
JS

- ЦВЕТНОЕ ИЗДАНИЕ
- ЧЕТКИЕ ПОШАГОВЫЕ ИНСТРУКЦИИ
- СОЗДАЙТЕ САЙТ С НУЛЯ
- ОПЫТ В РАЗРАБОТКЕ НЕ НУЖЕН

JAVASCRIPT

ДЛЯ НАЧИНАЮЩИХ



Майк МакГрат



**Мировой
компьютерный
бестселлер**



Mike McGrath



JS

JAVASCRIPT

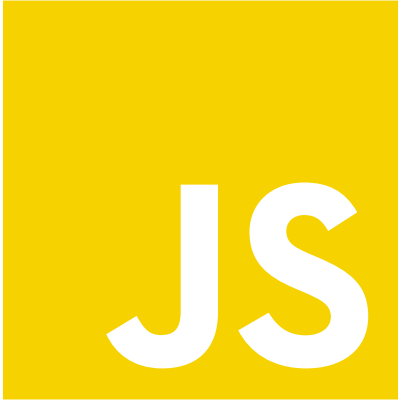
create dynamic web pages



edition | **6**



Майк МакГрат



- ЦВЕТНОЕ ИЗДАНИЕ
- ЧЕТКИЕ ПОШАГОВЫЕ ИНСТРУКЦИИ
- СОЗДАЙТЕ САЙТ С НУЛЯ
- ОПЫТ В РАЗРАБОТКЕ НЕ НУЖЕН

JAVASCRIPT

ДЛЯ НАЧИНАЮЩИХ



издание **6**



БОМБОРА
ИЗДАТЕЛЬСТВО

Москва 2023

УДК 004.43
ББК 32.973.26-018.1
М15

Mike McGrath

JavaScript in easy steps 6th edition

Copyright © 2020 by In Easy Steps Limited

Translated and reprinted under a licence agreement from the Publisher: In Easy Steps, 16
Hamilton Terrace, Holly Walk, Leamington Spa, Warwickshire, U.K. CV32 4LY



МакГрат, Майк.

М15 JavaScript для начинающих / Майк МакГрат ; [перевод с английского М. А. Райтмана]. — 6-е издание. — Москва : Эксмо, 2023. — 232 с. : ил. — (Мировой компьютерный бестселлер).

ISBN 978-5-04-121621-4

Цветное руководство по JavaScript для начинающих позволит в короткое время освоить этот язык программирования и приступить к созданию красивых и функциональных сайтов. Вся информация представлена схематично и снабжена наглядными примерами, а код и другие элементы, необходимые для обучения, читатели могут скачать и использовать совершенно бесплатно.

УДК 004.43
ББК 32.973.26-018.1

ISBN 978-5-04-121621-4

© Райтман М. А., перевод на русский язык, 2023
© Оформление. ООО «Издательство «Эксмо», 2023

Содержание

1

Введение в JavaScript

9

Знакомство с JS	10
Добавление JavaScript в HTML-документ	11
Вывод JavaScript	12
Структура кода	14
Избегайте ключевых слов	17
Хранение значений	19
Создание функций	21
Назначение функций	24
Область видимости	26
Замыкания	29
Заключение	31

2

Распространенные операции

33

Преобразование типов	34
Арифметические операторы	36
Операторы присваивания	38
Операторы сравнения	40
Логические операторы	42
Условный (тернарный) оператор	44
Побитовые операции	46
Приоритет операторов	48
Заключение	51

3

Управляющие конструкции в JavaScript

53

Ветвление с помощью оператора if	54
Альтернативное ветвление	56
Ветвление с помощью оператора switch	58
Цикл for	60
Цикл while	62
Цикл do..while	64
Выход из циклов	66
Обработка ошибок	68
Заключение	71

4

Управление объектами

73

Пользовательские объекты	74
Расширенные функции	76
Встроенные объекты	78
Создание массивов	81
Обход элементов в цикле	83
Методы управления элементами в массиве	86
Сортировка элементов массива	88
Получение даты	90
Получение календаря	92
Получение времени	95
Установка даты и времени	97
Сопоставление текста с шаблоном	99
Заключение	102

5

Работа с числовыми и строковыми типами данных 105

Вычисление площади	106
Сравнение чисел	108
Округление чисел	110
Генерация случайных чисел	112
Объединение строк	114
Разбиение строк	117
Поиск символов	119
Обрезка строк	121
Заключение	123

6

Открытие окон и методы объекта window

125

Введение в DOM	126
Свойства объекта window	128
Диалоговые окна	130
Прокрутка	132
Всплывающие окна	135
Создание таймера	137
Сведения о браузерах	139
Включение/отключение функций	141
Расположение	144
История	146
Заключение	148

7

Методы и свойства объекта document

151

Работа с документом	152
Свойства интерфейса Document	154
Получение элементов	156
Работа с текстом	158
Управление файлами cookie	161
События загрузки	163
Ответ на события мыши	166
Генерация событий	168
Добавление переключателей	170
Добавление элементов выбора	172
Сброс формы	174
Проверка и отправка формы	177
Заключение	179

8

Разработка веб-приложений

181

Введение в JSON	182
Промисы	184
Получение данных	187
Разработка интерфейса	189
Заполнение ячеек в таблице	192
Заполненная таблица	194
Обновление приложений	196
Заключение	199

9

Написание скриптов

201

Запрос данных	202
Встраиваемая векторная графика	204
Работа с холстами	207
Хранение данных	209
Перемещение элементов	211
Связь между объектами window	214
Местоположение пользователей	216
Заключение	219

Предметный указатель

221

Как пользоваться этой книгой

С помощью примеров вы узнаете, как использовать встроенные функции JavaScript, поддерживаемые современными веб-браузерами, а снимки экрана проиллюстрируют результаты, полученные с помощью примеров кода. Необходимые фрагменты кода выделены цветом.

Синим цветом выделен JavaScript-код; **красным** — имена, присвоенные программистом; **черным** — текст; **зеленым** — комментарии к коду:

```
let sum = ( 9 + 12 ) / 3 // Эквивалентно 21 / 3.  
document.getElementById( 'info' ).innerHTML += 'Grouped sum: ' + sum
```

Синим цветом выделены HTML-теги; **черным** — текст; **оранжевым** — значения атрибутов элементов в HTML- и JavaScript-коде:

```
<p id="info">JavaScript in easy steps</p>
```

Кроме того, для идентификации каждого исходного файла, описанного в пошаговых инструкциях, на полях указаны значки и имена соответствующих файлов:



page.html



data.json



external.js



data.xml



echo.pl



banner.svg

Чтобы избежать повторений, исходный код HTML-документов, приведенных в примерах, указан не полностью. Однако он представляет собой весь фрагмент документа, к которому применяется указанный JavaScript-код. Вы можете скачать ZIP-архив, содержащий все полные файлы исходных кодов, выполнив следующие простые шаги:

1. Перейдите на сайт <http://addons.eksmo.ru/it/js.zip> — загрузка начнется автоматически
2. Извлеките содержимое архива в любое удобное место на вашем компьютере.

Если вы не довольны полученным результатом, просто сравните свой код с кодом исходных файлов.

1

Введение в JavaScript

*Добро пожаловать
в удивительный
и захватывающий мир
программирования
на JavaScript! В этой главе
вы узнаете, как добавлять
сценарии в HTML-
документы, используя
переменные и функции
JavaScript.*

- 8 Знакомство с JS
- 9 Добавление JavaScript в HTML-документ
- 10 Вывод данных JavaScript
- 12 Структура кода
- 15 Избегайте ключевых слов
- 17 Хранение значений
- 19 Создание функций
- 22 Назначение функций
- 24 Область видимости
- 27 Замыкания
- 29 Заключение



Знакомство с JS

Язык программирования JavaScript («JS») — это объектно-ориентированный язык сценариев, объекты которого встроены в программное обеспечение веб-браузера, например, Google Chrome, Microsoft Edge, Firefox, Opera и Safari. Для обеспечения функциональности это позволяет при загрузке страницы в браузере интерпретировать содержащиеся на веб-странице сценарии. В целях безопасности язык JavaScript не может считывать или записывать файлы, за исключением файлов cookie, в которых хранится минимальный объем данных.

Самая первая реализация JavaScript была создана Бренданом Эйхом (Айком) (Brendan Eich) в компании Netscape. Этот язык программирования был впервые представлен в декабре 1995 года и первоначально назывался «LiveScript». Однако вскоре из коммерческих соображений LiveScript был переименован в JavaScript, получив соответствующую лицензию у Sun Microsystem.



Брендан Эйх, создатель языка программирования JavaScript, а также соучредитель проекта Mozilla, помог запустить веб-браузер Firefox.

До введения JavaScript браузер вызывал сценарии на стороне сервера, а в связи с долгим ответом снижалась производительность. Эта проблема решилась с помощью вызова сценариев на стороне клиента.

Популярность языка JavaScript быстро росла. Но между компаниями Netscape и Microsoft возникли разногласия по поводу его лицензирования, поэтому Microsoft представила собственную версию языка под названием «JScript». Несмотря на то что новая версия JScript очень похожа на JavaScript, она имеет некоторые расширенные функции. В июне 1997 года Ecma International предложила стандартизировать JavaScript, и в результате появился «ECMAScript». Это помогло стабилизировать основные функции. Однако название не прижилось среди юзеров, поэтому язык программирования все же получил название JavaScript.

В книге представлены:

- Основы языка — синтаксис, ключевые слова, операторы, структура и встроенные объекты.

- Функциональность веб-страницы показывает, как объектная модель документа (DOM) браузера обеспечивает взаимодействие с пользователем.
- Веб-приложения — адаптивные веб-приложения и текстовый формат представления данных в нотации объекта JavaScript (JSON).

Добавление JavaScript в HTML-документ

Вы можете добавить JavaScript-код в HTML-документ, поместив его между тегами `<script>` и `</script>`, например:

```
<script>  
document.getElementById( 'message' ).innerText = 'Hello World!'  
</script>
```

HTML-документ может включать в себя несколько сценариев, которые помещаются в раздел заголовка или тела документа. Однако рекомендуется размещать сценарии в конце основного раздела (непосредственно перед закрывающим тегом `</body>`), чтобы браузер мог отобразить веб-страницу до выполнения сценария.

Также в HTML-документ можно добавить JavaScript код, расположенный во внешнем файле с расширением `.js`. Это позволяет нескольким различным веб-страницам вызывать один и тот же сценарий. Подключение внешнего сценария выполняется с помощью атрибута `src` тега `<script>` следующим образом:

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

Этот код также можно поместить в раздел заголовка или тела документа, и браузер будет рассматривать сценарий так, как если бы он был написан непосредственно в этой позиции в HTML-документе.

Если атрибуту `src` тега `<script>` присваивается только имя файла внешнего сценария, необходимо, чтобы файл сценария находился в одной папке (каталоге)



В теге `<script>` применим атрибут `type="text/javascript"`. Однако это не требуется, поскольку JavaScript — язык сценариев для HTML по умолчанию.



Не включайте теги `<script>` и `</script>` во внешний файл JavaScript. В них необходимо добавлять только код сценария.



Внешние файлы сценариев упрощают сопровождение кода. Почти все примеры, представленные в этой книге, автономные, поэтому код сценария можно вставлять между тегами в HTML-документе.

с HTML-документом. Если сценарий находится в другом месте, вы можете указать относительный путь к файлу, например:

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

Если сценарий расположен в другом месте сайта, можно указать абсолютный путь к файлу, например:

```
<script src="https://www.example.com/js/external_script.js">
</script>
```

Кроме того, можно указать содержимое, которое будет отображаться на веб-странице только в том случае, если пользователь в своем веб-браузере отключил JavaScript, включив в HTML-документ элемент `<noscript>`, например:

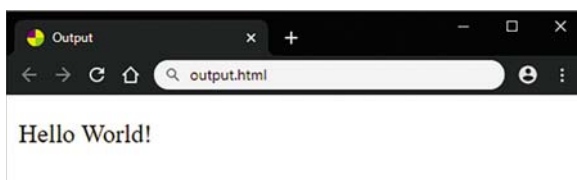
```
<noscript>JavaScript is Not Enabled!</noscript>
```

Вывод JavaScript

В JavaScript существует несколько вариантов отображения данных. Например:

```
document.getElementById( 'message' ).innerText = 'Hello World!'
```

Атрибут `id` определяет уникальный идентификатор HTML-элемента. Свойство `innerText` определяет содержимое HTML-документа.



Также для отображения данных можно использовать всплывающее окно сообщений.

```
window.alert( 'Hello World!' )
```

Метод `alert()` выводит на экран диалоговое окно с сообщением, указанным в коде в круглых скобках `()`.



Обратите внимание на использование оператора. (точка) в описании свойств или методов объекта с использованием «точечной нотации».



В процессе изучения языка JavaScript предпочтительнее выводить данные в консоль браузера, например:

```
console.log( 'Hello World!' )
```

Метод `log()` объекта `console` выводит в окно консоли содержимое, указанное в круглых скобках `()`. Фактически в любом браузере есть специальная панель веб-разработчика, доступ к которой осуществляется нажатием клавиши F12. Поскольку Google Chrome — самый популярный браузер на момент написания книги, я использовал его для демонстрации JavaScript, а консоль этого браузера используется для отображения данных.

1

Создайте HTML-документ, содержащий пустой абзац и программный код для отображения данных тремя способами.

```
<p id="message"></p>
<script>
document.getElementById( 'message' ).innerText =
    'Hello World!'
window.alert( 'Hello World!' )
console.log( 'Hello World!' )
</script>
```

2

Сохраните HTML-документ, затем откройте его в браузере, чтобы проанализировать полученный результат, как показано на рисунке ниже.

3

Нажмите клавишу F12 или воспользуйтесь меню браузера, чтобы открыть **Developers Tools** (Инструменты разработчика).



Существует также метод `document.write()`, который перезаписывает весь заголовок и тело веб-страницы. Стоит отметить, что его использование считается плохой практикой.



При возникновении ошибок в коде консоль предоставляет полезные сообщения, что очень удобно для отладки программы.

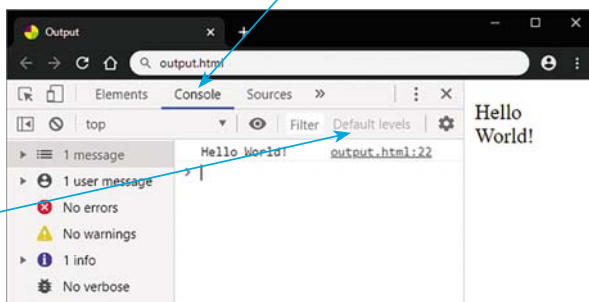





output.html

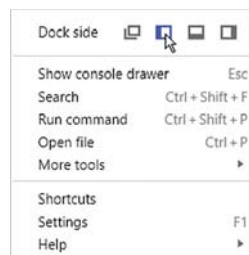


Убедитесь, что в консоли отображается содержимое, а также имя HTML-документа и номер строки, где есть соответствующий JavaScript-код.

- 4 Выберите вкладку **Console** (Консоль), чтобы увидеть содержимое, записанное в консоли.



- 5 Нажмите кнопку  **Show/Hide** (Показать/Скрыть), чтобы скрыть или показать боковую панель. Нажмите кнопку  **Customize** (Настроить), чтобы выбрать способ закрепления консоли в окне браузера. Затем нажмите кнопку  **Clear** (Очистить), чтобы очистить все содержимое консоли.



Структура кода



Более подробную информацию о ключевых словах вы можете найти на стр. 17–18, а об операторах, значениях и выражениях вы узнаете позже.

JavaScript-код состоит из ряда инструкций, называемых «операторами». Обычно все инструкции на странице выполняются сверху вниз.

Каждая инструкция может содержать любой из следующих элементов:

- **Ключевые слова** — слова, имеющие особое значение в языке JavaScript.
- **Операторы** — специальные символы, выполняющие операции с одним или несколькими операндами.
- **Значения** — текстовые строки, числа, логическое значение **true** или **false**, значения **undefined** и **null**.

- Выражения — любая часть исходного кода программы, которая вычисляет значение.

Ранее в коде JavaScript каждый оператор должен был заканчиваться символом ; (точка с запятой), а каждое предложение — символом . (точка) в английской раскладке. Теперь это необязательно, поэтому, если вы не хотите писать несколько операторов в одной строке, то это правило можно опустить. В таком случае операторы необходимо разделять точкой с запятой, например:

оператор; оператор; оператор

Некоторые разработчики JavaScript по-прежнему предпочитают заканчивать каждый оператор символом ; (точка с запятой). В приведенных в книге примерах он опускается, но выбор остается за вами.

Интерпретатор JavaScript игнорирует отступы и пробелы. Поэтому, чтобы улучшить читаемость кода, необходимо использовать пробелы. Например, при сложении двух чисел:

total = 100 + 200 вместо **total=100+200**

Операторы JavaScript обычно сгруппированы с помощью фигурных скобок {}, разделяющих код на функциональные блоки, которые можно при необходимости многократно вызывать. Для удобства и читаемости кода рекомендуется делать отступы двумя пробелами, например:

```
{  
  оператор  
  оператор  
  оператор  
}
```

Синтаксис JavaScript — это набор правил, по которым строится программа. В JavaScript существуют два типа значений: фиксированные и переменные. Фиксированные числовые и текстовые строковые значения называются литералами:



Результат вычисления выражения — это значение, тогда как оператор выполняет действие.



Для удобства и читаемости кода создавайте отступы с помощью клавиши Пробел, так как при отладке кода в текстовых редакторах отступы, созданные клавишей Tab, могут обрабатываться по-разному.



Для использования строковых литералов рекомендуется выбрать один вид кавычек и придерживаться его. В наших примерах используются одинарные кавычки.

- Числовые литералы — целые числа, например 100, или числа с плавающей запятой, например 3,142.
- Строковые литералы — строка символов, заключенная в двойные кавычки, например "JavaScript Fun", или одинарные кавычки, например 'JavaScript Fun'.

Переменные — это некий контейнер, внутри которого можно хранить значения для дальнейшего использования в программе. В JavaScript предусмотрен способ объявления переменных с помощью ключевого слова **let**. Например, с помощью кода **let total** создается переменная с именем «total». Переменной можно присвоить значение, используя оператор присваивания **=**, например:

```
let total = 300
```

Прочие операторы JavaScript используются для создания выражений, которые будут вычислять одно значение. Как правило, выражение заключается в круглые скобки **()**. Например, приведенное ниже выражение содержит числа и оператор сложения **+** и вычисляет одно значение 100:

```
( 80 + 20 )
```

Также выражения могут содержать значения переменных. Например, для вычисления одного значения 100 выражения могут содержать предыдущее значение переменной, оператор вычитания и число:

```
( total - 200 )
```



Иногда возникает необходимость закомментировать строки кода, чтобы предотвратить их выполнение при отладке.

JavaScript чувствителен к регистру символов, поэтому переменные с именами **total** и **Total** — это две совершенно разные переменные.

Хорошей практикой считается добавление к коду пояснительных комментариев. Они делают код более понятным для других пользователей, а также для вас при его повторном просмотре. Все, что находится в одной строке после символа **//** или между символов **/*** и ***/** в одной или нескольких строках, будет проигнорировано.

```
let total = 100 // Этот код БУДЕТ выполнен.  
/* let total = 100  
   Этот код НЕ БУДЕТ выполнен. */
```

Избегайте КЛЮЧЕВЫХ СЛОВ

В программе для переменных и функций вы можете указывать собственные имена. Лучше всего давать переменным осмысленные имена, которые будут отражать суть переменной или функции. Имя переменной может содержать буквы, цифры и символы подчеркивания, но никогда не начинается с цифры. Также в нем не используются пробелы. Запрещается использовать в качестве имен переменных приведенные ниже в таблице ключевые слова, которые имеют особое значение:



Ключевые (зарезервированные) слова JavaScript			
abstract	arguments	await	boolean
break	byte	case	catch
char	class	const	continue
debugger	default	delete	do
double	else	enum	eval
export	extends	false	final
finally	float	for	function
goto	if	implements	import
in	instanceof	int	interface
let	long	native	new
null	package	private	protected
public	return	short	static
super	switch	synchronized	this
throw	throws	transient	true
try	typeof	var	void
volatile	while	with	yield

Объекты, свойства и методы JavaScript

Array	Date	eval	function
hasOwnProperty	Infinity	isFinite	isNaN
isPrototypeOf	length	Math	NaN
name	Number	Object	prototype
String	toString	undefined	valueOf

HTML-имена, объекты окна и свойства

alert	all	anchor	anchors
area	assign	blur	button
checkbox	clearInterval	clearTimeout	clientInformation
close	closed	confirm	constructor
crypto	decodeURI	decodeURIComponent	defaultStatus
document	element	elements	embed
embeds	encodeURIComponent	encodeURIComponent	escape
event	fileUpload	focus	form
forms	frame	innerHeight	innerWidth
layer	layers	link	location
mimeType	navigate	navigator	frames
frameRate	hidden	history	image
images	offscreenBuffering	open	opener
option	outerHeight	outerWidth	packages
pageXOffset	pageYOffset	parent	parseFloat
parseInt	password	pkcs11	plugin
prompt	propertyIsEnum	radio	reset
screenX	screenY	scroll	secure
select	self	setInterval	setTimeout
status	submit	taint	text
textarea	top	unescape	untaint
window			

Атрибуты событий в HTML

Например:

onclick	ondblclick	onfocus	onfocusout
onkeydown	onkeypress	onkeyup	onload
onmousedown	onmouseup	onmouseover	onmouseout
onmousemove	onchange	onreset	onsubmit

Хранение значений

Переменная — это контейнер, общий для каждого языка сценариев и языка программирования, в котором хранятся данные и в процессе работы могут быть извлечены. В отличие от строго типизированных переменных в большинстве других языков программирования, которые должны декларировать определенный тип данных, переменные JavaScript в использовании намного проще. JavaScript — язык со слабой типизацией, поэтому переменные могут содержать любой тип данных:

Тип данных	Пример	Описание
String	'Hello World!'	Последовательность символов
Number	3.142	Целое число или число с плавающей запятой
Boolean	true	Значение true (1) или false (0)
Object	console	Пользовательский или встроенный объект
Function	log()	Пользовательская функция, встроенная функция или метод объекта
Symbol	Symbol()	Уникальный идентификатор
null	null	Отсутствие какого-либо объектного значения
undefined	undefined	Пустое значение

Для объявления переменных в JavaScript используются ключевые слова **let**, **const** или **var**, за которыми следует пробел и выбранное вами имя. По мере выполнения программного кода ключевое слово **let** позволяет присваивать переменным новые значения, тогда как **const** (константа) не предполагает никаких изменений. Ключевое слово **var** использовалось в JavaScript до введения ключевого слова **let**. Однако сейчас его лучше избегать, так как оно позволяет дважды объявлять одну и ту же переменную в одном и том же контексте.



Имя переменной — это псевдоним для значения. С помощью него можно ссылаться на ее сохраненное значение.



Для удобства рекомендуется выбирать понятные и осмысленные имена переменных.

Объявление переменной через **let** позволяет в программе создать переменную, значение которой может быть присвоено позже. Директива **let** объявляет переменную с блочной областью видимости с возможностью инициализировать ее значением.

```
let myNumber           // Объявление переменной.
myNumber = 10          // Инициализация переменной.
let myString = 'Hello World!' // Объявление и инициализация переменной.
```

Также в одной строке можно объявить несколько переменных:

```
let i, j, k           // Объявление трех переменных.
let num = 10, char = 'C' // Объявление и инициализация двух переменных.
```

Однако константы должны быть инициализированы во время объявления:

```
const myName = 'Mike'
```

После инициализации JavaScript автоматически устанавливает тип переменной для присвоенного значения. Последующее назначение другого типа данных выполняется в программе позже, чтобы изменить тип данных. Текущий тип переменной можно узнать по ключевому слову **typeof**.



variables.html

- 1 Создайте HTML-документ. Объявите и проинициализируйте несколько переменных разного типа.

```
const firstName = 'Mike'
const valueOfPi = 3.142
let isValid = true
let jsObject = console
let jsMethod = console.log
let jsSymbol = Symbol( )
let emptyVariable = null
let unusedVariable
```

- 2 Для вывода типа данных каждой переменной добавьте операторы:

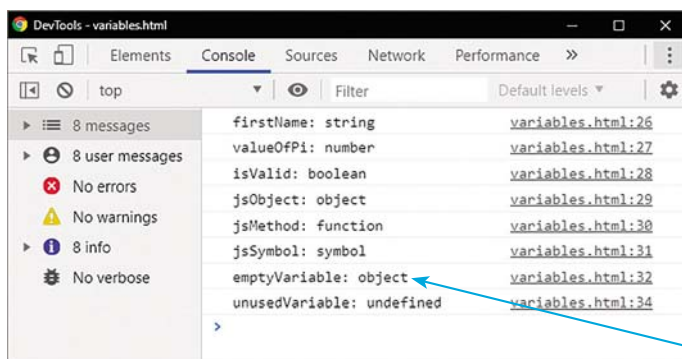
```

console.log( 'firstName: ' + typeof firstName )
console.log( 'valueOfPi: ' + typeof valueOfPi )
console.log( 'isValid: ' + typeof isValid )
console.log( 'jsObject: ' + typeof jsObject )
console.log( 'jsMethod: ' + typeof jsMethod )
console.log( 'jsSymbol: ' + typeof jsSymbol )
console.log( 'emptyVariable: ' + typeof emptyVariable )
console.log( 'unusedVariable: ' + typeof unusedVariable )

```

3

Сохраните HTML-документ, затем откройте его в браузере и запустите консоль. Проанализируйте полученные результаты — типы данных.



Оператор конкатенации + в данном примере используется для вывода объединенной текстовой строки.

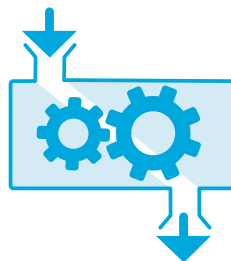


Вы будете удивлены, увидев, что переменная, которой присвоено значение `null`, описывается как тип `object`, а не тип `null`. Это распространенная ошибка языка JavaScript.

Создание функций

Объявление функции состоит из одного или нескольких операторов, сгруппированных вместе в фигурные скобки `{}`. Функция вызывает свои инструкции и возвращает в результате единственное значение. Функции могут вызываться по требованию программы. Чтобы отличать встроенные функции от определяемых пользователем, функции, которые являются свойством объекта, например `console.log()`, называются методами. Встроенные и определяемые пользователем функции содержат завершающие круглые скобки, которые могут принимать значения аргумента. Например, аргумент, переданный в скобках метода `console.log()`.

