Объект *history* предназначен для хранения истории посещений веб-страниц в браузере. Нам этот объект доступен через объект *window*.

Все сведения о посещении пользователя хранятся в специальном стеке (*history stack*). С помощью свойства *length* можно узнать, как много веб-станиц хранится в стеке.

Для перемещения по страницам в истории в объекте *history* определены методы *back*() (перемещение к прошлой посмотренной странице) и *forward*() (перемещение к следующей просмотренной странице)

Также в объекте *history* определен специальный метод *go*(), который позволяет перемещаться вперед и назад по истории на определенное число страниц. Положительное число определяет перемещение вперед, а отрицательное – назад.

Объект *location*

Объект *location* содержит информацию о расположении текущей веб-страницы: *URL*, информацию о сервере, номер порта, протокол. С помощью свойств объекта можно получить следующую информацию:

- ***href***: полная строка запроса к ресурсу;
- ***pathname***: путь к ресурсу;
- ***origin***: общая схема запроса;
- ***protocol***: протокол;
- ***port***: порт, используемый ресурсом;
- ***host***: хост;
- ***hostname***: название хоста;

– ***hash***: если строка запроса содержит символ решетки (#), то данное свойство возвращает ту часть строки, которая идет после этого символа;

– ***search***: если строка запроса содержит знак вопроса (?), например, то данное свойство возвращает ту часть строки, которая идет после знака вопроса.

Также объект *location* предоставляет ряд методов, которые можно использовать для управления путем запроса:

– ***assign***(***url***): загружает ресурс, который находится по пути ***url***.

– ***reload***(***forcedReload***): перезагружает текущую веб-страницу. Параметр ***forcedReload*** указывает, надо ли использовать кэш браузера. Если параметр равен ***true***, то кэш не используется.

– ***replace***(***url***): заменяет текущую веб-станицу другим ресурсом, который находится по пути ***url***. В отличие от метода ***assign***, который также загружает веб-станицу с другого ресурса, метод *replace* не сохраняет предыдущую веб-страницу в стеке истории переходов *history*, поэтому мы не сможем вызвать метод *history.back*() для перехода к ней.

Объект *navigator*

Объект *navigator* содержит информацию о браузере и операционной системе, в которой браузер запущен. Он определяет ряд свойств и методов, основным из которых является свойство *userAgent*, представляющее браузер пользователя.

Объект *navigator* хранит свойство *geolocation*, с помощью которого можно получить географическое положение пользователя. Для получения положения используется метод *getCurrentPosition*(). Этот метод принимает два параметра: функцию, которая срабатывает при удачном запуске, и функцию, которая срабатывает при ошибке запроса геоданных.

```
function success(position) {
let latitude = position.coords.latitude;
let longitude = position.coords.longitude;
let altitude = position.coords.altitude;
let speed = position.coords.speed;

document.write("Широта: " + latitude + "<br/>");
document.write("Долгота: " + longitude + "<br/>");
```

```
document.write("Высота: " + altitude + "<br/>");
document.write("Скорость перемещения: " + speed + "<br/>");
};

function error(obj) {
document.write("Ошибка при определении положения");
};
navigator.geolocation.getCurrentPosition(success, error);
```

Таймеры

Для выполнения действий через определенные промежутки времени в объекте *window* предусмотрены функции таймеров. Есть два типа таймеров: одни выполняются только один раз, а другие постоянно через промежуток времени.

Для одноразового выполнения действий через промежуток времени предназначена функция *setTimeout*().

Функция *setTimeout*() может принимать два параметра:

```
let timerId = setTimeout(someFunction, period)
```

Параметр *period* указывает на промежуток, через который будет выполняться функция из параметра *someFunction*. А в качестве результата функция возвращает *id* таймера.

Для остановки таймера применяется функция *clearTimeout*(*id*).

Функции *setInterval*() и *clearInterval*() работают аналогично функциям *setTimeout*() и *clearTimeout*() с той лишь разницей, что *setInterval*() постоянно выполняет определенную функцию через промежуток времени.

Функция *requestAnimationFrame*() действует аналогично *setInterval*() за тем исключением, что он больше предназначен для анимации, работы с графикой и имеет ряд оптимизаций, которые улучшают его производительность.

В метод *window.requestAnimationFrame*() передается функция, которая будет вызываться определенное количество раз (обычно 60) в секунду.

7.11 Сериализация объектов

Для хранения и обмена данными между клиентом и сервером чаще всего используется формат *JSON*. *JSON* (*JavaScript Object Nota-*

tion) – способ записи данных в виде строки. Поскольку формат *JSON* является только текстовым, его можно легко отправлять на сервер и с сервера и использовать в качестве формата данных на любом языке программирования. Например, клиент использует *JavaScript*, а сервер написан на *Ruby*/*PHP*/*Java* или любом другом языке.

JavaScript предоставляет следующие методы для сериализации и десериализации объектов:

- ***JSON.stringify*(*obj*)** используется для преобразования объектов в строку ***JSON***;
- ***JSON.parse*(*str*)** используется для преобразования строки в формате ***JSON*** обратно в объект.

JSON поддерживает три типа данных: примитивные значения, объекты и массивы. Примитивные значения представляют стандартные строки, числа, значение *null*, логические значения *true* и *false*.

```
<script>
   const student = {
      name: "Alex",
      age: 21,
      marks: {
        OAIP: 9,
        "System programming": 8,
        RPI: 9,
        ACS: 8
 }
};

let str = JSON.stringify(student);
console.log(str);

  /*{"name":"Alex","age":21,"marks":{"OAIP":9,"System
programming":8, "RPI":9,"ACS":8}}   */
let alex = JSON.parse(str);
console.log(alex);      // {name: "Alex", age: 21, marks: {…}}
</script>
```

Web storage

Для хранения данных на стороне клиента в *HTML5* была внедрена новая концепция – *web-storage*. *Web storage* состоит из двух компонентов: *session storage* и *local storage*.

Session storage представляет временное хранилище информации, которая удаляется после закрытия браузера.

Local storage представляет хранилище для данных на постоянной основе. Данные из *local storage* автоматически не удаляются и не имеют срока действия. Объем *local storage* составляет в *Chrome* и *Firefox* 5 *M*б для домена.

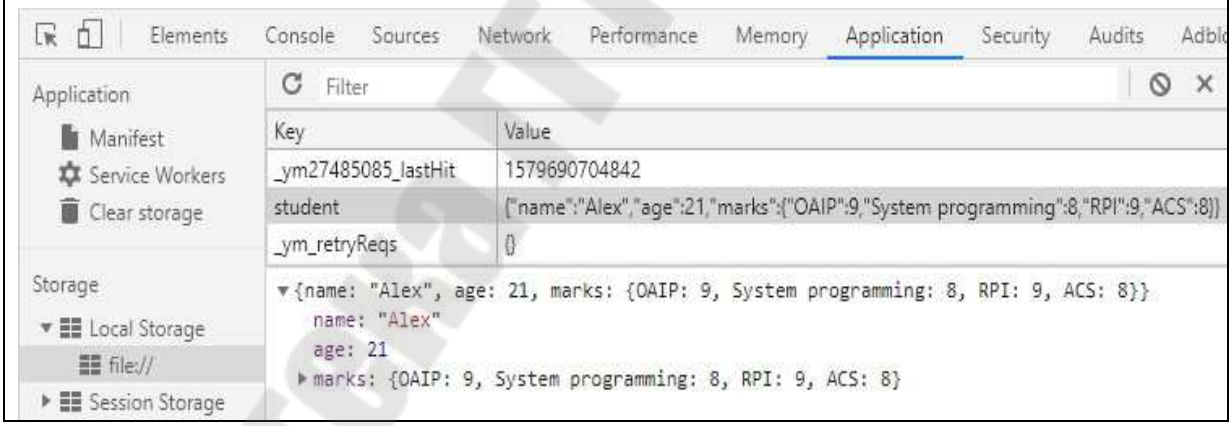
Все данные в *web storage* представляют набор пар ключ-значение. То есть каждый объект имеет уникальное имя-ключ и определенное значение.

Для работы с *local storage* в *javascript* используется объект *localStorage*, а для работы с *session storage* – объект *sessionStorage*. Эти объекты предомтавляют следующие методы для работы с данными:

- ***setItem***() – для сохранения данных;
- ***getItem***() – для получения сохраненных данных;
- ***removeItem***() – для удаления объект из хранилища;
- ***clear***() – для полного удаления всех объектов из хранилища.

