Н. А. Рындин

ТЕХНОЛОГИИ РАЗРАБОТКИ КЛИЕНТСКИХ WEB-ПРИЛОЖЕНИЙ НА ЯЗЫКЕ JAVASCRIPT

Учебное пособие

Воронеж 2020

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО
ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное бюджетное
образовательное учреждение высшего образования
«Воронежский государственный технический университет»

Н. А. Рындин

ТЕХНОЛОГИИ РАЗРАБОТКИ КЛИЕНТСКИХ WEB-ПРИЛОЖЕНИЙ НА ЯЗЫКЕ JAVASCRIPT

Учебное пособие

Воронеж 2020

УДК
ББК

Рецензенты:

кафедра программирования и информационных технологий Воронежского государственного университета (зав. кафедрой д-р техн. наук, проф. С. Д. Махортов); д-р техн. наук, проф. О. Я. Кравец

Рындин, Н. А.
Технологии разработки клиентских WEB-приложений на языке JavaScript: учеб. пособие / Н. А. Рындин: ФГБОУ ВО «Воронежский государственный технический университет». Воронеж: Изд-во ВГТУ, 2020. 50 с.

ISBN

Учебное пособие содержит теоретический и практический материал согласно программе дисциплины «Технологии разработки клиентских WEB-приложений», а также практические задания. Предназначено для студентов 2 курса магистратуры.

Издание предназначено для студентов обучающихся по направлению магистратуры 09.04.02 «Информационные системы и технологии» (направленность «Разработка WEB-ориентированных информационных систем»).

Ил. 3. Табл. 9. Библиогр.: 8 назв.

УДК
ББК

Печатается по решению редакционно-издательского совета Воронежского государственного технического университета

ISBN

Основы языка JavaScript

Введение

JavaScript — основной язык в web-разработке, т.к. все современные web-браузеры включают в себя его интерпретатор. Развитие экосистемы языка, обеспечило ему чрезвычайную применимость, т.к. существует масса фреймворков, позволяющих использовать JavaScript-код на различных платформах: от серверов до микроконтроллеров. Благодаря этому и другим преимуществам JavaScript занимает 7-е место в рейтинге популярности языков программирования TIOBE (октябрь 2020 года).

JavaScript был создан Бренданом Эйхом в компании Netscape в 1995 году. Название «JavaScript» является торговой маркой, зарегистрированной компанией Sun Microsystems (ныне Oracle), и используется для обозначения реализации языка, созданной компанией Netscape (ныне Mozilla). Компания Netscape представила язык для стандартизации европейской ассоциации производителей компьютеров ECMA (European Computer Manufacturer's Association), но из-за юридических проблем с торговыми марками стандартизованная версия языка получила название «ECMAScript». Таким образом можно считать, что ECMAScript (сокращенно — ES) — стандарт, а JavaScript — наиболее популярная версия реализации этого стандарта.

Стоит отметить, что существует некоторая путаница между JavaScript и Java. Оба языка имеют C-образный синтаксис, в остальном — это совершенно разные языки программирования.

Версии языка

JavaScript — динамично развивающийся язык. Начиная с издания ES 6 обновление стандарта ECMAScript происходит ежегодно. На сегодняшний день опубликованы одиннадцать изданий стандарта.

Первые три версии ECMA Script (ES1, ES2, ES3) были ежегодными обновлениями, тогда как издание ES4 так и не было выпущено из-за политических разногласий. И только спустя десятилетие издание ES5 было выпущено с несколькими существенными дополнениями.

Как было упомянуто ранее, начиная с версии ES 6, было принято решение о переходе на ежегодные обновления стандарта, после чего издание ES 6 было переименовано в ES 2015, таким образом все последующие издания стандарта нумеруются по году их выпуска.

Таблица 1

Версии ECMAScript

Версия	Год публикации	Название издания	Описание издания
1	1997	ES 1	Первое издание
2	1998	ES 2	Только редакционные изменения
3	1999	ES 3	Добавлены: • регулярные выражения; • конструкция try… catch
4		ES 4	Не выпущен
5	2009	ES 5	Добавлены: • директива use strict; • аксессоры getters и setters; • поддержка JSON; • возможность использовать зарезервированные слова в качестве ключей свойств и ставить запятые в конце массива; • многострочные строковые литералы
5.1	2011	ES 5	Редакционные изменения
6	2015	ES 6 / ES 2015	Добавлены: • let и const; • деструктурирующее присваивание; • стрелочные функции; • промисы; • метод Array.find (); • метод Array.findIndex ()
7	2016	ES 2016	Добавлены: • оператор возведения в степень (**); • метод Array.prototype.includes

Версии ECMAScript

8	2017	ES 2017	Добавлены: • функция string padding; • новые свойства объекта; • поддержка асинхронности (async/await); • разделение памяти и объект Atomics

Продолжение табл. 1

Версия	Год публикации	Название издания	Описание издания
9	2018	ES 2018	Добавлены: • rest / spread параметры; • асинхронные итераторы; • метод Promise.finally(); • Улучшение функционала регулярных выражений
10	2019	ES 2019	Добавлены: • метод Array.prototype.{flat,flatMap}; • метод Object.fromEntries; • метод String.prototype.{trimStart, trimEnd}; • optional catch binding
11	2020	ES 2020	Добавлены: • метод String.prototype.matchAll; • динамический import(); • тип BigInt; • метод Promise.allSettled; • объект globalThis; • механизм выполнения цикла for-in; • оператор Optional Chaining; • оператор Nullish Coalescing

Важно отметить, что, начиная со стандарта ES 5, все последующие стандарты не имеют обратной совместимости с более ранними версиями. Поэтому, для избежания ошибок в обработке кода и приведения кода к современному стандарту крайне рекомендуется использовать директиву use sctrict, которая ставится в начале скрипта.

Среда разработки

Разработчики JavaScript имеют доступ к различным редакторам кода, которые делают работу более эффективной. В этом случае наиболее замечательная функциональность в принципе включена во все распространенные редакторы (эта функциональность включает в себя подсветку и завершение кода, интеграцию с Git, поддержку плагинов). Тем не менее, у большинства разработчиков есть свой любимый редактор, который отличается от других небольшими, но важными деталями. Приведем и рассмотрим наиболее популярные редакторы.

Visual Studio Code

Visual Studio Code (сокращенно — VSCode) — это бесплатный редактор от компании Microsoft с открытым исходным кодом. VSCode отлично подходит для начинающих разработчиков JavaScript, так как он имеет хороший набор готовых к использованию функций без необходимости установки дополнительных плагинов. Но этот редактор популярен не только среди начинающих программистов. Это может быть идеальным выбором для более опытных пользователей, которым нужно просто начать быстро, не тратя много времени на настройку.

Уникальной особенностью VSCode является возможность использовать его в браузере. Таким образом, вы можете использовать редактор на планшете и в то же время иметь ту же среду, в которой вы работали в настольной версии. Чтобы эта функция работала, вам все равно нужно настроить сервер кодов в сети, к которой у вас есть доступ, но вам придется только один раз поработать с настройками, и это того стоит.

Atom

Другой бесплатный редактор кода, Atom, был разработан GitHub. На самом деле это специализированная версия браузера Chrome, преобразованная в редактор кода.

Готовый к использованию редактор Atom не такой мощный, но есть большое количество доступных плагинов, благодаря которым можно получить буквально всю необходимую функциональность. Создавая удобную среду разработки для себя, вы неизбежно соберете довольно большой персональный набор плагинов.

Используя плагин телетайпа, несколько человек могут работать над одним файлом одновременно, и у каждого будет несколько курсоров на экране. Эта функциональность может использоваться для наставничества, парного программирования или просто совместной работы. Это одна из особенностей, которые выделяют Atom.

Интеграция git в Atom хорошо реализована (было бы странно ожидать иначе от программного обеспечения, разработанного GitHub). Вам также может пригодиться плагин git-plus, позволяющий запускать команды с помощью сочетаний клавиш без использования терминала git.

Атом настолько настраиваемый, что вы можете даже отредактировать файл .less, чтобы выбрать цвета, которые вам нравятся больше всего. Это значительное преимущество для тех, кто любит настраивать каждую деталь своего окружения.

Скрипт .coffe, который запускается при запуске, позволяет быстро изменить поведение редактора.

Sublime Text

Sublime Text — это платный редактор с бесплатной пробной версией. Он не имеет большого количества предустановленных плагинов, но в целом эти плагины, во всем их разнообразии, для любого типа потребностей. Некоторые пакеты, такие как SideBarEnhancements (который позволяет переименовывать, перемещать, копировать и вставлять файлы и папки), вероятно, должны быть интегрированы в редактор, но вы можете загрузить и установить их.

Как и в случае с Atom, первоначальная настройка этого редактора может занять некоторое время. Но когда все настроено, все работает плавно, плавно.

Sublime Text это легковесный редактор, благодаря чему он очень быстр и может работать с большими проектами и объемными файлами.

VIM

Vim — это бесплатный и легко настраиваемый редактор кода. Это был первый текстовый редактор, разработанный для Unix и получивший название vi. Позже его функциональность была значительно расширена - так выглядел Vim. Этот редактор доступен в большинстве дистрибутивов Linux.

Vim имеет функции поиска и подсветки синтаксиса, а также очень легкий, благодаря чему он может обрабатывать даже очень большие файлы. Только здесь потребуется некоторое время, чтобы настроить и подготовиться к работе.

Этот редактор также имеет графический интерфейс, но не по умолчанию. Даже включение поддержки мыши требует определенных манипуляций. По умолчанию Vim управляется клавишами и сочетаниями клавиш.

Выполнение скриптов на JavaScript

JavaScript является интерпретируемым языком. Самый простым способ запустить скрипт, написанный на этом языке — воспользоваться интерпретатором, который встроен в любой современный web-браузер. После этого уже сам браузер делает все необходимое, чтобы выполнить код. Скрипт можно написать непосредственно в консоли JavaScript, которая, на примере браузеров Chrome, Safari и Firefox, обычно располагается в разделе меню «Разработка», либо интегрировать скрипт непосредственно в HTML-страницу.