Количество аргументов, передаваемых функции, как правило, должно соответствовать количеству





Обратите внимание, что в предпочтительном формате объявления функции открывающая фигурная скобка `{` находится в той же строке, что и ключевое слово `function`.



Вы можете опустить оператор `return` или использовать ключевое слово `return` без указания значения, тогда функция в результате вернет значение `undefined`.



functions.html

параметров, указанных в скобках. Например, определяемая пользователем функция, требующая только один аргумент, выглядит так:

```
function имя функции ( параметр ) {
    // Здесь будет ваш код.
}
```

Функция может иметь несколько параметров, которые указываются через запятую в скобках. При необходимости вы можете указать значение по умолчанию, которое будет использоваться, когда вызов функции не передает аргумент, например:

```
function имя функции ( параметр, параметр = значение ) {
    // Здесь будет ваш код.
}
```

Выбирая собственные имена параметров, необходимо следовать тем же правилам, что и при выборе имен переменных. Имена параметров можно использовать в функции для ссылки на значения аргументов, переданных при вызове функции.

Функциональный блок может включать оператор **`return`**. Он предназначен для возвращения значения выражения в качестве результата выполнения функции. Как только выполнение программы доходит до этого места, функция останавливается, и значение возвращается в вызвавший ее код.

```
function имя функции ( параметр, параметр = значение ) {
    // Здесь будет ваш код.

    return результат
}
```

В функциональном блоке также может находиться вызов других функций.

1

Создайте HTML-документ. Создайте функцию, возвращающую значения переданного аргумента, возведенного в квадрат (во вторую степень).

```
function square ( arg ) {
  return arg * arg
}
```

- 2 Добавьте функцию, возвращающую результат сложения.

```
function add ( argOne, argTwo = 10 ) {
  return argOne + argTwo
}
```

- 3 Теперь добавьте функцию, возвращающую результат возведения в квадрат и сложения, как показано ниже.

```
function squareAdd ( arg ) {
  let result = square( arg )
  return result + add( arg )
}
```

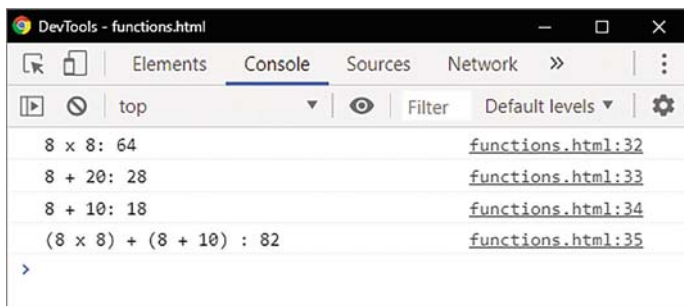
- 4 Добавьте операторы, которые вызывают функции и выводят возвращаемые значения на экран.

```
console.log( '8 x 8: ' + square( 8 ) )
console.log( '8 + 20: ' + add( 8, 20 ) )
console.log( '8 + 10: ' + add( 8 ) )
console.log( '(8 x 8) + (8 + 10): ' + squareAdd( 8 ) )
```

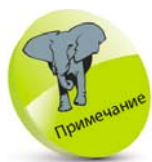
- 5 Сохраните HTML-документ, затем откройте его в браузере и запустите консоль. Проанализируйте полученный результат — возвращаемые функциями значения.



Обратите внимание, что в нашем случае используется значение второго параметра по умолчанию (10), когда при вызове функции передается только одно значение аргумента.



Символ * — это оператор арифметического умножения в JavaScript.



При присвоении переменной именованной функции указывайте в операторе только имя функции.



Переменные, которые были объявлены с использованием прежнего ключевого слова `var`, также были подняты. Переменные, которые были объявлены с помощью ключевых слов `let` или `const`, не поднимались.

Назначение функций

Функции — важный инструмент в JavaScript. Это некоторый фрагмент кода, который можно описать один раз, а затем вызывать на выполнение в разных частях программы любое число раз.

Важно понимать, что оператор `()` — это компонент оператора вызова, который фактически вызывает функцию. Это означает, что оператор может назначить функцию переменной, указав только имя функции. Затем переменную можно использовать для вызова функции. Однако будьте внимательны, если вы попытаетесь назначить функцию переменной, указав ее имя, за которым следует символ `()`. Такая функция будет вызвана, и будет присвоено значение, возвращаемое этой функцией.

Поднятие

Хотя код читается интерпретатором JavaScript сверху вниз, на самом деле он выполняет два этапа. На первом этапе ищутся объявления функций и запоминается все, что находится в процессе, то есть поднятие. Второй этап — это когда код фактически выполняется интерпретатором.

Поднятие позволяет вызовам функций появляться в коде до объявления функции, так как интерпретатор уже распознал функцию на первом этапе. Однако на первом этапе не распознаются функции, которые были присвоены переменным с помощью ключевых слов `let` или `const`!

Анонимные функции

При присвоении функции переменной имя функции можно не указывать, так как ее можно вызвать в операторе, указав имя переменной и оператор `()`. Такие функции называются анонимными функциональными выражениями. Их синтаксис выглядит следующим образом:

```
let переменная = function ( параметры ) { операторы ;  
return значение }
```

Анонимные функциональные выражения также можно сделать самовызывающимися, заключив функцию целиком в круглые скобки () и добавив в конце выражения оператор (). Это означает, что при первой загрузке сценария браузером операторы автоматически выполняются один раз. Синтаксис функционального выражения с автоматическим запуском выглядит так:

```
( function ( ) { операторы ; return значение } ) ( )
```

В приведенных в книге примерах для выполнения кода при загрузке скрипта обычно используются самовызывающиеся функции.

1

Создайте HTML-документ. Вызовите функцию, которая еще не была объявлена.

```
console.log( 'Hoisted: ' + add( 100, 200 ) )
```

2

Добавьте функцию, которая вызывается выше.

```
function add( numOne, numTwo ) {  
  return numOne + numTwo  
}
```

3

Добавьте функцию, которая назначает указанную выше функцию переменной, а потом вызывает назначенную функцию.

```
let addition = add  
console.log( 'Assigned: ' + addition( 32, 64 ) )
```

4

Назначьте аналогичную, но анонимную функцию переменной и вызовите назначенную функцию.

```
let anon = function ( numOne, numTwo ) {  
  let result = numOne + numTwo ; return result  
}  
console.log( 'Anonymous: ' + anon( 9, 1 ) )
```

5

Присвойте значение, возвращаемое самовызывающейся функцией, переменной и отобразите полученное значение.



Самовызывающиеся функциональные выражения также известны как немедленно вызываемые функции (IIFE — часто произносится как «iffy»).



anonymous.html

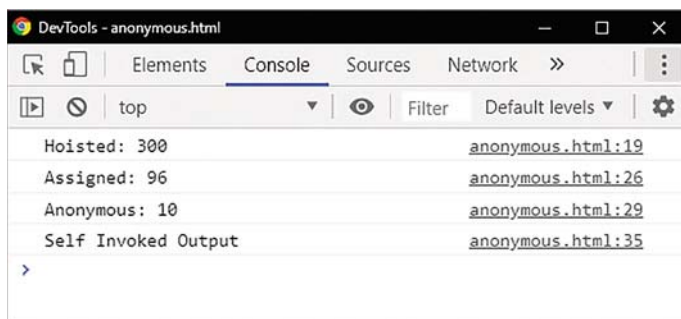


Смысл самовызывающихся функций может быть не сразу понятен, но к концу этой главы их важность станет более ясной и очевидной.

```
let iffy = ( function ( ) {
  let str = 'Self Invoked Output' ; return str
})()
console.log( iffy )
```

6

Сохраните HTML-документ, затем откройте его в браузере и запустите консоль. Проанализируйте полученные результаты — возвращаемые функцией значения.



Область видимости

Все переменные в JavaScript имеют определенную область видимости, в пределах которой они могут действовать. Таких областей две — глобальная и локальная.

Глобальная область видимости

Переменная или функция, созданная в этой области видимости, доступна из любой точки программы. Это означает, что переменные существуют постоянно и доступны для функций в одной области. На первый взгляд это может показаться очень удобным, однако имеется серьезный недостаток, заключающийся в том, что одноименные переменные могут конфликтовать. Представьте, что вы создали глобальную переменную **myName**, которой вами было присвоено имя, но затем подключился внешний код, в котором другой разработчик создал глобальную переменную **myName** с таким же именем. Обе переменные существуют в одной области программы, то есть конфликтуют. Таких ситуаций лучше избегать, поэтому для хранения примитивных значений

не рекомендуется создавать глобальные переменные (всех типов данных, за исключением **Object** и **Function**).

Локальная область видимости

Переменные, созданные внутри функциональных блоков, доступны локально на протяжении всего цикла функции. Они существуют только во время ее выполнения, а затем удаляются. Их область программы ограничена — от места, в котором они были созданы, до последней фигурной скобки `}` или момента возврата из функции. Рекомендуется объявлять переменные в начале функционального блока, чтобы их лексическая область видимости соответствовала времени жизни функции. Это означает, что переменные с одинаковыми именами могут существовать внутри отдельных функций без создания конфликта. Например, локальная переменная **myName** может успешно существовать в программе внутри отдельных функций и внутри функций во включенных внешних кодах. В программе для хранения значений рекомендуется создавать только локальные переменные.

Наиболее эффективный вариант

Объявление глобальных переменных с использованием ключевого слова **var** позволяет конфликтующим переменным с одинаковыми именами перезаписывать присвоенные им значения без предупреждения. В этом случае в результате использования более новых ключевых слов **let** и **const** выдается ошибка **Uncaught SyntaxError**. Поэтому в программе для хранения значений рекомендуется создавать переменные, объявленные с использованием ключевых слов **let** или **const**.

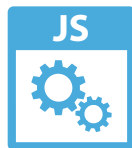
1

Создайте внешний код, в котором вызывается функция для вывода значения глобальной переменной.

```
let myName = 'External Script'  
function readName( ) {console.log( myName ) }  
readName( )
```



Информация по обнаружению и обработке ошибок более подробно описана на стр. 68–69.



external.js



scope.html



Вызовы функций `readName()` и `getName()` остаются в коде без редактирования.

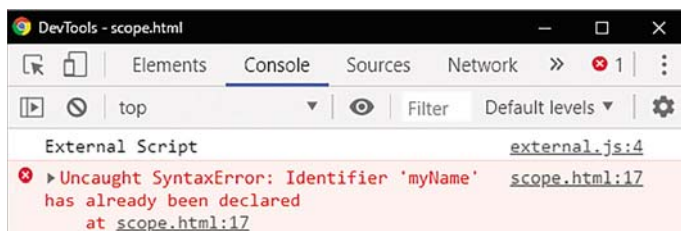
2

Создайте HTML-документ, который подключает внешний код и добавляет аналогичный.

```
<script src="external.js"></script>
<script>
let myName = 'Internal Script'
function getName() {console.log( myName ) }
getName()
</script>
```

3

Сохраните оба файла в одной папке, затем откройте HTML-документ, чтобы увидеть сообщение об ошибке конфликта в консоли.

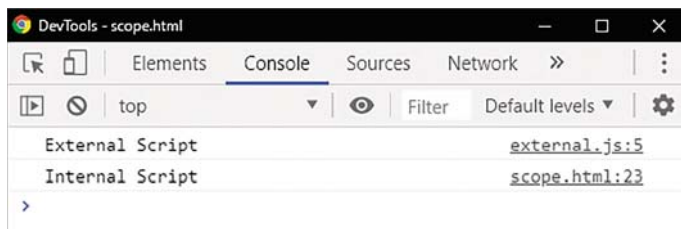


4

Отредактируйте оба скрипта, чтобы преобразовать глобальные переменные в локальные. Во избежание конфликта обновите браузер.

```
function readName() {
  let myName = 'External Script'; console.log (
    myName )
}

function getName() {
  let myName = 'Internal Script'; console.log (
    myName )
}
```



Замыкания

На предыдущем примере была продемонстрирована опасность создания глобальных переменных для хранения значений в JavaScript. Однако возникает необходимость сохранить значения, которые остаются постоянно доступными, — например, чтобы запоминать увеличение счетчика в ходе выполнения программы. Как это сделать без использования глобальных переменных для хранения примитивных значений? Ответ заключается в использовании замыканий.

Замыкание — это комбинация функции и лексического окружения вместе со всеми доступными внешними переменными. В JavaScript замыкания создаются каждый раз при создании функции, во время ее создания.

1

Создайте HTML-документ с кодом, который присваивает глобальной переменной самовызывающуюся анонимную функцию.

```
const add = ( function () {  
  // Здесь будет ваш код.  
})();
```

2

Добавьте операторы для инициализации локальной переменной и назначения функции локальной переменной в той же области.

```
let count = 0  
const nested = function () { return count = count + 1  
}
```

3

Теперь добавьте оператор для возврата внутренней функции — присвоения внутренней функции глобальной переменной.

```
return nested
```

4

Наконец добавьте три идентичных вызова функций к внутренней функции, которая теперь назначена глобальной переменной.

```
console.log( 'Count is ' + add() )
```



closure.html



Самовызывающиеся функции описаны на стр. 24. Они выполняют свои операторы только один раз. В данном случае вы можете использовать `console.log(add)`, чтобы подтвердить, что выражение функции было присвоено внешней переменной.



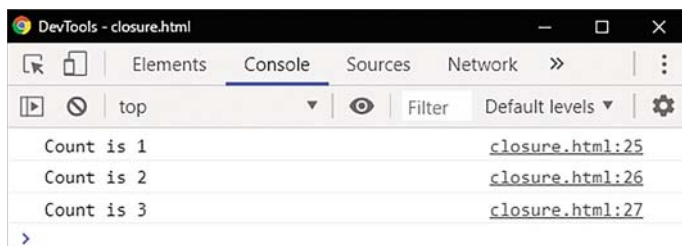
Все объекты JavaScript наследуют свойства и методы от свойства `prototype`. Стандартные объекты JavaScript, такие как функции, для создания объекта вызывают функцию внутреннего конструктора.



Не беспокойтесь, если вы сразу не сможете понять, как работают замыкания. Сначала это покажется сложным, но со временем все станет гораздо проще и понятнее. Вы можете продолжить изучение, а затем вернуться к этому инструменту.

```
console.log('Count is ' + add())
console.log('Count is ' + add())
```

- 5 Сохраните HTML-документ, затем откройте его в браузере и запустите консоль. Проанализируйте полученные результаты — значения, возвращенные при закрытии.

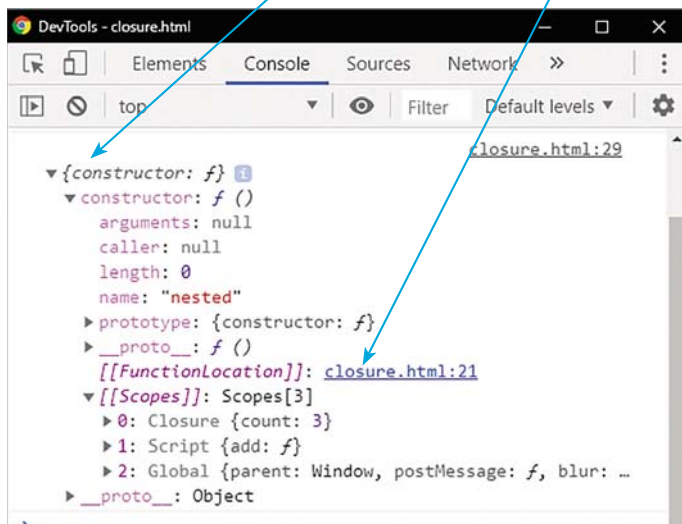


Понять концепцию замыканий непросто — может показаться, что переменная `count` в нашем примере должна быть удалена, когда самовывзывающаяся функция завершит выполнение. Чтобы разобраться, как работают замыкания, вы можете изучить свойство `prototype`.

- 6 В конце кода добавьте следующий оператор.

```
console.log(add.prototype)
```

- 7 Сохраните HTML-документ, затем обновите браузер и разверните раскрывающийся список `constructor`, чтобы увидеть области.



Присмотритесь внимательнее, и вы обнаружите, что назначенная функция имеет особую область видимости (замыкание) в дополнение к обычной локальной (скрипт или код) и внешней (глобальной) области. Таким образом, переменная **count** остается доступной через назначенную функцию. Однако на нее невозможно сослаться каким-либо другим способом.

Считается важным использование замыканий для сокрытия постоянных переменных от других областей кода. Точно так же частные переменные могут быть скрыты в других языках программирования и доступны только при использовании методов чтения.

Заключение

- JavaScript-код может быть включен в HTML-документ напрямую или из внешнего файла с помощью тегов `<script>` `</script>`.
- JavaScript позволяет отображать результаты вывода в диалоговом окне предупреждения или в окне консоли браузера.
- Операторы JavaScript могут содержать ключевые слова, операторы, значения и выражения.
- Интерпретатор JavaScript игнорирует отступы и пробелы.
- Операторы JavaScript могут быть сгруппированы в функциональные блоки с помощью фигурных скобок `{}`, которые при необходимости вызываются для выполнения.
- Имена переменных и функций могут состоять из букв, цифр и символов подчеркивания, но запрещается использовать в качестве имен переменных ключевые слова.
- Переменные JavaScript могут содержать следующие типы данных: строка (`string`), число (`number`), булев или логический тип (`boolean`),

объекты (object), функция (function), символы (symbol), значение (null) и специальное значение (undefined).

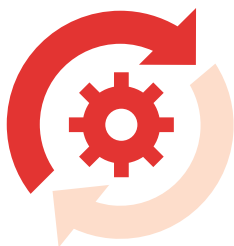
- Переменным, объявленным с использованием ключевого слова **let**, можно дать новые значения. Ключевое слово **const** не предполагает изменений.
- Объявление функции состоит из одного или нескольких операторов, сгруппированных вместе в фигурные скобки { }. Функция вызывает свои инструкции и возвращает в результате единственное значение.
- В функциональном выражении в круглых скобках () могут содержаться параметры значений аргументов, передаваемых от вызывающей стороны.
- Функциональный блок может включать оператор **return** для указания данных, которые должны быть переданы обратно вызывающей стороне.
- Оператор () вызывает функцию.
- Поднятие позволяет вызовам функций появляться в коде до объявления функции.
- Функция может быть анонимной, то есть не иметь имени.
- Лексическая область видимости — это область, где была создана переменная, которая может быть глобальной, локальной или замыканием.
- Замыкание — это функция, вложенная во внешнюю функцию, которая сохраняет доступ к переменным, объявленным во внешней функции.

2

Распространенные операции

*В этой главе вы узнаете
об операторах языка
JavaScript и о том, как они
используются в скриптах.*

- 32 Преобразование типов
- 34 Арифметические операторы
- 36 Операторы присваивания
- 38 Операторы сравнения
- 40 Логические операторы
- 42 Условный (тернарный) оператор
- 44 Побитовые операции
- 46 Приоритет операторов
- 49 Заключение



Преобразование типов

Во избежание непредвиденных результатов перед выполнением операций в JavaScript важно определить типы данных значений, с которыми вы работаете. Например, значение **42** — это число, а значение **'42'** — строка, поэтому в результате сложения **'42' + 8** получится строковое выражение **'428'**, а не число **50**. JavaScript предоставляет несколько способов преобразования типов данных без изменения исходного значения.

Преобразование строки в число

Встроенная функция **parseInt()** принимает строку в качестве аргумента и возвращает целое число в соответствии с указанным основанием системы счисления. Например, **parseInt('42')** вернет число **42**, поэтому в результате сложения **42 + 8** получится число **50**.

Встроенная функция **parseFloat()** принимает строку в качестве аргумента и возвращает десятичное число (число с плавающей запятой).

Оба метода позволяют буквенным символам следовать за числовой частью указанной строки и удалять их из результата — например, **parseInt('42nd Street')** в результате возвращает число **42**.



Преобразование или приведение типов данных может быть явным или неявным. Например, **<42> + 8** в результате выполнения возвращает строку **<428>** — это неявное приведение. Метод **String(42)** возвращает в результате строку **'42'** — это явное приведение.

Если в начале указанной строки ни одна из этих функций не находит числовое значение, результатом будет **NaN** (Not-A-Number) — свойство JavaScript, означающее «не число». С помощью функции **isNaN()** вы можете проверить, является ли переменная или литерал нечисловым значением. В начале указанного значения она попытается найти число и вернет **false**, если число обнаружено (даже в указанной строке). В противном случае, если этого не произойдет, в результате вернется значение **true**.

Преобразование числа в строку

Метод **String()** — строковое представление числа, указанного в круглых скобках, например, **String(42)** вернет в результате строковый тип данных **'42'**.

Чтобы вернуть строковое представление сохраненного числового типа данных, к имени переменной можно добавить вызов метода `toString()`. Например, если переменной с именем «num» был присвоен номер, метод `num.toString()` в результате вернет строковое представление этого сохраненного числа.

1

Создайте HTML-документ с самовызывающейся анонимной функцией, объявите и инициализируйте три переменные.

```
( function () {  
    let sum, net = '25', tax = 5.00  
    // Здесь будет ваш код.  
})()
```

2

Добавьте операторы, которые создают разные типы данных и выводят результат в выходной строке.

```
sum = net + tax  
console.log( 'sum: ' + sum + ' ' + typeof sum )  
sum = parseFloat( net ) + tax  
console.log( 'sum: ' + sum + ' ' + typeof sum )  
console.log( 'tax: ' + tax + ' ' + typeof tax )  
tax = tax.toString()   
console.log( 'tax: ' + tax + ' ' + typeof tax )  
net = '$' + net  
console.log( 'net: ' + net + ' ' + parseInt( net ) )  
console.log( 'net Not a Number? ' + isNaN( net ) )
```

3

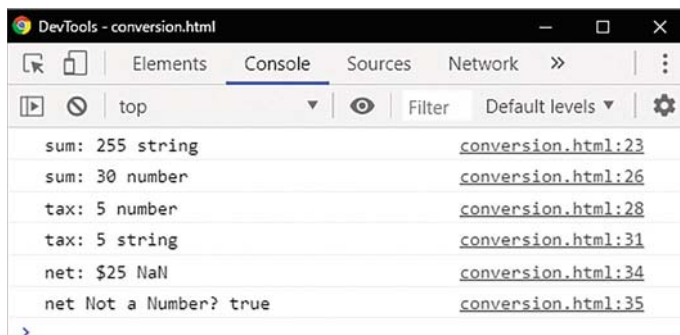
Сохраните HTML-документ, затем откройте его в браузере и запустите консоль. Проанализируйте результаты вывода.

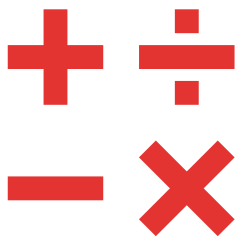


conversion.html



Если вы вызовете функцию `isNaN(net)` до того, как к строке будет добавлен префикс '\$', результат будет `false` (ложь), так как в начале строки функция находит число '25'.





Арифметические операторы

Перед вами наиболее распространенные в JavaScript арифметические операторы:

Оператор	Описание
+	Оператор сложения возвращает сумму числовых операндов или объединяет строки
-	Оператор вычитания
*	Оператор умножения
/	Оператор деления
%	Остаток от деления
++	Инкремент
--	Декремент
**	Возведение в степень



Оператор возведения в степень ****** возвращает результат первого операнда, возведенного в степень.

36

Данные, обрабатываемые сценарием JavaScript, называются операндами. Например, в выражении **5+2** оператору **+** передаются значения операндов **5** и **2**. Обратите внимание, что в зависимости от типа операндов оператор **+** выполняет два типа операций. Числовые операнды добавляются для получения суммы чисел, однако при сложении строковых операндов в результате возвращается объединенная строка.

Оператор **%** возвращает целый остаток от деления левого операнда на правый. Деление на **2** вернет либо **1**, либо **0**.



Пример использования оператора **%** с нечетными или четными числами можно найти на стр. 45.

Оператор инкремент **++** и оператор декремент **--** увеличивают и уменьшают значение операнда на единицу соответственно и возвращают новое значение. Эти операторы чаще всего используются для подсчета итераций цикла двумя разными способами. Если форма операции постфикс (оператор после операнда), значение операнда возвращается, а затем увеличивается или уменьшается на единицу. Если применяется префиксная форма (оператор перед операндом), значение операнда возвращается увеличенным или уменьшенным на единицу.

1

Создайте HTML-документ с самовызывающейся функцией.

```
( function () {  
  // Здесь будет ваш код.  
})();
```



arithmetic.html

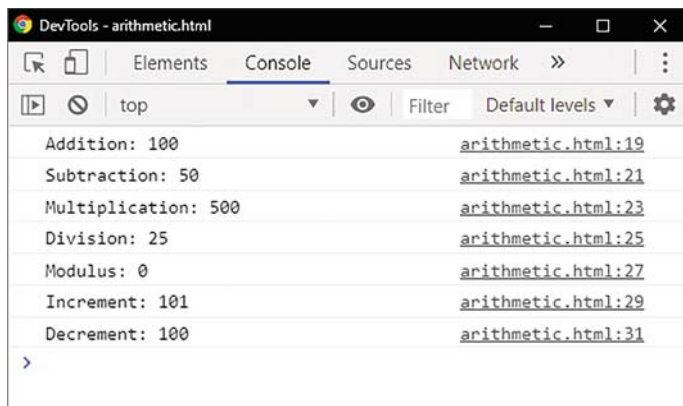
2

Добавьте операторы, которые присваивают значения переменным, используя каждый арифметический оператор, и проанализируйте результат.

```
let sum = 80 + 20 ; console.log( 'Addition: ' + sum )  
  
let sub = sum - 50 ; console.log( 'Subtraction: ' + sub )  
  
let mul = sum * 5 ; console.log( 'Multiplication: ' + mul )  
  
let div = sum / 4 ; console.log( 'Division: ' + div )  
  
let mod = sum % 2 ; console.log( 'Modulus: ' + mod )  
  
let inc = ++sum ; console.log( 'Increment: ' + inc )  
  
let dec = --sum ; console.log( 'Decrement: ' + dec )
```

3

Сохраните HTML-документ, затем откройте его в браузере и запустите консоль. Проанализируйте результаты вывода — значения, полученные в результате выполнения арифметических операций.





Оператор равенства `===` сравнивает значения. Более подробная информация об операторах сравнения приведена на стр. 40.

Операторы присваивания

Ниже в таблице представлены наиболее распространенные в JavaScript операторы присваивания. Все операторы, кроме `=`, — это сокращенная форма более длинных выражений, поэтому каждому оператору приводится эквивалент.