Например:

```
localStorage.setItem("student", JSON.stringify(student));
```



7.12 Библиотека *jQuery*

Современное веб-программирование и создание веб-приложений уже невозможно представить без использования языка *JavaScript*. Однако в настоящее время, все чаще используется не чистый код *JavaScript*, а *JavaScript*-фреймворки и библиотеки. Одной из таких библиотек является *jQuery*.

Подключив библиотеку *jQuery* вместо десятков команд на *JavaScript* можно написать несколько команд. Команды основаны на селекторах и классах *CSS*.

Библиотеку *jQuery* можно скачать с сайта *http://code.jquery.com/*. Если нет желания скачивать библиотеку её можно подключить с *CDN* (*Content Delivery Network*) компаний *Google* или *Microsoft*.

Вся работа с библиотекой сводится к выбору (*query*) *HTML*-элемента и выполнения над ним различных действий.

Основной синтаксис:

```
$(selector).action(),
```

где $() – функция доступа (определения) *jQuery*. (*jQuery*() – полное название функции $), *selector* – "*query*(запрос)" для определения *HTML* элементов, *action*() – действия над элементом (элементами).

Примеры:

```
$(this).hide()    // скрывает текущий элемент.
$("p").hide()     // скрывает все абзацы.
$(".test").hide() // скрывает все элементы класса "test".
$("#test").hide() // скрывает элемент с id="test".
```

Создание элементов *DOM:*

```
$("<div>Hi</div>");
$("<div>")
```

Можно использовать кавычки, можно апострофы. Но наличие тегов обязательно. Задавать просто текст нельзя. Сама по себе эта команда ничего не выведет, надо этот узел привязать к родительскому элементу. Делается это следующим образом:

```
$("<div>Hi</div>").insertAfter("#a1")
```

Здесь *a*1 – это идентификатор объекта, после которого необходимо вставить элемент.

Работа с полученным набором значений

В терминах *jQuery* эти наборы называют обернутыми.

Для определения количество выбранных элементов используется свойство *length* или метода *size*, который также возвращает число выбранных элементов:

```
let num1 = $("tr:nth-child(odd)").length;
let num2 = $("tr:nth-child(odd)").size();
```

Для получения доступа к элементу выборки можно использовать индекс или использовать функцию *eq*(*index*). Второй способ более удобен так как функция возвращает объект:

```
let firstElement = $("tr:nth-child(odd)")[0];.
let elem = $("tr:even").eq(0);.
```

Для перебора элементов выборки можно использовать стандартные методы *JavaScript*, а также использовать специальный метод *each*:

```
$(function(){
   $("tr:even").each(function(index, elem){
      console.log(index + ". " + elem.innerHTML);
});
});
```

В метод *each* в качестве параметра передается безымянная функция, которая принимает два параметра: *index* – индекс элемента в наборе и *elem* – сам элемент.

Манипулирование объектами на странице

Библиотека *jQuery* предлагает нам инструментарий для манипуляции свойствами и атрибутами элементов:

- ***prop***(***property***, ***[value]***) получает или изменяет значение свойства элемента;
- ***removeProp***(***property***) удаляет свойство;
- ***attr***(***attribute***б ***[value]***) получить или изменяет значение атрибута элемента;
- ***css***(***property***, ***[value]***) применяется для работы со стилями элемента;
- ***html***() используется для получения или установки разметки;
- ***text***() используется для получения или установки текста;

- ***val***() используется для получения значений элементов форм.

Обработка событий

Различные браузеры по-своему могут обрабатывать события, *jQuery* пытается сгладить эти неприятности.

Модель событий *jQuery* обладает следующими свойствами:

- поддерживает единый метод установки событий;
- позволяет устанавливать несколько обработчиков для события;
- использует стандартные названия типов событий;
- предоставляет единые методы отмены события и блокирования действий по умолчанию.

jQuery предоставляет специальный метод *on*, который позволяет создать и зарегистрировать обработчики как для существующих, так и для будущих элементов, которых еще не в структуре *DOM*. Этот метод имеет следующие варианты использования:

- ***on***('событие', 'селектор', обработчик_события);
- ***on***('событие', 'селектор', данные_события, обработчик_события) то же самое, что и в предыдущем случае, плюс параметр для передачи данных события в объект *Event* (*e.data*).

Для удаления обработчика используется команда ***unbind***().

Метод *trigger*() используется для вызова обработчиков событий вручную. Например, вызвав метод *trigger* для нужного элемента мы можем также и вызвать обработчик события. Он имеет следующие варианты использования:

- ***trigger***('событие'): вызов обработчика для данного события;
- ***trigger***(объект_***Event***): вызов обработчика с использованием объекта ***Event***, который содержит данные о том, какой именно обработчик надо вызвать.

Кроме специализированных методов *jQuery* предоставляет прямые методы для обработки событий. Эти методы, как правило, носят наименование обрабатываемого события, а в качестве параметра принимают функцию обработчика данного события:

```
$('button').first().click(function(){
    $(this).css('background-color', 'silver');
});
```

Эффекты анимации

К базовым эффектам в *jQuery* относятся эффекты скрытия и отображения элементов, которые достигаются с помощью методов *show*(), *hide*() и *toggle*().

Все эти методы могут использоваться в трех вариантах (рассмотрим на примере метод *hide*):

- ***hide***(): метод без параметров
- ***hide***(***[duration]***[, ***complete]***): принимает два необязательных параметра. Параметр *duration* указывает как долго анимация элемента будет длиться. По умолчанию его значение равно 400 миллисекунд. Параметр *complete* представляет функцию, вызываемую методом по завершению анимации
- ***hide***(***[duration] [***, ***easing]***[, ***complete]***): то же самое, только добавляется параметр ***easing***, который принимает название функции плавности анимации в виде строки. По умолчанию его значение равно "*swing*". Также можно использовать значения '*slow*' и '*fast*', которые соответствуют длительности эффекта в 600 и 200 миллисекунд.

Эффекты скольжения позволяют нам плавно скрыть или раскрыть элемент. Эффекты скольжения реализованы в виде методов *slideUp*(), *slideDown*() и *slideToggle*().

Эффекты прозрачности позволяют нам, плавно изменяя прозрачность элемента, скрыть его или отобразить. Эффекты прозрачности реализованы с помощью методов *fadeOut*(), *fadeIn*(), *fadeTo*() и *fadeToggle*().

Для создания более сложных по характеру эффектов используется метод *animate*():

```
animate(properties [,duration] [,easing] [,complete])
```

Обязательный параметр *properties* содержит набор *css*-свойств, у которых указываются финальные значения. Параметр *duration* указывает, как долго будет длиться изменение прозрачности элемента. Параметр *easing* принимает название функции плавности анимации в виде строки. Параметр *complete* представляет функцию обратного вызова, вызываемую методом по завершении анимации.

7.13 Взаимодействие с сервером. Технология *AJAX*

Ajax представляет технологию для отправки запросов к серверу из клиентского кода ***JavaScript*** без перезагрузки страницы. Сам термин расшифровывается как ***Asynchronous JavaScript And XML***,

т.е. изначально ***AJAX*** предполагал асинхронное взаимодействие клиента и сервера посредством данных в формате ***XML***.

В настоящее время основным форматом данных для обмена данных между клиентом и сервером является формат ***JSON***.

Объект *XMLHttpRequest*

Для создания приложений, использующих ***Ajax***, применяются различные способы, но самым распространенным способом является использование объекта ***XMLHttpRequest***:

```
const request = new XMLHttpRequest();
```

После создания объекта ***XMLHttpRequest*** можно отправлять запросы к серверу. Но для начала необходимо вызвать метод ***open***() для инициализации:

```
request.open("GET", "http://localhost/test.txt", false);
```

Метод ***open***() принимает три параметра: тип запроса (***GET***, ***POST***, ***HEAD***, ***PUT***), адрес запроса и третий, необязательный параметр – логическое значение ***true*** или ***false***, указывающее, будет ли запрос осуществляться в асинхронном режиме. По умолчанию, если третий параметр не используется, то запрос отправляется в асинхронном режиме, что позволяет параллельно с выполнением запроса выполнять также и другой код ***JavaScript***.