JavaScript подключается к странице происходит так же, как подключение файла CSS — код можно писать непосредственно внутри разметки страницы, а

можно вынести в отдельный файл. Для написания кода внутри разметки используется HTML-тега <script>. Пример подключения JavaScript-кода внутри разметки:

```
        <head>
<script>
setTimeout(delay,                                                    5000);
function                                delay()                          {
  alert(«Hello,                                                    World»);
}
</script>
</head>
```

В остальных случаях код выносят в отдельный файл, подключаемый в HTML-коде:

```
<script src="/path/to/somescript.js"></script>
```

В качестве значения атрибута src указывается путь к файлу, содержащему скрипт. Браузер скачивает и выполняет его. Можно указать скрипт, который расположен на другом сервере:

```
<script src="https://sample.url.com/path/to/somescript.js"></script>
```

Синтаксис, переменные и выражения

Синтаксис языка

Этот раздел объясняет основные синтаксические принципы JavaScript.
Несколько примеров синтаксиса:

```
// Две косые черты начинают однострочные комментарии

var x; // объявляем переменную

x = 3 + y; // присваиваем значение переменной `x`

foo (x, y); // вызываем функцию `foo` с параметрами` x` и `y`

obj.bar (3); // вызываем метод `bar` объекта` obj`

// Условный оператор
if (x === 0) {// Значение `x` равно нулю?
   x = 123;
}

// Определение функции `baz` с параметрами` a` и `b`
function baz (a, b) {
   вернуть a + b;
}
```

Важно обратить внимание на два разных использования знака равенства:

- один знак равенства (=) используется для присвоения значения переменной;
- двойной знак равенства (==) используется для сравнения на равенство (сравнивается значение с приведением типов);
- тройной знак равенства (===) используется для сравнения на идентичность (сравниваются значения и типы).

Точки с запятой необязательны в JavaScript. Тем не менее, рекомендуют всегда включать их, потому что в противном случае JavaScript может неправильно угадать конец оператора.

Точки с запятой завершают операторы, но не блоки. В одном случае вы увидите точку с запятой после блока: выражение функции — это выражение, которое заканчивается блоком. Если такое выражение стоит последним в утверждении, за ним следует точка с запятой.

JavaScript имеет два вида комментариев: однострочные и многострочные. Однострочные комментарии начинаются с // и заканчиваются концом строки:

```
x++; // однострочный комментарий
```

Многострочные комментарии ограничены /* и */:

```
/* Это
многострочный
комментарий */
```

Идентификаторы — это имена, которые играют различные синтаксические роли в JavaScript. Например, имя переменной является идентификатором. Идентификаторы чувствительны к регистру.

Следующие идентификаторы являются зарезервированными словами — они являются частью синтаксиса и не могут использоваться в качестве имен переменных (включая имена функций и имена параметров).

Таблица 2

Зарезервированные идентификаторы

arguments	break	case	catch
class	const	continue	debugger
default	delete	do	else
enum	export	extends	false
finally	for	function	if
implements	import	in	instanceof
interface	let	new	null

Зарезервированные идентификаторы

package	private	protected	public
return	static	super	switch
this	throw	true	try
typeof	var	void	while

Типы данных и переменные языка

Примитивное значение — это элемент любого из встроенных типов. Объект — это элемент еще одного встроенного типа Object.

Примитивы:

Undefined — неопределенный тип.

Boolean — примитивный тип данных в информатике, которые могут принимать два возможных значения, иногда называемых истиной (true) и ложью (false).

Number — числовой тип данных в формате 64-битного числа двойной точности с плавающей запятой.

String — строковый тип данных.

BigInt — числовой тип данных, который может представлять данные в формате длинной арифметики.

Null — специальный примитив, используемый не только для данных но и в качестве указателя на финальную точку в Цепочке Прототипов.

Object — простая структура, используемая не только для хранения данных данных, но и для создания других структур, где любая структура создаётся с использованием ключевого слова new: new Object, new Array, new Map, new Set, new WeakMap, new WeakSet, new Date и множество других структур.

Function — случай, упрощающий определение типа для функций, несмотря на то, что все функции конструктивно унаследованы от Object.

Константы (const) используются для задания постоянных значений. В JavaScript имеется несколько типов констант, соответствующих его встроенным типам, а именно:

- нулевая константа null типа Null;
- логические константы true (истина) и false (ложь) типа Boolean;
- строковые константы типа String, например, "Привет всем!";
- числовые константы типа Number, например, 21 или 3.1415926.

Переменные используются в качестве символических имен, принимающих различные значения. Имена переменных задаются идентификаторами. Переменная создается в момент ее декларации.

Мы можем объявить переменные для хранения данных с помощью ключевых слов var, let или const.

let — это современный способ объявления.

var — это устаревший способ объявления.

const – похоже на let, но значение переменной не может изменяться.

Динамическая типизация

JavaScript является слабо типизированным или динамическим языком. Это значит, что вам не нужно определять тип переменной заранее. Тип определится автоматически во время выполнения программы. Также это значит, что вы можете использовать одну переменную для хранения данных различных типов:

```
var qw = 42; // сейчас qw типа Number
var qw = "car"; // а теперь qw типа String
var qw o = true;  // qw становится типа Boolean
```

Инициализатор — это любое выражение, значение которого присваивается переменной при ее создании.

Каждая переменная имеет свою область действия, в которой она определенна и в которой она имеет значение. Область действия переменной определяется положением ее декларации в тексте программы.

Выражения и операции

Выражения в JavaScript, как и в других языках программирования, представляют собой комбинации переменных, констант и операций, дающие осмысленный результат. Этот результат может быть числом, текстовой строкой, логическим значением или объектом. Соответственно все выражения JavaScript подразделяются на арифметические, строковые, логические и объектные.

Существует два типа выражений: те, которые присваивают значение некоторой переменной (например, x = 2 + 3), и те, которые просто имеют некое значение (например, 2 + 3). Первый тип выражений называется операциями присваивания.

Все операции в JavaScript подразделяются на следующие:

- операции сравнения;
- арифметические операции;
- битовые операции;
- логические операции;
- строковые операции;
- операции присваивания;
- прочие операции.

Операции сравнения сравнивают два операнда и возвращают логическое значение, означающее результат этого сравнения. Строки сравниваются в лексикографическом порядке в кодировке Unicode. Если типы операндов различны, то делается попытка преобразовать их к одному типу. При этом:

- операции «больше», «меньше», «не больше» и «не меньше» сначала пытаются преобразовать операнды в числа, а, если это невозможно, то в строки, а затем производят их сравнение;
- операции «равно» и «не равно» пытаются преобразовать операнды в строки, затем в числа и в логические значения, а затем производят их сравнение;
- операции «тождественно» и «не тождественно» не преобразовывают типы данных: два операнда считаются тождественно равными, если они имеют одинаковые типы и одинаковые значения.

Таблица 3

Операции сравнения

Оператор	Описание	Примеры, возвращающие true
Равно (==)	Возвращает true, если операнды равны	3 == var1 "3" == var1 3 == '3'
Не равно (!=)	Возвращает true, если операнды не равны	var1 != 4 var2 != "3"
Строго равно (===)	Возвращает true, если операнды равны и имеют одинаковый тип. См. также Object.is и sameness in JS	3 === var1
Строго не равно(!==)	Возвращает true, если операнды не равны и/или имеют разный тип.	var1 !== "3" 3 !== '3'
Больше (>)	Возвращает true, если операнд слева больше операнда справа.	var2 > var1 "12" > 2
Больше или равно (>=)	Возвращает true, если операнд слева больше или равен операнду справа.	var2 >= var1 var1 >= 3
Меньше (<)	Возвращает true, если операнд слева меньше операнда справа.	var1 < var2 "2" < 12

Операции сравнения

Меньше или равно (<=)	Возвращает true, если операнд слева меньше или равен операнду справа.	var1 <= var2 var2 <= 5

Арифметические операции применяются к числовым операндам и возвращают числовое значение, означающее результат операции. Если типы операндов различны, то делается попытка преобразовать их к числовому типу.

При этом:

- операция «сложение» выполняется только тогда, когда оба операнда являются числами или логическими значениями. Если хотя бы один операнд является строкой, то производится конкатенация строк;
- остальные операции преобразуют операнды в числа, а затем выполняют операцию;
- операции «инкремент» и «декремент» применяются только к переменным.

Таблица 4

Арифметические операции

Оператор	Имя	Функция	Пример
+	сложение	Объединение чисел в одно целое.	6 + 9
-	вычитание	Вычитает правое число от левого.	20 - 15
*	умножение	Умножает два числа вместе.	3 * 7
/	деление	Делит левое число на правое.	10 / 5
%	модуль числа	Возвращает значение остатка при делении первого числа на второе. Результат будет иметь тот же знак, что и первое число.	11 % 3 = 2 (поскольку число 3 вмещается три раза, остатком будет число 2)

Арифметические операции

**	показатель степени	Возводит базовое число в указанную степень, то есть количество базовых чисел, указанных экспонентой, умножается вместе. Впервые он был представлен в ES 2016.	5 ** 5 (возвращает 3125, или как: 5*5*5*5*5)

При этом:

- операция «сложение» выполняется только тогда, когда оба операнда являются числами или логическими значениями. Если хотя бы один операнд является строкой, то производится конкатенация строк;
- остальные операции преобразуют операнды в числа, а затем выполняют операцию;
- операции «инкремент» и «декремент» применяются только к переменным.

На сегодняшний день JavaScript поддерживает единственную строковую операцию, а именно конкатенацию строк, которая обозначается символом «+». Если хотя бы один операнд является строкой, то результатом операции является слияние строк-операндов.

Битовые операции применяются к числовым операндам, представленным как двоичные числа (т. е. как цепочки из 32 битов), и возвращают числовое значение, означающее результат операции. Перед выполнением операции операнды преобразуются в целые числа, а результат операции преобразуется в Number.

Таблица 5

Битовые операции

Оператор	Использование	Описание
Побитовое И	a & b	Возвращает 1 в тех разрядах, которые у обоих операндов были равны 1.
Побитовое ИЛИ	a | b	Возвращает 1 в тех разрядах, которые хотя бы у одного из операндов были равны 1.