Оператор	Пример	Эквивалент
<code>=</code>	<code>a = b</code>	<code>a = b</code>
<code>+=</code>	<code>a += b</code>	<code>a = (a + b)</code>
<code>-=</code>	<code>a -= b</code>	<code>a = (a - b)</code>
<code>*=</code>	<code>a *= b</code>	<code>a = (a * b)</code>
<code>/=</code>	<code>a /= b</code>	<code>a = (a / b)</code>
<code>%=</code>	<code>a %= b</code>	<code>a = (a % b)</code>
<code>**=</code>	<code>a **= b</code>	<code>a = (a ** b)</code>

Важно понимать, что оператор `=` означает «присваивать», а не «равно». Для сравнения оператор равенства в JavaScript выглядит следующим образом: `===`.

На примере в таблице оператор присваивания `=` присваивает переменной `a` значение, содержащееся в переменной `b`, и в результате возвращается новое сохраненное значение.

Оператор присваивания `+=` считается наиболее полезным и может использоваться для добавления новой строки к существующей. На следующем примере `let str='JavaScript'` и `str+='Fun'` мы видим, что теперь переменная хранит объединенную строку `'JavaScript Fun'`.

На примере в таблице оператор присваивания `+=` добавит значение, содержащееся в переменной `a`, к значению, содержащемуся в переменной `b`, а затем назначит итоговую сумму, чтобы она стала новым значением, сохраненным в переменной `a`.

Все остальные комбинированные операторы присваивания работают аналогично. Каждый из них сначала выполняет арифметическую операцию с двумя

операндами, а затем присваивает результат этой операции первой переменной, и она становится ее новым сохраненным значением.



assignment.html

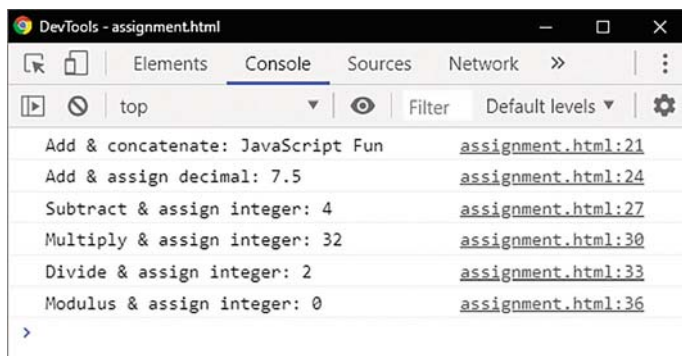
- 1 Создайте HTML-документ с самовызывающейся анонимной функцией, объединяющей две строки.

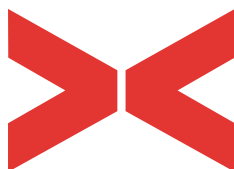
```
( function () {  
    let msg = 'JavaScript' ; msg += ' Fun'  
    console.log( 'Add & concatenate: ' + msg )  
    // Здесь будет ваш код.  
})()
```

- 2 Добавьте операторы, использующие для выполнения арифметических операций комбинированные операторы, и выведите результат.

```
let sum = 5.00 ; sum += 2.50  
console.log( 'Add & assign decimal: ' + sum )  
  
sum = 8 ; sum -= 4  
console.log( 'Subtract & assign integer: ' + sum )  
  
sum = 8 ; sum *= 4  
console.log( 'Multiply & assign integer: ' + sum )  
  
sum = 8 ; sum /= 4  
console.log( 'Divide & assign integer: ' + sum )  
  
sum = 8 ; sum %= 4  
console.log( 'Modulus & assign integer: ' + sum )
```

- 3 Сохраните HTML-документ, затем откройте его в браузере и запустите консоль. Проанализируйте вывод — результат выполнения операторов присваивания.





Пример использования оператора меньше `<` в структуре цикла можно найти на стр. 61.



Также существуют стандартные оператор равенства `==` и оператор неравенства `!=`. Однако они могут привести к непредвиденным результатам. В отличие от операторов строгого равенства, они не гарантируют, что сравниваемые значения относятся к одному типу данных. Например, в результате сравнения `25== '25'` возвращается значение `true`, а в результате сравнения `25=== '25'` — `false`. Для точных сравнений всегда используйте трехсимвольные операторы.

Операторы сравнения

Ниже в таблице представлены наиболее распространенные в JavaScript операторы сравнения.

Оператор	Описание
<code>===</code>	Строго равно
<code>!==</code>	Строго не равно
<code>></code>	Больше
<code><</code>	Меньше
<code>>=</code>	Больше или равно
<code><=</code>	Меньше или равно

Оператор равенства `===` сравнивает два операнда и возвращает логическое значение `true` (истина) в том случае, если операнды строго равны. В противном случае возвращается логическое значение `false` (ложь). Если операнды — одинаковые числа, они равны; если операнды представляют собой строки, содержащие одинаковые символы в одинаковых позициях, они равны; если операнды — логические значения, которые оба принимают значение `true` или оба принимают значение `false`, они равны. Оператор не равно `!==` возвращает значение `true` (истина) в том случае, если операнды не равны, используя те же правила, что и для оператора `===`.

Операторы строго равно и строго не равно полезны при сравнении двух значений для выполнения «условного ветвления», когда необходимо выполнить различные действия в зависимости от условий.

Оператор больше `>` возвращает значение `true` в том случае, если значение левого операнда больше, чем правого. Оператор меньше `<` возвращает значение `true` в том случае, если значение левого операнда меньше, чем правого. При добавлении символа `=` после оператора `>` или оператора `<` в результате возвращается значение `true` еще в том случае, если два операнда равны.

Операторы больше `>` и меньше `<`, как правило, используются для проверки значения переменной счетчика в структуре цикла.

- 1 Создайте HTML-документ с самовызывающейся анонимной функцией, объявите и проинициализируйте три переменные.

```
( function () {  
  let comparison, sum = 8, str = 'JavaScript'  
  // Здесь будет ваш код.  
} ) ()
```



comparison.html

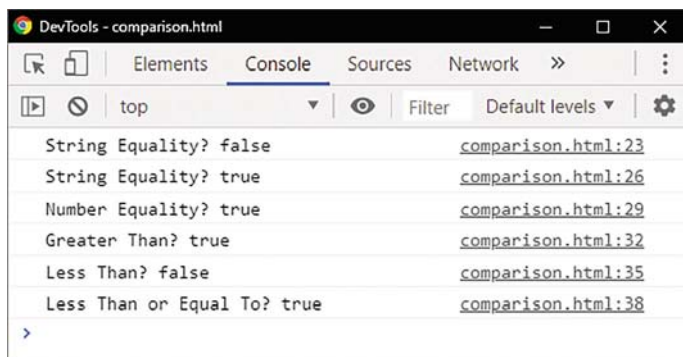
- 2 Добавьте операторы, использующие операторы сравнения и выведите результат.

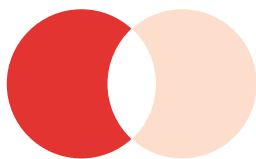
```
comparison = str === 'JAVASCRIPT'  
console.log( 'String Equality? ' + comparison )  
  
comparison = str === 'JavaScript'  
console.log( 'String Equality? ' + comparison )  
  
comparison = sum === 8  
console.log( 'Number Equality? ' + comparison )  
  
comparison = sum > 5  
console.log( 'Greater Than? ' + comparison )  
  
comparison = sum < 5  
console.log( 'Less Than? ' + comparison )  
  
comparison = sum <= 8  
console.log( 'Less Than or Equal To? ' + comparison )
```



JavaScript чувствителен к регистру, поэтому регистр символов должен совпадать, чтобы сравниваемые строки были равны.

- 3 Сохраните HTML-документ, затем откройте его в браузере и запустите консоль. Проанализируйте вывод — результат выполнения операторов сравнения.





Логические операторы

В JavaScript существуют три логических оператора.

Оператор	Описание
&&	Логическое И
 	Логическое ИЛИ
!	Логическое НЕ

Логические операторы обычно используются вместе с операндами, имеющими логическое значение **true** или **false**, или значениями, которые могут быть в них преобразованы.



Термин «булев» относится к системе логического мышления, разработанной английским математиком Джорджем Булем (1815–1864).

Оператор логическое И **&&** будет оценивать два операнда и возвращать значение **true**, если оба операнда истинны, а иначе — **false**. Это часто используется при условном ветвлении, когда необходимо выполнить различные действия в зависимости от условий.

В отличие от логического оператора **&&**, которому необходимо, чтобы оба операнда были истинными, оператор логическое ИЛИ **||** находит первое истинное значение и возвращает значение **true**, а иначе — **false**. Это полезно для выполнения сценариев определенного действия в зависимости от условий.

Третий логический оператор — это оператор логического НЕ **!**, который сначала приводит аргумент к логическому типу **true/false**, а затем возвращает противоположное значение. Например, если переменная с именем «tog» имеет значение **true**, то **!tog** вернет в результате значение **false**. Это полезно для инвертирования значения переменной в последовательных итерациях цикла, например **tog=!a**. А при каждой итерации значение будет меняться на противоположное. Например, включение и выключение светового переключателя.

1

Создайте HTML-документ с самовызывающейся анонимной функцией, объявите и проинициализируйте три переменные.

```
( function () {
  let result, yes = true, no = false
  // Здесь будет ваш код.
} ) ()
```



logic.html

2

Добавьте операторы, использующие логические операторы, и выведите результат.

```
result = yes && yes
console.log( 'Are both true? ' + result )

result = yes && no
console.log( 'Are both still true? ' + result )

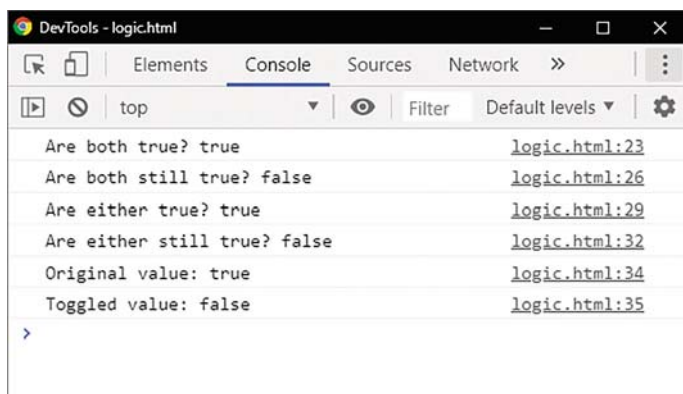
result = yes || no
console.log( 'Are either true? ' + result )

result = no || no
console.log( 'Are either still true? ' + result )

console.log( 'Original value: ' + yes )
yes = !yes
console.log( 'Toggled value: ' + yes )
```

3

Сохраните HTML-документ, затем откройте его в браузере и запустите консоль. Проанализируйте вывод — результат выполнения логических операторов.



Обратите внимание, что выражение `false&&false` возвращает значение `false`, а не `true`, как может показаться, действуя по принципу «два заблуждения — еще не правда».

Условный (тернарный) оператор



Возможно, наиболее полюбившийся оператор создателя JavaScript — это условный оператор?:. Он также известен как **тернарный**, то есть состоящий из трех операндов.

Первый операнд вычисляется и используется как логическое значение. Если он имеет значение **true**, то вычисляется и возвращается значение выражения во втором операнде. Если же значение **false**, то вычисляется и возвращается значение выражения в третьем операнде. Вычисляется всегда только какой-то один из операндов, второй или третий, и никогда оба. Его синтаксис выглядит так:

условие ? если-true-выполнить-это : если-false-выполнить-это

Если в зависимости от условий требуется выполнить несколько действий, каждый указанный оператор может быть вызовом функции для выполнения нескольких операторов в каждой из них. Например, вызов функций для выполнения нескольких операторов в соответствии с логическим значением переменной с именем «flag»:

```
flag === true ? doThis() : doThat()
```

В этом примере нет необходимости использовать оператор равенства **===** и ключевое слово **true**, так как операторы, вычисляющие выражение для логического значения, автоматически выполняют эту проверку. Поэтому пример можно упростить:

```
flag ? doThis() : doThat()
```

В качестве альтернативы два оператора, указанные для тернарного оператора, могут присвоить значение переменной в зависимости от условий проверки, например:

```
flag ? str = 'Go left' : str = 'Go right'
```

Это синтаксически верно, но также можно упростить:

```
str = flag ? 'Go left' : 'Go right'
```



Тернарный оператор состоит из трех операндов. Первый предшествует символу **?**, второй — между символами **?** и **:**, третий — после символа **:**.

Если в условии проверяется четность числового значения, два оператора могут предоставлять варианты в зависимости от того, четное число или нечетное.

1

Создайте HTML-документ с самовызывающейся анонимной функцией, объявите и проинициализируйте две переменные.

```
( function () {  
  const numOne = 8, numTwo = 3  
  // Здесь будет ваш код.  
} ) ()
```



ternary.html

2

Добавьте операторы для вывода строки, обозначающей количество.

```
let verb = ( numOne !== 1 ) ? 'are' : 'is'  
console.log( 'There' + verb + numOne )
```

3

Для вывода строк, корректно описывающих четность двух значений переменных, добавьте следующие операторы.

```
let parity = ( numOne % 2 !== 0 ) ? 'Odd' : 'Even'  
console.log( numOne + ' is ' + parity )  
  
parity = ( numTwo % 2 !== 0 ) ? 'Odd' : 'Even'  
console.log( numTwo + ' is ' + parity )
```

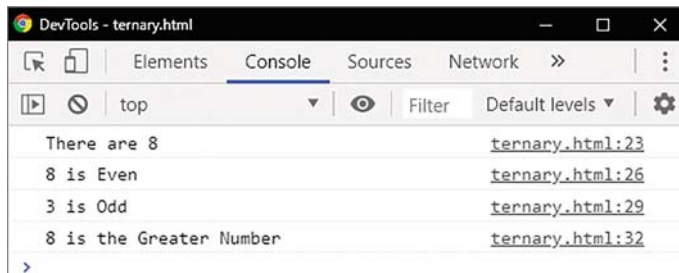
4

Добавьте операторы для вывода строки, сообщающей большее из двух значений переменных.

```
let max = ( numOne > numTwo ) ? numOne : numTwo  
console.log( max + ' is the Greater Number' )
```

5

Сохраните HTML-документ, затем откройте его в браузере и запустите консоль. Проанализируйте результаты вывода.



Тернарный оператор может возвращать значения любого типа данных, будь то строка, число, логическое значение и т. д.



Многие авторы книг по программированию на JavaScript не дают информацию по побитовым операторам. Однако полезно понять, что они из себя представляют и как их можно использовать.



Байт состоит из 8 бит, и каждая половина байта известна как полубайт (4 бита). Двоичные числа, представленные в примерах таблицы, описывают значения, хранящиеся в полубайте.

Побитовые операции

Побитовые операторы JavaScript интерпретируют операнды как последовательность из 32 битов (нулей и единиц). Каждый бит передает десятичный компонент только в том случае, если он содержит единицу. Компоненты рассматриваются справа налево от «младшего значащего бита» (LSB) до «старшего значащего бита» (MSB). Ниже представлено двоичное число в восьмибитном представлении — десятичное число 50, что обозначено битами, установленными со значением 1 ($2 + 16 + 32 = 50$).

Биты	8	7	6	5	4	3	2	1
Десятичное представление	128	64	32	16	8	4	2	1
Двоичное представление	0	0	1	1	0	0	1	0

В следующей таблице перечислены все побитовые операторы, используемые в JavaScript.

Оператор	Название	Binary number operation:
	Побитовое ИЛИ (OR)	Возвращает 1 в тех разрядах, которые хотя бы у одного из операндов были равны 1 . Пример: $1010 \mid 0101 = 1111$
&	Побитовое И (AND)	Возвращает 1 в тех разрядах, которые у обоих операндов были равны 1 . Пример: $1010 \& 1100 = 1000$
~	Побитовое НЕ (NOT)	Заменяет каждый бит операнда на противоположный. Возвращает 1 , если бит не равен 1 , и 0 — если бит равен 1 . Пример: $\sim 1010 = 0101$
^	Побитовое исключающее ИЛИ (XOR)	Возвращает 1 в тех позициях, которые только у одного из операндов были равны 1 . Пример: $1010 \wedge 0100 = 1110$

Оператор	Название	Binary number operation:
<<	Левый сдвиг	Сдвигает двоичное представление числа на некоторое количество разрядов влево, добавляя справа нули. Пример: 0010 << 2 = 1000
>>	Правый сдвиг, переносящий знак	Сдвигает двоичное представление числа на некоторое количество разрядов вправо. Освобождающиеся разряды заполняются знаковым битом. Пример: 1000 >> 2 = 0010
>>>	Правый сдвиг с заполнением нулями	Сдвигает двоичное представление числа на некоторое количество разрядов вправо. Освобождающиеся разряды заполняются нулями. Пример: 1000 >>> 2 = 0010

1

Создайте HTML-документ с самовызывающейся анонимной функцией, объявите и проинициализируйте две переменные.

```
( function ( ) {
  let numOne = 10, numTwo = 5
  // Здесь будет ваш код.
} ) ( )
```

2

Добавьте операторы для простого вывода строк, подтверждающих начальные значения, хранящиеся в каждой переменной.

```
console.log( 'numOne: ' + numOne )
console.log( 'numTwo: ' + numTwo )
```

3

Добавьте операторы, чтобы с помощью побитовых операций поменять местами значения, хранящиеся в каждой переменной.

```
numOne = numOne ^ numTwo
// 1010 ^ 0101 = 1111 = (десятичное число 15)
```



bitwise.html

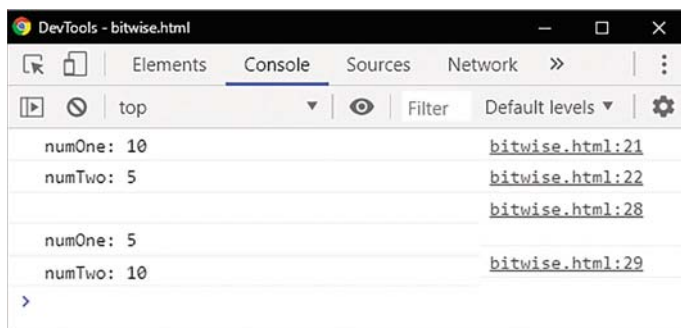


Обратите внимание, каким образом в нашем примере для разрыва строки используется специальная escape-последовательность `\n`.

- 4 Добавьте операторы для вывода разрыва строки и строк, чтобы подтвердить окончательные значения, хранящиеся в каждой переменной.

```
console.log( '\n' + 'numOne: ' + numOne )
console.log( 'numTwo: ' + numTwo )
```

- 5 Сохраните HTML-документ, затем откройте его в браузере и запустите консоль. Проанализируйте результаты вывода.



Приоритет операторов



Операторы JavaScript имеют разные уровни приоритета. Приоритет операторов определяет порядок, в котором операторы выполняются. Операторы с более высоким приоритетом выполняются раньше операторов с более низким приоритетом.

В таблице сверху вниз перечислены операторы каждого типа в порядке от самого высокого до самого низкого уровня приоритета:

Оператор	Описание
()	Группировка
.	Оператор доступа к объекту
[]	Оператор доступа к массиву
()	Вызов функции
++ -	Постфиксный инкремент, постфиксный декремент
++ -	Префиксный инкремент, префиксный декремент
! ~	Логическое отрицание, побитовое отрицание
**	Возведение в степень
* %	Умножение, деление, остаток от деления
+ -	Сложение, вычитание
<< >> >>>	Побитовый сдвиг
< <= > >=	Сравнение
=== !== !=	Строго равно, равно, строго не равно, не равно
&	Побитовое И
^	Побитовое исключающее ИЛИ
	Побитовое ИЛИ
&&	Логическое И
	Логическое ИЛИ
?:	Тернарный оператор
=	Операторы присваивания
+= -= *= /=%=	
&= ^= =	
<<= >>= >>>=	
,	
	Запятая

Приоритет



Оператор [] описан в разделе о массивах, начиная со стр. 81. Также обратите внимание, что оператор. (точка) используется в точечной нотации, например `console.log()`. Он нужен для доступа к свойствам или методам объекта (массива или функции).

1

Создайте HTML-документ с самовызывающейся анонимной функцией, которая инициализирует переменную с результатом разгруппированного выражения и выводит его значение.

```
( function () {
  let sum = 9 + 12 / 3           // Эквивалентно 9 + 4.
  console.log( 'Ungrouped sum: ' + sum )
})()
```

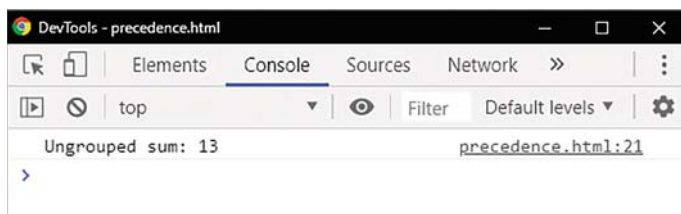


precedence.html



Для определения порядка вычислений в выражениях рекомендуется использовать круглые скобки ().

- 2 Сохраните HTML-документ, затем откройте его в браузере и запустите консоль. Проанализируйте полученные результаты.

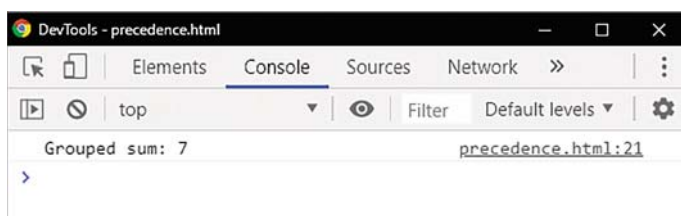


Сначала вычисляется деление, так как оператор деления имеет более высокий приоритет по сравнению с оператором сложения, поэтому результат равен 13. Также вы можете определить порядок приоритета, сгруппировав выражение в круглых скобках, и оно будет вычислено первым, так как оператор () имеет самый высокий приоритет.

- 3 Чтобы установить порядок вычисления, вы можете изменить операторы в функции, и сложение будет вычисляться перед делением.

```
let sum = ( 9 + 12 ) / 3 // Эквивалентно 21 / 3.
console.log( 'Grouped sum: ' + sum )
```

- 4 Сохраните HTML-документ, затем откройте его в браузере и запустите консоль. Проанализируйте результаты вывода.



Заключение

- Функции `parseInt()` и `parseFloat()` преобразовывают строки в числа. Функции `String()` и `toString()` преобразовывают числа в строки.
- Функция `isNaN()` определяет, является ли литерал или переменная нечисловым значением `NaN`.
- Арифметические операторы выполняют математические операции: сложение `+`, остаток от деления `%`, инкремент `++`, декремент `--` и возведение в степень `**`.
- Если операция используется как постфикс (оператор после операнда), значение операнда возвращается, а затем увеличивается или уменьшается на единицу. Если используется префиксная форма (оператор перед операндом), значение операнда возвращается увеличенным или уменьшенным на единицу.
- Оператор `=` может быть объединен с арифметическим оператором для выполнения арифметической операции, а затем присвоения ее результата.
- Оператор присваивания `+=` считается наиболее полезным и может использоваться для добавления строки к уже существующей.
- Операнды можно сравнивать. Существуют следующие операторы сравнения: строго равно `===`, строго не равно `!==`, больше `>` или меньше `<`.
- Комбинированные операторы сравнения `<=` и `>=` возвращают значение `true`, если оба операнда равны.
- Оператор логическое И `&&` проверяет два операнда и возвращает `true`, если оба истинны. Оператор логическое ИЛИ `||` находит первое истинное значение и возвращает значение `true`, а иначе — `false`.

- Оператор логического НЕ! сначала приводит аргумент к логическому типу **true/false**, а затем возвращает противоположное значение.
- Тернарный оператор?: состоит из трех операндов. Первый вычисляется и используется как логическое значение. Условие тернарного оператора задается в первом операнде. Если он имеет значение **true**, то вычисляется и возвращается значение выражения во втором операнде. Если же значение **false**, то вычисляется и возвращается значение выражения в третьем операнде.
- Побитовые операторы JavaScript для выполнения двоичной арифметики могут манипулировать отдельными битами двоичной последовательности.
- Операторы JavaScript имеют разные уровни приоритета. Приоритет операторов определяет порядок, в котором операторы выполняются.
- Для принудительного вычисления определенных частей выражения перед другими используются круглые скобки.

3

Управляющие конструкции в JavaScript

*В этой главе вы узнаете
об управляющих
конструкциях в JavaScript,
их назначении и правилах
использования.*

- 52 Ветвление с помощью оператора if
- 54 Альтернативное ветвление
- 56 Ветвление с помощью переключателей (оператор switch)
- 58 Цикл for
- 60 Цикл while
- 62 Цикл do..while
- 64 Выход из циклов
- 66 Обработка ошибок
- 69 Заключение



Ветвление с помощью оператора if

Иногда возникает необходимость выполнить различные действия в зависимости от условий.

В JavaScript проверка определенных условий выполняется с помощью ключевого слова **if**. Условный оператор **if** реализует выполнение конкретных действий при условии, что некоторое логическое выражение принимает значение **true** или **false**. Если результат равен **true**, выполняется одна инструкция, в противном случае — другая.

Необходимо, чтобы проверяемое условие после ключевого слова **if** было заключено в круглые скобки.

Синтаксис оператора **if** выглядит следующим образом:

if (условие) код выполняется, если условие истинно

Выполняемые инструкции могут быть представлены одной строкой или блоком кода, заключенными в фигурные скобки.

```
if ( условие )
{
    код выполняется, если условие истинно
    код выполняется, если условие истинно
    код выполняется, если условие истинно
}
```

Оценка условий и выполнение действий в соответствии с их результатом — это настоящий мыслительный процесс. Например, запрограммируем действия, которые вы можете выполнить в летний день:

```
let temperature = readThermometer( )
const tolerable = 25
```

```
if ( temperature > tolerable )
{
    turn_on_air-conditioning( )
    get_a_cool_drink( )
    stay_in_shade( )
}
```



Рекомендуется заключать в фигурные скобки даже отдельные операторы — для сохранения согласованного стиля.

Проверка условий эквивалентна следующему синтаксису `if(условие === true)`. Но поскольку JavaScript автоматически преобразует результат выражения в логическое значение, т. е. выполняет проверку равенства для истинного значения, нет необходимости включать в скобки оператор `===true`.