Кроме того, метод ***open***() может принимать еще два параметра: логин и пароль пользователя, если для выполнения запроса нужна аутентификация:

```
request.open("GET", "http://localhost/home.php", true,
"login", "password");
```

После инициализации запроса методом ***open***() необходимо отправить запрос с помощью метода ***send***():

```
request.send();
```

Объект ***XMLHttpRequest*** имеет ряд свойств, которые позволяют проконтролировать выполнение запроса:

- ***status***: содержит статусный код ответа ***HTTP***, который пришел от сервера. С помощью статусного кода можно судить об успешности запроса или об ошибках, которые могли бы возникнуть при его выполнении. Например, статусный код 200 указывает на то, что запрос прошел успешно. Код 403 говорит о необходимости авторизации для выполнения запроса, а код 404 сообщает, что ресурс не найден и так далее;
- ***statusText***: возвращает текст статуса ответа, например, "200 *OK*";
- ***responseType***: возвращает тип ответа;
- ***response***: возвращает ответ сервера;
- ***responseText***: возвращает текст ответа сервера;
- ***responseXML***: возвращает *xml*, если ответ от сервера в формате *xml*.

При асинхронном запросе объект *XMLHttpRequest* использует свойство *readyState* для хранения состояния запроса. Состояние запроса представляет собой число:

- 0: объект ***XMLHttpRequest*** создан, но метод ***open***() еще не был вызван для инициализации объекта
- 1: метод ***open***() был вызван, но запрос еще не был отправлен методом ***send***();
- 2: запрос был отправлен, заголовки и статус ответа получены и готовы к использованию;
- 3: ответ получен от сервера;
- 4: выполнение запроса полностью завершено (даже если получен код ошибки, например, 404).

Событие ***readystatechange*** возникает каждый раз, когда изменяется значение свойства ***readyState***.

Отправка запросов

GET-запрос характеризуется тем, что данные могут отправляться в строке запроса:

```
<script>
// объект для отправки
var user = {
   name: "Alex",
   age: 21
```

```
};

const request = new XMLHttpRequest();
function reqReadyStateChange() {
 if (request.readyState == 4) {
    var status = request.status;
 if (status == 200) {
document.getElementById("output").innerHTML=request.responseText;
       }
    }
}
// строка с параметрами для отправки
var body = "name=" + user.name + "&age="+user.age;
request.open("GET", "http://localhost:8080/postdata.php?"+body);
request.onreadystatechange = reqReadyStateChange;
request.send();
</script>
```

Для отправки берем свойства объекта *user* и формируем из их значений сроку с параметрами "*name*=" + *user.name* + "*&age*=" + *user.age*. Затем эта строка добавляется к строке запроса в методе: *open*("*GET*", "*http*://*localhost*:8080/*postdata.php*?"+*body*)

Для отправки данных методом *POST* надо установить заголовок *Content-Type* с помощью метода *setRequestHeader*(). В данном случае заголовок имеет значение *application/x-www-form-urlencoded*. Данные встраиваются в тело запроса при посылке запроса.

```
var user = {
   name: "Tom",
   age: 23
};

const request = new XMLHttpRequest();
function reqReadyStateChange() {
  if (request.readyState == 4 && request.status == 200)
document.getElementById("output").innerHTML=request.responseText;
}
let body = "name=" + user.name + "&age="+user.age;

request.open("POST", "http://localhost:8080/postdata.php");
request.setRequestHeader('Content-Type', 'application/x-www-form-
urlencoded');
```

```
request.onreadystatechange = reqReadyStateChange;
request.send(body);
```

7.14 Promise в *Ajax*-запросах

Для более удобного способа организации асинхронного кода в современном *JavaScript* используется объект *Promise*, который обертывает асинхронную операцию в один объект и позволяет определить действия, выполняющиеся при успешном или неудачном выполнении этой операции.

Promise – это специальный объект, который содержит своё состояние. Вначале *pending* («ожидание»), затем – одно из: *fulfilled* («выполнено успешно») или *rejected* («выполнено с ошибкой»).

Синтаксис создания *Promise*:

```
const promise = new Promise(function(resolve, reject) {
// Эта функция будет вызвана автоматически
// В ней можно делать любые асинхронные операции,
// А когда они завершатся – нужно вызвать одно из:
// resolve(результат) при успешном выполнении

// reject(ошибка) при ошибке
})
```

Для обработки результата объекта *Promise* вызывается метод *then*(), который принимает два параметра: функцию, вызываемую при успешном выполнении запроса, и функцию, которая вызывается при неудачном выполнении запроса. Метод *then*() также возвращает объект *Promise*. Поэтому при необходимости мы можем применить к его результату цепочки вызовов метода *then*: *promise*.*then*().*then*().

Рассмотрим пример использования промиса с использованием промисификация – это когда берут асинхронную функциональность и делают для неё обёртку, возвращающую промис.

После промисификации использование функциональности зачастую становится гораздо удобнее. Например:

```
function httpGet(url) {
   return new Promise(function(resolve, reject) {
    var xhr = new XMLHttpRequest();
    xhr.open('GET', url, true);
```

```
    xhr.onload = function() {
    if (this.status == 200) {
        resolve(this.response);
    }
    else {
      var error = new Error(this.statusText);
      error.code = this.status;
      reject(error);
    }
 };

   xhr.onerror = function() {
     reject(new Error("Network Error"));
};
    xhr.send();
});
}

httpGet("/article/promise/user.json")
.then(
  response => alert(`Fulfilled: ${response}`),
  error => alert(`Rejected: ${error}`)
);
```

7.15 Метод *Fetch*

Современные браузеры поддерживает метод *fetch* – новый встроенный метод для *AJAX*-запросов, призванный заменить *XMLHttpRequest*. Он гораздо мощнее, чем приведенная выше функция *httpGet*.

Типичный запрос с помощью *fetch* состоит из двух операторов *await*:

```
let response = await fetch(url, options);
                    // завершается с заголовками ответа
let result = await response.json();
                    // читать тело ответа в формате JSON
```

Или, без *await*:

```
fetch(url, options)
.then(response => response.json())
```

```
.then(result => /* обрабатываем результат */)
```

Метод ***fetch*** следующие опции для формирования запроса:

- ***method*** – *HTTP*-метод;
- ***headers*** – объект с запрашиваемыми заголовками;
- ***body*** – данные для отправки (тело запроса) в виде текста, *FormData*, *BufferSource*, *Blob* или *UrlSearchParams*.

Для получения тела ответа используются следующие методы:

- ***response.text***() – возвращает ответ как обычный текст;
- ***response.json***() – преобразовывает ответ в *JSON*-объект;
- ***response.formData***() – возвращает ответ как объект *FormData* (кодировка ***form***/***multipart***, см. следующую главу);
- ***response.blob***() – возвращает объект как *Blob* (бинарные данные с типом);
- ***response.arrayBuffer***() – возвращает ответ как *ArrayBuffer* (низкоуровневые бинарные данные).

8 JAVASCRIPT И *ECMASCRIPT*

ECMAScript – это стандарт, на котором основан *JavaScript*. Его часто называют *ES*. *ECMAScript* 2015, также известный как *ES*6, является фундаментальной версией стандарта *ECMAScript*. Опубликованный через четыре года после последней версии стандарта *ECMAScript* 5.1, он также обозначил переход с номера издания на номер года. Поэтому его следует называть не *ES*6 (хотя все называют его таковым), а *ES*2015.

*ES*5 разрабатывался 10 лет, с 1999 по 2009 год и, как таковой, он был фундаментальным и крайне важным пересмотром языка. Поскольку этот длинный промежуток времени прошел между *ES*5.1 и *ES*6, выпуск полон важных новых функций и существенных изменений в рекомендуемых лучших практиках разработки программ на *JavaScript*.

Рассмотрим наиболее важные изменения.

Стрелочные функции

С момента появления стрелочных функций, последние изменили то, как выглядит и работает большая часть *JavaScript* кода.

Визуально, это простое и долгожданное изменение. Например, функция:

```
const name = function name() {
  console.log("my name is Alex");
};
```

превращается в:

```
const name = () => {
  console.log("my name is Alex");
};
```

Если тело функции является однострочным, то можно убрать фигурные скобки:

```
const name = () => console.log("my name is Alex");
```

Если только один параметр, то можно убрать круглые скобки:

```
const name = param => console.log(`my name is ${param}`);
```

Это не серьезное изменение, обычные функции продолжают работать так же, как и раньше.

Новая область *this*

Значение *this* определяется тем, каким образом вызвана функция. В *JavaScript* ключевое слово *this* относится к объекту, к которому оно принадлежит.