Битовые операции

Побитовое исключающее ИЛИ	a ^ b	Возвращает 1 в тех позициях, которые только у одного из операндов были равны 1.
Побитовое НЕ	~ a	Инвертирует биты операнда.
Сдвиг влево	a << b	Сдвигает двоичное представление числа a на b разрядов влево заполняя освободившиеся справа разряды нулями.
Арифметический сдвиг вправо	a >> b	Сдвигает двоичное представление числа a на b разрядов вправо. Освобождающиеся разряды заполняются знаковым битом.
Логический сдвиг вправо	a >>> b	Сдвигает двоичное представление числа a на b разрядов вправо. Освобождающиеся разряды заполняются нулями.

Логические операции применяются к логическим операндам и возвращают логическое значение, означающее результат операции. Если типы операндов различны, то делается попытка преобразовать их к логическому типу.

Таблица 6

Логические операции

Оператор	Использование	Описание
Логическое И (&&)	expr1 && expr2	Возвращает значение expr1, если оно может быть преобразовано в false; иначе возвращает значение expr2. Таким образом, при использовании с величинами типа Boolean оператор && вернет true, если оба операнда могут быть преобразованы в true; иначе оператор && вернет false.

Продолжение табл. 6

Оператор	Использование	Описание
Логическое ИЛИ (||)	expr1 || expr2	Возвращает значение expr1, если оно может быть преобразовано в true; иначе возвращает значение expr2. Таким образом, при использовании с величинами типа Boolean оператор || вернет true если хоть один из них равен true; в других случаях вернет false.
Логическое НЕ (!)	!expr	Возвращает false если значение expr можно привести к true; в противоположном случае возвращает true.

Операции присваивания присваивают левому операнду результат операции, который определяется правым операндом и самой операцией. Базовая операция присваивания имеет вид a = b, что означает: присвоить переменной a значение операнда b.

Таблица 7

Операции присваивания

Имя	Сокращенный оператор	Смысл
Присваивание	x = y	x = y
Присваивание со сложением	x += y	x = x + y
Присваивание с вычитанием	x -= y	x = x - y
Присваивание с умножением	x *= y	x = x * y
Присваивание с делением	x /= y	x = x / y
Присваивание по модулю	x %= y	x = x % y
Присваивание с левым сдвигом	x <<= y	x = x << y
Присваивание с правым сдвигом	x >>= y	x = x >> y
Присваивание с беззнаковым сдвигом вправо	x >>>= y	x = x >>> y
Присваивание с побитовым AND	x &= y	x = x & y

Операции присваивания

Имя	Сокращенный оператор	Смысл
Присваивание с побитовым XOR	x ^= y	x = x ^ y
Присваивание с побитовым OR	x |= y	x = x | y

Прочие операции включают большой набор разнообразных операций, к которым относятся такие операции, как условная операция, операция запятая, операция delete, операция new и другие. Более подробно узнать о них моно по одной из предложенных в конце раздела ссылок.

Управляющие конструкции языка JavaScript

Условный оператор IF

Не все программы являются прямыми дорогами. Например, мы можем захотеть создать ветвящуюся дорогу, где программа выбирает правильную ветвь в зависимости от ситуации. Это называется условным исполнением.

Условный оператор создается с помощью ключевого слова if в JavaScript. В простом случае мы хотим, чтобы какой-то код выполнялся тогда и только тогда, когда выполняется определенное условие. Например, мы можем захотеть показать квадрат ввода только в том случае, если на самом деле это число.

```
let theNumber = Number(prompt("Pick a number"));
if (!Number.isNaN(theNumber)) {
  console.log("Your number is the square root of " +
              theNumber * theNumber);
}
```

С этой модификацией, если вы введете «parrot», никакой вывод не отображается.

Ключевое слово if выполняет или пропускает оператор в зависимости от значения логического выражения. Решающее выражение записывается после ключевого слова в скобках, за которым следует инструкция для выполнения.

Функция Number.isNaN — это стандартная функция JavaScript, которая возвращает true только в том случае, если задан аргумент NaN. Функция Number возвращает значение NaN, когда вы даете ей строку, которая не представляет действительное число. Таким образом, условие переводится как «если theNumber не является числом, делайте это».

Оператор после if в этом примере заключен в фигурные скобки ({и}). Скобки можно использовать для группировки любого количества операторов в один оператор, называемый блоком. Вы также могли бы опустить их в этом

случае, так как они содержат только один оператор, но чтобы избежать необходимости думать о том, нужны ли они, большинство программистов JavaScript используют их в каждом заключенном в оболочку выражении, подобном этому. Мы в основном будем придерживаться этого соглашения в этой книге, за исключением случайных однострочных.

```
if (1 + 1 == 2) console.log("It's true");
// → It's true
```

Вы часто будете использовать не только код, который выполняется, когда условие выполнено, но и код, который обрабатывает другой случай. Этот альтернативный путь представлен второй стрелкой на диаграмме. Вы можете использовать ключевое слово else вместе с if для создания двух отдельных альтернативных путей выполнения.

Если у вас есть более двух путей на выбор, вы можете «связать» несколько пар if / else вместе. Вот пример:

```
let num = Number(prompt("Pick a number"));
if (num < 10) {
  console.log("Small");
} else if (num < 100) {
  console.log("Medium");
} else {
  console.log("Large");
}
```

Программа сначала проверит, меньше ли num, чем 10. Если это так, она выбирает эту ветвь, показывает «Small» и все готово. Если это не так, он принимает ветвь else, которая сама содержит секунду if. Если второе условие (<100) выполняется, это означает, что число находится в диапазоне от 10 до 100, и отображается «Среднее». Если это не так, выбирается вторая и последняя ветвь.

Оператор цикла DO/WHILE

Рассмотрим программу, которая выводит все четные числа от 0 до 12. Один из способов написать это следующим образом:

```
console.log(0);
console.log(2);
console.log(4);
console.log(6);
console.log(8);
console.log(10);
console.log(12);
```

Это работает, но идея написания программы состоит в том, чтобы заставить что-то меньше работать, а не больше. Если бы нам нужны были все четные числа меньше 1000, такой подход был бы неосуществимым. Нам нужен способ запуска фрагмента кода несколько раз. Эта форма управления потоком называется циклом.

Зацикливание потока управления позволяет нам вернуться к некоторой точке программы, где мы были раньше, и повторить ее с нашим текущим состоянием программы. Если мы объединяем это с привязкой, которая имеет значение, мы можем сделать что-то вроде этого:

```
let number = 0;
while (number <= 12) {
  console.log(number);
  number = number + 2;
}
// → 0
// → 2
//   … etcetera
```

Оператор, начинающийся с ключевого слова while, создает цикл. За словом while следует выражение в скобках, а затем утверждение, очень похожее на if. Цикл продолжает вводить этот оператор до тех пор, пока выражение выдает значение, которое дает значение true при преобразовании в логическое значение.

Привязка чисел демонстрирует, как привязка может отслеживать ход выполнения программы. Каждый раз, когда цикл повторяется, число получает значение, которое на 2 больше, чем его предыдущее значение. В начале каждого повторения его сравнивают с числом 12, чтобы определить, завершена ли работа программы.

В качестве примера, который действительно делает что-то полезное, теперь мы можем написать программу, которая вычисляет и показывает значение 210 (от 2 до 10-й степени). Мы используем две привязки: одну для отслеживания нашего результата и одну для подсчета того, как часто мы умножали этот результат на 2. Цикл проверяет, достигла ли вторая привязка еще 10, и, если нет, обновляет обе привязки.

```
let result = 1;
let counter = 0;
while (counter < 10) {
  result = result * 2;
  counter = counter + 1;
}
console.log(result);
// → 1024
```

Счетчик также мог бы начинаться с 1 и проверяться на <= 10, но по причинам, которые станут понятны в главе 4, хорошей идеей будет привыкнуть к счету от 0.

Цикл do - это структура управления, похожая на цикл while. Он отличается только одним моментом: цикл do всегда выполняет свое тело хотя бы один раз и начинает тестировать, должен ли он останавливаться только после этого первого выполнения. Чтобы отразить это, тест появляется после тела цикла.

```
let yourName;
do {
  yourName = prompt("Who are you?");
} while (!yourName);
console.log(yourName);
```

Эта программа заставит вас ввести имя. Он будет спрашивать снова и снова, пока не получит что-то, что не является пустой строкой. Применяя оператор преобразует значение в логический тип перед его отрицанием, а все строки, кроме "", преобразуются в true. Это означает, что цикл продолжается, пока вы не укажете непустое имя.

Оператор цикла FOR

Многие циклы следуют шаблону, показанному в примерах. Сначала создается привязка «счетчик» для отслеживания хода цикла. Затем следует цикл while, обычно с тестовым выражением, которое проверяет, достигло ли счетчик своего конечного значения. В конце тела цикла счетчик обновляется для отслеживания прогресса.

Поскольку этот шаблон очень распространен, JavaScript и аналогичные языки предоставляют несколько более короткую и всеобъемлющую форму цикла for.

```
for (let number = 0; number <= 12; number = number + 2) {
  console.log(number);
}
// → 0
// → 2
//   … etcetera
```

Эта программа в точности эквивалентна предыдущему примеру печати четных чисел. Единственное изменение состоит в том, что все операторы, связанные с «состоянием» цикла, группируются после for.

Скобки после ключевого слова for должны содержать две точки с запятой. Часть перед первой точкой с запятой инициализирует цикл, обычно путем определения привязки. Вторая часть - это выражение, которое проверяет, должен ли цикл продолжаться. Последняя часть обновляет состояние цикла после каждой итерации. В большинстве случаев это короче и понятнее, чем конструкция while.

Это код, который вычисляет 2^{10}, используя вместо вместо:

```
let result = 1;
for (let counter = 0; counter < 10; counter = counter + 1) {
  result = result * 2;
}
console.log(result);
// → 1024
```

Цикл for ... in используется для просмотра свойств объекта. Поскольку мы еще не обсуждали объекты, вам может не понравиться этот цикл. Но как только вы поймете, как объекты ведут себя в JavaScript, вы обнаружите, что этот цикл очень полезен.

```
for (variablename in object) {
   statement or block to execute
}
```

В каждой итерации одному свойству объекта присваивается одно имя переменной, и этот цикл продолжается до тех пор, пока все свойства объекта не будут исчерпаны.

Оператор прерывания BREAK

Наличие условия зацикливания создает false - не единственный способ завершения цикла. Существует специальное утверждение, называемое **break**, которое сразу же выпрыгивает из замкнутого цикла.