1

Создайте HTML-документ с самовызывающейся функцией, которая начинается с инициализации логической переменной.

```
let flag = true
```

2

Вставьте операторы для выполнения условий логического значения переменной.

```
if( !flag )
{
    console.log( 'Power is OFF' )
}
if( flag )
{
    console.log( 'Power is ON' )
}
```

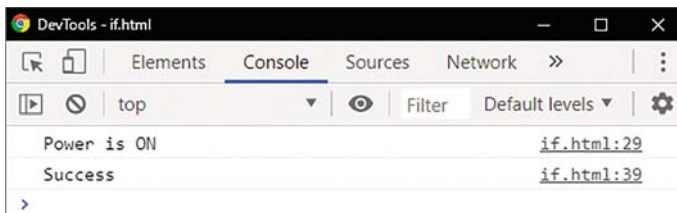
3

Вставьте операторы для выполнения условий выражения, сравнивающего два целых числа.

```
if( 7 < 2 )
{
    console.log( 'Failure' )
}
if( 7 > 2 )
{
    console.log( 'Success' )
}
```

4

Сохраните HTML-документ, затем откройте его в браузере и запустите консоль. Проанализируйте полученные результаты.



if.html



В этом примере оператор **NOT** используется для инвертирования проверки условий, чтобы он стал эквивалентным `if (flag===false)`.



Код сценария, создающий функциональный блок, в нашем примере опущен. Более подробную информацию по созданию анонимных самовызывающихся функций вы можете прочитать на стр. 24.



Альтернативное ветвление

Оператор **if**, который проверяет условие для логического значения и выполняет свои операторы только тогда, когда результат равен **true**, обеспечивает единственную ветвь для выполнения действий. Альтернативная ветвь, по которой действия могут быть выполнены при ложном результате, обозначается ключевым словом **else**.

После оператора **if** следует оператор **else**, например:

if (условие) выполняется, если условие истинно
else выполняется, если условие ложно

Оператор **if else** может также выполнять несколько инструкций, если заключить их в фигурные скобки, например:

```
if ( условие )
{
    выполняется, если условие истинно
    выполняется, если условие истинно
}
else
{
    выполняется, если условие ложно
    выполняется, если условие ложно
}
```



Как только условие в операторе **if else** оказывается равным **true**, выполняются связанные с ним операторы, затем поток продолжается после оператора **if else** — без проверки последующих операторов **else**.

Иногда возникает необходимость проверить несколько вариантов условия. Для этого используется блок **else if**, например:

```
if ( условие )
{
    выполняется, если условие истинно
}
else if ( условие )
{
    выполняется, если условие истинно
}
```

```

else if ( условие )
{
    выполняется, если условие истинно
}
else
{
    выполняется, если условие ложно
}

```

Оператор **if else if** может многократно проверять переменную или условия. Последний блок **else** срабатывает по умолчанию, когда не выполнено ни одно условие.

1

Создайте HTML-документ с самовызывающейся функцией, объявите и проинициализируйте две переменные.

```

let flag = false
const num = 10

```



else.html

2

Для выполнения условий логического значения первой переменной вставьте следующие операторы.

```

if( !flag )
{
    console.log( 'Power is OFF' )
}
else
{
    console.log( 'Power is ON' )
}

```

3

Для выполнения условий числового значения второй переменной добавьте следующие операторы.

```

if( num === 5 )
{
    console.log( 'Number is Five' )
}
else if( num === 10 )
{
    console.log( 'Number is Ten' )
}

```

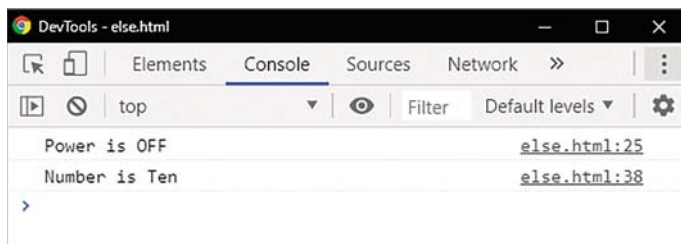


Оператор **if** может содержать блок **else**. Он выполняется, когда условие ложно. Блок **else if** используется при необходимости проверить несколько вариантов условия.

```
else  
{  
    console.log( 'Number is Neither Five nor Ten' )  
}
```

4

Сохраните HTML-документ, затем откройте его в браузере и запустите консоль. Проанализируйте полученные результаты — условия, которые были выполнены, а которые проигнорированы.



Ветвление с помощью оператора switch

Оператор **if else** подходит для проверки лишь нескольких условий, однако может стать неудобным при столкновении со множеством условий. В такой ситуации наиболее эффективно использовать оператор **switch**.

Оператор **switch** работает необычным образом. Он сравнивает выражение с вариантами, перечисленными внутри нее, а затем выполняет соответствующие действия. Если совпадение найдено, оператор **switch** выполнит один или несколько операторов, связанных с этим значением, в противном случае он выполнит один или несколько операторов, указанных как «операторы по умолчанию».

Оператор **switch** начинается с заключения выражения, которое необходимо проверить, в круглые скобки после ключевого слова **switch**. Затем следуют фигурные скобки {}, которые содержат возможные



совпадения. Каждое значение соответствия следует за ключевым словом **case** и использует символ двоеточия: для связывания одного или нескольких выполняемых операторов. Необходимо отметить, что каждый случай должен заканчиваться оператором **break**, который выполняет выход из блока **switch**.

Опциональный оператор **break** может располагаться в каждом из случаев, однако он необязателен. В случае отсутствия оператора **break** выполняются следующие операторы из блока **switch**.

Синтаксис оператора **switch** выглядит таким образом:

switch (выражение)

```
{  
  case значение-1 : операторы, которые выполняются  
    при нахождении совпадения ; break  
  case значение-2 : операторы, которые выполняются  
    при нахождении совпадения ; break  
  case значение-3 : операторы, которые выполняются  
    при нахождении совпадения ; break  
  default : операторы, которые выполняются, если совпадения не найдено  
}
```

Ограничений на количество блоков **case**, которые могут быть включены в блок операторов **switch**, не существует. Поэтому его использование — это идеальный вариант, чтобы сопоставить любое из десятков, сотен или даже тысяч различных значений.

1

Создайте HTML-документ с самовызывающейся функцией, объявите и проинициализируйте переменную.

let **day**

2

Затем добавьте оператор **switch**, чтобы присвоить значение переменной после проверки выражения.

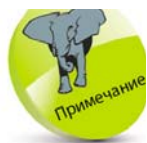
```
switch( 5 - 2 )  
{  
  case 1 : day = 'Monday' ; break
```



Игнорирование оператора **break** позволяет сценарию выполнять операторы, связанные с последующими несоответствующими значениями блока **case**.



switch.html



Строковые значения, предлагаемые в качестве возможных совпадений в блоке `case`, должны быть заключены в кавычки, как и все другие строковые значения.

```
case 2 : day = 'Tuesday' ; break
case 3 : day = 'Wednesday' ; break
case 4 : day = 'Thursday' ; break
case 5 : day = 'Friday' ; break
default : day = 'Weekend'
}
```

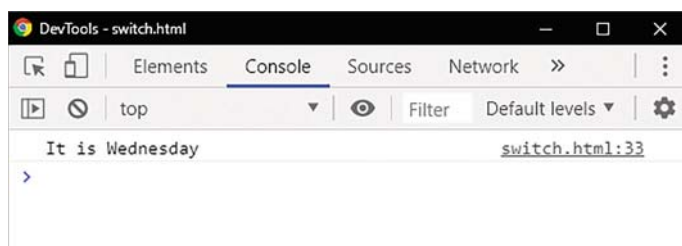
3

Добавьте оператор для вывода значения, назначенного оператором `case` при совпадении.

```
console.log( 'It is ' + day )
```

4

Сохраните HTML-документ, затем откройте его в браузере и запустите консоль. Проанализируйте полученные результаты.



Цикл for



Цикл — это структура, содержащая условие и один или несколько операторов, которые повторно выполняются, пока выполняется условие. Повторение процесса с целью получения результата называется итерацией. Если условие не выполняется, дальнейшие итерации также игнорируются, и выполнение продолжается со следующего оператора, идущего за структурой цикла.

Цикл **for** — наиболее распространенная структура цикла в JavaScript. Он имеет такой синтаксис:

```
for ( инициализатор ; условие ; модификатор ) { операторы на выполнение }
```

В скобках после ключевого слова **for** указываются три выражения, управляющие числом итераций цикла:

- Инициализатор — инструкция, указывающая начальное значение переменной, которая будет использоваться для подсчета количества итераций цикла. Обычно эту переменную счетчика называют просто «i».
- Условие — выражение, которое при каждой итерации проверяется на логическое значение **true**. Если выражение принимает значение **true**, выполняются операторы цикла для завершения итерации. Если выражение принимает значение **false**, операторы не выполняются, и цикл завершается. Обычно в условии проверяется значение переменной счетчика цикла.
- Модификатор — оператор, который изменяет значение в условии, поэтому в какой-то момент в результате вернется значение **false**. Обычно при этом увеличивается или уменьшается значение переменной счетчика цикла.

Например, структура цикла **for** при выполнении набора операторов сто раз может выглядеть следующим образом:

```
let i
for ( i = 0 ; i < 100 ; i++ ) { операторы-на-выполнение }
```

В этом случае при каждой итерации переменная счетчика увеличивается, пока ее значение не достигнет значения **100**, после чего в результате вернется значение **false**, и цикл завершится.

- 1 Создайте HTML-документ с самовызывающейся функцией, объявите и проинициализируйте переменную счетчика цикла.

```
let i = 0
```

- 2 Вставьте структуру цикла **for**, которая будет выполнять 10 итераций и при каждой итерации выводить на экран значение счетчика цикла.

```
for( i = 1 ; i < 11 ; i++ )
```



Если модификатор не позволяет в какой-то момент вернуть значение **false**, создается бесконечный цикл, т. е. цикл будет выполняться бесконечное число раз.



for.html

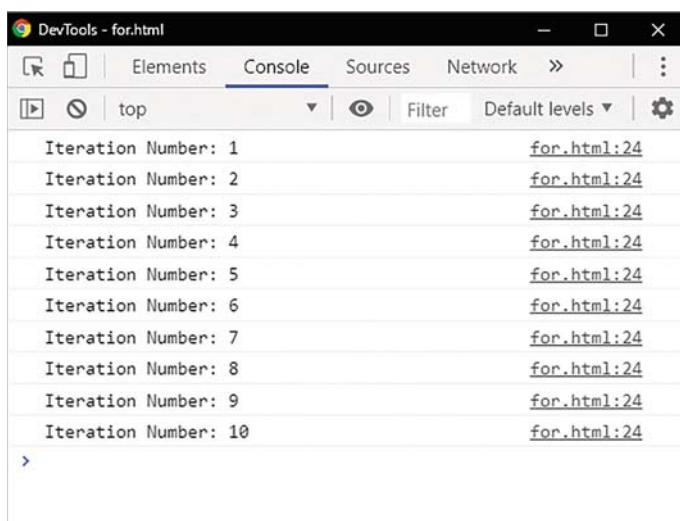


Мы рассмотрели обычный цикл `for`. Существует также специальный цикл `for in`, который используется для перебора свойств объекта. На стр. 76 вы можете найти о нем более подробную информацию.

```
{
  console.log( 'Iteration Number: ' + i )
}
```

3

Сохраните HTML-документ, затем откройте его в браузере и запустите консоль. Проанализируйте полученные результаты — итерации цикла.



Цикл while

Структура цикла `for`, описанная на стр. 000, идеальна, когда количество требуемых итераций известно. В ином случае предпочтительнее использовать структуру цикла `while`. Так выглядит его синтаксис:

```
while( условие )
{
  операторы-на-выполнение
  модификатор
}
```

В скобках после ключевого слова `while` указывается условие или выражение, логическое значение которого проверяется каждый раз перед заходом в цикл. Операторы, которые должны выполняться при



каждой итерации, заключаются в фигурные скобки вместе с оператором, который изменяет значение в условии. Поэтому в определенный момент в результате вернется значение **false**, и цикл завершится. Пока результат равен **true**, операторы будут выполняться на каждой итерации цикла.

Если на первой итерации условие принимает значение **false**, цикл немедленно завершается, поэтому операторы, указанные в фигурных скобках, никогда не выполняются. Циклы **while** и циклы **for** иногда называют циклами предварительного тестирования, поскольку их тестовое условие проверяется до выполнения каких-либо операторов.

Цикл **while** может быть выполнен для определенного количества итераций, аналогично циклу **for**, используя переменную счетчика в качестве условия проверки и увеличения ее значения на каждой итерации. Например, структура цикла **while** для выполнения набора операторов 100 раз может выглядеть следующим образом:

```
let i = 0
while ( i < 100 )
{
    операторы-на-выполнение
    i++;
}
```

На каждой итерации переменная счетчика увеличивается до тех пор, пока значение не достигнет **100**, после чего результат будет равен **false**, и цикл завершится.

- 1 Создайте HTML-документ с самовызывающейся функцией, объявите и проинициализируйте переменную счетчика цикла.

```
let i = 10
```

- 2 Вставьте структуру цикла **while**, которая будет выполнять итерации и при каждой итерации выводить на экран значение счетчика цикла, пока не достигнет значения 0.



При отсутствии модификатора в структуре цикла **while** возникнет бесконечный цикл, т. е. цикл будет выполняться бесконечное число раз.



while.html

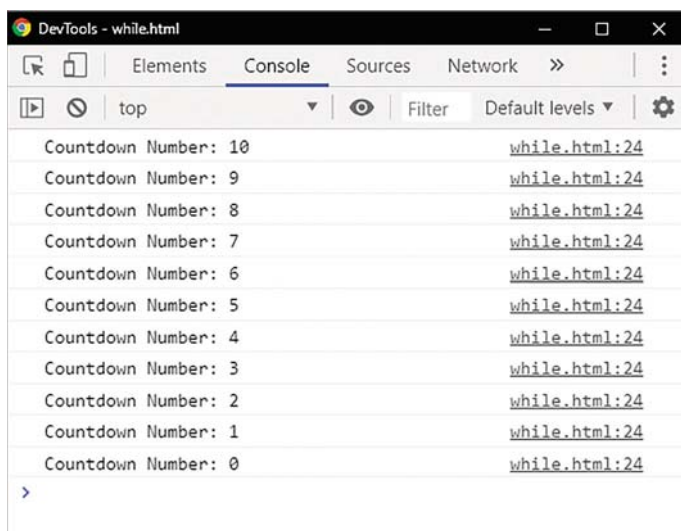


Каждый цикл `while` необходимо заключать в фигурные скобки, так как содержится как минимум два оператора — один выполняемый оператор и модификатор.

```
while( i > -1 )
{
    console.log( 'Countdown Number: ' + i )
    i--
}
```

3

Сохраните HTML-документ, затем откройте его в браузере и запустите консоль. Проанализируйте полученные результаты.



Цикл `do..while`



Другой пример циклической структуры в JavaScript — это цикл `do..while`. Он похож на инвертированную версию цикла `while`, описанную на стр. 62–63. Цикл `do..while` используется тогда, когда известно, что цикл необходимо запустить хотя бы один раз. Его синтаксис выглядит так:

```
do
{
    операторы-на-выполнение
    модификатор
}
while( условие )
```

В скобках после ключевого слова **while** указывается условие или выражение, логическое значение которого проверяется каждый раз после захода в цикл. Операторы, которые должны выполняться при каждой итерации, заключаются в фигурные скобки вместе с оператором, который изменяет значение в условии. Поэтому в определенный момент в результате вернется значение **false**, и цикл завершится. Пока результат равен **true**, операторы будут выполняться на каждой итерации цикла.



Если на первой итерации условие принимает значение **false**, цикл немедленно завершается, поэтому операторы, указанные в фигурных скобках, выполняются только один раз. Цикл **do..while** иногда называют циклом с постусловием, так как в отличие от предыдущих циклов он вначале исполняет команды, а затем проверяет заданное условие.

Цикл **do..while** рекомендуется использовать только в том случае, если операторы необходимо выполнить хотя бы один раз.

Цикл **do..while** можно принудительно заставить выполнять итерации определенное количество раз, аналогично циклу **for**, используя переменную счетчика в качестве условия проверки и увеличивая ее значение на каждой итерации. Например, структура цикла **do while** для выполнения набора операторов 100 раз может выглядеть следующим образом:

```
let i = 0
do
{
  операторы-на-выполнение
  i++
}
while ( i < 100 )
```

Переменная счетчика увеличивается на каждой итерации до тех пор, пока ее значение не достигнет **100**, после чего в результате вернется значение **false**, и цикл завершится.

1

Создайте HTML-документ с самовызывающейся функцией. Объявите и проинициализируйте переменную счетчика цикла.

```
let i = 2
```



do.html

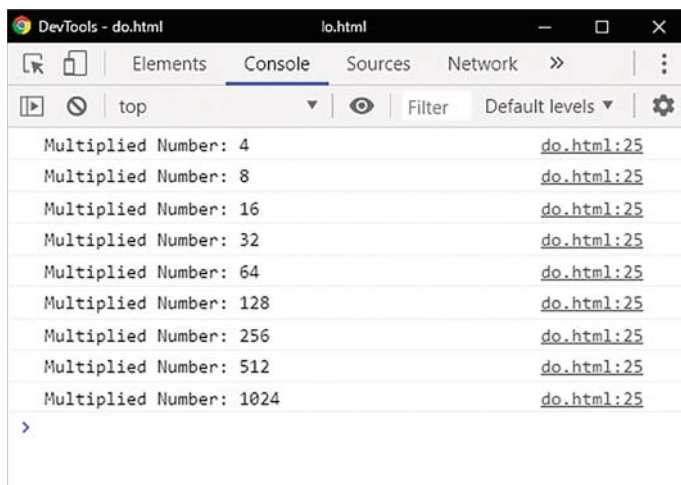


Обратите внимание, что окончательное значение превышает предельное значение условия, так как оно записывается на выходе перед выполнением проверки.

- 2 Вставьте структуру цикла **do..while**, которая будет выполнять итерации и при каждой итерации выводить на экран значение счетчика цикла, пока не превысит значения 1000.

```
do
{
    i *= 2
    console.log( 'Multiplied Number: ' + i )
}
while( i < 1000 )
```

- 3 Сохраните HTML-документ, затем откройте его в браузере и запустите консоль. Проанализируйте полученные результаты.



Выход из циклов

Оператор **break** прерывает выполнение текущего цикла при обнаружении заданного условия. В циклах он обычно используется для немедленного выхода, когда в зависимости от каких-либо условий требуется завершить выполнение цикла.

Если оператор **break** используется в цикле, вложенном во внешний цикл, блок операторов выполняется во внешнем цикле.



Оператор **continue** прерывает выполнение итерации текущего или отмеченного цикла и продолжает его выполнение на следующей итерации.

Если оператор **continue** используется в цикле, вложенном во внешний цикл, блок операторов выполняется на следующей итерации внутреннего цикла.

- 1
- Создайте HTML-документ с самовызывающейся функцией. Объявите и проинициализируйте переменную счетчика цикла.

```
let i = 0
let j = 0
```

- 2
- Добавьте цикл **for**, содержащий внутренний вложенный цикл **for**.

```
for ( i = 1 ; i < 3 ; i++ )
{
  console.log( 'Outer Loop: ' + i )
  for ( j = 1 ; j < 4 ; j++ )
  {
    // Здесь будет ваш код.
    console.log( '\tInner Loop: ' + j )
  }
}
```

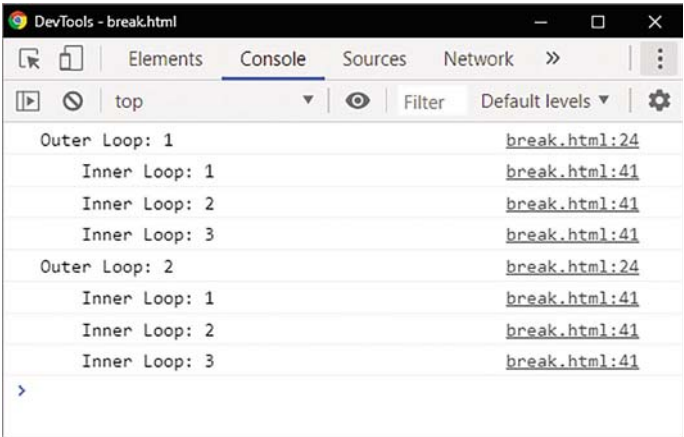
- 3
- Сохраните HTML-документ, затем откройте его в браузере и запустите консоль. Проанализируйте полученные результаты — две итерации внешнего цикла и три итерации внутреннего цикла.



break.html



В данном случае escape-последовательность `\t` используется для перемещения позиции печати к следующей позиции горизонтальной табуляции.



В контексте оператора **switch** оператор **break** обычно используется в конце каждого блока для его завершения.



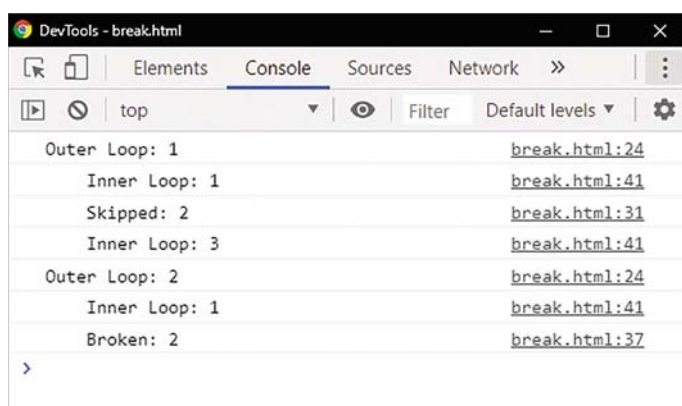
Оператор `break` удобно использовать, когда в определенной ситуации необходимо выйти из цикла.

- 4 Во внутренний цикл вставьте операторы, чтобы пропустить его итерацию и выйти из внешнего цикла.

```
if( ( i === 1 ) && ( j === 2 ) ) {
    console.log( '\tSkipped: ' + j )
    continue
}

if( ( i === 2 ) && ( j === 2 ) ) {
    console.log( '\tBroken: ' + j )
    break
}
```

- 5 Сохраните HTML-документ, затем откройте его в браузере и запустите консоль. Проанализируйте полученные результаты — выход из цикла.



Обработка ошибок



В JavaScript существует конструкция **`try..catch`**, позволяющая обрабатывать ошибки (исключения). Операторы, которые должны быть выполнены, внутри блока **`try..catch`** заключаются в фигурные скобки, а исключения передаются в качестве аргумента в следующий блок **`catch`**. Конструкция **`try..catch`** может содержать секцию **`finally`**, содержащую операторы, которые должны выполняться после обработки ошибок.

JavaScript имеет встроенный объект ошибки, который предоставляет информацию при ее возникновении. JavaScript распознает следующие ошибки: **Error**, **EvalError**, **InternalError**, **RangeError**, **ReferenceError**, **SyntaxError**, **TypeError** и **URIError**. Они могут быть автоматически созданы и переданы в блок **catch** с помощью ключевого слова **new** и конструктора, а затем переданы при использовании ключевого слова **throw**.

Каждый объект ошибки имеет два свойства: **name** и **message**, позволяющие устанавливать или возвращать сообщение об ошибке.

В качестве альтернативы для идентификации ошибки строка может быть передана в блок **catch** с помощью оператора **throw**. Вы можете генерировать ошибки (исключения). Исключения могут быть строкой, числом, логическим значением или объектом JavaScript.

1

Создайте HTML-документ с самовызывающейся функцией. Объявите и проинициализируйте переменную.

```
let day = 32
```



catch.html

2

Для распознавания недопустимых целочисленных значений вставьте блок **try**.

```
try
{
    if( day > 31 )
    {
        throw new RangeError( 'Day Cannot Exceed 31' )
    }

    if( day < 1 )
    {
        throw 'invalid'
    }
}
```

3

Для обработки недопустимых целочисленных значений к блоку **try** добавьте блок **catch**.



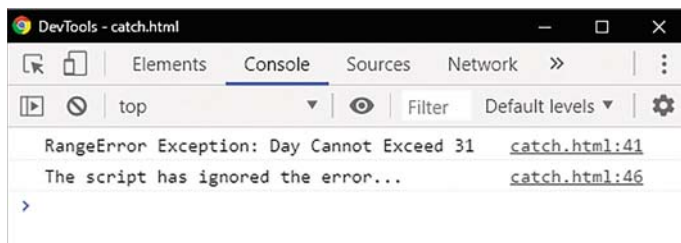
Чтобы обнаружить автоматическую ошибку `ReferenceError`, удалите или закомментируйте объявление переменной `day`, затем сохраните и обновите этот пример.

```
catch( err )
{
    if( err === 'invalid' )
    {
        console.log( 'Variable has invalid value of ' + day )
    }
    else
    {
        console.log( err.name + ' Exception: ' +
            err.message )
    }
}
```

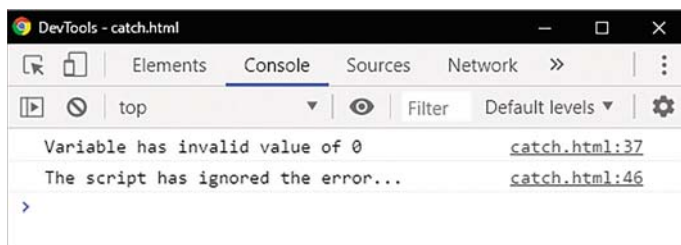
- 4 Для вывода окончательного сообщения к блоку `catch` добавьте блок `finally`.

```
finally
{
    console.log( 'The script has ignored the error...' )
}
```

- 5 Сохраните HTML-документ, затем откройте его в браузере и запустите консоль. Проанализируйте результаты — полученное сообщение об ошибке.



- 6 Чтобы увидеть обнаруженную ошибку, измените значение переменной на 0, затем сохраните HTML-документ и обновите страницу в браузере.



Заключение

- Оператор **if** проверяет условие на наличие логического значения **true** или **false**.
- Альтернативная ветвь, по которой действия могут быть выполнены при ложном результате, обозначается ключевым словом **else**.
- С помощью оператора **switch** можно сравнить выражение сразу с несколькими условиями.
- В блоке **switch** каждый блок **case** должен заканчиваться оператором **break**, чтобы прервать выполнение.
- Блок **switch** может содержать оператор **default**. Оператор **default** выполняется, если ни один вариант не совпал.
- В цикле **for** указывается инициализатор, условие, которое должно быть проверено на логическое значение **true** или **false**, и модификатор.
- Модификатор — это оператор, который изменяет значение в условии, поэтому в какой-то момент в результате вернется значение **false**, и происходит выход из цикла.
- Циклы **while** и **for** проверяют условие перед выполнением блока операторов.
- Цикл **do..while** проверяет условие после того, как блок операторов был выполнен.
- Оператор **break** прерывает выполнение текущего цикла.
- Оператор **continue** прерывает выполнение текущей итерации в цикле и продолжает на следующей итерации.
- Структура **try..catch** используется для обработки возникающих в сценарии ошибок (исключений).