Оно имеет разные значения в зависимости от того, где используется:

- в методе, *this* относится к объекту, по отношению к которому вызван метод;
- в глобальном контексте выполнения, *this* ссылается к глобальному объекту;
- в функции, значение *this* зависит от того, каким образом вызвана функция;
- в обычной функции, *this* относится к глобальному объекту;
- в стрелочных функциях, *this* привязан к окружению, в котором была создана функция;

- в строгом режиме функции, *this* не определен, если предварительно он не был установлен в контексте исполнения;
- в событии, *this* относится к элементу, который получил событие;
- такие методы, как *call*() и *apply*() могут ссылать *this* на любой объект.

let* и *const

Есть несколько способов объявить переменные в *ES6*: объявить переменные с помощью *var*, плюс сейчас можем использовать *let* и *const*.

Эти два способа имеют некоторые атрибуты, которые будут полезны при создании переменных.

Сделаем краткий обзор, как отличаются *var*, *let* и *const*.

Во-первых, переменные *var* могут быть переопределены или обновлены. Например:

```
var width = 100;
console.log(width);   // 100
width = 200;
console.log(width);   // 200
```

Также они не будут «кричать» о создании одного и того же имени переменной дважды в одной и той же области видимости, так как переменные *var* могут быть обновлены или переопределены.

Также важно помнить, как переменные *var* определены. По сути, *scoping* или область видимости означает «Где эти переменные доступны для меня?».

В случае переменных *var* они являются областью действия функции, что означает, что они доступны только внутри функции, в которой они созданы. Однако, если они не объявлены в функции, то они имеют глобальную область видимости и доступны во всем окне.

var – традиционно функциональная область.

let – это новое объявление переменной, которая имеет ***ограниченую область видимости***.

Это означает, что объявление переменных *let* в цикле *for*, внутри блока *if* или в обычном блоке, не позволит этой переменной

«покинуть» блок, в то время как *var* поднимется до определения функции.

const похож на *let*, но он неизменный, константа (почти!). Если попытаться обновить его, то это не сработает, потому что нельзя обновить переменную *const* так, как *let*.

Например, есть объект:

```
const person = {
  name: "Alex",
  age: 28
};
```

Обновить переменную ***const***, набрав *person = { name: 'Alexey' }*, не получится. Однако свойства переменной *const* могут измениться. Это потому, что весь объект не является неизменным. Его просто нельзя переназначить полностью. Например:

```
const person = {
  name: "Alex",
  age: 28
};
  person.name = "Alexey";
```

Это сработает, но нельзя перезаписать всю переменную.

Promises

Promises (промисы) позволяют устранить известный «*callback hell*».

Промисы использовались разработчиками *JavaScript* задолго до *ES*2015, только с помощью множеств различных библиотек, а стандарт нашел общий язык между ними.

Используя промис, можно переписать этот код:

```
setTimeout(function () {
  console.log("I promised to run after 1s");
  setTimeout(function () {
    console.log("I promised to run after 2s");
  }, 1000);
}, 1000);
```

на следующий:

```
const wait = () =>
  new Promise((resolve, reject) => {
    setTimeout(resolve, 1000);
  });

wait()
  .then(() => {
    console.log("I promised to run after 1s");
    return wait();
  })
  .then(() => console.log("I promised to run after 2s"));
```

Промис может быть выполнен успешно – ***resolved*** или отклонен – ***rejected***.

Когда успешно выполняешь промис, запускается ***then***(), а когда отклоняешь – запускается ***catch***().

Промис позволяет обрабатывать результаты асинхронных операций так, как если бы они были синхронными.

Generators

Generators – это особый тип функций, который дает возможность приостановить свое выполнения, а потом позже возобновить. Это позволяет запускать другой код в момент приостановления функции.

function **со звёздочкой** ***function**** определяет функцию-генератор.

Классы

Традиционно *JavaScript* – единственный основной язык с наследованием на основе прототипов.

Наследование очень простое и напоминает другие языки объектно-ориентированного программирования:

```
class Person {
  constructor(name) {
    this.name = name;
  }


  hello() {
    return "Hello, I am " + this.name + ".";
```

```
  }
}

class Developer extends Person {
  hello() {
    return super.hello() + " I am a Frontend Developer.";
  }
}
const me = new Developer("Alex");
console.log(me.hello());
          // Hello, I am Alex. I am a Frontend Developer.
```

Классы не имеют явных объявлений переменных класса, но мы должны инициализировать любую переменную в конструкторе.

Конструктор

Классы имеют специальный метод с именем ***constructor***, который вызывается, когда класс инициализируются с помощью ***new***.

Родительский *constructor* наследуется автоматически, если у потомка нет своего метода *constructor*.

Если же потомок имеет свой *constructor*, то, чтобы унаследовать конструктор родителя, нужно использовать *super*() с аргументами для родителя.

super

Слово *super* используется для вызова функций, принадлежащих родителю объекта.

Getters and setters

Геттеры могут быть объявлены следующим образом:

```
class Person {
  get fullName() {
    return `${this.firstName} ${this.lastName}`;
  }
}
```

get связывает свойство объекта с функцией, которая вызывается при обращении к этому свойству.

Сеттеры пишутся так же:

```
class Person {
  set age(years) {
    this.theAge = years;
  }
}
```

set связывает свойство объекта с функцией, он будет вызываться при попытке установить это свойство.

Модули

До *ES*2015 существовало не менее трех основных модулей, конкурирующих со стандартами, которые разделили сообщество:

- *AMD*;
- *RequireJS*;
- *CommonJS*.

*ES*2015 стандартизировал их в общий формат.

Импорт модулей

import используется для импорта ссылок на значения, экспортированные из внешнего модуля, и осуществляется через конструкцию *import ... from ...*:

```
import defaultExport from "module-name";
import * as name from "module-name";
import * from 'module-name'
import { namedExport } from 'module-name'
import "module-name";
```

Экспорт модулей

Экспорт используется для экспорта функций, объектов. Можно писать модули и экспортировать что угодно в другие модули, используя ключевое слово ***export***:

```
export { name1, name2, …, nameN };
export default выражение;
export * from …;
          export default function
```

Шаблонные литералы (*Template Literals*)

Шаблонные литералы – это новый синтаксис для создания строк:

```
const aString = `A string`;
```

Они обеспечивают способ встраивания выражений в строки, эффективно интерполируя значения, используя синтаксис ${имя_переменной}:

```
const name = "Alex";
const string = `Hey ${name}`;   // Hey Alex
```

Можно выполнять более сложные выражения:

```
const string = `total count is: ${1 + 2 + 3}`;
const string2 = `result is ${foo() ? "x" : "y"}`;
```

Также строки могут занимать несколько строк:

```
const string3 = `Hey
this

string
is awesome!`;
```

Параметры по умолчанию

Функции теперь поддерживают параметры по умолчанию:

```
const sayMyName = function(name = "Alex") {
  console.log(name);
};
sayMyName();    // Alex
```

Если при вывозе функции *name*() не передали аргумент, то тогда она возьмет значение по умолчанию "*Alex*".

Оператор распространения *spread*

Можно расширить или объединить массив, объект или строку, используя оператор распространения ... .

Начнем с массива.

У нас есть один массив *const a = [1, 2, 3]* , который мы хотим модифицировать, добавив несколько значений:

```
const a = [1, 2, 3];
const b = [...a, 4, 5, 6];        // [1, 2, 3, 4, 5, 6]
```

Теперь у нас новый массив *b* который содержит все значения *a*, плюс свои.

Также можно создать копию массива:

```
const c = [...a];
```

spread также работает и для объектов.

Клонировать объект можно так:

```
const newObj = { ...oldObj };
```

Используя строки, оператор распространения *spread* создает массив с каждым символом в строке:

```
const hey = "hey";
const arrayized = [...hey]; // ['h', 'e', 'y']
```

У этого оператора есть несколько довольно полезных свойств. Наиболее важным из них является возможность очень просто использовать массив в качестве аргумента функции:

```
const f = (one, two) => {
  console.log(one, two); // 1, 2
};
const a = [1, 2];
f(...a);
```

Деструктуризация

Деструктуризация – это выражение *JavaScript*, которое позволяет извлекать данные из массивов и объектов. Например, когда с данного объекта необходимо извлечь только некоторые значения и поместить их в переменные, можем сделать это так:

```
const person = {
```

```
  firstName: "Alex",
  dev: true,
  age: 28
};

const name = person.firstName;
const age = person.age;

console.log(name, age); // Alex, 28
```

Но тогда выходит код, повторяющийся снова и снова, и нам необходмо будет создать переменную для каждого значения, которое находится внутри объекта или массива.

Вместо этого можно создать несколько переменных, и структурировать их в одну строку следующим образом:

```
const { firstName, age } = person;
```

Здесь создаем переменные с именами *firstName*, *age* и берем их значения из объекта *person*.