Эта программа иллюстрирует оператор break. Он находит первое число, которое больше или равно 20 и делится на 7.

```
for (let current = 20; ; current = current + 1) {
  if (current % 7 == 0) {
    console.log(current);
    break;
  }
}
// → 21
```

Использование оператора remainder (%) - это простой способ проверить, делится ли число на другое число. Если это так, остаток от их деления равен нулю.

Конструкция for в этом примере не имеет части, которая проверяет конец цикла. Это означает, что цикл никогда не остановится, пока не будет выполнен оператор break внутри.

Если бы вы удалили этот оператор break или случайно написали условие завершения, которое всегда выдает true, ваша программа застряла бы в бесконечном цикле. Программа, застрявшая в бесконечном цикле, никогда не закончит свою работу, что обычно плохо.

Если вы создадите бесконечный цикл в одном из примеров на этих страницах, вас, как правило, спросят, хотите ли вы остановить скрипт через несколько секунд. Если это не помогло, вам придется закрыть вкладку, в которой вы работаете, или в некоторых браузерах закрыть весь браузер, чтобы восстановиться.

Ключевое слово continue похоже на break в том смысле, что оно влияет на ход цикла. Когда в теле цикла встречается продолжение, управление выпрыгивает из тела и продолжает следующую итерацию цикла.

Оператор выбора SWITH

Код нередко выглядит так:

```
if (x == "value1") action1();
else if (x == "value2") action2();
else if (x == "value3") action3();
else defaultAction();
```

Существует конструкция, называемая switch, предназначенная для выражения такой «отправки» более прямым способом. К сожалению, синтаксис, который JavaScript использует для этого (который он унаследовал от линии языков программирования C / Java), несколько неловкий — цепочка операторов if может выглядеть лучше. Вот пример:

```
switch (prompt("What is the weather like?")) {
  case "rainy":
    console.log("Remember to bring an umbrella.");
    break;
  case "sunny":
    console.log("Dress lightly.");
  case "cloudy":
    console.log("Go outside.");
    break;
  default:
    console.log("Unknown weather type!");
    break;
}
```

Вы можете поместить любое количество меток в блок, открытый переключателем. Программа начнет выполнение с метки, соответствующей значению, которое было дано переключателю, или по умолчанию, если не найдено подходящего значения. Он будет продолжать выполняться даже через другие метки, пока не достигнет оператора break. В некоторых случаях, таких как «солнечный» случай в примере, это можно использовать для совместного использования некоторого кода между случаями (рекомендуется выходить на улицу как в солнечную, так и в облачную погоду). Но будьте осторожны — такой разрыв легко забыть, что приведет к тому, что программа выполнит код, который вы не хотите выполнять.

Практическое задание

Задание 1

а) Объявите две переменных: x и z.

б) Запишите строку «Лабораторная работа» в переменную x.

в) Скопируйте значение из x в z.

г) Выведите значение на экран z, используя функцию alert.

Задание 2

Вычислить площадь треугольника по всем известным формулам и вывести результат на экран.

Задание 3

$$x = ln\left|(y - \sqrt{|z|})(z - \frac{y}{a + \frac{z^2}{4}})\right|$$

Вычислить значение переменной по следующей формуле:

Работа с функциями и классами

Обзор функций JavaScript

Функция — это именованный набор операторов в JavaScript, которые выполняют как простые так и сложные задачи.

Функция в JavaScript определяется с помощью ключевого слова function, за которым следует имя, после чего следуют скобки (). Имена функций могут содержать буквы, цифры, знаки подчеркивания и знаки доллара (те же правила, что и для переменных). Скобки могут включать имена параметров, разделенные запятыми: (параметр1, параметр2, ...). Код, который должен быть выполнен функцией, помещен в фигурные скобки: {}.

Далее представлены примеры объявления обычных функций и их вызовов:

```
//Объявление функции со статусом «отложенного использования»
//декларация функции
function myFunction(p1,                    p2)                    {
  return p1 * p2;
}

myFunction(3,2) // возвращаемое значение будет равно 6

//анонимное функциональное выражение
const ordinary2 = function (p1, p2) {
return p1 * p2;
};
ordinary2(4,2) // возвращаемое значение будет равно 8

//именнованное функциональное выражение
const ordinary3 = function myName(p1, p2) {
return p1 * p2;
};
ordinary3(4,3) // возвращаемое значение будет равно 12
```

В данных примерах в функцию передаются 2 параметра p1 и p2, а возвращает эта функция произведение этих переданных параметров.

Ключевое слово return без выражения после него приведет к тому, что функция вернет undefined. Функции, которые вообще не имеют оператора return, аналогично возвращают undefined.

Также объявленные функции могут храниться в свойстве обычного объекта:

```
const obj = { addAsMethod: myFunction };
obj.addAsMethod(2, 4)  // возвращаемое значение будет равно 8
```

Помимо этого, функции в пределах своей логики могут вызвать как другие функции, так и сами себя (рекурсивные функции), пример работы рекурсивной функции:

```
function funcExpr(num) {
  if (num < 15) {
    console.log(num); //вывод числа в консоль браузера
    return funcExpr(++num);
  }
return num;
}
funcExpr(12);
```

В результате после вызова функции funcExpr в консоль браузера будет выведено 4 числа начиная с 12 по 15.

Также стоит иметь ввиду при использовании функциональных выражений имена функций доступны в пределах этого выражения, например:

```
const func = function funcExpr() { return funcExpr };
func() // вернёт функцию funcExpr()
funcExpr() // будет выведена ошибка, что функция не определена ReferenceError
```

В JavaScript есть так называемые стрелочные функции, которые обеспечивают более простой способ создания функций.

Синтаксис стрелочных функций

Пусть есть анонимное функциональное выражение:

```
const f = function (x, y, z) { return 123 };
```

Аналогично такое выражение можно записать используя стрелочную функцию:

```
const f = (x, y, z) => { return 123 };
```

Здесь тело функции стрелки является блоком. Но оно также может являться выражением. Следующий пример аналогичен предыдущему.

```
const f = (x, y, z) => 123;
```

Если функция имеет один параметр, то она может быть объявлена как в примере ниже:

```
const id = x => x;
```

Чаще всего стрелочные функции используются для передачи в качестве параметров другим функциям или методам.

```
> [1,2,3].map(x => x+1)
[ 2, 3, 4 ]
```

Этот пример показывает одно из преимуществ таких функций — краткость. Иначе было бы необходимо использовать анонимное функциональное выражение.

```
[1,2,3].map(function (x) { return x+1 });
```

В блоке функции JavaScript можно объявлять не только переменные, но и другие функции – вложенные, особенностью такого использования внутренних функций является доступ к внешним переменным.

```
function hypotenuse(a, b) {
function square(x) { return x*x; }
return Math.sqrt(square(a) + square(b));
}
```

К примеру функция square может обращаться к переменным a и b и имеет возможность изменять их.

Обзор классов JavaScript

Для того чтобы объявить класс необходимо использовать ключевое слово class и указать название класса, далее идут фигурных скобки в которых представлено тело класса.

Пример объявления:

```
class User {
// тело класса
}
```

Не обязательно указывать название класса, используя выражения класса, можно назначить класс переменной. Пример:

```
const UserClass = class{
// тело класса
}
```

Класс может содержать в себе constructor (параметр 1, параметр 2, …) — это метод, который инициализирует экземпляр класса.

Пример использования:

```
class User {
name;
constructor(name){
 this.name = name; // создание поля экземпляра и присваение ему начального значения
}}
const user = new User('Hello') // созданние объекта экземпляра класса
user.name; // обращение к публичному полю(name), чьё значение равно 'Hello'
```

В классах JavaScript, есть статические поля и методы. Для их создания необходимо перед объявлением поля или метода указать ключевое слово static. Статические поля и методы могут быть приватными. Рассмотрим класс с примерами статического поля и метода.

```
class User {
static #takenNames = [];
static isNameTaken(name) {
return User.#takenNames.includes(name);
}
name = 'Unknown';
constructor(name) {
this.name = name;
User.#takenNames.push(name);
}
}
const user = new User('Jon Snow');
User.isNameTaken('Jon Snow'); // => true
User.isNameTaken('Arya Stark'); // => false
```

isNameTaken() — это статический метод, который использует статическое приватное поле User.#takeNames для проверки принятого значения name.

Чаще всего статические поля и методы используются, когда их функционал общий для всех объектов и относится непосредственно к классу

Классы JavaScript поддерживают инкапсуляцию, наследование, полиморфизм.

Инкапсуляция предполагает сокрытие некоторых внутренностей класса, например, полей, методов(функций внутри классов). Скрытие поля осуществляется через префикс #.

Пример объявления и использования класса с приватным полем:

```
class User {
#name; // приватное поле
constructor(name) {
this.#name = name;
}
getName() {
return this.#name;
}
}
const user = new User('Hello');
```

Использовать приватное поле напрямую нельзя, возникнет синтаксическая ошибка.

Чтобы получить значение приватного поля необходимо обратиться к методу getName:

```
user.#name; // SyntaxError is thrown
user.getName();
```

Наследование в классах осуществляется с помощью ключевого слово extends.

Предыдущий пример класса User можно расширить следующим образом:

```
class ContentWriter extends User {
posts = [];
}
```

Класс ContentWriter наследует конструктор, поля и методы класса User.

Если необходимо вызвать конструктор родительского класса, то это можно сделать с помощью специальной функции super(). А если нужно обратиться к методу родителя, то также можно вызвать super.getName().

Полиморфизм в JavaScript эквивалентно используется также, как и в других языках программирования. Т.к. JavaScript является динамически типизированным языком необходимо с осторожностью работать с объектами, которые используют полиморфный класс — необходимо определить специфичное для типа данных поведение.

Практическое задание

Задание №1

Необходимо реализовать функцию, которая на вход принимает строку, а возвращает подстроку, являющейся самым длинным полиндром в этой строке. Полиндромом называется такая последовательность символов или чисел,

которая одинаково читается в обоих направлениях, например, слово "топот", "шалаш" или 12321.