- Каждый объект ошибки имеет два свойства: **name** и **message**, позволяющие устанавливать или возвращать сообщение об ошибке.
- В качестве альтернативы для идентификации ошибки строка может быть передана в блок **catch** с помощью оператора **throw**.
- Конструкция **try..catch** может содержать секцию **finally** с операторами, которые должны выполняться после обработки ошибок.

4

Управление объектами

*В этой главе вы научитесь
создавать объекты
и использовать встроенные
объекты JavaScript.*

- 72 Пользовательские объекты
- 74 Расширенные функции
- 76 Встроенные объекты
- 79 Создание массивов
- 81 Обход элементов в цикле
- 84 Методы управления элементами в массиве
- 86 Сортировка элементов массива
- 88 Получение даты
- 90 Получение календаря
- 93 Получение времени
- 95 Установка даты и времени
- 97 Сопоставление текста с шаблоном
- 100 Заключение



Пользовательские объекты

Объекты реального мира окружают нас повсюду, и каждый из них имеет атрибуты и модель поведения, которые мы можем описать:

- Атрибуты — это особенности, характеризующие объект.
- Поведение — это действия, которые могут выполняться над объектом или которые может выполнять сам объект.

Например, объект автомобиль можно описать с помощью атрибутов «марка» и «модель», а также модель поведения «ускорение» и «торможение».



Эти элементы могут быть представлены в JavaScript с помощью пользовательского объекта **car**, содержащего свойства **make** и **model**, а также методы **accelerate()** и **brake()**.

Значения присваиваются объекту в виде разделенного запятыми списка пар **name: value** (имя: значение), заключенного в фигурные скобки {}, например:

```
let car = { make: 'Jeep', model: 'Wrangler',
  accelerate: function () { return this.model + ' drives away' },
  brake: function () { return this.make + ' pulls up' }
}
```

Доступ к свойству объекта можно получить двумя способами: с помощью точечной записи **objectName.propertyName** или **objectName['propertyName']**

Методы объекта могут быть вызваны с помощью точечной нотации **objectName.methodName()**.



Пробелы игнорируются в списке пар **name: value** (имя: значение), но не забывайте эти пары разделять запятыми.

Ключевое слово **this** относится к объекту, к которому принадлежит. В нашем примере оно относится к объекту **car**, поэтому, используя запись **this.model**, мы ссылаемся на свойство **car.model**, а **this.make** — на свойство **car.make**.

1

Создайте HTML-документ с самовызывающейся функцией. Объявите и проинициализируйте переменную, содержащую определение объекта.

```
let car = {  
  // Здесь будет ваш код.  
}
```

2

Для определения свойств объекта вставьте следующие операторы.

```
make: 'Jeep',  
model: 'Wrangler',
```

3

Для определения методов объекта вставьте следующие операторы.

```
accelerate: function () {  
  return this.model + ' drives away' },  
brake: function () {  
  return this.make + ' pulls up' }
```

4

Для вывода строки, содержащей значения свойств объекта, добавьте после символа закрывающей фигурной скобки **}** следующий оператор.

```
console.log('Car is a ' + car.make + ' ' + car['model'])
```

5

Для вызова каждого метода объекта добавьте следующие операторы.

```
console.log(car.accelerate())  
console.log(car.brake())
```

6

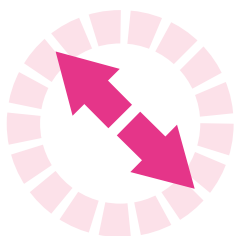
Сохраните HTML-документ, затем откройте его в браузере и запустите консоль. Проанализируйте полученные



object.html

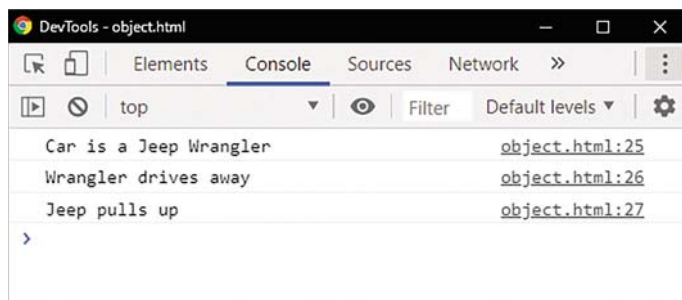


Для вызова метода необходимо добавить круглые скобки **()**. В противном случае метод вернет определение функции.



extend.html

результаты — значения свойств объекта и возвращаемые методами строки.



Расширенные функции

Пользовательские объекты могут быть в любое время легко расширены и изменены. Для этого нужно всего лишь назначить новое значение с помощью точечной нотации, например: **имяОбъекта.имяСвойства**.

Специальный цикл **for in** проходит через все перечисляемые свойства объекта, по каждому отдельному элементу и имеет следующий синтаксис:

```
for ( свойство in имяОбъекта ) { console.log( свойство ) }
```

Чтобы на каждой итерации сослаться на значение каждого свойства, его имя следует указать после имени объекта в квадратных скобках, например: **имяОбъекта[свойства]**.

1

Создайте HTML-документ с самовызывающейся функцией. Объявите и проинициализируйте переменную, которая точно воссоздает объект из предыдущего примера.

```
let property, car = {  
  make: 'Jeep' ,  
  model: 'Wrangler' ,  
  accelerate: function () {  
    return this.model + ' drives away' ;  
  brake: function () {  
    return this.make + ' pulls up' }  
  }
```

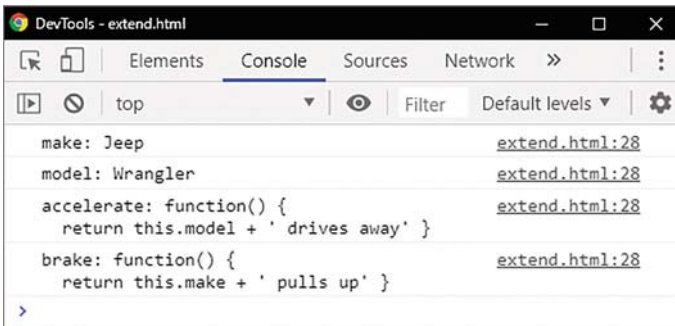
2

Добавьте оператор цикла для перечисления имен и значений каждого свойства и метода.

```
for( property in car ) {
    console.log( property + ':' + car[ property ] )
}
```

3

Сохраните HTML-документ, затем откройте его в браузере и запустите консоль. Проанализируйте полученные результаты — ключи и значения объекта.



Фактически в этом примере свойству может быть присвоено любое допустимое имя.

4

Добавьте операторы для присвоения новых значений двум существующим свойствам объекта.

```
car.make = 'Ford'
car.model = 'Bronco'
```

5

Для присвоения объекту дополнительных свойств и методов добавьте следующие операторы.

```
car.engine = 'V6'
car.start = function ( ) {
    return this.make + ' motor is running' }
```

6

В соответствии с правилами добавьте к выходным строкам, содержащим значения свойств объекта, следующие операторы.

```
console.log( '\nCar is a ' + car.make + '
                                     ' + car[ 'model' ] )
console.log( 'Engine Type: ' + car.engine )
```

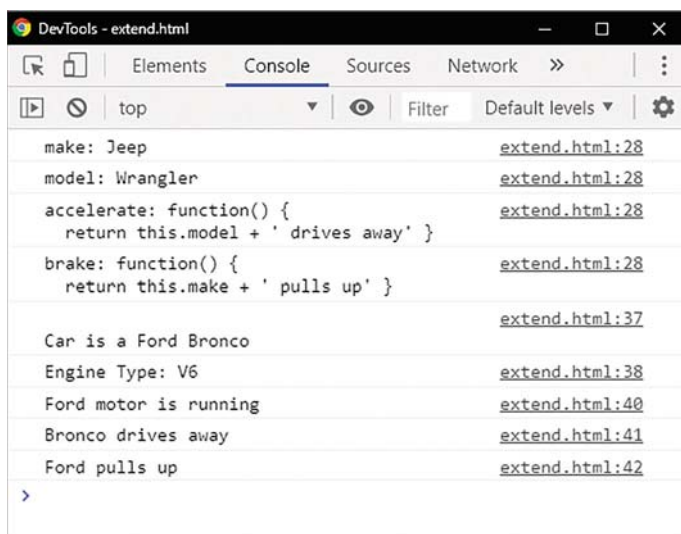


Чтобы включить в вывод новую строку (разрыв строки), в этом примере используется escape-последовательность `\n`.

- 7 Добавьте операторы для вызова каждого метода объекта.

```
console.log( car.start() )
console.log( car.accelerate() )
console.log( car.brake() )
```

- 8 Сохраните HTML-документ, затем откройте его в браузере и запустите консоль. Проанализируйте полученные результаты — значения свойств расширенного объекта и возвращаемые его методами строки.



Встроенные объекты



В таблице ниже перечислены встроенные стандартные объекты JavaScript. Каждый из них, за исключением объекта **Math**, имеет одноименный конструктор, который можно использовать для создания объекта с помощью ключевого слова **new**. Например, создание объекта **Date** выглядит так: **let now=new Date()**.

Однако не рекомендуется для всех типов объектов использовать ключевое слово **new** и конструктор, кроме объектов **Error** и **Date**. JavaScript распознает,

какой тип объекта может быть создан с помощью присвоенного значения, только если это значение не строка **string**, число **number** или логическое значение **boolean**. Такие типы данных называются примитивами. Они не имеют свойств или методов, поэтому оператор **typeof** возвращает для них значения **string**, **number** и **boolean**, а не **object**.

Все объекты в JavaScript наследуют свойства и методы из прототипа объекта высокого уровня **Object.prototype**. Например, с объектом более низкого уровня, таким как **Date.prototype**, это осуществляется методом **toLocaleString()**, поэтому, чтобы получить строку даты в локальном формате, вы можете к объекту **Date** добавить вызов метода **toLocaleString()**.

Также можно вызывать унаследованные методы для примитивных строковых (**string**), числовых (**number**) и логических (**boolean**) значений, так как JavaScript автоматически вызывает метод из эквивалентного объекта. Значит, чтобы получить числовую строку в локальном формате, вы можете к числовому литералу добавить вызов метода **toLocaleString()**.



Если вы с самого начала работаете с этой книгой, то уже должны были изучить объекты **String**, **Number**, **Boolean**, **Error** и **Object**. Объект **Array** описан на стр. 81, объект **Date** — на стр. 90, объект **RegExp** — на стр. 99, а объект **Math** — на стр. 106.

Объект

String (создается с помощью ключевого слова **new**)

Number (создается с помощью ключевого слова **new**)

Boolean (создается с помощью ключевого слова **new**)

Object — определенный вами объект

Date — используется для работы с датой и временем

Array — используется для хранения упорядоченных коллекций данных

RegExp — создает объект регулярного выражения для сопоставления текста с шаблоном

Math — встроенный объект, хранящий в своих свойствах и методах различные математические константы и функции

Error — создает объект ошибки



extend.html



Объект `Math` — единственный объект JavaScript, который не является конструктором.

- 1 Создайте HTML-документ с самовызывающейся функцией. Объявите и проинициализируйте три переменные и присвойте им примитивные литеральные значения.

```
let jsString = 'Text' // неверно — new String( 'Text' ).
```

```
let jsNumber = 125000 // неверно — new
                        Number( 125000 ).
```

```
let jsBoolean = true // неверно — new
                      Boolean( true ).
```

- 2 Для создания объектов присвойте следующие значения.

```
let jsObject = {firstName: 'Mike' , lastName: 'McGrath' }
let jsDate = new Date( )
let jsArray = [ 1, 2, 3 ]
let jsRegExp = /ineasysteps/i
let jsMath = Math
let jsError = new Error( 'Error!' )
```

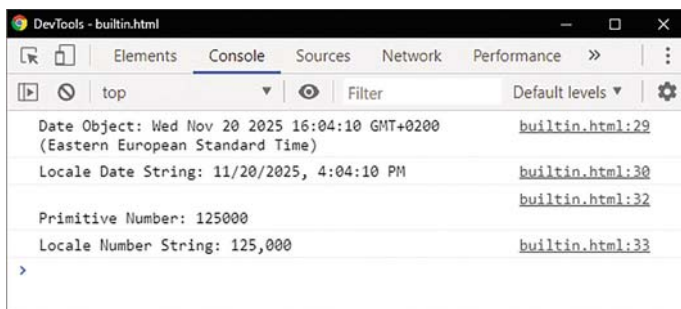
- 3 Добавьте операторы для вывода содержимого объекта `Date` и строки в локальном формате.

```
console.log( 'Date Object: ' + jsDate )
console.log( 'Locale Date String: ' +
            jsDate.toLocaleString( ) )
```

- 4 Добавьте операторы для вывода примитивного числового значения и числовой строки в локальном формате.

```
console.log( '\nPrimitive Number: ' + jsNumber )
console.log( 'Locale Number String: ' +
            jsNumber.toLocaleString( ) )
```

- 5 Сохраните HTML-документ, затем откройте его в браузере и запустите консоль. Проанализируйте полученные результаты.



Создание массивов

Объект **Array** — это встроенный объект JavaScript, который используется для хранения упорядоченных коллекций данных (с различными типами данных). Объекты массивов создаются путем присвоения переменным литеральных значений, указанных в квадратных скобках `[]` и разделенных запятыми. Синтаксис объекта **Array** выглядит следующим образом:

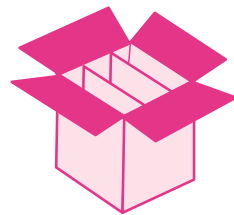
```
let имя массива = [ значение1 , значение2 , значение3 ]
```

В отличие от пользовательских объектов, где каждое свойство содержит имя, элементы массива нумеруются автоматически, начиная с нуля. Первому элементу присваивается 0, второму — 1, третьему — 2 и т.д. Эту систему нумерации часто называют «индексом с отсчетом от нуля».

Доступ к элементам массива осуществляется с помощью оператора `[]`. Например, `colors[0]` ссылается на значение в первом элементе массива с именем «colors».

Можно создать пустой массив и позже присвоить значения его элементам, например:

```
let colors = []
colors[0] = 'Red'
colors[1] = 'Green'
colors[2] = 'Blue'
```





Все имена встроенных объектов начинаются с символа верхнего регистра, поэтому следует различать «Array» и «array».



array.html

Можно создать массив, используя конструктор `Array()`. Однако такой способ использовать нежелательно, так как это может привести к неожиданным результатам. Например, создание массива, инициализирующего первый элемент:

```
let jsArray = new Array( 10 )
```

Конечно, вы можете ожидать, что `jsArray[0]` будет ссылаться на целочисленное значение 10 первого элемента, но на самом деле он возвращает **неопределенное** значение. Почему? Это аномалия, которая возникает только тогда, когда вы указываете один целочисленный аргумент для конструктора, что заставляет JavaScript создавать массив из 10 пустых элементов! Создание массива с помощью `jsArray = [10]` не дает такого эффекта и создает массив с первым элементом, содержащим целочисленное значение 10, как и предполагалось.

- 1 Создайте HTML-документ с самовызывающейся функцией, которая начинается с создания массива — неверно.

```
let jsArray = new Array( 10 )
```

- 2 Для вывода значения первого элемента массива и перечисления массива добавьте следующие операторы.

```
console.log( jsArray[ 0 ] )  
console.log( jsArray )
```

- 3 Для объявления переменной и переменной, инициализированной массивом, добавьте следующие операторы.

```
let month, summer = [ 'June', 'July', 'August' ]
```

- 4 Добавьте циклическую структуру для вывода номера индекса и значения каждого элемента.

```
for ( month in summer )
{
    if ( month !== '' )
    {
        console.log( month + ': ' + summer[ month ] )
    }
}
```

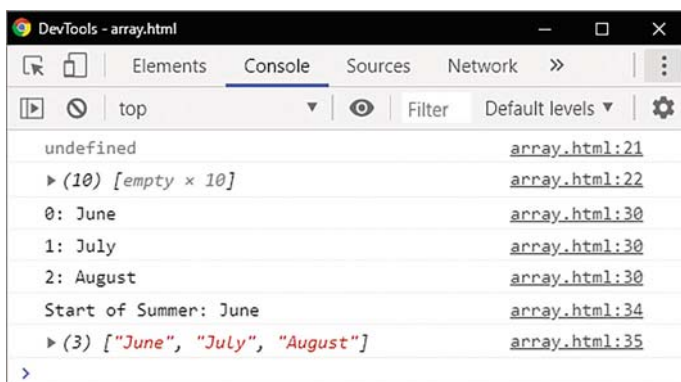


Рекомендуется заключать тело цикла `for in` в оператор `if`. Это гарантирует, что элемент не пустой.

- 5 Наконец, для вывода значения первого элемента массива и перечисления массива добавьте следующие операторы.

```
console.log( 'Start of Summer: ' + summer[ 0 ] )
console.log( summer )
```

- 6 Сохраните HTML-документ, затем откройте его в браузере и запустите консоль. Проанализируйте полученные результаты — вывод содержимого элементов массива.



Обход элементов в цикле

Как правило, при работе с массивами используются циклы. Для заполнения элементов массива значениями можно использовать любой из них. Таким образом можно заполнить элементы даже очень больших массивов, используя при этом удивительно простой код.

Также их используют для быстрого считывания значений каждого элемента массива и выполнения каких-либо действий с этим значением.





elements.html



Длина этого массива — 11, так как он состоит из 11 элементов, даже если нулевой элемент пустой.

Чтобы узнать длину массива, необходимо обратиться к его свойству **length**. В результате индексирования с отсчетом от нуля длина массива всегда будет на единицу больше, чем номер индекса последнего элемента, поэтому его можно легко использовать в условии для завершения цикла.

- 1 Создайте HTML-документ с самовызывающейся функцией. Объявите и проинициализируйте три переменные.

```
let i, result, boolArray = [ ]
```

- 2 Выведите простой заголовок.

```
console.log( 'Fill Elements...'
```

- 3 Добавьте цикл для заполнения 10 элементов массива логическими значениями и вывода их порядковых номеров и каждого сохраненного значения.

```
for( i = 1; i < 11; i++ )
{
    boolArray[ i ] = ( i % 2 === 0 ) ? true : false
    console.log( 'Element ' + i + ': ' + boolArray[ i ] )
}
```

- 4 Выведите еще один заголовок и присвойте переменной пустое строковое значение.

```
console.log( 'Read Elements...' )
result = ''
```

- 5 Добавьте цикл для присвоения индексных номеров любых элементов, содержащих значение **true**.

```
for( i = 1 ; i < boolArray.length ; i++ )
{
    if( boolArray[ i ] ) { result += i + ' | ' }
}
```

- 6 Выведите строку, чтобы отобразить порядковые номера элементов, содержащих значение **true**.

```
console.log( 'True in Elements: ' + result )
```

7

Присвойте строковой переменной пустое значение.

```
result = "
```

8

Добавьте цикл для присвоения индексных номеров любых элементов, содержащих значение **false**.

```
for( i = 1 ; i < boolArray.length ; i++ )
{
    if( !boolArray[ i ] ) { result += i + ' | ' }
}
```

9

Наконец, выведите строку, чтобы отобразить порядковые номера элементов, содержащих значение **false**.

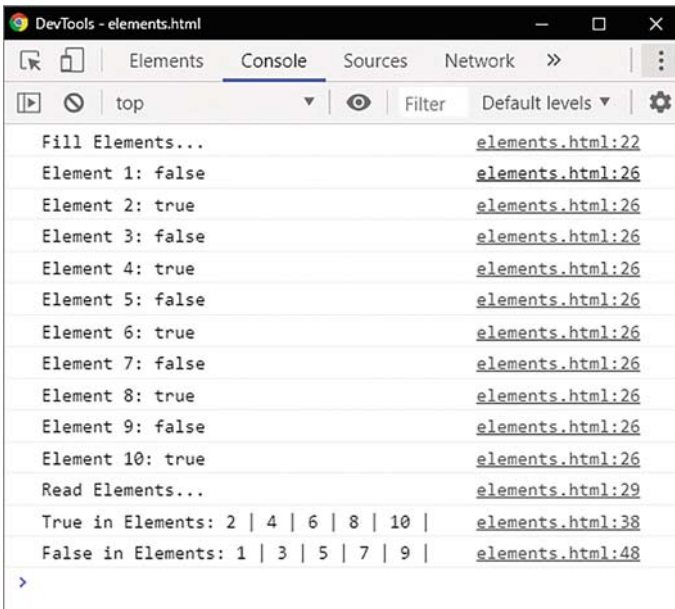
```
console.log( 'False in Elements: ' + result )
```

10

Сохраните HTML-документ, затем откройте его в браузере и запустите консоль. Проанализируйте полученные результаты — запись и чтение элементов массива.



Условия для логического значения не должны включать инструкцию `===true`, так как цикл выполняется автоматически.





Метод `join()` объединяет множество элементов в одну строку. Оператор `+` объединяет несколько значений элементов.



Метод `slice()` возвращает новый массив, который содержит копии элементов, вырезанные из исходного массива.

Методы управления элементами в массиве

Объекты JavaScript имеют свойства и методы. В дополнение к свойству `length` у каждого объекта массива имеются методы, которые можно использовать для управления элементами в массиве. Ниже в таблице перечислены такие методы и их краткое описание.

Метод	Описание
<code>join(separator)</code>	Объединяет все элементы массива в одну строку и разделяет их запятыми.
<code>pop()</code>	Удаляет последний элемент из массива и возвращает его значение.
<code>push(value , value)</code>	Добавляет один и более элементов в конец массива и возвращает новую длину массива.
<code>reverse()</code>	Обращает порядок следования элементов массива. Первый элемент становится последним, а последний — первым.
<code>shift()</code>	Удаляет первый элемент из массива и возвращает его значение.
<code>slice(begin , end)</code>	Возвращает новый массив, содержащий копию части исходного массива.
<code>sort()</code>	Сортирует элементы массива и возвращает отсортированный массив.
<code>splice(position , number , value , value)</code>	Изменяет содержимое массива, удаляя существующие элементы и/или добавляя новые.
<code>unshift(value , value)</code>	Добавляет один и более элементов в начало массива и возвращает новую длину массива.

Если значения не определены методами `push()` или `unshift()`, в массив добавляется один пустой элемент. Чтобы изменить несколько элементов, список значений, разделенных запятыми, может быть указан в методах `push()`, `unshift()` и `splice()`.

1

Создайте HTML-документ с самовывзывающейся функцией и объявите массив.

```
let seasons = [ 'Spring', 'Summer', 'Fall', 'Winter' ]
console.log( 'Elements: ' + seasons )
```



slice.html

2

Выведите измененный список элементов.

```
console.log( 'Joined: ' + seasons.join( ' & ' ) )
```

3

Извлеките последний элемент из массива.

```
console.log( 'Popped: ' + seasons.pop( ) )
console.log( 'Elements: ' + seasons )
```



4

Верните последний элемент обратно в массив.

```
console.log( 'Pushed: ' + seasons.push( 'Winter' ) )
console.log( 'Elements: ' + seasons )
```

5

Затем выведите только два значения элемента.

```
console.log( 'Sliced: ' + seasons.slice( 1, 3 ) )
```

6

Замените значение в третьем элементе.

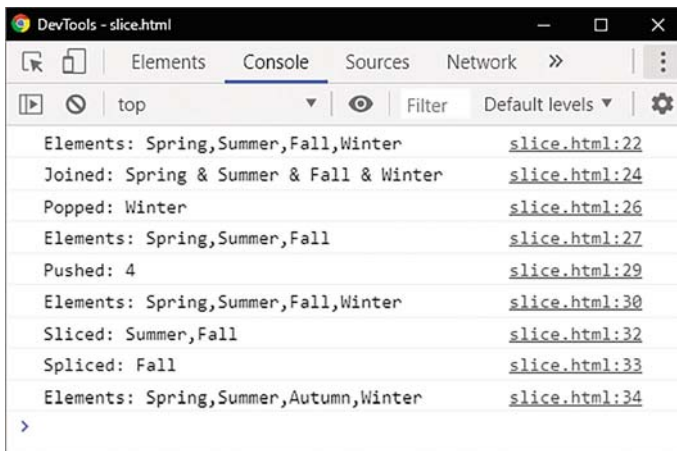
```
console.log( 'Spliced: ' + seasons.splice( 2, 1, 'Autumn' ) )
console.log( 'Elements: ' + seasons )
```

7

Сохраните HTML-документ, затем откройте его в браузере и запустите консоль. Проанализируйте полученные результаты.

Чтобы удалить указанное количество элементов в указанной позиции, используйте метод `slice()` без значения замены. Нумерация всех оставшихся элементов, которые следуют в этом массиве, автоматически изменится.

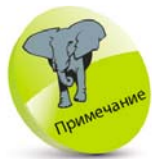
87



Методы `shift()` и `unshift()` работают аналогично методам `pop()` и `push()`, но с первым элементом, а не с последним. Методы `reverse()` и `sort()` рассматриваются в следующем примере на стр. 88–89.

Сортировка элементов массива

Иногда возникает необходимость расположить значения элементов массива в определенном порядке. Метод **sort()** сортирует элементы массива и возвращает отсортированный массив. При необходимости можно указать аргумент функции, определяющий порядок сортировки.



Всегда помните, что при использовании метода **sort()** меняется порядок значений, хранящихся в элементах массива.

Если функция сравнения не указана, элементы сортируются путем преобразования их в строки и сравнения строк в порядке следования кодовых точек Unicode, сравнивая каждый первый символ, затем каждый второй символ и т.д. Если элементы содержат совпадающие строки, различающиеся только регистром символов, строка с наибольшим количеством символов верхнего регистра получает нижнюю позицию индекса, а перед ней появляется строка с меньшим количеством символов верхнего регистра.

Такой вид сортировки подходит для строковых значений, но совершенно не годится для числовых. Например, при сортировке трех значений 30, 100, 20 результат будет 100, 20, 30, так как первые символы разные, значения сортируются только при их сравнении. Как правило, необходимо, чтобы все числовые значения были отсортированы в порядке возрастания или убывания. Поэтому для определения порядка сортировки в методе **sort()** необходимо указать функцию сравнения.



Работа метода **sort()** по умолчанию эквивалентна функции сравнения, сравнивающей аргументы **x** и **y** со следующими операторами:

```
if ( x > y ) return 1
else if ( x < y ) return -1 else
return 0.
```

Если функция сравнения указана, элементы массива будут сортироваться в соответствии с ее возвращенным значением. Если первое значение больше второго, то вернется значение больше 0, и первому значению будет присвоена более высокая позиция индекса. И наоборот, функция сравнения возвращает значение меньше 0, а первому значению присваивается более низкая позиция индекса. Если оба значения равны, то в результате возвращается 0, и позиции элементов остаются неизменными. Когда

все сравнения будут выполнены, элементы расположатся в порядке возрастания значений. При необходимости сортировки элементов по убыванию можно использовать метод `reverse()`. Он обращает порядок следования элементов массива.