Иными словами, берем свойства *firstName*, *age* и помещаем их в две новые переменные, которые будут видны родительскому блоку (или *window*).

Также есть возможность переименовать переменную, если она вам не нравится или имеет непонятное имя, важно, что бы имя было не занято в области видимости.

Например:

```
const { firstName: name, age } = person;
```

Таким образом, извлекли значение с *firstName* или *person.firstName* и дали ему новое имя *name*.

Деструктуризация так же работает на массивах. Например, если нужны только первый и второй элементы:

```
const a = [1, 2, 3, 4, 5];
const [first, second] = a;

console.log(first, second); // 1, 2
```

Еще один вариант как выбрать нужные элементы из массива:

```
const [first, second, , , fifth] = a;

console.log(first, second, fifth); // 1, 2, 5
```

Этот синтаксис создает три новые переменные, получая элементы с индексами 0, 1, 4 из массива *a*.

Более простой синтаксис. Расширение литералов объекта

Стал более простым синтаксис для переменных, если они имеют одинаковые имена. Вместо этого кода:

```
const name = "Alex";
const x = {
  name: name
};
```

можно написать так:

```
const name = "Alex";
const x = {
  name
};
```

Прототип

Прототип может быть указан следующим образом:

```
const person = { name: "Alex" };
const developer = {
  __proto__: person
};
super()
const person = { name: "Alex", say: () => "Hello " };
const developer = {
  __proto__: person,
  say() {
    return super.say() + this.name;
  }
};
developer.say();   // Hello Alex
```

Динамические свойства

Имя свойства объекта может быть указано динамически.

```
const x = {
  ["a" + "_" + "b"]: "z"
};
x.a_b;    // z
```

Цикл *for-of*

*ES*5 еще в 2009 году представил цикл *forEach*(). Единственное, что в этом цикле нет никакого способа сделать разрыв *break*, как в цикле *for*.

*ES*2015 представил цикл *for-of*, который сочетает в себе краткость *forEach* с возможностью *break* разрыва:

```
for (const v of ["a", "b", "c"]) {
  console.log(v);
}
// получает индекс и значение, используя `entries()`
for (const [i, v] of ["a", "b", "c"].entries()) {
  console.log(i, v);
  if (i === 0) {
    break;
  }
}
```

Метод *entries()* – это метод, который возвращает *iterator*, позволяя пройтись по всем его ключам/значениям. В нашем случае используется деструктуризация массива, где *i* – это индекс элемента в массиве, а *v* – это его значение.

Map* и *Set

Map и *Set* (и их соответствующие сборщики мусора *WeakMap* и *WeakSet*) являются официальными реализациями двух очень популярных структур данных.

Map содержит пары ключ-значение и сохраняет порядок вставки.

Любое значение может быть использовано в качестве ключа.

Set позволяет сохранять уникальные значения любого типа.

Новые строковые методы

Любое строковое значение получило несколько новых методов экземпляра:

- *repeat*() – повторяет строки указанное количество раз:
  'Ho'.repeat(3) // *HoHoHo*;
- *codePointAt*() – возвращает не отрицательное целое число, которое является закодированным в *UTF*-16 значением кодовой точки.

Новые методы объекта

*ES*6 представил несколько статических методов в пространстве имен объекта:

- *Object.is*() – метод определяет, являются ли два значения одинаковыми, *var isSame = Object.is*(*value*1, *value*2);
- *Object.assign*() – используется для поверхностного копирования всех свойств объекта в целевой объект, *Object.assign*(*target*, *...sources*);
- *Object.setPrototypeOf* – устанавливает прототип объекта *Object.setPrototypeOf*(*obj*, *prototype*).

9 ВЫБОР ПРЕПРОЦЕССОРА *CSS*

CSS* препроцессор (*CSS preprocessor*)** – это надстройка над ***CSS, которая предоставляет новые функции и возможности для *CSS* с помощью добавленных синтаксических конструкций.

Основная задача препроцессора – это предоставление удобных синтаксических конструкций для разработчика, чтобы упростить, и тем самым, ускорить разработку и поддержу стилей в проектах.

CSS препроцессоры преобразуют код, написанный с использованием препроцессорного языка, в чистый и валидный *CSS*-код.

При помощи препроцессоров можно писать код, который нацелен на:

- читабельность для человека;
- структурированность и логичность;
- производительность.

И это лишь малая часть того, что может дать препроцессор.

Рассмотрим такое понятие, как «синтаксический сахар».

Синтаксический сахар (от англ. *syntactic sugar*) – это дополнения синтаксиса языка программирования, которые не вносят каких-то существенных изменений или новых возможностей, но делают этот язык более читабельным для человека.

Синтаксический сахар вводит в язык альтернативные варианты записи заложенных в этот язык конструкций. Под альтернативными вариантами записи стоит понимать более короткие или удобные конструкции для человека, которые, в конечном итоге, будут преобразовываться препроцессором в исходный язык, без синтаксического сахара. Если попытаться применить это понятие к *CSS*-препроцессорам, то оно, в общем случае, полностью описывает их суть.

Итог: основной задачей препроцессоров является упрощение и ускорение разработки.

Разновидности препроцессоров

Среди наиболее популярных препроцессоров выделяют:

- ***Less*;**
- ***Sass* (*Scss*);**
- ***Stylus***.

Также более менее заметными инструментами в данной области являются ***Closure Stylesheets***, ***CSS Crush***. Они менее популярны на данный момент, но все таки ими пользуются некоторые разработчики.

Смысл использования препроцессоров – это читабельность кода, структурирование и повышение производительности.

Существуют и другие причины использования препроцессоров. Это доступная документация, простота использования, структура и логичность кода, модульность.

Less

Самый популярный препроцессор.

Основан в 2009 году Алексис Сельер (*Alexis Sellier*) и написан на *JavaScript* (изначально был написан на *Ruby*, но Алексис вовремя сделал правильный шаг).

Имеет все базовые возможности препроцессоров и даже больше, но не имеет условных конструкций и циклов в привычном для нас понимании.

Основным плюсом является его простота, практически стандартный для *CSS* синтаксис и возможность расширения функционала за счёт системы плагинов.

***Sass* (*SCSS*)**

Самый мощный из *CSS*-препроцессоров. Имеет довольно большое сообщество разработчиков. Основан в 2007 году как модуль для *HAML* и написан на *Ruby* (есть порт на *C++*). Имеет куда больший ассортимент возможностей в сравнении с *Less*.

Возможности самого препроцессора расширяются за счёт многофункциональной библиотеки *Compass*, которая позволяет выйти за рамки *CSS* и работать, например, со спрайтами в автоматическом режиме.

Имеет два синтаксиса:

- ***Sass*** (***Syntactically Awesome Style Sheets***) – упрощённый синтаксис *CSS*, который основан на идентации. Считается устаревшим.
- ***SCSS*** (***Sassy CSS***) – основан на стандартном для *CSS* синтаксисе.

Stylus

Самый молодой, но в тоже время самый перспективный *CSS*-препроцессор. Основан в 2010 году. Говорят, это самый удобный и расширяемый препроцессор, а ещё он гибче *Sass*.

Написан на *JavaScript*. Поддерживает варианты синтаксиса от подобного *CSS* до упрощённого (отсутствуют :, ;, {} и некоторые скобки).

10 ПЛАТФОМЫ И ФРЕЙМВОРИ

Сегодня фреймворки стали неотъемлемой частью процесса веб-разработки из-за растущих стандартов веб-приложений и сложности используемых технологий.

Именно поэтому фреймворки отлично принимаются веб-разработчиками по всему миру для создания интуитивно понятных, интерактивных и функциональных веб-приложений.

Узнаем подробнее о них, а также о лучших фронтенд и бэкенд фреймворках, которые можно использовать.

Программная платформа, или фреймворк – это платформа для разработки программных приложений. Это абстракция, в которой программное обеспечение, предлагающее общие функциональные возможности, может быть при желании изменено благодаря возможности добавить дополнительный пользовательский код и, следовательно, предоставить программное обеспечение для конкретных приложений. Он предоставляет разработчикам основу для разработки и развертывания своих приложений и представляет собой универсальную программную среду для многократного использования.

Преимущества использования фреймворка заключаются в экономии времени, масштабируемом кодировании и безопасности.

Программные платформы могут включать компиляторы, вспомогательные программы, наборы инструментов, готовые библиотеки кода и *API* (интерфейсы прикладного программирования), которые объединяют различные компоненты для облегчения разработки системы или проекта.