В данной задачи необходимо предусмотреть такую ситуацию:

1. Когда в введённой строке может быть несколько полиндромов, например в строке "библейский потоп", есть как минимум 2 подстроки с полиндромами "потоп" и "биб", но в результате, функция должна вернуть "потоп";
2. Если есть несколько полиндромов с одинаковым количеством символов или чисел, то функция должна вернуть их все, например, если в введённой строке есть подстроки "топот" и "шалаш", то функция должна их вывести, если больше нет полиндомов с количеством символов больше 5.

В результате необходимо показать несколько примеров, которые демонстрируют работу пункта 1 и пункта 2.

Задание №2

Требуется создать класс Car (автомобиль), в котором будут поддерживаться следующие особенности:

1. Класс хранит значения фирмы и марки автомобиля, а также текущее значение скорости движения (эти параметры должны попадать в конструктор класса при инициализации объекта);
2. В классе должна быть реализована возможность увеличения скорости автомобиля на заданное количество единиц (км/ч) и снижение скорости аналогичным образом;
3. Также необходимо уметь выводить полную информацию о характеристиках автомобиля: его фирме, модели и текущее значение скорости.

Создайте хотя бы один объект класса автомобиль. Увеличьте и уменьшите скорость, как минимум по одному разу и выведите промежуточные значения каждого изменения объекта(пункт 3) в консоль браузера.

Задание №3

Требуется создать класс TV (телевизор), в котором будут поддерживаться следующие особенности:

1. Класс хранит значения фирмы, модели, текущий уровень громкости звука и активный канал передачи;
2. В классе должна быть реализована возможность увеличения и уменьшения громкости звука, установления текущего активного канала и сброс настроек (громкости и канала передачи) до начальных;
3. Также необходимо предусмотреть невозможность задачи параметров, если они выходят за границы заданного вами диапазона значений. Например, нельзя задать громкость свыше 100 единиц и соотвественно меньше 0 и каналу больше 300 и ниже 0;

4. Также необходимо уметь выводить полную информацию о характеристиках телефизора: текущий уровень громкости звука, активный канал передачи, значения фирмы и модели.

По аналогии с классом автомобиль, создайте хотя бы один объект телевизор и увеличте и уменьшите его громкость и замените канал передачи, после выполнения этих действий сбросьте текущие параметры до начальных. С каждым изменением объекта необходимо выводить полную информацию о состоянии объекта (пункт 4) в консоль браузера.

Стандартные встроенные объекты JavaScript

Числа и даты

Объекты, имеющие дело с числами, датами и математическими вычислениями.

- Number;
- Math;
- Date.

Math — это набор полезных функций, связанных с числом, таких как: Math.max (максимум), Math.min (минимум), и Math.sqrt (квадратный корень).

Объект Math используется в качестве контейнера для группирования связанных функций. Существует только один объект Math, и он почти никогда не используется в качестве значения. Скорее, он предоставляет пространство имен, так что все эти функции и значения не должны быть глобальными привязками.

Наличие слишком большого количества глобальных связей «загрязняет» пространство имен. Чем больше имен занято, тем больше вероятность того, что вы случайно перезапишите значение какой-либо существующей привязки. Например, весьма вероятно, что вы захотите назвать что-то max в одной из ваших программ. Поскольку встроенная в JavaScript функция max безопасно хранится внутри объекта Math, нам не нужно беспокоиться о ее перезаписи.

Многие языки остановят вас или, по крайней мере, предупредят, когда вы определяете привязку с именем, которое уже занято. JavaScript делает это для привязок, которые вы объявили с помощью let или const, но, наоборот, не для стандартных привязок и не для привязок, объявленных с помощью var или function.

Вернуться к объекту Math. Если вам нужно сделать тригонометрию, математика может помочь. Он содержит cos (косинус), sin (синус) и tan (тангенс), а также их обратные функции, acos, asin и atan соответственно. Число π (pi) - или, по крайней мере, самое близкое приближение, которое соответствует числу JavaScript - доступно как Math.PI. Существует старая программная традиция написания имен постоянных значений во всех заглавных буквах.

```
function randomPointOnCircle(radius) {
  let angle = Math.random() * 2 * Math.PI;
  return {x: radius * Math.cos(angle),
          y: radius * Math.sin(angle)};
}
console.log(randomPointOnCircle(2));
// → {x: 0.3667, y: 1.966}
```

Кстати, если дан один и тот же вход - возможно, что они производят числа, которые кажутся случайными. Для этого машина сохраняет некоторое скрытое значение, и всякий раз, когда вы запрашиваете новое случайное число, она выполняет сложные вычисления для этого скрытого значения, чтобы создать новое значение. Он сохраняет новое значение и возвращает некоторое число, полученное из него. Таким образом, он может генерировать все новые, трудно предсказуемые числа способом, который кажется случайным.

Если мы хотим получить целое случайное число вместо дробного, мы можем использовать Math.floor (который округляется до ближайшего целого числа) в результате Math.random.

```
console.log(Math.floor(Math.random() * 10));
// → 2
```

Умножение случайного числа на 10 дает нам число, большее или равное 0 и ниже 10. Поскольку Math.floor округляет в меньшую сторону, это выражение с равной вероятностью будет генерировать любое число от 0 до 9.

Есть также функции Math.ceil (для «потолка», который округляет до целого числа), Math.round (до ближайшего целого числа) и Math.abs, который принимает абсолютное значение числа, то есть отрицает отрицательные значения, но оставляет положительные, как они есть.

Примитивные значения (например, 3.14 или 2014) не могут иметь свойств и методов (потому что они не являются объектами).

Но с JavaScript методы и свойства также доступны для примитивных значений, потому что JavaScript обрабатывает примитивные значения как объекты при выполнении методов и свойств.

Number () может использоваться для преобразования переменных JavaScript в числа:

```
Number(true);         // returns 1
Number(false);        // returns 0
Number("10");                    //              returns              10
Number("              10");      //              returns              10
Number("10            ");        //              returns              10
Number("        10          ");  //              returns              10
Number("10.33");                 //            returns             10.33
Number("10,33");                 //             returns              NaN
Number("10            33");      //             returns              NaN
Number("John");       // returns NaN
```

Number () также может конвертировать дату в число:

```
Number(new Date("2017-09-30"));    // returns 1506729600000
```

Вышеуказанный метод Number () возвращает количество миллисекунд с 1.1.1970.

Экземпляр объекта Date, представляющего собой момент времени.

```
new Date();
new Date(value);
new Date(dateString);
new Date(year, month[, day[, hour[, minute[, second[, millisecond]]]]]);
```

Объекты Date могут быть созданы только путём вызова функции Date в качестве конструктора: обычный вызов функции (то есть, без использования оператора new) вернёт строку вместо объекта Date; в отличие от других объектных типов JavaScript, объекты Date не имеют литерального синтаксиса.

Если функция Date вызывается в качестве конструктора с более, чем одним аргументом, значения, большие логического диапазона (например, 13 в качестве номера месяца или 70 для значения минут) «переметнутся» на соседние значения. Например, вызов new Date(2013, 13, 1) эквивалентен вызову new Date(2014, 1, 1), оба создадут дату 2014-02-01 (нумерация месяцев начинается с нуля). Тоже самое действует и для других значений: вызов new Date(2013, 2, 1, 0, 70) эквивалентен вызову new Date(2013, 2, 1, 1, 10) — оба вызова создадут дату 2013-03-01T01:10:00.

Value — целое значение, представляющее количество миллисекунд, прошедших с 1 января 1970 00:00:00 по UTC (эпохи Unix).

DateString — строковое значение, представляющее дату. Строка должна быть в одном из форматов, распознаваемых методом Date.parse() (совместимые с IETF RFC 2822 временные метки [на английском, на русском], а также версия ISO8601 [на английском, на русском]).

Year — целое значение, представляющее год. Значения с 0 по 99 отображаются на года с 1900 по 1999. Смотрите пример ниже.

Month — целое значение, представляющее месяц, начинается с 0 для января и кончается 11 для декабря.

Day — необязательный параметр. Целое значение, представляющее день месяца.

Hour — Необязательный параметр. Целое значение, представляющее часы дня.

Minute — необязательный параметр. Целое значение, представляющее минуты времени.

Second — необязательный параметр. Целое значение, представляющее секунды времени.

Millisecond — необязательный параметр. Целое значение, представляющее миллисекунды времени.

Обработка текста

Объекты для манипулирования текстом:

- String;
- RegExp.

Объект String позволяет работать с серией символов; он оборачивает строковый примитивный тип данных Javascript несколькими вспомогательными методами.

Поскольку JavaScript автоматически конвертирует между строковыми примитивами и объектами String, вы можете вызывать любой из вспомогательных методов объекта String в строковом примитиве.

Используйте следующий синтаксис для создания объекта String:

```
var val = new String(string);
```

Параметр String - это серия символов, которые были правильно закодированы.

Ниже представлен список свойств объекта String и их описание.

Таблица 8

Описание свойств String

Свойство	Описание
constructor	Возвращает ссылку на функцию String, которая создала объект.
length	Возвращает длину строки.

Описание свойств String

proto-type	Свойство prototype позволяет добавлять свойства и методы к объекту.

Таблица 9

Описание методов String

Метод	Описание
String.from-CharCode()	Возвращает строку, созданную из указанной последовательности значений Юникода.
String.fromCo-dePoint()	Возвращает строку, созданную из указанной последовательности кодовых точек Юникода.
String.raw()	Возвращает строку, созданную из сырой шаблонной строки.

Регулярное выражение - это последовательность символов, образующая шаблон поиска.

```
let re1 = new RegExp("abc");
let re2 = /abc/;
```

Шаблон поиска можно использовать для операций поиска текста и замены текста.

Регулярное выражение может быть одним символом или более сложным шаблоном.

Регулярные выражения могут использоваться для выполнения всех типов операций текстового поиска и замены текста.

В JavaScript регулярные выражения часто используются с двумя строковыми методами: search () и replace ().

Метод search () использует выражение для поиска совпадения и возвращает позицию совпадения.

Метод replace () возвращает измененную строку, в которой шаблон заменяется.

Метод search () ищет строку для указанного значения и возвращает позицию совпадения:

Используйте строку для поиска "W3schools" в строке:

```
var str = "Visit W3Schools!";
var n = str.search("W3Schools");
```

Метод replace () заменяет указанное значение другим значением в строке:

```
var str = "Visit Microsoft!";
var res = str.replace("Microsoft", "W3Schools");
```

При использовании конструктора RegExp шаблон записывается как обычная строка, поэтому обычные правила применяются для обратной косой черты.