Если функция сравнения сравнивает числовые значения, то для получения необходимого результата следует вернуть результат вычитания второго и первого переданного значения.

1

Создайте HTML-документ с самовызывающейся функцией и объявите два массива.

```
let hues = [ 'Red', 'RED', 'red', 'Green', 'Blue' ]  
let nums = [ 1, 20, 3, 17, 14, 0.5 ]
```



slice.html

2

Выведите значения каждого массива и значения их элементов, отсортированных по ключу.

```
console.log( 'Colors: ' + hues )  
console.log( 'Dictionary Sort: ' + hues.sort( ) )  
console.log( '\nNumbers: ' + nums )  
console.log( 'Dictionary Sort: ' + nums.sort( ) )
```

3

Добавьте оператор для вывода числовых значений, отсортированных после вызова функции сравнения.

```
console.log( 'Numerical Sort: ' + nums.sort( sortNums ) )
```

4

Для вывода числовых значений в порядке убывания добавьте в функциональном блоке следующий оператор.

```
console.log( 'Reversed Sort: ' + nums.reverse( ) )
```

5

Добавьте функцию сравнения.

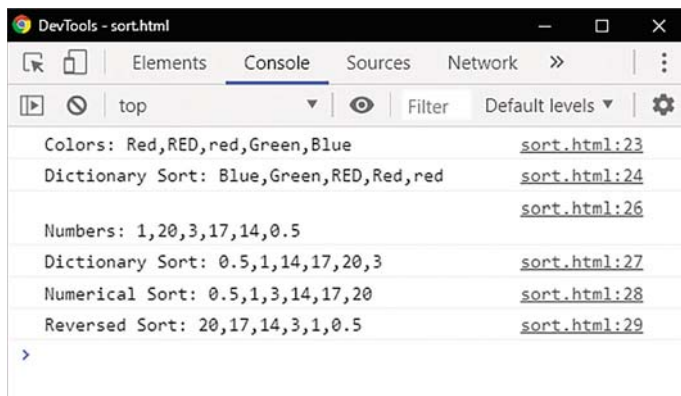
```
function sortNums( numOne, numTwo ) {  
  return numOne - numTwo  
}
```



В методе `sort()` функция сравнения указывается только по имени. Не используйте закрывающиеся скобки после имени функции сравнения в аргументе метода `sort()`.

6

Сохраните HTML-документ, затем откройте его в браузере и запустите консоль. Проанализируйте полученные результаты — отсортированные элементы.



Получение даты

Встроенный объект **Date** представляет конкретную дату, время и часовой пояс. Он создается с помощью оператора **new**, конструктора объекта **Date()** и присвоения имени переменной. Конструктор **Date()**, вызванный без аргументов, создаст объект **Date** со значением, которое будет соответствовать текущей дате и времени. Объект **Date** поддерживает ряд UTC (универсальных) методов, а также методов локального времени. UTC, также известный как среднее время по Гринвичу (GMT), относится к времени, установленному мировым стандартом времени.

Дата и время в JavaScript в основном определяют-ся как количество миллисекунд, прошедших с 1 января 1970 года 00:00:00 GMT, то есть прошедшего с начала эпохи UNIX. Это число получается из объекта **Date** с помощью метода **getTime()** и может быть вычтено из числа другого объекта **Date** для вычисления прошедшего периода между двумя точками сценария. Например, для вычисления периода, необходимого для выполнения цикла.

Строковое представление объекта **Date** можно получить с помощью методов **toString()** или **toUTCString()**, преобразующего дату в строку с использованием часового пояса UTC.

С помощью метода **getTimezoneOffset()** JavaScript может определить, в каком часовом поясе находится пользователь при условии, что в системе корректно настроен локальный часовой пояс. Этот метод возвращает смещение часового пояса относительно часового пояса UTC в минутах для текущей локали. Расчет выполняется в минутах, а не часах, так как некоторые часовые пояса смещены с интервалом, отличным от одного часа. Например, Ньюфаундленд, Канада — UTC -3:30 (UTC -2:30 в летнее время).

Значение смещения часового пояса можно использовать для локальных настроек часовых поясов США, но для перехода на летнее время их необходимо скорректировать посредством вычитания 60 (минут). Далее в примере демонстрируется вызов методов **getMonth()** и **getDate()** объекта **Date** для корректировки значения смещения часового пояса, если летнее время не действует в текущую дату.

1

Создайте HTML-документ с самовызывающейся функцией. Объявите и проинициализируйте три переменные.

```
const now = new Date( )
let offset = now.getTimezoneOffset( )
let dst = 60
```

2

Добавьте операторы для отключения летнего времени с 3 ноября по 10 марта.

```
if( ( now.getMonth( ) < 3 ) && ( now.getDate( ) < 10 ) )
{ dst = 0 }
if( ( now.getMonth( ) > 9 ) && ( now.getDate( ) > 2 ) )
{ dst = 0 }
```

3

Добавьте операторы, чтобы установить часовой пояс.



На стр. 92–98 рассмотрены приемы, демонстрирующие, как использовать компоненты объекта **Date** и его распространенные методы.



date.html



Метод **getMonth()** возвращает месяц и дату по местному времени. Нумерация месяцев начинается с нуля. В нашем примере март находится на позиции 2. Подробнее об этом см. стр. 93.



Локальный часовой пояс пользователя можно использовать для перенаправления браузера на страницу, относящуюся к этому часовому поясу. Например, пользователей в часовом поясе Тихого океана можно перенаправить на страницу, содержащую только дистрибуторов из Калифорнии. Однако следует учитывать, что информация о системном времени может быть легко изменена пользователем на любое время, дату или часовой пояс, поэтому не обязательно сообщать фактическое местоположение.

```
switch( offset )
{
  case ( 300 - dst ) : offset = 'East Coast' ; break
  case ( 360 - dst ) : offset = 'Central' ; break
  case ( 420 - dst ) : offset = 'Mountain' ; break
  case ( 480 - dst ) : offset = 'Pacific' ; break
  default : offset = 'All'
}
```

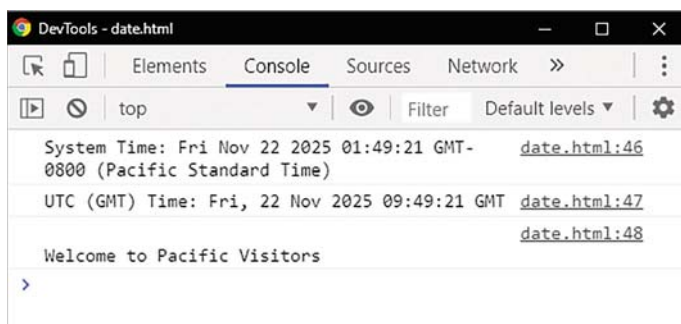
4

Добавьте операторы для вывода информации о дате и времени и приветственное сообщение.

```
console.log( 'System Time: ' + now.toString() )
console.log( 'UTC (GMT) Time: ' + now.toUTCString() )
console.log( '\nWelcome to ' + offset + ' Visitors' )
```

5

Сохраните HTML-документ, затем откройте его в браузере и запустите консоль. Проанализируйте полученные результаты.



Получение календаря



Объект **Date** предоставляет отдельные методы для получения следующих значений: год, месяц, день месяца и день недели.

Метод **getFullYear()** объекта **Date** возвращает год в виде четырехзначного числа, например **2025**. Метод **getDate()** объекта **Date** возвращает день месяца. В первый день месяца возвращается число **1**.

Метод	Описание
<code>getFullYear()</code>	Возвращает год указанной даты по местному времени (yyyy)
<code>getMonth()</code>	Возвращает месяц указанной даты по местному времени (0–11)
<code>getDate()</code>	Возвращает день указанной даты по местному времени (1–31)
<code>getDay()</code>	Возвращает день недели указанной даты по местному времени (0–6).

В зависимости от региональных настроек методы `getMonth()` и `getDay()` возвращают числовые значения индекса, которые необходимо преобразовать в названия месяца и дня на местном языке. Такое преобразование легко выполнить для месяцев с помощью создания массива всех названий месяцев, начиная с января, а затем используя номер индекса, возвращаемого `getMonth()`, для ссылки на соответствующее название месяца из элемента массива.

Аналогичным образом можно выполнить преобразование для дней недели с помощью создания массива названий всех дней недели, начиная с воскресенья, а затем с использованием номера индекса, возвращаемого методом `getDay()`, для ссылки на соответствующее название дня из элемента массива.

Затем различные компоненты могут быть объединены в строку даты, упорядоченную в соответствии с необходимым форматом даты для любой локали.

1

Создайте HTML-документ с самовызывающейся функцией. Объявите и проинициализируйте три переменные.

```
const days = [ 'Sun', 'Mon', 'Tue', 'Wed', 'Thu', 'Fri', 'Sat' ]
const months = [ 'Jan', 'Feb', 'Mar', 'Apr', 'May', 'Jun', 'Jul',
'Aug', 'Sep', 'Oct', 'Nov', 'Dec' ]
const now = new Date( )
```

2

Добавьте операторы для извлечения отдельных компонентов даты с помощью методов объекта `Date`.



calendar.html



Нумерация месяцев начинается с нуля (0), а не с единицы (1) — так, например, месяцу март присвоен индекс [2], а не [3].

```
let year = now.getFullYear( )
let month = now.getMonth( )
let dayNumber = now.getDate( )
let dayName = now.getDay( )
```

- 3 Добавьте операторы для преобразования извлеченных номеров индексов в значения названий месяца и дня недели.

```
month = months[ month ]
dayName = days[ dayName ]
```

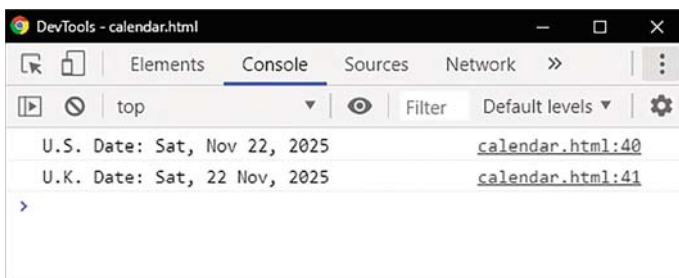
- 4 Объедините компоненты даты в строки даты — в американском и британском форматах.

```
let usDate = dayName + ', ' + month + ' ' +
              dayNumber + ', ' + year
let ukDate = dayName + ', ' +
              dayNumber + ' ' + month + ', ' + year
```

- 5 Для вывода каждой строки даты добавьте следующие операторы.

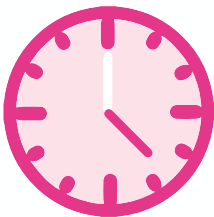
```
console.log( 'U.S. Date: ' + usDate )
console.log( 'U.K. Date: ' + ukDate )
```

- 6 Сохраните HTML-документ, затем откройте его в браузере и запустите консоль. Проанализируйте полученные результаты — форматы записи даты.



Получение времени

Объект JavaScript **Date** предоставляет отдельные методы для получения следующих значений: часы, минуты, секунды и миллисекунды.



Метод	Описание
<code>getHours()</code>	Возвращает часы даты по местному времени (0–23)
<code>getMinutes()</code>	Возвращает минуты даты по местному времени (0–59)
<code>getSeconds()</code>	Возвращает секунды даты по местному времени (0–59)
<code>getMilliseconds()</code>	Возвращает миллисекунды даты по местному времени (0–999)

Метод `getHours()` объекта **Date** возвращает часы даты по местному времени в 24-часовом формате в виде значения в диапазоне 0–23. Методы `getMinutes()` и `getSeconds()` возвращают значение в диапазоне 0–59. Метод `getMilliseconds()` возвращает значение в диапазоне 0–999.

Значения могут быть объединены в строку, однако для улучшения читабельности возникает необходимость добавлять ноль к значениям минут и секунд. Например, 10:05:02 предпочтительнее 10:5:2.

В зависимости от времени суток (утро, день или вечер) можно добавить аббревиатуры. В случае 12-часового формата времени используются сокращения AM или PM. Например, 13:00 можно преобразовать в 1:00 PM.

- 1
- Создайте HTML-документ с самовызывающейся функцией. Объявите и проинициализируйте пять переменных.

```
const now = new Date( )
let hour = now.getHours( )
let minute = now.getMinutes( )
let second = now.getSeconds( )
let millisecond = now.getMilliseconds( )
```



Значения времени основаны на системном времени пользователя, которое может быть неточным.



time.html



Объект `Date` также предоставляет методы для получения даты и времени по всемирному координированному времени (UTC) — например, методы `getUTCMonth()` и `getUTCHours()`.

- Для удобочитаемости с помощью следующих операторов добавьте 0 в значениях минут и секунд, значение которых меньше 10.

```
if( minute < 10 ) { minute = '0' + minute }
if( second < 10 ) { second = '0' + second }
```

- Объедините все составляющие времени в строку, затем выведите ее.

```
let time = 'It is now: ' + hour + ':' + minute + ':' +
           second + ' and ' + millisecond + ' milliseconds'
console.log( time )
```

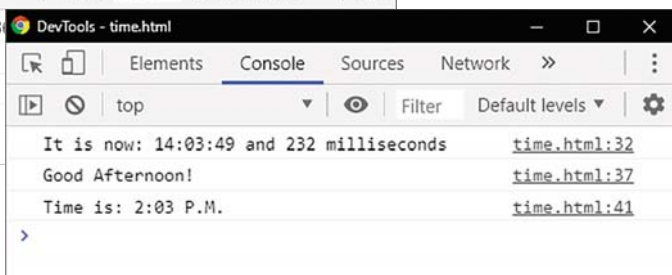
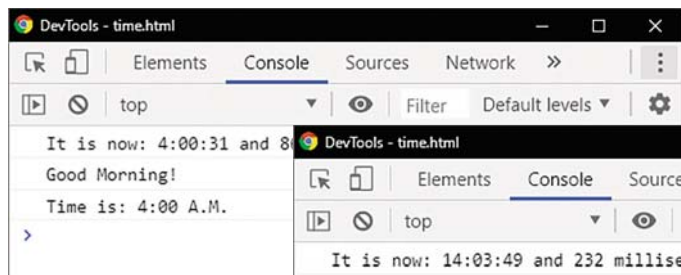
- В зависимости от времени суток выведите приветствие.

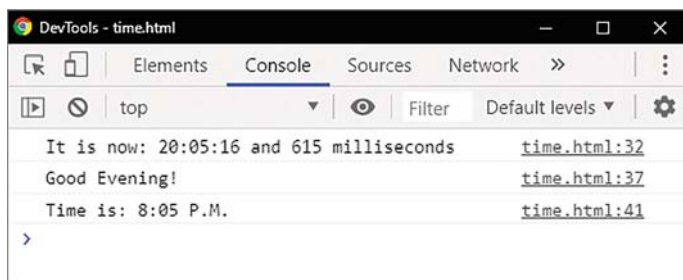
```
let greeting = 'Good Morning!'
if( hour > 11 ) { greeting = 'Good Afternoon!' }
if( hour > 17 ) { greeting = 'Good Evening!' }
console.log( greeting )
```

- Выведите время в 12-часовом формате.

```
let suffix = ( hour > 11 ) ? ' P.M.' : ' A.M.'
if( hour > 12 ) { hour -= 12 }
console.log( 'Time is: ' + hour + ':' + minute + suffix )
```

- Сохраните HTML-документ, затем откройте его в браузере и запустите консоль. Проанализируйте полученные результаты — формат записи времени.





Установка даты и времени

Конструктор **Date()** может дополнительно указывать от двух до семи аргументов для установки значений для каждого из его компонентов, например:

new Date(год, месяц, день, часы, минуты, секунды, миллисекунды)

Если указаны только год и месяц, соответствующим компонентам устанавливается единица (1), а остальным — ноль (0).

Объект **Date** также предоставляет отдельные методы для установки значений даты и времени:

Метод	Описание
<code>setDate()</code>	Устанавливает день (1–31)
<code>setFullYear()</code>	Устанавливает год (уууу)
<code>setMonth()</code>	Устанавливает месяц (0–11)
<code>setHours()</code>	Устанавливает часы (0–23)
<code>setMinutes()</code>	Устанавливает минуты (0–59)
<code>setSeconds()</code>	Устанавливает секунды (0–59)
<code>setMilliseconds()</code>	Устанавливает миллисекунды (0–999)

Метод `setMonth()` устанавливает месяц в диапазоне, где 0 = январь — 11 = декабрь. При необходимости с помощью метода `setFullYear()` можно установить месяц и день, используя следующий синтаксис:

date.setFullYear(год , числоМесяца , числоДня)



Метод `toString()` возвращает строковое значение любого объекта JavaScript.



setdate.html



Также существует метод `setTime()`, который устанавливает объект `Date` в значение времени, представленным количеством миллисекунд, прошедших с 1 января 1970 года 00:00:00 по UTC. Каждый день — 86400 000 миллисекунд, поэтому `setTime(86400000)` устанавливает дату 1 января 1970 года.

Значения каждого компонента даты и времени могут быть установлены с помощью объекта `Date`. Кроме того, существуют методы для вывода различных строк, отображающих дату и время. Метод `toString()` преобразует дату в строковое значение; метод `toUTCString()` преобразует дату в ее эквивалент в формате UTC; метод `toLocaleString()` отображает дату в зависимости от используемой локали. Методы `toDateString()` и `toTimeString()` могут отображать компоненты даты и времени.

- 1 Создайте HTML-документ с самовызывающейся функцией и создайте объект `Date` «4 июля».

```
const holiday = new Date( 2025, 6, 4 )
console.log( 'Object: ' + holiday )
```

- 2 Чтобы в полдень наступило Рождество, необходимо изменить отдельные компоненты даты. Для этого добавьте операторы.

```
holiday.setFullYear( 2028 )
holiday.setMonth( 11 )
holiday.setDate( 25 )
holiday.setHours( 12 )
holiday.setMinutes( 0 )
holiday.setSeconds( 0 )
holiday.setMilliseconds( 0 )
```

- 3 Для вывода измененной даты, времени и эквивалента в формате UTC (GMT) добавьте следующие операторы.

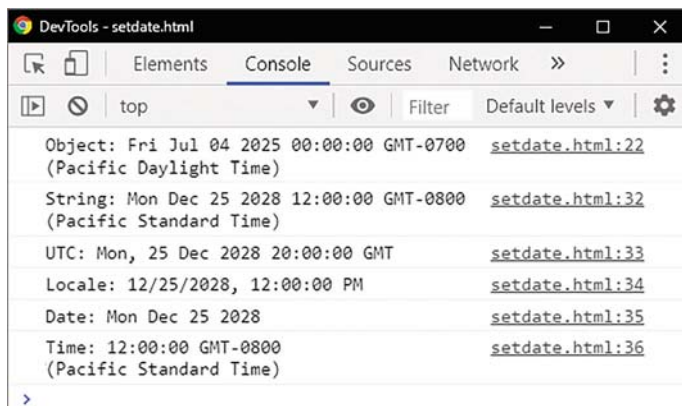
```
console.log( 'String: ' + holiday.toString() )
console.log( 'UTC: ' + holiday.toUTCString() )
```

- 4 Добавьте операторы для вывода измененной даты и времени в формате строк локали, даты и времени.

```
console.log( 'Locale: ' + holiday.toLocaleString() )
console.log( 'Date: ' + holiday.toDateString() )
console.log( 'Time: ' + holiday.toTimeString() )
```

5

Сохраните HTML-документ, затем откройте его в браузере и запустите консоль. Проанализируйте полученные результаты — установленные дату и время.

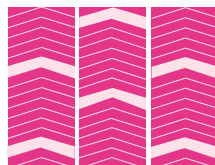


Сопоставление текста с шаблоном

Объект **RegExp** — это встроенный объект JavaScript, который создает объект регулярного выражения для сопоставления текста с шаблоном. Регулярные выражения используются как средство для поиска и замены текста при помощи шаблонов.

Шаблон регулярного выражения может состоять целиком из символьных литералов, заключенных между символами `/`. Например, регулярное выражение `/wind/` найдет совпадение в строке «windows» — шаблон строго совпадает со строкой в тексте. Как правило, шаблон регулярного выражения состоит из набора символьных литералов и специальных символов или метасимволов.

Для выполнения поиска без учета регистра шаблон может также включать модификатор `i` после последнего символа `/` или модификатор `g` для выполнения глобального поиска всех совпадений шаблона. Метод `test()` выполняет поиск сопоставления регулярного выражения



Тема регулярных выражений обширна и выходит за рамки этой книги. Здесь на тему регулярных выражений приводятся лишь краткие сведения.



Диапазон символов [a-z] соответствует только строчным символам латинского алфавита, а диапазон [a-z0-9] включает еще и цифры.

указанной строке. При обнаружении совпадения метод `test()` возвращает значение `true`, в противном случае — `false`. Метод `exec()` выполняет поиск сопоставления регулярного выражения в указанной строке и возвращает массив с результатами или значение `null`. `Index` — индекс сопоставления в строке — начинается с нуля.

Специальный символ	Значение	Пример
.	Любой символ	ja.pt
^	Начало строки	^ja
\$	Конец строки	...pt\$
*	Повторение фрагмента 0 или более раз	ja*
+	Повторение фрагмента 1 или более раз	ja+
?	Фрагмент либо присутствует, либо отсутствует (0 или 1)	ja?
{}	Множественное повторение	ja{3}
[]	Класс символов	[a-z]
\	Специальная последовательность	\d
	Фрагмент слева или фрагмент справа	a b
()	Группировка выражений	(...)



regexp.html

- 1

Создайте HTML-документ с самовызывающейся функцией. Объявите и проинициализируйте три переменные.

```
const system = 'Windows', suite = 'Office' pattern = /ice/i
```
- 2

Добавьте операторы для вывода результатов поиска.

```
console.log( 'In ' + system + '?' + pattern.test( system ) )
console.log( 'In ' + suite + '?' + pattern.test( suite ) )
```
- 3

Добавьте операторы для вывода совпадающего текста и позиции. В противном случае — сообщение о том, что совпадения не найдены.

```
let result = pattern.exec( suite )
if( result )
{
  console.log( 'Found ' + result + ' at ' + result.index )
}
else { console.log( 'No Match Found' ) }
```



4

Добавьте операторы для вывода результата проверки некорректного адреса электронной почты.

```
let email = 'mike@example'
const format = /\.+\@.+\.+/
console.log( email + ' Valid? ' + format.test( email ) )
```

Используемое в этом примере регулярное выражение проверяет только самые основные требования к формату электронной почты.

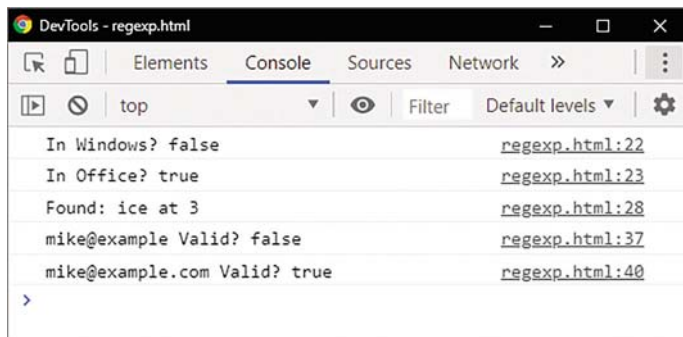
5

Добавьте операторы для исправления адреса и вывода результата проверки.

```
email += '.com'
console.log( email + ' Valid? ' + format.test( email ) )
```

6

Сохраните HTML-документ, затем откройте его в браузере и запустите консоль. Проанализируйте полученные результаты — регулярные выражения.



Нумерация индексов в строке начинается с нуля, поэтому четвертый символ находится в позиции индекса 3.

Заключение

- Значения присваиваются объекту в виде разделенного запятыми списка **name: value** (имя: значение), заключенного в фигурные скобки {}.
- На значения свойств объекта можно ссылаться с использованием точечной нотации, или указав их в квадратных скобках [].
- Методы объекта вызываются с помощью добавления после имени круглых скобок ().
- Ключевое слово **this** относится к объекту, к которому оно принадлежит.
- Пользовательские объекты расширяются путем присвоения нового значения с использованием точечной записи для ссылки на свойство.
- Пользовательские объекты могут быть в любое время легко расширены и изменены. Для этого нужно лишь назначить новое значение с помощью точечной нотации.
- Цикл **for in** используется для перечисления всех свойств и методов объекта.
- Все объекты в JavaScript наследуют свойства и методы из прототипа объекта **Object.prototype**.
- Объект **Array** — это встроенный объект JavaScript, который используется для хранения упорядоченных коллекций данных (с различными типами данных). Элементы массива нумеруются автоматически, начиная с нуля.
- Объекты массивов создаются путем присвоения переменным литеральных значений, указанных в квадратных скобках [] и разделенных запятыми.
- Доступ к элементам массива осуществляется с помощью оператора [].

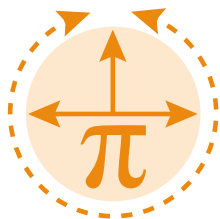
- Чтобы узнать длину массива, необходимо обратиться к его свойству **length**.
- Встроенный объект **Date** представляет конкретную дату, время и часовой пояс.
- Конструктор **Date()** может дополнительно указывать от двух до семи аргументов для установки значений для каждого из его компонентов.
- Объект **Date** также предоставляет отдельные методы для установки значений даты и времени.
- Объект **RegExp** — это встроенный объект JavaScript, который создает объект регулярного выражения для сопоставления текста с шаблоном.
- Методы **test()** и **exec()** выполняют поиск сопоставления регулярного выражения в указанной строке.

5

Работа с числовыми и строковыми типами данных

*В этой главе вы
познакомитесь с методами
встроенных объектов Math
и String.*

104	Вычисление площади
106	Сравнение чисел
108	Округление чисел
110	Генерация случайных чисел
112	Объединение строк
115	Разбиение строк
117	Поиск символов
119	Обрезка строк
121	Заключение



Все методы объекта **Math** описаны на стр. 108.

Вычисление площади

Объект **Math** — встроенный объект JavaScript, хранящий в своих свойствах и методах различные константы и функции. Ниже в таблице приведены константы, их описание и значение.

Константа	Описание
Math.E	Число Эйлера или Непера, основание натуральных логарифмов, приблизительно равно 2,71828
Math.LN2	Натуральный логарифм из 2, приблизительно равен 0,69315
Math.LN10	Натуральный логарифм из 10, приблизительно равен 2,30259
Math.LOG2E	Двоичный логарифм из E, приблизительно равен 1,44269
Math.LOG10E	Десятичный логарифм из E, приблизительно равен 0,43429
Math.PI	Число Пи, приблизительно равно 3,14159
Math.SQRT1_2	Квадратный корень из 0,5, приблизительно равен 0,70711
Math.SQRT2	Квадратный корень из 2, приблизительно равен 1,41421

Нет необходимости создавать экземпляр объекта **Math**, так как он присутствует по умолчанию, поэтому доступ к математическим константам и методам осуществляется через объект **Math** и с помощью точечной нотации.