Создание программного обеспечения сложный процесс. Он включает в себя множество задач, таких как проектирование, кодирование и тестирование. Только при кодировании программистам приходится заботиться о разделе объявлений, синтаксисе, выражениях, сборке мусора, исключениях и многом другом.

Программные фреймворки облегчают жизнь разработчикам веб-сайтов и приложений, позволяя им контролировать процесс разработки программного обеспечения с помощью единой платформы.

Преимущества использования программных фреймворков:

- экономия времени;
- масштабируемый код;
- безопасность.

Фронтенд относится к области приложения или веб-сайта, с которой посетители взаимодействуют напрямую. Интерфейсные веб-платформы снижают сложность работы программистам, которым приходится вручную создавать код, чтобы описывать поведение и взаимодействие между пользователями и приложением (сайтом). Эти фреймворки предлагают предварительно написанные коды, которые разработчики могут использовать как готовую базу.

Бэкенд фреймворки – это библиотеки серверных языков программирования. Они помогают в создании внутренней конфигурации веб-приложений.

Внутренние веб-платформы предоставляют инструменты, которые помогают в разработке таких задач, как авторизация пользователей, безопасность, маршрутизация *URL*-адресов и взаимодействие с базой данных. Использование этих фреймворков дает разработчикам преимущество, убирая необходимость настраивать и создавать все с нуля.

Представим десять лучших Бэкенд и Фронтенд фреймворков.

Фреймворк	Категория	Язык програм-мирования	Популярные приложения
1	*2*	*3*	*4*
React	Фронтенд	*Javascript*	*Facebook* *Yahoo* *Khan Academy*
Angular	Фронтенд	*Typescript*	*Gmail* *Forbes*
Vuejs	Фронтенд	*Javascript*	*Chargebee* *Yousign* *Infermedica*
jQuery	Фронтенд	*Javascript*	*Upwork* *LinkedIn* *Udemy*
Emberjs	Фронтенд	*Javascript*	*TED* *Netflix* *Square*
Django	Бэкенд	*Python*	*National Geographic* *Mozilla* *Pinterest*

1	*2*	*3*	*4*
Laravel	Бэкенд	*PHP*	*Deltanet Travel* *Neighborhood Lender* *World Walking*
Ruby on Rails	Бэкенд	*Ruby*	*Twitter* *Zendesk Github*
Cake PHP	Бэкенд	*PHP*	*Coconala* *Goodfirms* *Croogo*
Express JS	Бэкенд	*Node*	*Uber* *Groupon* *GoDaddy*

1. ***React* (*Frontend*)**

React, также известный как *ReactJS* или *React.js*, является интерфейсной библиотекой *JavaScript* с открытым исходным кодом для создания компонентов пользовательского интерфейса. Он поддерживается *Facebook*, а также сообществом отдельных компаний и разработчиков. *React* можно использовать в качестве основы при разработке мобильных или одностраничных приложений.

Однако его используют только для управления состоянием и передачи этого состояния в *DOM*. Это означает, что для создания приложений *React* потребуются дополнительные библиотеки для маршрутизации и определенные функции на стороне клиента.

Преимущества:

- бесперебойная и стабильная производительность с использованием виртуального *DOM*;
- возможность повторного использования компонентов облегчает совместную работу, а также их повторное использование в других частях приложения;
- идеальная альтернатива написанию компонентов при использовании *React Hook*, поскольку она позволяет разработчикам писать компоненты без классов, а также позволяет легко учить *React*;
- общий процесс создания компонентов сценариев упрощается с использованием *JSX*, бесплатного расширения синтаксиса;
- инструменты разработки *React* полезны и продвинуты;

- *react* ориентирован на *SEO*, а платформа поставляется с полным набором инструментов для разработчиков.

Характеристики:

- стабильный и безопасный код;
- гибкость в Интернете и на мобильных устройствах;
- однонаправленная привязка данных;
- виртуальный ДОМ;
- повышенная производительность и скорость;
- легкость в расширении и обучении;
- высокая совместимость с обширными библиотеками;
- *JSX* – расширение синтаксиса *JavaScript*;
- отладка выполняется быстрее и проще.

2. ***Angular* (*Frontend*)**

Список лучших фреймворков для разработки фронтенда будет точно неполным без *Angular*. Это фреймворк основан на *TypeScript*. Разработанный компанией *Google*, он был официально запущен в 2016 году. *Angular* был разработан, чтобы преодолеть разрыв между растущими требованиями технологий и традиционными концепциями, которые показывали отличные результаты.

Angular уникален своей функцией двусторонней привязки данных, что означает синхронизацию в реальном времени между представлением и моделью. Таким образом, когда в модели происходит какое-либо изменение, оно мгновенно отражается на представлении, и наоборот.

Angular идеально подходит не только для веб-приложений или мобильных приложений, с его помощью можно разрабатывать многостраничные и растущие веб-приложения. Такие компании, как *Blender*, *Forbes*, *BMW* и *Xbox* уже развертывают приложения, разработанные с использованием *Angular*.

Преимущества:

- фреймворк *Angular* имеет встроенную функциональность для мгновенного обновления изменений, внесенных в модель, с использованием представления и наоборот;
- архитектура, основанная на компонентах, обеспечивает более высокое качество кода;
- компоненты являются многократно используемыми и легко управляются с помощью инъекции зависимостей;

- разработчики могут легко обнаружить ошибки, сохранить код четким и понятным, а также устранить ошибки по мере их ввода, поскольку *Angular* построен на *TypeScript*;
- асинхронные вызовы данных могут быть легко обработаны с помощью *RxJS* (библиотека, широко используемая в *Angular*);
- долгосрочная поддержка *Google* (*LTS*) гарантирует, что *Google* планирует и дальше развивать фреймворк, поэтому разработчики имеют доступ к обширному сообществу для поддержки и обучения.

Характеристики:
- двусторонняя привязка к данным;
- кроссплатформенность;
- архитектура *MVC* (*Model-View-Controller*);
- модульная структура разработки;
- иерархическая инъекция зависимостей;
- простота в сопровождении;
- виртуальная прокрутка;
- меньше кода;
- обеспечивает большую гибкость;
- высокая производительность для одностраничных приложений большого объема;
- основана на *TypeScript* (расширенная версия языка программирования *JavaScript*);
- долгосрочная поддержка *Google*.

3. ***Vuejs***

Vue.js является моделью представления-*viewmodel* (*MVVM*), фреймворком *JavaScript* с открытым исходным кодом для разработки одностраничных приложений и пользовательских интерфейсов. Он был создан Эваном Вами, и структура поддерживается непосредственно им и его активными членами основной команды.

Vue – это простая и понятная платформа, которая хорошо справляется с проблемами, с которыми сталкиваются разработчики *Angular*. Она помогает пользователям выполнять множество задач и с легкостью справляется с динамичными и простыми процессами, включая мобильные и веб-приложения, а также развивающимися веб-приложениями.

Несмотря на то, что фреймворк создан для оптимизации производительности приложений и решения сложных задач, он не

пользуется большой популярностью среди рыночных гигантов. Однако *Xiaomi*, *Alibaba* и *Reuters* используют эту структуру. Несмотря на меньшее количество желающих из Силиконовой долины, *Vue* продолжает расти в отношении количества пользователей.

Преимущества:

- простота – это огромное преимущество фреймворка, которое позволяет разработчикам достигать хороших результатов при необходимости кодировать всего несколько строк;
- однофайловые компоненты могут хранить все коды в одном файле и требуют относительно небольших затрат;
- *Vue.js* может быть легко интегрирован в другие фреймворки, такие как *React*;
- удобный и простой в освоении, так как программистам требуется только знать основы *CSS*, *HTML* и *JavaScript*;
- фреймворк можно использовать с обычными редакторами;
- все его функции легко доступны, и приложения могут быть настроены в соответствии с конкретными потребностями;
- обеспечивает большую гибкость и меньше ограничений;
- хорошая документация;
- большое сообщество, предлагающее поддержку в обучении и часто публикующее обновленную информацию.

Характеристики:

- виртуальный *DOM*;
- встроенные *CSS*-переходы и анимации;
- привязка данных с помощью *v-bind*;
- шаблоны на основе *HTML*;
- малый размер (большая совместимость) ;
- простой синтаксис и интеграция;
- предлагает организованную документацию
- легко понять;
- наблюдатели (обрабатывает изменения данных) ;
- *Vue*-маршрутизатор (выполняет навигацию между страницами).