Вторая нотация, где шаблон появляется между символами косой черты, обрабатывает обратную косую черту несколько иначе. Во-первых, поскольку косая черта завершает шаблон, нам нужно поставить обратную косую черту перед любой косой чертой, которую мы хотим стать частью шаблона. Кроме того, обратные слэши, которые не являются частью специальных кодов символов (например, \ n), будут сохранены, а не проигнорированы, как в строках, и изменят значение шаблона. Некоторые символы, такие как вопросительные знаки и знаки плюс, имеют специальные значения в регулярных выражениях и должны начинаться с обратной косой черты, если они предназначены для представления самого символа.

Структурированные данные

Буферы данных:

- ArrayBuffer;
- DataView;
- JSON.

Поскольку свойства воспринимают только свое значение, а не содержат его, объекты и массивы хранятся в памяти компьютера как последовательности битов, содержащих адреса — место в памяти — их содержимого. Таким образом, массив с другим массивом внутри него состоит (по крайней мере) из одной области памяти для внутреннего массива, а другой — для внешнего массива, содержащего (среди прочего) двоичное число, которое представляет позицию внутреннего массива.

Если вы хотите сохранить данные в файл для дальнейшего использования или отправить их на другой компьютер по сети, вам нужно каким-то образом преобразовать эти запутанные адреса памяти в описание, которое можно сохранить или отправить. Полагаю, вы могли бы переслать всю память

компьютера вместе с адресом интересующей вас ценности, но это не кажется лучшим подходом.

Популярный формат сериализации называется JSON (произносится «Jason»), что означает JavaScript Object Notation. Он широко используется в качестве формата хранения данных и обмена данными в Интернете, даже на языках, отличных от JavaScript.

JSON похож на способ написания массивов и объектов в JavaScript с некоторыми ограничениями. Все имена свойств должны быть заключены в двойные кавычки, и допускаются только простые выражения данных — без вызовов функций, привязок или чего-либо, что связано с фактическими вычислениями. Комментарии не допускаются в JSON.

Запись в журнале может выглядеть так, если она представлена в виде данных JSON:

```
{
  "squirrel": false,
  "events": ["work", "touched tree", "pizza", "running"]
}
```

JavaScript предоставляет нам функции JSON.stringify и JSON.parse для преобразования данных в этот формат и обратно. Первый принимает значение JavaScript и возвращает строку в кодировке JSON. Вторая берет такую строку и преобразует ее в значение, которое она кодирует.

```
let string = JSON.stringify({squirrel: false,
  events: ["weekend"]});
console.log(string);
// → {"squirrel":false,"events":["weekend"]}
console.log(JSON.parse(string).events);
// → ["weekend"]
```

Объект ArrayBuffer используется для представления общего буфера исходных двоичных данных фиксированной длины.

Это массив байтов, который в других языках часто называют «байтовым массивом». Вы не можете напрямую манипулировать содержимым ArrayBuffer; вместо этого вы создаете один из типизированных объектов массива или объект DataView, который представляет буфер в определенном формате, и используете его для чтения и записи содержимого буфера.

Конструктор ArrayBuffer () создает новый ArrayBuffer заданной длины в байтах. Вы также можете получить буфер массива из существующих данных, например, из строки Base64 или из локального файла.

В этом примере мы создаем 8-байтовый буфер с представлением Int32Array, ссылающимся на буфер:

```
const buffer = new ArrayBuffer(8);
const view = new Int32Array(buffer);
```

Форматы многобайтовых чисел представлены в памяти по-разному в зависимости от архитектуры машины - см. Endianness для объяснения. Средства доступа к DataView обеспечивают явный контроль доступа к данным независимо от порядка выполнения компьютера.

```
var littleEndian = (function() {
  var buffer = new ArrayBuffer(2);
  new DataView(buffer).setInt16(0, 256, true /* littleEndian */);
  return new Int16Array(buffer)[0] === 256;
})();
console.log(littleEndian); // true or false
```

Поскольку JavaScript в настоящее время не включает стандартную поддержку 64-битных целочисленных значений, DataView не предлагает собственные 64-битные операции. В качестве обходного пути вы можете реализовать собственную функцию getUint64 () для получения значения с точностью до Number.MAX_SAFE_INTEGER, что может быть достаточно для определенных случаев.

```
function getUint64(dataview, byteOffset, littleEndian) {
  // split 64-bit number into two 32-bit (4-byte) parts
  const left =  dataview.getUint32(byteOffset, littleEndian);
  const right = dataview.getUint32(byteOffset+4, littleEndian);
  const combined = littleEndian? left + 2**32*right : 2**32*left + right;
  if (!Number.isSafeInteger(combined))
    console.warn(combined, 'exceeds MAX_SAFE_INTEGER. Precision may
be lost');
  return combined;
}
```

В качестве альтернативы, если вам нужен полный 64-битный диапазон, вы можете создать BigInt. Кроме того, хотя встроенные BigInts намного быстрее, чем эквиваленты пользовательских библиотек, BigInts всегда будут намного медленнее, чем 32-разрядные целые числа в JavaScript, из-за характера их переменного размера.

```
const BigInt = window.BigInt, bigThirtyTwo = BigInt(32), bigZero = BigInt(0);
function getUint64BigInt(dataview, byteOffset, littleEndian) {
  const left = BigInt(dataview.getUint32(byteOffset|0, !!littleEndian)>>>0);
  const right = BigInt(dataview.getUint32((byteOffset|0) + 4|0, !!littleEndian)>>>0);
  return littleEndian ? (right<<bigThirtyTwo)|left : (left<<bigThirtyTwo)|right;
}
```

Практическое задание

Задание 1

Напишите функцию, которая в качестве параметра получает строку, а как результат свой работы вычисляет, в строке больше 10 символов или меньше.

Задание 2

Используя методы объекта Date, вывести на экран время в формате: час:минута:секунда (пример: 14:23:11)

При этом, если секунды и минуты попадают в интервал от 0 до 10, они должны выводиться с нулем впереди. Т.е. вместо 18:7:3 у Вас должно выводиться 18:07:03.

Задание 3

Дана JSON строка '["Коля", "Вася", "Петя"]'. Преобразуйте ее в массив JavaScript и выведите на экран элемент "Петя".

Задание 4

Дана строка 'ahb acb aeb aeeb adcb axeb'. Напишите регулярное выражение, которая найдет строки ahb, acb, aeb по шаблону: буква 'a', любой символ, буква 'b'.

Асинхронность

При разработки клиентской части приложения особая роль отведена асинхронности языка. Потребность в ней обусловлена многими причинами, например, когда необходимо получить данные от сервера, необходимо ожидать ответа. И если за определённый промежуток времени ничего не получено, то необходимо, например, продолжить выполнение сценария, чтобы не было зависания приложения. Или наоборот, если нет необходимости ожидать ответа, то надо просто выполнить запрос и продолжить дальнейшую работу вашей программы. Ещё, например, возможна ситуация, когда вам будет необходимо

осуществлять обращение к какому-либо компоненту страницы, чтобы проверять его статус для выполнения какой-либо дополнительной логики.

Асинхронное программирование является одной из особенностей в JavaScript. Это означает, что если какое-либо действие занимает некоторое время, то ваша программа может продолжать выполнять дальнейшие действия, пока не закончатся параллельный процесс. Как только это действие будет завершено, вы можете начать что-то делать с результатом. Это оказывается отличной особенностью для таких функций, как поиск данных. В JavaScript есть несколько способов работы с асинхронностью: callback функции, Promises и асинхронное ожидание (async-await).

Callback функции

Callback - это функция обратного вызова, которая является аргументом вызова другой функции или метода. Ниже представлен её пример:

```
const myArray = ['a', 'b'];
const callback = (x) => console.log(x);
myArray.forEach(callback);
// Output:
// 'a'
// 'b'
```

Обратные вызовы являются шаблоном для обработки асинхронных результатов. Чаще всего они используются только для обработки полученных данных. В качестве примера рассмотрим функцию readFile(), которая читает текстовый файл и асинхронно возвращает его содержимое.

```
readFile('some-file.txt', {encoding: 'utf8'},
  (error, data) => {
    if (error) {
      assert.fail(error);
      return;
    }
    assert.equal(data, 'The content of some-file.txt\n');
  });
```

Существует один обратный вызов, который обрабатывается как при успешном выполнении чтения файла, так и при возникновении ошибки. Если первый параметр не нулевой, то произошла ошибка. В противном случае результат будет содержаться во втором аргументе функции.

Использование функции на основе Promise

Promise существует, чтобы сделать сложность выполнения асинхронных запросов более управляемой. Объект Promise представляет возможное

завершение операции в JavaScript. Он может быть либо успешно выполнен, либо отклонён. В первом случае, можно обработать его возвращенное значение с помощью метода then, в другом случае можно использовать перехват ошибки если такая имеется.

Следующий код является примером использования функции addAsync (), основанной на Promise

```
addAsync(3, 4)
  .then(result => { // success
    assert.equal(result, 7);
  })
  .catch(error => { // failure
    assert.fail(error);
  });
```

В данном примере в объекте promise регистрируется 2 callback функции:

- метод .then регистрирует callback, который перехватывает результат выполнения функции addAsync;
- метод .catch регистрирует callback, который перехватывает ошибки, если в процессе выполнения функции addAsync таковые были.

Функция на основе Promise возвращает объект Promise и отправляет результат в соответствующий callback.

В общих случаях объекты Promise чаще всего используются для заполнения данных в какой-либо контейнер, либо для регистрирования слушателей (листенеров).

Ниже представлена реализация функции addAsync, которая складывает 2 числа x и y.

```
function addAsync(x, y) {
  return new Promise(
    (resolve, reject) => { // (A)
      if (x === undefined || y === undefined) {
        reject(new Error('Must provide two parameters'));
      } else {
        resolve(x + y);
      }
    });
}
```

addAsync немедленно вызывает конструктор Promise. Реальная реализация этой функции находится в обратном вызове, который передается этому

конструктору (строка A). Этот обратный вызов предоставляется с двумя функциями:

- resolve используется для передачи результата (в случае успеха);
- reject используется для передачи ошибки (в случае сбоя).