Константы этого объекта, как правило, используются для вычисления каких-либо математических операций. В таблице выше перечислены наиболее распространенные из них.

1

Создайте HTML-документ с самовызывающейся функцией. Объявите и проинициализируйте переменную.

```
let radius = 4
console.log( '\nRadius of Circle: ' + radius )
```



constants.html

2

Добавьте операторы для выполнения математического расчета и вывода результата.

```
let area = Math.PI * ( radius * radius )
console.log( '\nArea of Circle: ' + area )
```

3

Добавьте операторы для выполнения еще одного математического расчета и вывода результата.

```
let circumference = 2 * ( Math.PI * radius )
console.log( '\nPerimeter of Circle: ' + circumference )
```

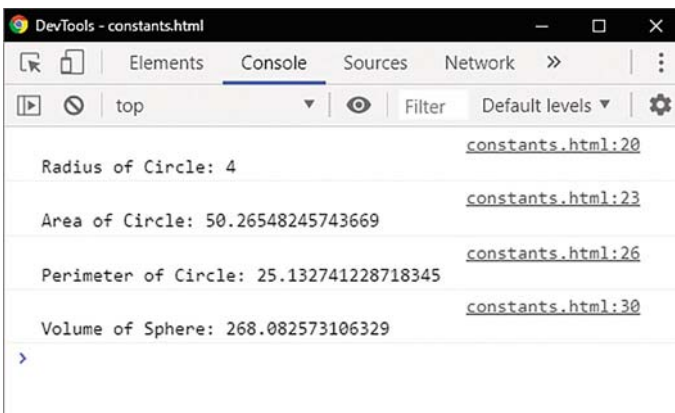
4

Добавьте операторы для выполнения окончательного математического расчета и вывода результата.

```
let cube = ( radius * radius * radius )
let volume = ( ( 4 / 3 ) * Math.PI ) * cube
console.log( '\nVolume of Sphere: ' + volume )
```

5

Сохраните HTML-документ, затем откройте его в браузере и запустите консоль. Проанализируйте полученные результаты — математические расчеты.



Имена констант объекта `Math` записываются полностью прописными символами. Например, `Math.PI`, а не `Math.pi`.



Сравнение чисел

Перед вами наиболее распространенные и важные методы встроенного объекта **Math**.

Метод	Описание
<code>Math.abs()</code>	Возвращает абсолютное значение числа
<code>Math.acos()</code>	Возвращает арккосинус числа
<code>Math.asin()</code>	Возвращает арксинус числа
<code>Math.atan()</code>	Возвращает арктангенс числа
<code>Math.atan2()</code>	Возвращает арктангенс от частного своих аргументов
<code>Math.ceil()</code>	Возвращает значение числа, округленное к большему целому
<code>Math.cos()</code>	Возвращает косинус числа
<code>Math.exp()</code>	Возвращает E^x , где x — аргумент, а E — число Эйлера
<code>Math.floor()</code>	Возвращает значение числа, округленное к меньшему целому
<code>Math.log()</code>	Возвращает натуральный логарифм числа
<code>Math.max()</code>	Возвращает наибольшее число из своих аргументов
<code>Math.min()</code>	Возвращает наименьшее число из своих аргументов
<code>Math.pow()</code>	Возвращает основание в степени экспоненты
<code>Math.random()</code>	Возвращает псевдослучайное число
<code>Math.round()</code>	Возвращает значение числа, округленное до ближайшего целого
<code>Math.sin()</code>	Возвращает синус числа
<code>Math.sqrt()</code>	Возвращает квадратный корень числа
<code>Math.tan()</code>	Возвращает тангенс числа



math.html

1

Создайте HTML-документ с самовызывающейся функцией. Объявите и проинициализируйте две переменные.

```
let square = Math.pow( 5, 2 )    // 52 или (5 x 5)
let cube = Math.pow( 4, 3 )      // 43 или (4 x 4 x 4)
```

2

Добавьте операторы для вывода наибольшего и наименьшего значений из двух положительных чисел.

```
console.log( '\nLargest Positive: ' +
              Math.max( square, cube ) )
console.log( '\nSmallest Positive: ' +
              Math.min( square, cube ) )
```

3

Добавьте операторы для изменения полярности каждого числа — положительные числа должны стать отрицательными.

```
square *= -1
cube *= -1
```

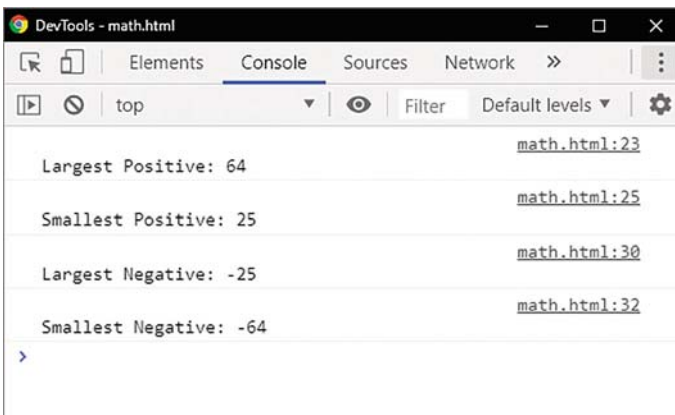
4

Добавьте операторы для вывода наибольшего и наименьшего значений из двух отрицательных чисел.

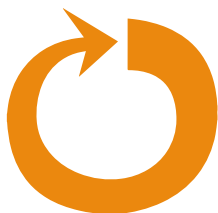
```
console.log( '\nLargest Negative: ' +
              Math.max( square, cube ) )
console.log( '\nSmallest Negative: ' +
              Math.min( square, cube ) )
```

5

Сохраните HTML-документ, затем откройте его в браузере и запустите консоль. Проанализируйте полученные результаты — сравнение чисел.



При сравнении двух отрицательных чисел большее будет расположено правее, то есть ближе к началу отсчета (ближе к 0).



Округление чисел

Встроенный объект **Math** предоставляет три метода округления чисел с плавающей запятой до целых значений. Каждый метод принимает значение с плавающей запятой в качестве аргумента и возвращает целое число. Метод **Math.ceil()** округляет аргумент до большего целого. Метод **Math.floor()** округляет аргумент до ближайшего меньшего значения. Метод **Math.round()** возвращает число, округленное к ближайшему целому.

При работе со значениями с плавающей запятой важно понимать несоответствие между компьютерным математическим стандартом с плавающей запятой, как определено IEEE (Институт инженеров по электротехнике и электронике), и общепринятой математической точностью. Это связано с тем, что некоторые десятичные числа нельзя точно преобразовать в двоичную форму. Например, десятичное число **81,66** не может быть точно преобразовано в двоичное, поэтому выражение **81,66*15** в результате возвращает **1224,8999999999999**, а не точное значение **1224,9**.

Некоторые языки программирования обеспечивают автоматическое округление для устранения несоответствий с плавающей запятой. Однако в JavaScript такой метод не используется, поэтому во избежание ошибочных результатов будьте внимательными, особенно при работе с денежными величинами. Рекомендуется сначала умножить значение с плавающей запятой на 100, затем выполнить арифметическую операцию и, наконец, разделить результат на 100, чтобы вернуться к тому же десятичному представлению.



Метод **Math.round()** по умолчанию возвращает число, округленное к ближайшему целому, поэтому **Math.round(7.5)** возвращает число 8, а не 7; **Math.round(-7.5)** возвращает число -7, а не -8.

Аналогичную процедуру можно использовать для ограничения чисел с плавающей запятой до двух десятичных знаков. После умножения значения на 100 можно использовать метод **Math.round()** для округления значения. Затем использовать деление на 100 и в результате вернуть два десятичных знака.

Все математические операции могут быть записаны как отдельные шаги. Также для определения

порядка вычислений можно использовать скобки. Например, преобразование числа с плавающей запятой в переменной с именем «num» выглядит следующим образом:

```
num = num * 100
num = Math.round( num )
num /= 100
```

или

```
num = ( Math.round( num * 100 ) ) / 100
```

1

Создайте HTML-документ с самовызывающейся функцией. Объявите и проинициализируйте переменную.

```
let bodyTemp = 98.6
```



round.html

2

Добавьте операторы для вывода ближайших целых чисел к значению с плавающей запятой.

```
console.log( 'Ceiling: ' + Math.ceil( bodyTemp ) )
console.log( 'Floor: ' + Math.floor( bodyTemp ) )
console.log( 'Round: ' + Math.round( bodyTemp ) )
```

3

Добавьте операторы для вывода неверного результата выражения и его исправленного эквивалента.

```
console.log( '\nImprecision: ' + ( 81.66 * 15 ) )
console.log( 'Corrected: ' +
  ( ( ( 81.66 * 100 ) * 15 ) / 100 ) )
```

4

Добавьте операторы для вывода числа с плавающей запятой и его преобразованного значения.

```
console.log( '\nFloat: ' + Math.PI )
console.log( 'Commutated: ' +
  ( ( Math.round ( Math.PI * 100 ) / 100 ) ) )
```

5

Сохраните HTML-документ, затем откройте его в браузере и запустите консоль.



Сначала вычисляются выражения во внутренних скобках.

Проанализируйте полученные результаты — округление чисел.

DevTools - round.html	
Elements	Console
top	Filter Default levels
Ceiling: 99	round.html:21
Floor: 98	round.html:22
Round: 99	round.html:23
Imprecision: 1224.8999999999999	round.html:25
Corrected: 1224.9	round.html:26
Float: 3.141592653589793	round.html:28
Commutated: 3.14	round.html:29

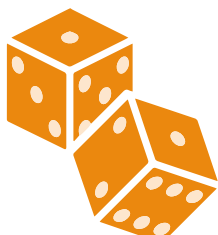
Генерация случайных чисел

Метод **Math.random()** возвращает псевдослучайное число с плавающей запятой от 0 (включительно) до 1 (но не включая 1). Этот метод можно использовать для создания различных эффектов веб-страницы, для которых требуется сгенерированное случайное число. Умножение случайного числа с плавающей запятой увеличит его диапазон. Например, умножение на 10 увеличивает диапазон от 0 до 10.

Как правило, полезно округлить случайное число с помощью метода **Math.ceil()**, чтобы диапазон стал от 1 до 10.

Процесс определения диапазона для случайного числа может быть записан как отдельные шаги. Также для определения порядка вычислений можно использовать скобки. Например, процесс определения диапазона от 1 до 10 для переменной с именем «rand» выглядит следующим образом:

```
let rand = Math.random( )
rand *= 10
rand = Math.ceil( rand )
```



или

```
let rand = Math.ceil( Math.random( ) * 10 )
```

В указанном диапазоне может быть сгенерирован ряд уникальных случайных чисел. Например, случайный выбор номеров лотереи в диапазоне от 1 до 59.

1

Создайте HTML-документ с самовызывающейся функцией. Объявите и проинициализируйте пять переменных.

```
let i, rand, temp, nums = [ ]  
let str = '\n\nYour Six Lucky Numbers: '
```



random.html

2

Добавьте цикл для заполнения элементов массива от 1 до 59 соответствующими номерами индексов.

```
for( i = 1 ; i < 60; i++ ) { nums[ i ] = i }
```

3

Добавьте цикл для случайного выбора чисел в элементах массива.

```
for( i = 1 ; i < 60; i++ )  
{  
    rand = Math.ceil( Math.random( ) * 59 )  
    temp = nums[ i ]  
    nums[ i ] = nums[ rand ]  
    nums[ rand ] = temp  
}
```



Шаг 3 содержит алгоритм, который перетасовывает числа и гарантирует, что значения элементов не повторяются.

4

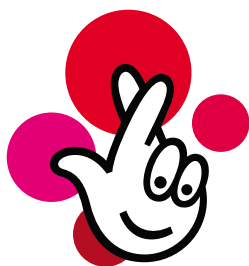
Добавьте цикл для добавления списка из шести значений элементов через дефис.

```
for( i = 1 ; i < 7 ; i++ )  
{  
    str += nums[ i ]  
    if( i !== 6 ) { str += ' - ' }  
}
```

5

Добавьте оператор для вывода строковой переменной.

```
console.log( str )
```

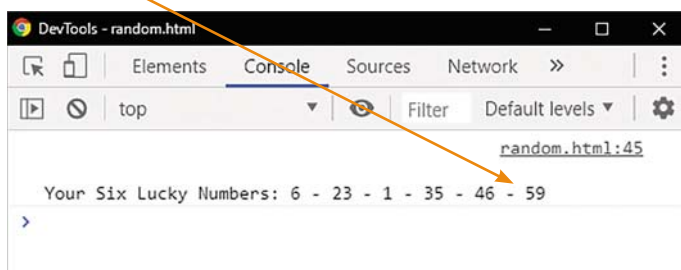
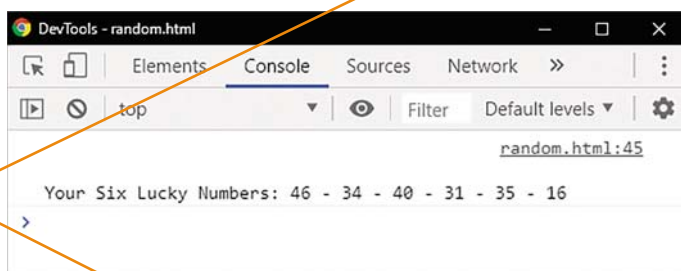
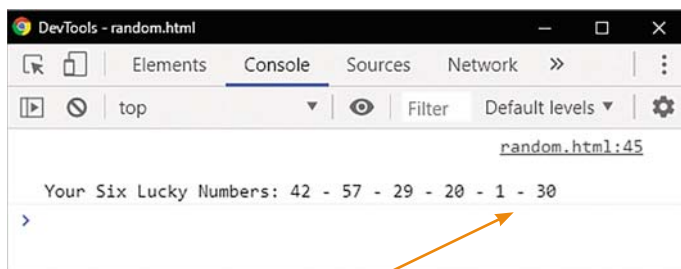


В этом примере случайные числа находятся в диапазоне от 1 до 59 — для игры в UK Lotto или в US New York Lotto.



6

Сохраните HTML-документ, затем откройте его в браузере и запустите консоль. Проанализируйте полученные результаты — уникальный набор случайных чисел в указанном диапазоне при выполнении каждого шага.



Объединение строк

В JavaScript существует объект **String**, предоставляющий полезные методы для управления строковыми значениями. Однако нет необходимости создавать его экземпляры с помощью ключевого слова **new** и конструктора **String()**, поскольку его методы можно просто применять к строковым переменным с использованием точечной нотации. Например, метод **str.toUpperCase()** возвращает значение

строки, преобразованное в верхний регистр. Метод **str.toLowerCase()** возвращает значение строки, преобразованное в нижний регистр.

Существует также свойство **length**, в котором хранится общее количество символов в строке.

Во многих приведенных ранее примерах для объединения нескольких строк используется оператор конкатенации **+**. Также можно использовать метод **concat()** для добавления одной или нескольких строк, представленных в виде списка аргументов, разделенных запятыми.

Некоторые разработчики для объединения строк и переменных используют встроенную функцию **eval()**. Но мы вам этого не советуем. Функция **eval()** напрямую вызывает компилятор JavaScript, чтобы скомпилировать свой строковый аргумент в оператор JavaScript.

- Если строка представляет собой выражение, функция **eval()** вычислит это выражение — например, **eval('1+1')** возвращает значение **2**.
- Если строка представляет собой оператор или последовательность операторов, функция **eval()** выполнит последний оператор — например, код **eval('let num=100; alert(num)')** создает диалоговое окно с предупреждением.



Такое использование функции **eval()** требует больших затрат на производительность кода. Однако ее можно заменить более современными и эффективными методами. Кроме того, использование функции **eval()** — это невероятный риск для безопасности. Злоумышленнику слишком легко запустить какой угодно код, когда вы используете **eval()**. Например, код **eval('while(true); alert(')')** вызовет бесконечный цикл, блокирующий браузер.



Никогда не используйте функцию **eval()**. Существует даже известное выражение — «eval can be evil» («eval — это зло»).



string.html

- 1 Создайте HTML-документ с самовызывающейся функцией. Объявите и проинициализируйте три переменные.

```
let topic = 'JavaScript'
let series = 'in easy steps'
let title = ''
```

- 2 Добавьте операторы для вывода преобразованных версий регистра первых двух значений строковой переменной.

```
console.log( topic + ' > ' + topic.toLowerCase() )
console.log( series + ' > ' + series.toUpperCase() )
```

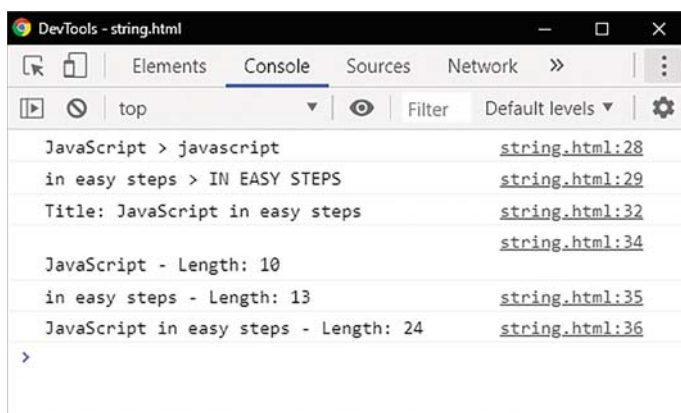
- 3 Вставьте операторы, чтобы добавить пробел и вторую строку к первой, затем присвойте ее третьей переменной и выведите объединенную строку.

```
title = topic.concat( ' ', series )
console.log( 'Title: ' + title )
```

- 4 Добавьте операторы для вывода длины каждой строки.

```
console.log( '\n' + topic + ' - Length: ' +
    topic.length )
console.log( series + ' - Length: ' + series.length )
console.log( title + ' - Length: ' + title.length )
```

- 5 Сохраните HTML-документ, затем откройте его в браузере и запустите консоль. Проанализируйте полученные результаты.



Разбиение строк

В JavaScript существуют методы, которые позволяют из полной строки скопировать определенную ее часть. Они рассматривают строку как массив, в котором каждый элемент содержит символ или пробел, и на него можно ссылаться по позиции индекса. Как и в случае с массивами, нумерация индекса строки начинается с 0, поэтому первый символ находится в позиции 0.

Метод **substring()** возвращает подстроку строки между двумя индексами, или от одного индекса и до конца строки. Первый параметр определяет индекс, с которого производится копирование подстроки из исходной строки. Второй параметр, необязательный, определяет индекс, до которого происходит копирование подстроки. Затем будут скопированы все символы между начальной и конечной позициями.

Альтернативный способ копирования подстрок, метод **substr()**, позволяет извлечь из строки определенное количество символов, начиная с заданного индекса. Как и **substring()**, метод **substr()** может принимать один аргумент для указания позиции индекса, с которого следует начать копирование, и будет копировать все символы после этой позиции до конца строки. В отличие от **substring()**, второй параметр метода **substr()** указывает количество символов, которые будут скопированы из исходной строки.

Метод **slice()** позволяет вернуть новый массив, который содержит копии элементов, вырезанных из исходного массива.

Иногда бывает полезно копировать фрагменты строки, разделенные определенным символом. Символ-разделитель можно указать в качестве аргумента метода **split()**, который позволяет разбить строки на массив подстрок, используя заданную строку-разделитель для определения места разбиения. Второй параметр, необязательный, определяет ограничение на количество найденных



Метод **substr()** считается проще в использовании, чем метод **substring()**, так как необходимо только вычислить начальную позицию и длину подстроки, а не конечную позицию.



На стр. 161 приведен пример, в котором метод **split()** используется для разделения данных файлов cookie.



split.html



Символ обратной косой черты \ позволяет продолжить текст на следующей строке.

подстроку. Когда этот параметр задан, метод `split()` разбивает строку при каждом возникновении указанного разделителя, но останавливается, когда количество элементов в массиве достигнет заданного значения.

Ни один из этих методов не изменяет исходную строку, а просто делает копию ее определенного фрагмента.

- 1 Создайте HTML-документ с самовызывающейся функцией. Объявите и проинициализируйте переменную.

```
let definition = 'JavaScript is the original dialect of \
the ECMAScript standard language.'
```

- 2 Добавьте операторы для присвоения выбранных фрагментов строки второй переменной и выведите ее значение.

```
let str = definition.slice( 0, 27 )
str += definition.slice( 62, 70 )
console.log( str )
```

- 3 Добавьте операторы для вывода четырех отдельных фрагментов.

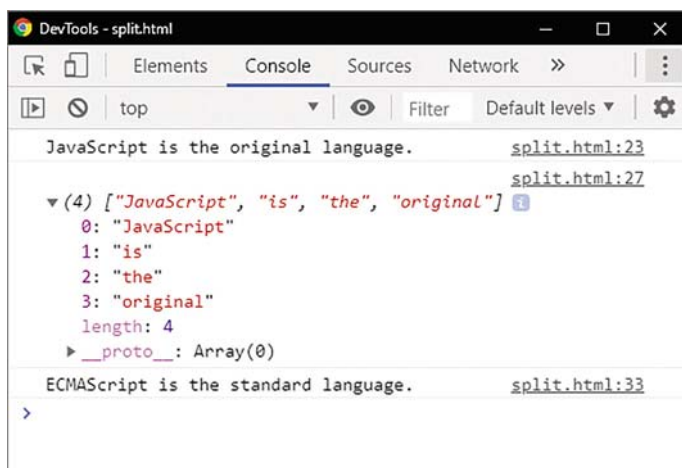
```
console.log( str.split( ' ', 4 ) )
```

- 4 Добавьте операторы для присвоения выбранных подстрок исходной строки второй переменной и выведите ее значение.

```
str = definition.substring( 42, 52 )
str += definition.substring( 10, 17 )
str += definition.substr( 52, 70 )
```

```
console.log( str )
```

- 5 Сохраните HTML-документ, затем откройте его в браузере и запустите консоль. Проанализируйте полученные результаты — фрагменты строки.



Чтобы вернуть массив отдельных символов, в методе `split()` в качестве разделителя укажите `"` (пустую строку без пробелов).

Поиск символов

В JavaScript существуют методы, позволяющие искать в строке определенный символ или подстроку. Метод `search()` возвращает позицию первого символа найденной подстроки, в противном случае (если соответствие не найдено) метод вернет значение `-1`. Метод `match()` производит поиск по заданной строке с использованием регулярного выражения и возвращает массив, содержащий результаты этого поиска. Если соответствие не будет найдено, то метод `match()` вернет значение `null`.

Метод `indexOf()` позволяет возвратить индекс искомого элемента в массиве при первом совпадении, или `-1`, если элемент не найден. Метод `lastIndexOf()` производит поиск необходимого элемента с конца строки.

Метод `charAt()` возвращает символ по заданному индексу внутри строки. Метод `charCodeAt()` возвращает Unicode символа по указанному индексу в строке. И наоборот, в методе `fromCharCode()` в качестве аргументов можно указать одно или несколько символов Unicode для возврата их строковых значений.

Кроме того, метод `replace()` выполняет внутри строки поиск с использованием регулярного выражения или строкового значения и возвращает новую строку, в которой будут заменены найденные значения.



Ключевое слово `null` не равно нулевому значению, так как `null` вообще не имеет никакого значения.



find.html



Используйте двойные кавычки, чтобы включать символ кавычки в строку, заключенную в одинарные кавычки.



Значения A-Z верхнего регистра Unicode — 65–90, а значения a-z нижнего регистра — 97–122.

- 1 Создайте HTML-документ с самовызывающейся функцией. Объявите и проинициализируйте строковую переменную.

```
let str = 'JavaScript in easy steps'
```

- 2 Добавьте операторы для вывода результатов двух поисков строк с учетом регистра.

```
console.log( "Script" Search: ' + str.search( 'Script' ) )
console.log( "script" Search: ' + str.search( 'script' ) )
```

- 3 Добавьте операторы для вывода результатов двух совпадений строк с учетом регистра.

```
console.log( "\nScript" Match: ' + str.match( 'Script' ) )
console.log( "script" Match: ' + str.match( 'script' ) )
```

- 4 Добавьте операторы для вывода первой и последней позиции индекса символа, если он найден в строке.

```
console.log( '\nindexOf "s": ' + str.indexOf( 's' ) )
console.log( 'indexOf "m": ' + str.indexOf( 'm' ) )
console.log( '\nlastIndexOf "s": ' + str.lastIndexOf( 's' ) )
console.log( 'lastIndexOf "m": ' + str.lastIndexOf( 'm' ) )
```

- 5 Добавьте операторы для вывода первого символа в строке и его значения Unicode, а также четырех символов, указанных в их значениях Unicode.

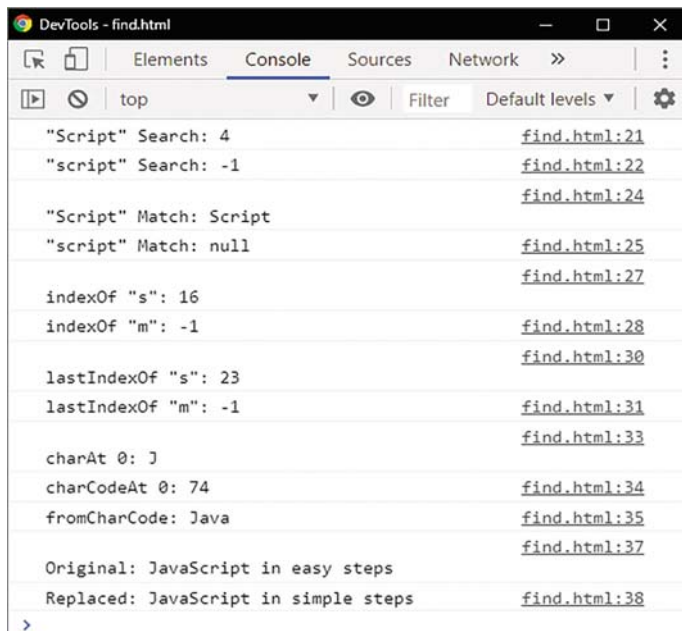
```
console.log( '\ncharAt 0: ' + str.charAt( 0 ) )
console.log( 'charCodeAt 0: ' + str.charCodeAt( 0 ) )
console.log( 'fromCharCode: ' +
    String.fromCharCode( 74, 97, 118, 97 ) )
```

- 6 Добавьте операторы для вывода исходной строки и ее измененной версии.

```
console.log( '\nOriginal: ' + str )
console.log( 'Replaced: ' +
    str.replace( 'easy', 'simple' ) )
```


7

Сохраните HTML-документ, затем откройте его в браузере и запустите консоль. Проанализируйте полученные результаты — фрагменты строки.