4. ***jQuery***

Запущенный в 2006 году, *jQuery* является одним из самых ранних фроненд-фреймворков, и, несмотря на дату запуска, он продолжает оставаться актуальным в современном мире технологий.

Этот фреймворк предлагает легкость и простоту использования, а также сводит к минимуму необходимость написания обширного кода *JavaScript*. Кроме того, существует обширное сообщество *jQuery*, на которое разработчики могут положиться в поисках решений.

Будучи по своей сути библиотекой, этот фреймворк используется для манипулирования *DOM* и *CSS*, а также для оптимизации интерактивности и функциональности сайта. Хотя в прошлом разработка мобильных приложений с помощью *jQuery* была невозможна, последние разработки помогли расширить границы.

Более того, последние разработки *jQuery* помогают разработчикам создавать нативные мобильные приложения с помощью системы пользовательского интерфейса на базе *HTML5* – *jQuery Mobile*. Кроме того, фронтенд-фреймворк дружественен к браузеру и поддерживает практически все браузеры.

Преимущества:

- широко используется благодаря своей простоте и легкости в использовании;
- интуитивно понятен и прост в освоении, поскольку библиотека построена с использованием более коротких и простых кодов;
- кроссбраузерная совместимость;
- понятный, мощный и простой синтаксис облегчает выбор элементов *DOM*;
- легкая и компактная Библиотека *jQuery*;
- библиотека с открытым исходным кодом ;
- классные эффекты и анимация;
- высокая расширяемость, страницы загружаются быстрее;
- *jQuery* удобен для *SEO* и облегчает работу с динамическим контентом.

Характеристики:

- манипуляция *DOM* ;
- *CSS* манипуляция;
- работа с *HTML*;
- выбор элемента *DOM*;
- утилиты;
- анимации и эффекты;
- методы событий *HTML*;
- *AJAX*;
- расширяемость с помощью подключаемых модулей;

- разбор *JSON*.

5. ***Emberjs* (*Frontend*)**

Веб-фреймворк *JavaScript* с открытым исходным кодом, *Ember.js* использует шаблонную систему. Это позволяет разработчикам разрабатывать масштабируемые одностраничные веб-приложения за счет включения в структуру лучших практик, распространенных идиом и шаблонов из других шаблонов экосистемы одностраничных приложений.

Square, *Apple Music*, *LinkedIn*, *Chipotle* и *Twitch* – вот некоторые из популярных веб-сайтов, на которых *Ember.js* используется. Несмотря на то, что это в основном считается основой для работы с Интернетом, *Ember.js* также помогает в создании мобильных и ПК приложений.

Apple Music является одним из наиболее ярких примеров *Ember.js* настольное приложение, которое является функцией приложения *iTunes* для ПК. Компания *Tilde Inc*. владеет торговой маркой *Ember*.

Преимущества:

- более быстрое развитие благодаря интерфейсу командной строки *Ember*;
- высокая производительность;
- двусторонняя привязка данных;
- *Ember Inspector* – собственный инструмент отладки;
- хорошая организованность;
- понятная документация;
- легко интегрируется с большими командами;
- стабильность без простоев;

Характеристики:

- *HTML* и *CSS* лежат в основе модели развития *Ember.js;*
- фреймворк используется для создания поддерживаемых и повторно используемых веб-приложений *JavaScript;*
- предоставляет инициализаторы экземпляра;
- инструмент *Ember Inspector* для отладки приложений *Ember;*
- маршруты являются основными функциями платформы и используются для управления *URL*-адресами;
- использует шаблоны для автоматического обновления модели в случае изменения содержимого приложений.

6. ***Django***

Django – это бесплатный веб-фреймворк на основе *Python* с открытым исходным кодом. Он следует архитектурному шаблону *MTV* (модель-шаблон-представления). *Django Software Foundation (DSF)* – это американская независимая организация, которая поддерживает *Django*.

Основная цель *Django* – упростить создание сложных веб-сайтов, управляемых базами данных. Фреймворк делает акцент на подключаемости и повторном использовании компонентов, низком уровне связности, меньшем количестве кода, принципе «не повторяйся» и быстрой разработке.

Python используется повсеместно, даже для файлов, настроек и моделей данных. Фреймворк также предоставляет административный интерфейс создания, чтения, обновления и удаления, который является необязательным и динамически генерируется с помощью самодиагностики. Интерфейс настраивается с помощью моделей администратора.

Преимущества:

- *Django* написан на языке *Python*, что означает, что его легко изучить;
- *Django* и *Python* являются основными решениями в *IT*-гигантах (*Google*, *NASA* и др.), крупнейших компаниях, *IoT* и *FinTech* компаниях в Кремниевой долине;
- включает ряд функциональных возможностей, таких как *MVC*-верстка, бесплатный *API*, невероятный ПЗУ, поддержку нескольких языков и нескольких сайтов, поддержка *AJAX*, работа с сессиями, легкая миграция баз данных и многое другое;
- обширные руководства и документация;
- интерфейс администрирования;
- огромное сообщество, которое предлагает ресурсы и обновления;
- масштабируемость и настраиваемость;
- безопасность;
- встроенная система шаблонов.

Характеристики:

- *Python* веб-фреймворк;
- отличная документация;
- высокая масштабируемость;
- *SEO*-поддержка;

- обеспечивает высокую безопасность;
- универсальная природа;
- тщательно протестирован на динамические изменения в отрасли;
- способствует быстрому развитию.

7. ***Laravel***

Созданный Тейлором Отвеллом, *Laravel* – это бесплатный веб-фреймворк *PHP* с открытым исходным кодом, основанный на *Symfony*. Его исходный код размещен на *GitHub*. Платформа была предназначена для разработки веб-приложений по архитектурному шаблону *MVC* (модель-представление-контроллер).

Некоторые из функций платформы – это утилиты, которые помогают в обслуживании и развертывании приложений, различные способы доступа к реляционным базам данных, ориентация на синтаксическое удобство и модульная система упаковки с выделенным менеджером зависимостей.

Преимущества:

- *Laravel* упрощает реализацию аутентификации, наряду с организацией аутентичной логики и контролем доступа к ресурсам;
- доступны драйверы для *Mailgun*, *SMTP*, *SparkPost*, почтовой функции *PHP*, *Amazon SES* и *Sendmail*, а также простой и понятный *API* через библиотеку *SwiftMailer*;
- поддерживает популярные бэкенды кэша, такие как *Redis* из коробки и *Memcached*;
- высокозащищенные веб-приложения, поскольку *Laravel* защищает от подделки межсайтовых запросов, внедрения *SQL* и межсайтовых сценариев;
- обработка исключений и ошибок уже настроена для всех новых проектов на базе *Laravel*;
- работа по тестированию автоматизирована и *phpunit.xml* файл уже настроен для приложения;
- унифицированный *API* для широкого спектра различных бэкендов очередей.

Характеристики:

- встроенные облегченные шаблоны и множество виджетов, включающих код *JS* и *CSS*;
- поддержка архитектуры *MVC*;
- надежная защита веб-приложений;

- функционально важное *ORM* (объектно-реляционное сопоставление) ;
- встроенные командные инструменты *Artisan*, которые отвечают за автоматизацию утомительных повторяющихся задач программирования;
- предустановленные модульные и объектно-ориентированные библиотеки.

8. ***Ruby on Rails***

Ruby on Rails, или просто *Ruby*, написан на языке *Ruby* под лицензией *MIT License*. Это серверный фреймворк для веб-приложений. Он представляет собой структуру *MVC* (модель-представление-контроллер) и предоставляет веб-сервис, стандартные структуры для базы данных и веб-страниц.

Ruby облегчает и поощряет использование веб-стандартов, таких как *XML* или *JSON* для передачи данных и *CSS*, *JavaScript* и *HTML* для взаимодействия с пользователем. Помимо *MVC*, фреймворк уделяет большое внимание использованию известных парадигм и инженерных паттернов, таких как *CoC* (конвенция над конфигурацией), *DRY* (не повторяйся) и паттерн активной записи.

Появление *Ruby on Rails* в 2005 году оказало большое влияние на разработку веб-приложений благодаря таким современным возможностям, как миграции, продуманные создатели таблиц базы данных и построение представлений для быстрой разработки приложений.

Даже сегодня влияние *Ruby on Rails* на другие веб-фреймворки остается очевидным: так много фреймворков на разных языках заимствуют его идею, включая *Catalyst* на *Perl*, *Django*на *Python*, *Grails* на *Groovy*, *Sails.js* на *Node.js*, *CakePHP*, *Yii* и *Laravel* на *PHP*, *Play* на *Scala* и *Phoenix* на *Elixir*.