Async функции

Грубо говоря, async функции предоставляют более удобный синтаксис для работы с Promises. Рассмотрим следующую async функцию:

```
async function fetchJsonAsync(url) {
  try {
    const request = await fetch(url); // async
    const text = await request.text(); // async
    return JSON.parse(text); // sync
  }
  catch (error) {
    assert.fail(error);
  }
}
```

Данный пример синхронного кода аналогичен примеру ниже, который напрямую использует Promises:

```
function fetchJsonViaPromises(url) {
  return fetch(url) // async
  .then(request => request.text()) // async
  .then(text => JSON.parse(text)) // sync
  .catch(error => {
    assert.fail(error);
  });
}
```

Несколько замечаний об асинхронной функции fetchJsonAsync():

- async функция помечена ключевым словом async;
- в синхронном коде async функции необходимо применять оператор ожидания, когда используется объект Promise. Этот оператор приостанавливает асинхронную функцию и возобновляет ее после выполнения этого объекта;
- результатом async функции всегда является Promise объект.

И fetchJsonAsync (), и fetchJsonViaPromises () вызываются одинаково, например:

```
fetchJsonAsync('http://example.com/person.json')
.then(obj => {
  assert.deepEqual(obj, {
    first: 'Jane',
    last: 'Doe',
  });
});
```

Со стороны практически невозможно определить разницу между async функцией и функцией, которая возвращает Promise.

Способы объявления async функций:

```
// объявления async функции
async function func1() {}

// выражение async функции
const func2 = async function () {};

// стрелочная async функция
const func3 = async () => {};

// async метод определённый в объектном литерале
const obj = { async m() {} };

// async метод определённый в классе
class MyClass { async m() {} }
```

Асинхронная функция - это любая функция, которая доставляет свой результат асинхронно, например, callback функция или функция на основе Promise.

Async функция определяется через специальный синтаксис, включающий ключевые слова async и await. Они также называется async/await функциями из-за этих двух ключевых слов. Async функции основаны на Promises и, соответственно являются асинхронными.

Отложенные вычисления

В JavaSript есть несколько функций для работы с отложенными вычислениями: setTimeout и setInterval. Разница их в том, что через заданный промежуток времени, setTimeout запускается только один раз, а setInterval повторяется с указанной периодичностью.

Ниже представлен синтаксис объявления функции setTimeout():

let timerId = setTimeout(func|code, [delay], [arg1], [arg2], ...)

func|code — чаще всего функция или одна команда, которая выполнится через указанный промежуток времени в параметре [delay].

[delay] — число в миллисекундах, для определения времени задержки, по умолчанию 0, если аргумент не указан.

[arg1], [arg1]… — здесь перечисляются аргументы функции, которые должны быть переданы в первый аргумент функции setTimeout(), то есть в функцию func.

Ниже представлен пример работы этой функции:

```
setTimeout(() => {
console.log("Hello world");
}, 3000);
```

В логах консоли браузера через 3 секунды будет выведено: Hello world.

Альтернативно, можно было реализовать следующим образом:

```
setTimeout(greeting, 3000);
function greeting() {
console.log("Hello world");
}
```

Также можно добавить аргументы в отложенную функцию:

```
setTimeout(greeting, 300, "Hello world");

function greeting(x) {
  console.log(x);
}
```

Если есть необходимость отмены выполнения отложенной функции, то можно воспользоваться функцией clearTimeout().

```
const timeout = setTimeout(greeting, 300, "Hello world");

if (true) {
clearTimeout(timeout);
}
function greeting(x) {
console.log(x);
}
```

Ниже представлен синтаксис объявления функции setInterval():

let timerId = setInterval(func|code, [delay], [arg1], [arg2], ...)

Аргументы аналогичны функции setTimeout(). Только рассмотренная ранее функция greeting будет выполняться каждые 3 секунды. Прервать

выполнение можно таким же способом как и clearTimeout только используя функцию clearInterval.

Практическое задание

Задание 1

Необходимо написать функцию которая складывает 2 числа и имеет следующие условности:

- суммирование происходит каждые 5 секунд, сначала в функцию попадает произвольных числа, а в следующих вызовах отложенной функции в первый аргумент попадает сумма чисел с предыдущей итерации суммирования, а второй аргумент остаётся прежним;
- сумму и количество вызов отложенной функции выводить на консоль браузера на каждой итерации;
- как только отложенная функция будет выполнена 5 раз прекратить периодическое выполнение.

Задание 2

Модифицировать полученную реализацию функции из первого задания в Promise объект с помощью конструктора и вернуть полученный объект из этой функции. Необходимо учесть следующие требования:

- если хотя бы один из аргументов функции неопределён или имеют не number тип, то отклонить выполнение reject, иначе выполнить сложение этих чисел;
- реализовать вызов такой функции с помощью async/await функции;
- реализовать вызов такой функции с помощью функции основанной на Promise;
- продемонстрировать оба варианта вызова таких функций как с у спешным выполнением сложения этих чисел, так и с вызовом ошибки.

Работа с DOM

Описание модели DOM

DOM (объектная модель документа) - это интерфейс, для HTML и XML документов распознаваемые браузером. JavaScript позволяет управлять, структурировать и стилизовать сайт. После того, как браузер читает HTML-документ, он создает дерево представления, называемое объектной моделью документа, и определяет, как к этому дереву можно получить доступ.

Необходимость использования DOM представляет собой возможность создавать приложения, которые обновляют данные страницы без необходимости обновления. Кроме того, есть возможность создавать приложения,

настраиваемые пользователем, а затем изменять макет страницы без обновления, а также перетаскивать, перемещать и удалять элементы.

На рисунке 1 представлен пример построенного дерева представления после прочтения html документа браузером. Здесь четыре основных элемента:

Document - обрабатывает все HTML документы.

Elements: все теги, которые находятся внутри вашего HTML или XML, превращаются в элемент DOM.

Text: все содержимое тегов.

Attributes: все атрибуты из определенного элемента HTML. На изображении атрибут class = "hero" является атрибутом элемента <p>.

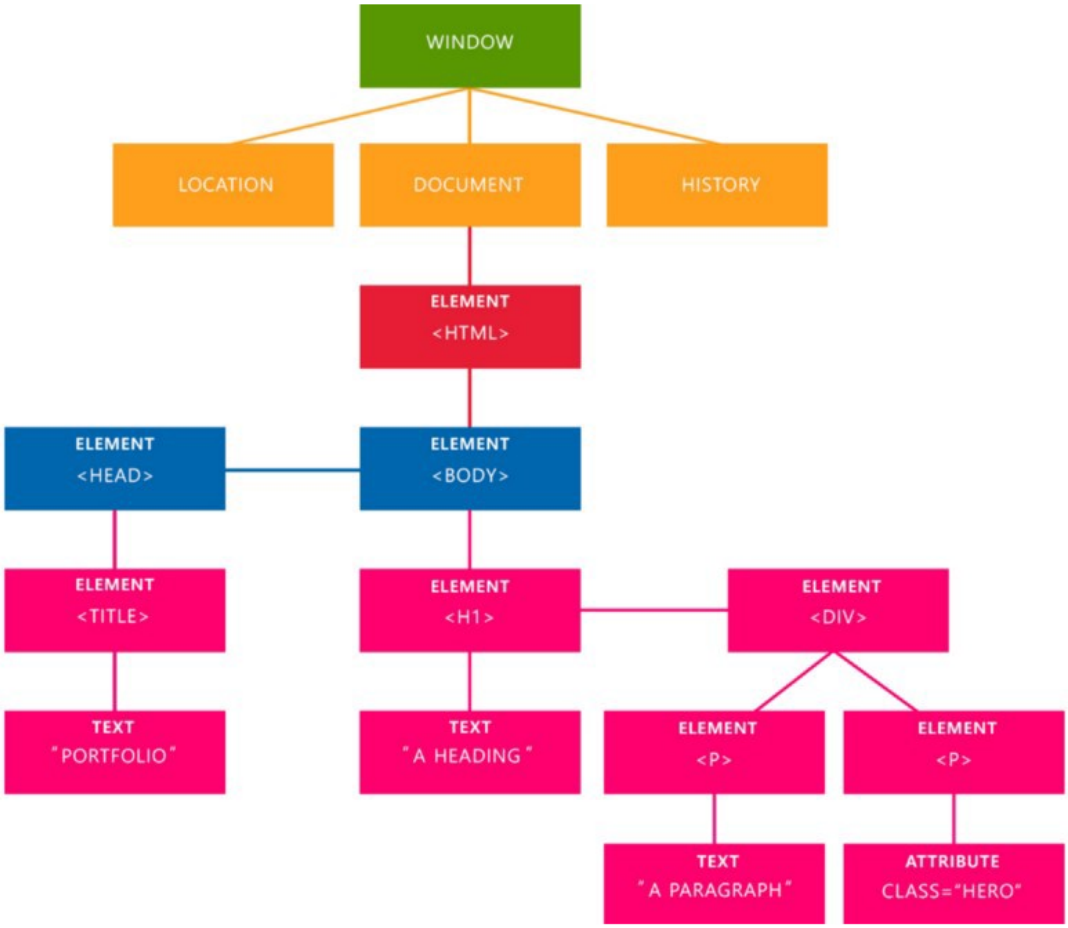

Рис. 1. Структура DOM

События и методы

Для того чтобы продемонстрировать способы работы с DOM элементами для примера рассмотрим следующий html документ.

```
<!DOCTYPE html>
<html lang="pt-br">

<head>
   <meta charset="UTF-8">
   <meta name="viewport" content="width=device-width, initial-scale=1.0">
   <meta http-equiv="X-UA-Compatible" content="ie=edge">
   <title>Entendendo o DOM (Document Object Model)</title>
</head>

<body>
   <div class="container">
      <h1><time>00:00:00</time></h1> <button id="start">Start</button> <button id="stop">Stop</button> <button
         id="reset">Reset</button>
   </div>
</body>

</html>
```

В JavaScript для работы с элементами DOM есть глобальный объект document. Разберём несколько несколько примеров.

getElementById()

Этот метод возвращает элемент, который содержит переданный идентификатор. Как известно, идентификаторы должны быть уникальными, поэтому это очень полезный способ получить только тот элемент, который необходим.

var myStart = document.getElementsById('start');

myStart — имя переменной, в которой находится объект элемента DOM с id start

start — идентификатор. И в случае, если у нас нет идентификатора с этим конкретным именем, то будет возвращено ноль.

getElementsByClassName()

Этот метод возвращает HTML коллекцию по имени класса.

var myContainer = document.getElementsByClassName('container');

.container — имя класса. В случае, если нет класса с таким конкретным именем, функция возвращает ноль.

getElementsByTagName()

Этот метод работает так же, как и вышеописанные методы: он также возвращает HTML коллекцию, но на этот раз с одним отличием: он возвращает все эти элементы с переданным именем тега.

var buttons = document.getElementsByTagName('button');

button — имя тега, который мы хотим получить.

querySelector()

Возвращает первый элемент, которому передан определенный селектор CSS. Селектор должен следовать синтаксису CSS. Если у нет селектора, то функция вернёт ноль.

var resetButton = document.querySelector('#reset');

#reset — переданный селектор, и если нет какого-либо селектора, который соответствует ему, возвращается ноль.

querySelectorAll()

Очень похоже на метод querySelector(), но с одним отличием: он возвращает все элементы, которые соответствуют переданному селектору CSS. Он также должен следовать синтаксису CSS. Если нет селектора, то функция вернёт ноль.

var myButtons = document.querySelector('#buttons');

#buttons — переданный селектор, если нет какого-либо селектора, который соответствует ему, возвращает ноль.

Это некоторые наиболее используемые методы DOM. Есть еще несколько методов, которые можно использовать, например, createElement (), который создает новый элемент на HTML-странице, и setAttribute (), который позволяет устанавливать новые атрибуты для элементов HTML.

Помимо методов у DOM элементов есть события. Они отвечают за возможность интерактивности на странице.

click

Одним из наиболее часто используемых событий является событие клика. Когда пользователь нажимает на определенный элемент, он реализует некоторые действия.

```
myStart.addEventListener('click', function(event) {
// Do something here.
}, false);
```

Параметры addEventListener ():

- тип события, которое необходимо (в данном случае событие клик);
- callback функция;

- UseCapture по умолчанию имеет значение false. Он указывает, хотите ли вы «захватить» событие.

select

Это события, когда вы хотите передать что-то, когда выбран определенный элемент. В этом случае передается простое предупреждение.

```
myStart.addEventListener('select', function(event) {
alert('Element selected!');
}, false);
```

Обход элементов DOM

Есть возможность обхода DOM элементов и использование некоторых свойств, которые будут представлены ниже. С помощью этих свойств можно возвращать элементы, комментарии, текст и т.д.

.childNodes

Это свойство возвращает nodeList дочерних узлов данного элемента. Оно возвращает текст, комментарии и так далее.

```
var container = document.querySelector('.container');
var getContainerChilds = container.childNodes;
```

.firstChild

Это свойство возвращает первого потомка данного элемента.

```
var container = document.querySelector('.container');
var getFirstChild = container.firstChild;
```

.nodeName

Это свойство возвращает имя полученнго элемента. В этом случае будет получен div

```
var container = document.querySelector('.container');
var getName = container.nodeName;
```

.nodeValue

Это свойство специфично для текстов и комментариев, так как оно возвращает или устанавливает значение текущего узла. В данном случае, поскольку был передан div, то вернется null.

```
var container = document.querySelector ('. container')
var getValue = container.nodeValue;
```

.nodeType

Это свойство возвращает тип данного элемента. В данном случае возвращается «1».

var container = document.querySelector('.container')
var getValue = container.nodeType;

В основном nodeType это _ELEMENT_NODE_ и возвращает ноль. Если в примере был бы атрибут, то вернулось бы «2» и его значение.

На рисунке 2 представлены все типы nodeTypes и nodeName и nodeValue

Node type		nodeName returns	nodeValue returns
1	Element	element name	null
2	Attr	attribute name	attribute value
3	Text	#text	content of node
4	CDATASection	#cdata-section	content of node
5	EntityReference	entity reference name	null
6	Entity	entity name	null
7	ProcessingInstruction	target	content of node
8	Comment	#comment	comment text

Рис. 2. Типы Node type, nodeName и nodeValue

Элементы DOM

Это свойства вместо описанных ранее, возвращают нам только элементы Dom.

.parentNode

Это свойство возвращает родителя данного узла.

var container = document.querySelector('.container')
var getParent = container.parentNode;

.firstElementChild

Возвращает первый дочерний узел полученного данного элемента.

var container = document.querySelector('.container')
var getValue = container.firstElementChild;

.lastElementChild

Возвращает последний дочерний узел полеченного элемента.

var container = document.querySelector('.container')
var getValue = container.lastElementChild;

Практическое задание

Необходимо по заданной структуре HTML документа и стилей CSS реализовать работу калькулятора, который представлен на рисунке 3.

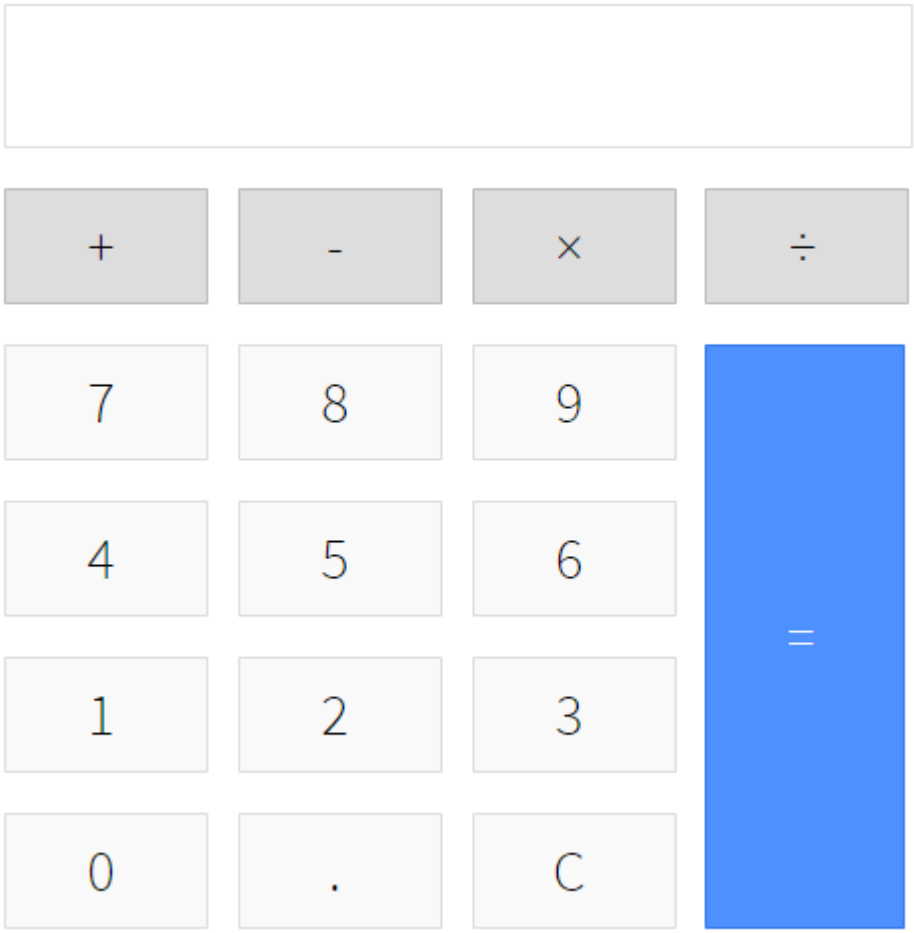

Рис. 4. Вид калькулятора

Заключение

Разработка WEB-ориентированных систем связана в первую очередь с программированием на специальных языках, предназначенных для создания приложений. Рассматриваемые технологии создания клиентских приложений на языке JavaScript, являются одним из инструментов для создания таких Internet-приложений, который имеет все большее распространение в мире в настоящее время. Знание и умение пользоваться этим инструментом является в настоящее время непременным условием для специалиста в области разработки WEB-ориентированных систем. В рамках данного учебного пособия изложены основные технологические приемы при создании клиентских приложений на языке JavaScript, освоение которых позволит самостоятельно разрабатывать отдельные Web-приложения, участвовать в разработке сайтов и порталов для различных отраслей.

Пособие не охватывает всех вопросов создания клиентских приложений, в частности не содержит материалов по использованию различных Frameworks. Это является отдельной тематикой, которой будет посвящено отдельное методическое руководство.

При написании учебного пособия использовались практические материалы, подготовленные магистрантами группы мРИС-191 Алиной Наздрачевой и Мариной Поздняковой, которые отработали контрольные задания и проверили временные рамки выполнения этих заданий на своих практических занятиях, а также подготовили ряд базовых материалов по теории программирования на языке JavaScript.

Библиографический список

1. Дронов, В. JavaScript в Web-дизайне - М.: БХВ-Петербург, 2017. - 880 с.
2. Макфарланд, Д. JavaScript. Подробное руководство - М.: Эксмо, 2015. - 608 с.
3. Машнин, Т. Web-сервисы Java - М.: БХВ-Петербург, 2017. - 560 с.
4. Минник, К. JavaScript для чайников - М.: Вильямс, 2016. - 320 с.
5. Закас, Н. JavaScript. Оптимизация производительности - М.: Символ-плюс, 2016. - 482 с.
6. Слепцова, Л. Д. JavaScript. Самоучитель - М.: Вильямс, 2017. - 448 с.
7. Херман, Д. Сила JavaScript. 68 способов эффективного использования JS - М.: Питер, 2016**.** - 907 с.
8. Флэнаган, Д. JavaScript. Карманный справочник - М.: Вильямс, 2015. - 320 с.

Оглавление

Учебное издание

Рындин Никита Александрович

ТЕХНОЛОГИИ РАЗРАБОТКИ КЛИЕНТСКИХ WEB-ПРИЛОЖЕНИЙ НА ЯЗЫКЕ JAVASCRIPT

В авторской редакции

Подписано в печать _____.
Формат 60x87/16. Бумага для множительных аппаратов.
Усл. печ. л. 11,5. Тираж 30 экз.
Зак. №

ФГБОУ ВО «Воронежский государственный технический университет»
394026 Воронеж, Московский просп., 14

Участок оперативной полиграфии издательства ВГТУ
394026 Воронеж, Московский проспект., 14