Преимущества:

- фреймворк на 100% бесплатный и работает под *Linux*;
- построен на архитектуре *MVC*, которая централизует бизнес-логику приложения и правила манипулирования данными;
- изменениями можно легко управлять;
- встроенные меры безопасности;
- исключительная производительность;
- высокая гибкость, поскольку веб-приложения могут использовать возможности бэкенда и фронтенда *Rails*;

- обеспечивает высокую производительность и позволяет разработчикам быстро создавать функции в сочетании с библиотеками сторонних разработчиков;

- большое сообщество, предлагающее решения и поддержку.

Характеристики:

- основан на архитектуре *MVC* (модель-представление-контроллер).

- *RoR* подчеркивает принципы *DRY* (не повторяйся) и *CoC* (конвенция над конфигурацией);

- большое сообщество *Rails* активно работает над устранением новых уязвимостей;

- модифицировать существующий код легко и просто;

- мощная и надежная библиотека, называемая *Active Record;*

- простой инструмент тестирования, называемый *RSpe;*

- объектно-ориентированный язык программирования, который лаконичен, прост и очень близок к английскому языку.

9. ***Express JS***

Выпущено как бесплатное программное обеспечение с открытым исходным кодом по лицензии *MIT*, *Express* или *Express.js,* является основой бэкенда веб-приложения для *Node.js*. Платформа предназначена для создания *API* и веб-приложений. *Express* был назван де-факто стандартной бэкенд платформой для *Node.js*.

Фреймворк описывается как сервер, вдохновленный Синатрой, Тиджеем Головайчуком, настоящим автором. Это означает, что *Express* относительно минимален с широким спектром функций, доступных в виде плагинов.

Express.js является серверным компонентом многих популярных комплектов разработчиков приложений, включая стек *MERN*, *MEAN* или *MEVN*, наряду с библиотекой или фреймворком *JavaScript*.

Преимущества:

- простота настройки и конфигурации;

- способствует просто и быстрой разработке;

- позволяет разработчикам определять маршруты вашего приложения на основе методов *URL* и *HTTP;*

- простота интеграции с различными движками шаблонов, такими как *Vash*, *Jade*, *EJS* и так далее;

- платформа включает в себя различные модули промежуточного программного обеспечения, которые могут быть

использованы для выполнения дополнительных задач по ответу и запросу;

- *Middleware* – это часть структуры, которая позволяет получить доступ к базе данных;
- может быть создан сервер *REST API;*
- простота подключения к базам данных, таким, как *Redis* и *MySQL;*
- простота обслуживания ресурсов и статических файлов вашего приложения.

Характеристики:

- *Middleware* является частью платформы, которая позволяет получать доступ к базе данных;
- более быстрая разработка на стороне сервера;
- усовершенствованный механизм маршрутизации;
- высокопроизводительные механизмы создания шаблонов, которые разработчики могут использовать для создания динамического контента для веб-страниц путем создания *HTML*-шаблонов на стороне сервера;
- механизм отладки.

10. ***Cake PHP***

CakePHP следует подходу модель-представление-контроллер (*MVC*) и представляет собой платформу с открытым исходным кодом. Она написана на *PHP* и смоделирована по концепциям *Ruby*. Она распространяется по лицензии *MIT*.

Платформа использует популярные концепции разработки программного обеспечения, а также шаблоны проектирования программного обеспечения, такие как модель-представление-контроллер, обычная конфигурация, фронтальный контроллер, сопоставление данных ассоциации и активная запись.

Преимущества:

- предварительная настройка не требуется, и поэтому *CakePHP* избавляет разработчиков от множества хлопот;
- платформа с открытым исходным кодом и легко доступная для всех;
- встроенный *ORM* (объектно-реляционное сопоставление) позволяет разработчикам преобразовывать данные между несовместимыми системами с использованием объектно-ориентированного языка программирования;

- автоматическая генерация *CRUD*;
- можно создавать повторно используемые части кода, которые можно использовать более одного раза для разных проектов;
- платформа имеет возможность протестировать все критические и уязвимые точки вашего приложения;
- обеспечивает очень безопасную среду;
- правильное наследование классов;
- шаблон *MVC*;
- удобная лицензия, которую можно легко расширить с помощью плагинов и компонентов.

Характеристики:

- гибкое и простое лицензирование;
- дружелюбное, активное и большое сообщество;
- интегрированный *CRUD* для взаимодействия с базой данных;
- фреймворк хорошо совместим с версиями 4, а также 5 *PHP*;
- генерация кода;
- автоматическая кодогенерация;
- встроенная проверка;
- архитектура *MVC*;
- гибкое и быстрое создание шаблонов (синтаксис *PHP* вместе с помощниками) ;
- локализация;
- гибкий *ACL*;
- гибкое кэширование;
- очистка данных;
- компоненты файлов *cookie*, электронной почты, сеанса, безопасности и обработки запросов;
- просмотр помощников для *JavaScript*, *AJAX*, *HTML*-форм и т.д.;
- работает из любого каталога веб-сайтов, практически без настройки *Apache.*

Выбор подходящего комплекта технологий для разработки веб-сайтов и приложений может оказаться непростой задачей, но многое зависит от того, какая технология сочетается с ним и каким вариантом использования приложения.

Несмотря на то, что большинство разработчиков обычно предпочитают работать с фреймворками, с которыми они знакомы, у

них часто не остается иного выбора, кроме как экспериментировать с новыми фреймворками.

К лучшим фронтенд фреймворкам относят:
- *React;*
- *Angula;*
- *Vue.JS;*
- *Ember.JS;*
- *jQuery.*

К лучшим бэкенд фреймворкам относят:
- *Django;*
- *Ruby on Rails;*
- *CakePHP;*
- *Lavavel;*
- *Express.JS.*

Список использованных источников

1. Front-end. Клиентская разработка для профессионалов. Node.js, ES6, REST / К. Аквино, Т. Ганди. – СПб.: Питер, 2017. – 512 с.
2. Савельев, А.О. HTML5. Основы клиентской разработки / А.О. Савельев, А.А. Алексеев. – 2-е изд., испр. – Москва : Национальный Открытый Университет «ИНТУИТ», 2016. – 272 с.
3. Самовендюк, Н. В. Разработка приложений для интернет. FRONT-END : практикум по выполнению лабораторных работ по одноименной дисциплине для студентов специальности 1-40 04 01 "Информатика и технологии программирования" дневной формы обучения / Н. В. Самовендюк. – Гомель : ГГТУ им. П. О. Сухого, 2021. – 108 с.
4. Режим доступа: https://professorweb.ru/my/javascript/js_theory/level2/2_2.php
5. Режим доступа: https://metanit.com/web/
6. Режим доступа: https://frontend-stuff.com/blog/es6/
7. Режим доступа: http://www.wisdomweb.ru/
8. Режим доступа: https://cyberleninka.ru/article/n/sovremennye-printsipy-i-podhody-k-frontend-arhitekture-veb-prilozheniy/pdf
9. Режим доступа: https://depix.ru/articles/sovremennye_tehnologii_verstki_i_front_end?
10. Режим доступа: https://blog.back4app.com/ru/10-%D0%BB%D1%83%D1%87%D1%88%D0%B8%D1%85-%D1%84%D1%80%D0%B5%D0%B9%D0%BC%D0%B2%D0%BE%D1%80%D0%BA%D0%BE%D0%B2-%D0%B4%D0%BB%D1%8F-%D1%84%D1%80%D0%BE%D0%BD%D1%82%D0%B5%D0%BD%D0%B4%D0%B0-%D0%B8-%D0%B1%D1%8D/
11. Режим доступа: https://dan-it.com.ua/blog/razrabotka-so-storony-front-end-chto-jeto-takoe-i-chem-otlichaetsja-ot-back-end/
12. Режим доступа: https://blog.webf.zone/contemporary-front-end-architectures-fb5b500b0231
13. Режим доступа: https://habr.com/ru/post/321312/

Титова Людмила Константиновна

СОВРЕМЕННЫЕ ТЕХНОЛОГИИ FRONT-END РАЗРАБОТКИ

**Учебно-методическое пособие
для студентов специальности
1-40 05 01 «Информационные системы и технологии
(по направлениям)»
дневной и заочной форм обучения**

Подписано к размещению в электронную библиотеку
ГГТУ им. П. О. Сухого в качестве электронного
учебно-методического документа 30.08.23.
Рег. № 47Е.
http://www.gstu.by