Метод `replace()` возвращает измененную версию исходной строки. При этом исходная строка не изменяется.

Обрезка строк

Метод `trim()` удаляет пробелы из начала и конца строки. Это особенно полезно для удаления всех пробелов, отступов или символов новой строки, случайно включенных пользователем.

Вырезав пробелы из строки, вы можете с помощью метода `startsWith()` проверить ее первые символы, а с помощью метода `endsWith()` — последние. Оба метода принимают в качестве аргумента символ или подстроку и ищут совпадение с учетом регистра. Если совпадение найдено, возвращается значение `true`, в противном случае — `false`.

Доступ к определенному символу осуществляется с помощью указания его позиции индекса в квадратных скобках `[]` — например, при использовании





На стр. 162 приведен пример использования метода `split()` для удаления пробелов из данных `cookie`.



trim.html



В качестве альтернативы сопоставление символов можно выполнить с помощью оператора равенства, например `str[0]=='L'`

записи `str[0]`, осуществляется доступ к первому символу строки в переменной с именем «`str`».

Метод `includes()` позволяет определить, содержит ли массив искомый элемент. В случае нахождения элемента метод возвращает логическое значение `true`, в противном случае `false`.

При необходимости создать новую строку, содержащую несколько копий существующей строки, просто укажите целочисленный аргумент для метода `repeat()`. Метод `repeat()` конструирует и возвращает новую строку, содержащую указанное количество соединенных вместе копий строки, на которой он был вызван.

- 1 Создайте HTML-документ с самовызывающейся функцией. Объявите и проинициализируйте переменную — со значением, содержащим пробелы по обоим концам строки.

```
let str = ' Love For All, Hatred For None. '
```

- 2 Добавьте операторы для вывода строки и проверьте ее начало и конец.

```
console.log( 'String: ' + str )
console.log( 'Starts With "L" ? ' + str.startsWith( 'L' ) )
console.log( 'Ends With "." ? ' + str.endsWith( '.' ) )
console.log( 'First Letter: ' + str[ 0 ] )
```

- 3 Добавьте оператор для присвоения переменной обрезанной строки.

```
str = str.trim( )
```

- 4 Добавьте операторы для вывода обрезанной строки и проверьте ее начало и конец.

```
console.log( 'Trimmed: ' + str )
console.log( 'Starts With "L"? ' + str.startsWith( 'L' ) )
console.log( 'Ends With "."? ' + str.endsWith( '.' ) )
console.log( 'First Letter: ' + str[ 0 ] )
```

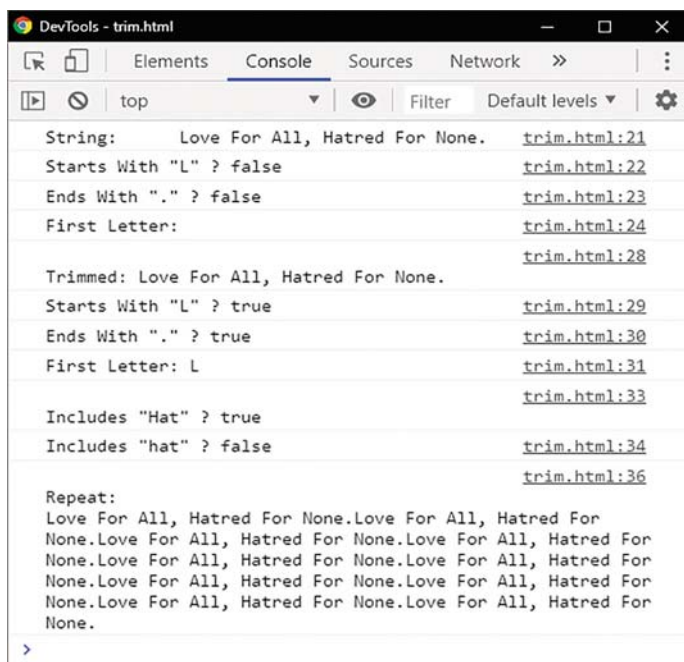
- 5 Добавьте операторы, чтобы увидеть подстроки внутри строки.

```
console.log( '\nIncludes "Hat"? ' + str.includes( 'Hat' ) )
console.log( 'Includes "hat"? ' + str.includes( 'hat' ) )
```

- 6 Добавьте оператор для вывода 10 копий об-
резанной строки.

```
console.log('\nRepeat:\n' + str.repeat( 10 ) )
```

- 7 Сохраните HTML-документ, затем откройте
его в браузере и запустите консоль. Проана-
лизируйте полученные результаты.



The screenshot shows the Chrome DevTools Console with the following output:

```
String:      Love For All, Hatred For None.   trim.html:21
Starts With "L" ? false                       trim.html:22
Ends With "." ? false                         trim.html:23
First Letter:                                trim.html:24
                                                    trim.html:28
Trimmed: Love For All, Hatred For None.
Starts With "L" ? true                       trim.html:29
Ends With "." ? true                         trim.html:30
First Letter: L                             trim.html:31
                                                    trim.html:33
Includes "Hat" ? true
Includes "hat" ? false                       trim.html:34
                                                    trim.html:36
Repeat:
Love For All, Hatred For None.Love For All, Hatred For
None.Love For All, Hatred For None.Love For All, Hatred For
None.Love For All, Hatred For None.Love For All, Hatred For
None.Love For All, Hatred For None.Love For All, Hatred For
None.Love For All, Hatred For None.Love For All, Hatred For
None.Love For All, Hatred For None.Love For All, Hatred For
None.
```



При использо-
вании этих методов
учитывается ре-
гистр.

Заключение

- Объект **Math** предоставляет для использования математические константы, например **Math.PI**, и методы, например **Math.max()**.
- Числа с плавающей запятой можно округлить до ближайшего целого с помощью методов **Math.floor()**, **Math.ceil()** и **Math.round()**.
- В JavaScript не используется метод автоматического округления.

- Во избежание несоответствий рекомендуется умножить числа с плавающей запятой на 100, выполнить арифметические операции, затем разделить результат на 100.
- Метод **Math.random()** возвращает случайное число с плавающей запятой в диапазоне от 0 до 1.
- В результате умножения случайного числа с плавающей запятой на 10 и округления результата с помощью **Math.ceil()** получается диапазон от 1 до 10.
- В JavaScript существуют полезные методы для управления строковыми значениями, например **toUpperCase()** и **toLowerCase()**.
- Существует также свойство **length**, в котором хранится общее количество символов в строке.
- Строки можно объединить с помощью оператора конкатенации **+** или метода **concat()**.
- Использование функции **eval()** — это невероятный риск для безопасности. Никогда не используйте ее!
- Аргументы методов **slice()** и **substring()** определяют начальную и конечную позиции. Аргументы метода **substr()** задают начальную позицию и количество символов для копирования.
- Метод **split()** разбивает объект **String** на массив строк путем деления строки указанной подстрокой.
- Для поиска символов в строке используются методы **search()**, **match()**, **indexOf()**, **lastIndexOf()** и **charAt()**.
- Метод **trim()** удаляет пробелы из начала и конца строки.
- Методы **startsWith()**, **endsWith()** и **includes()** выполняют поиск совпадений с учетом регистра в строке.
- Методы **replace()** и **repeat()** создают измененные строки.

6

Открытие окон и методы объекта window

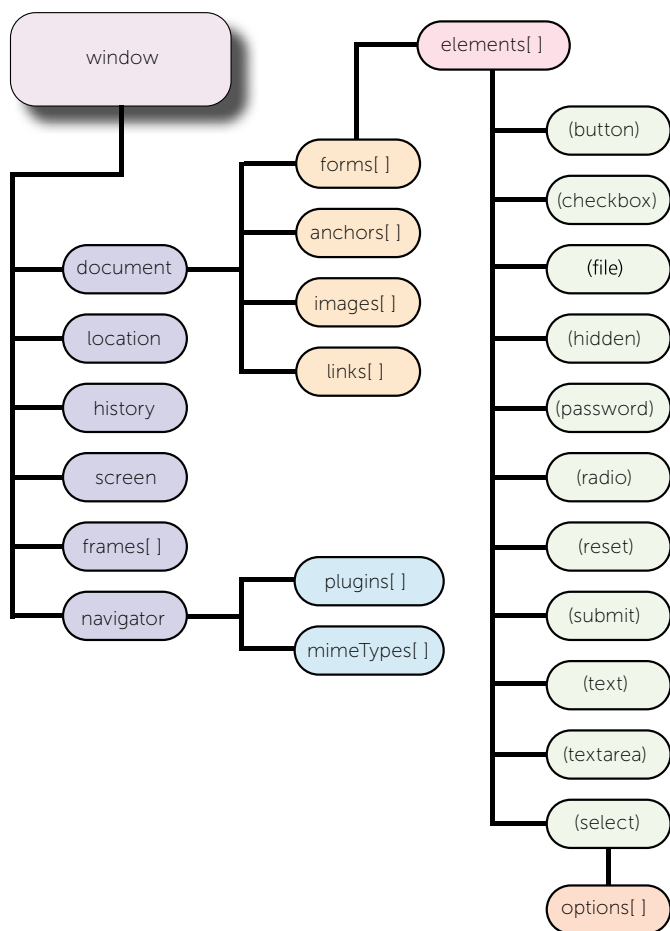
*В этой главе описываются
и демонстрируются методы
объекта window.*

- 124 Введение в DOM
- 126 Свойства объекта window
- 128 Диалоговые окна
- 130 Прокрутка
- 133 Всплывающие окна
- 135 Создание таймера
- 137 Сведения о браузерах
- 139 Включение/отключение функций
- 142 Расположение
- 144 История
- 146 Заключение



Введение в DOM

Браузер представляет все компоненты веб-страницы в иерархической древовидной структуре, называемой объектной моделью документа (DOM, Document Object Model). Каждый компонент отображается под объектом **window**. Ниже представлена древовидная структура:



Элементы, за которыми следуют квадратные скобки, представляют собой объекты массива, а элементы в обычных круглых скобках — это различные типы элементов формы.

Цикл **for in** используется для перечисления всех свойств объекта **window**, предоставляемых браузером. Список будет содержать основные свойства, общие для всех современных браузеров, а также дополнительные свойства для определенного браузера.

- 1 Создайте HTML-документ с пустым элементом списка.

```
<ol id="props" style="column-count:3"> </ol>
```

- 2 Создайте самовызывающуюся функцию. Объявите и проинициализируйте две переменные.

```
const list = document.getElementById( 'props' )  
let property = null
```

- 3 Добавьте цикл для заполнения списка элементами — свойствами объекта `window`.

```
for( property in window )  
{  
  if( property ) { list.innerHTML += '<li>' + property }  
}
```

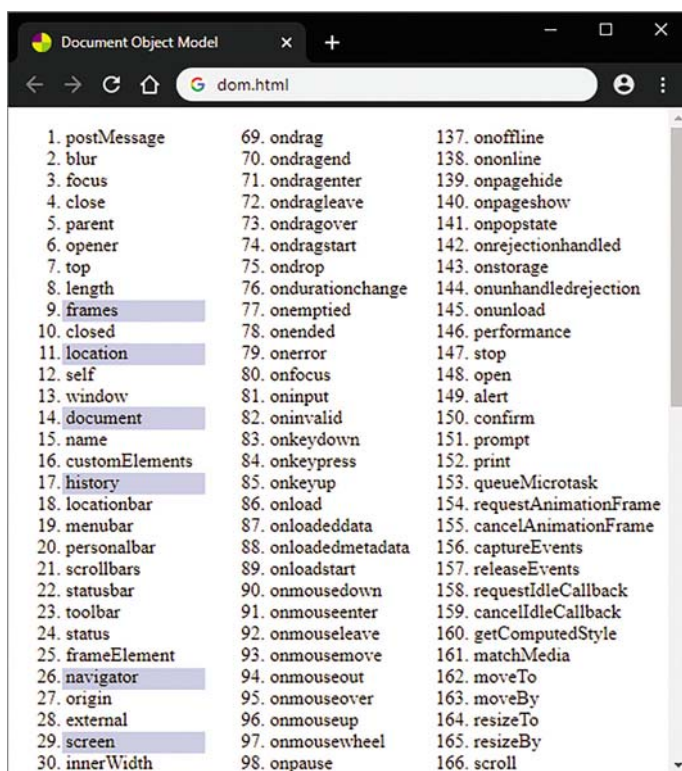
- 4 Сохраните HTML-документ, затем откройте его в браузере. Проанализируйте полученные результаты — свойства и методы объекта `window`.



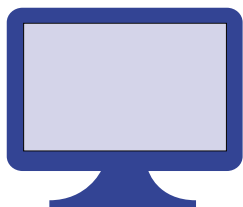
dom.html



Закрывающие теги `` необязательны, поэтому в этом цикле они не используются.



Прокрутите список вниз и проанализируйте все свойства и методы `window` — в нашем примере 204 элемента. Важные свойства выделены цветом.



Обратите внимание на стиль написания составных слов в программировании camelCase.

Свойства объекта window

Объект **window** имеет дочерний объект **screen**, который предоставляет свойства, описывающие разрешение экрана в зависимости от количества пикселей. Общие размеры экрана можно установить с помощью свойств **window.screen.width** и **window.screen.height**.

Свойства **window.screen.availWidth** и **window.screen.availHeight** возвращают количество пикселей, которые доступны браузеру по горизонтали и вертикали соответственно.

Свойство **window.screen.colorDepth** возвращает битовую глубину цветовой палитры для отображения изображений в битах на пиксель.

- **8 бит** — Low Color (неинтенсивная окраска, отображает только 256 цветов).
- **16 бит** — High Color (высококачественный цвет, отображает 65 536 цветов).
- **24 бита** — True Color (натуральный цвет, отображает миллионы цветов).
- **32 бита** — Deep Color (насыщенный цвет, отображает гамму из миллиарда или более цветов).

Современные компьютеры для цветного отображения используют 24-битное или 32-битное оборудование. Однако в устаревших компьютерах оборудование 16-битное. Только очень старые компьютеры и мобильные телефоны для цветного отображения используют 8-битное оборудование.

Свойство **window.screen.pixelDepth** возвращает битовую глубину экрана в пикселях. На современных компьютерах это то же значение, что и значение свойства **window.screen.colorDepth**. Всегда используйте его для определения цвета.

В настоящее время некоторые браузеры поддерживают объект **window.screen.orientation** со свойством

type, которое описывает текущую ориентацию экрана (альбомную или книжную) и указывает, основная ориентация или неосновная.

Поскольку объект **window** — глобальный объект, его имя может быть опущено при ссылке на дочерние объекты и их свойства. Например, вместо **window.screen.colorDepth** можно использовать запись **screen.colorDepth**.

1

Создайте HTML-документ с пустым абзацем.

```
<p id="props" style="font:1.5em sans-serif"></p>
```

2

Создайте самовызывающуюся функцию. Объясните и проинициализируйте шесть переменных.

```
const info = document.getElementById( 'props' )
let width = window.screen.width + 'px'
let height = window.screen.height + 'px'
let availW = window.screen.availWidth + 'px'
let availH = window.screen.availHeight + 'px'
let colors = 'Unknown'
```

3

Добавьте операторы, описывающие способности к цветовоспроизведению.

```
switch(window.screen.colorDepth)
{
  case 8 : colors = 'Low Color' ; break
  case 16 : colors = 'High Color' ; break
  case 24 : colors = 'True Color' ; break
  case 32 : colors = 'Deep Color' ; break
}
```

4

Добавьте операторы для отображения полученных результатов.

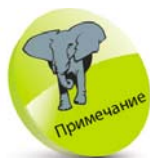
```
info.innerHTML = 'Screen Resolution: ' +
  width + ' x ' + height + '<br>'
info.innerHTML += 'Available Screen Size: ' +
  availW + ' x ' + availH + '<br>'
info.innerHTML += 'Color Capability: ' + colors + '<br>'
if( window.screen.orientation )
```



screen.html



Для браузеров с ограниченными цветовыми возможностями свойство **colorDepth** используется для изображений с низким разрешением.

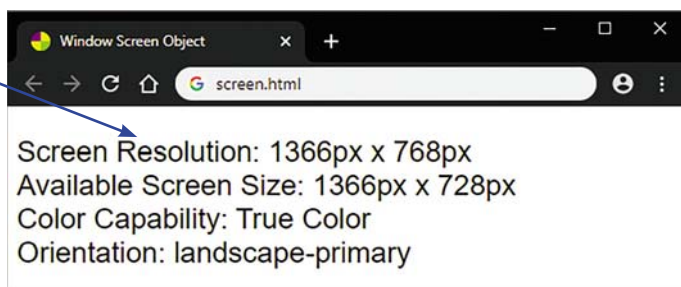


Доступная высота на 40 пикселей меньше высоты экрана, так как высота панели задач рабочего стола составляет 40 пикселей.

```
{
    info.innerHTML += 'Orientation: ' +
        window.screen.orientation.type
}
```

5

Сохраните HTML-документ, затем откройте его в браузере. Проанализируйте полученные результаты — общие сведения об экране.



Диалоговые окна

Существуют три метода, с помощью которых JavaScript отображает диалоговые сообщения для пользователя. Метод **window.alert()** показывает окно предупреждающего сообщения. Это окно с указанием сообщения и кнопкой ОК.

Метод **window.confirm()** отображает диалоговое окно с указанием сообщения, а также кнопок ОК и Cancel (Отмена). При нажатии любой из кнопок диалоговое окно закрывается. При нажатии на кнопку ОК возвращается значение **true**, а при нажатии на кнопку Cancel возвращается значение **false**.

Метод **window.prompt()** отображает диалоговое окно, предлагающее пользователю ввести данные. Это диалоговое окно с кнопкой ОК, Cancel и полем для ввода текста. При нажатии любой из них диалоговое окно закрывается. При нажатии на кнопку ОК возвращается значение в текстовом поле, а при нажатии на кнопку Cancel возвращается значение **null**. Второй аргумент также может быть передан методу **window.prompt()**, чтобы указать содержимое по умолчанию для текстового поля.

1

Создайте HTML-документ с пустым абзацем.

```
<p id="response" style="font:1.5em sans-serif"></p>
```



dialogs.html

2

Затем создайте самовывзывающуюся функцию. Объявите и проинициализируйте одну переменную.

```
const info = document.getElementById( 'response' )
```

3

Добавьте операторы для отображения сообщения в простом диалоговом окне.

```
window.alert( 'Hello from JavaScript' )
```

4

Затем добавьте операторы для запроса решения от пользователя и напишите ответ в текстовом поле.

```
info.innerHTML = 'Confirm: ' +  
window.confirm( 'Go or Stop?' )
```

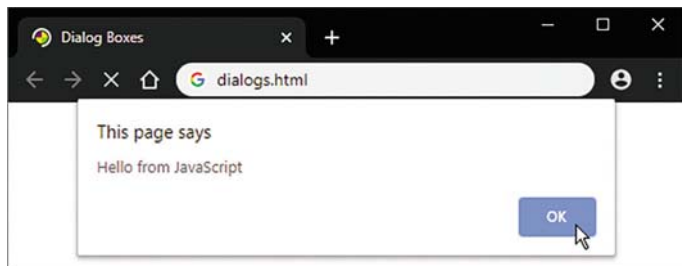
5

Добавьте операторы для запроса ввода текста от пользователя и запишите ответ в текстовом поле.

```
info.innerHTML += '<br>Prompt: ' +  
window.prompt( 'Yes or No?', 'Yes' )
```

6

Сохраните HTML-документ, затем откройте его в браузере. Проанализируйте полученные результаты — диалоговое окно.



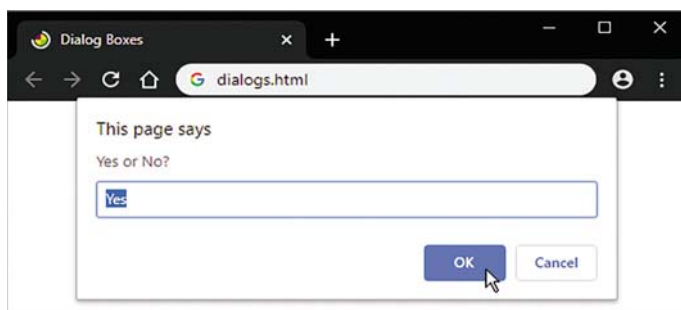
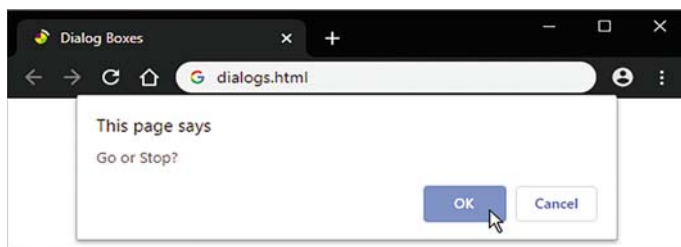


В программном коде создания диалогового окна подтверждения можно использовать оператор `if`. Например, `if (confirm('OK?')) {...} else {...}`.



Если в программном коде необходимо использовать ввод текста из диалогового окна запроса, рекомендуется удалить пробелы с обоих концов строки — см. стр. 121.

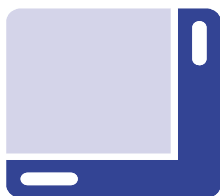
- 7 Для закрытия в каждом диалоговом окне нажмите кнопку ОК. Проанализируйте полученный результат.



Прокрутка

Метод `scrollBy()` позволяет прокручивать окно по горизонтали и вертикали, если содержимое выходит за его пределы. Параметры `X` и `Y` указывают смещение в пикселях по горизонтали и вертикали соответственно.

Если содержимое выходит за пределы окна по вертикали, вдоль правого края окна браузера появляется полоса прокрутки. Метод `scrollBy()` прокручивает страницу на указанное количество пикселей (указано в первом параметре) или до тех пор, пока не достигнет крайнего значения содержимого.



Если содержимое выходит за пределы окна по горизонтали, в нижней части окна браузера появляется полоса прокрутки. Метод **scrollBy()** будет прокручивать на определенное количество пикселей (указано во втором параметре) или до тех пор, пока не достигнет крайнего значения содержимого.

Существует также метод **scrollTo()**, который выполняет прокрутку документа до указанного координатора. Этот метод принимает два аргумента, определяющих координаты **X** и **Y**, до которых должен прокручиваться верхний левый угол окна, если содержимое выходит за его пределы по горизонтали и вертикали. Это можно использовать для перехода от координат по умолчанию **X=0, Y=0** к заданной позиции. Например, если данные в браузере отображаются в табличной форме, в первой ячейке первой строки в верхнем левом углу окна браузера, метод **scrollTo()** вместо этого может разместить определенную ячейку в верхнем левом углу окна браузера.

Объект **window** имеет свойство **scrollX**, возвращающее количество пикселей, в которых документ прокручивается по горизонтали в данный момент. Это обозначает положение ползунка вдоль полосы прокрутки в нижней части окна относительно его левого угла. Свойство **window.scrollX** — это альтернативное имя для более старого свойства **window.pageXOffset**, которое все еще существует в DOM, и может использоваться для обратной совместимости вместо **window.scrollX**.

Свойство **scrollY** возвращает количество пикселей, в которых документ прокручивается по вертикали в данный момент. Это обозначает положение ползунка на полосе прокрутки в правой части окна относительно его верхнего угла. Свойство **window.scrollY** — это альтернативное имя для более старого свойства **window.pageYOffset**, которое все еще существует в DOM, и может использоваться для обратной совместимости вместо **window.scrollY**.

1

Создайте HTML-документ с широким пустым абзацем, расположенным слева от окна.

```
<p id="info" style="width:2000px; margin-left:300px; font:1.2em sans-serif"></p>
```



Отрицательные значения, указанные в качестве параметров в методе **scrollBy()**, прокрутят страницу влево и вверх.



Эффект от метода **scrollby()** можно заметить только тогда, когда содержимое выходит за пределы окна, так как при этом появляются полосы прокрутки.



scroll.html



Обратите внимание на использование свойства элемента `clientHeight`. Существует также еще одно полезное свойство `clientWidth`.

- 2 Затем создайте самовызывающуюся функцию. Объявите и проинициализируйте две переменные.

```
const info = document.getElementById( 'info' )
let i = 0
```

- 3 Добавьте цикл для записи столбца из 40 чисел.

```
for( i = 1; i < 41; i++ )
{
    info.innerHTML += ( i + '<br>' )
}
```

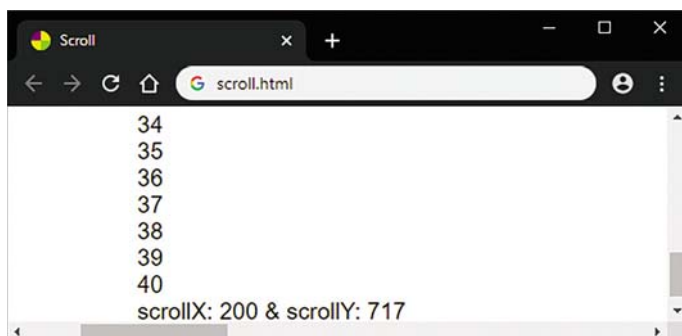
- 4 Затем добавьте операторы для прокрутки окна на 200 пикселей по горизонтали вправо и вертикально вниз по высоте элемента абзаца.

```
window.scrollBy( 200, info.clientHeight )
```

- 5 Наконец, вставьте оператор, чтобы добавить подтверждение положения ползунка в текущем окне.

```
info.innerHTML += 'scrollX: ' + window.scrollX +
    '& scrollY: ' + window.scrollY
```

- 6 Сохраните HTML-документ, затем откройте его в браузере. Проанализируйте полученные результаты — диалоговое окно с полосой прокрутки.



Всплывающие окна

С помощью метода `open()` можно открыть новое окно браузера. Для этого необходимо указать три параметра: URL-адрес документа, который будет загружен в новое окно, имя для нового окна и разделенный запятыми список свойств, описанных в таблице ниже.



Свойство	Описание
<code>directories</code>	Добавляет панель ссылок
<code>height</code>	Устанавливает высоту области документа в пикселях
<code>left</code>	Координата экрана окна X
<code>location</code>	Добавляет адресную строку
<code>menubar</code>	Добавляет стандартную строку меню
<code>resizable</code>	Позволяет изменять размер окна
<code>scrollbars</code>	При необходимости включает полосы прокрутки
<code>status</code>	Добавляет строку состояния
<code>toolbar</code>	Добавляет панель кнопок Forward (Вперед) и Back (Назад)
<code>top</code>	Координата экрана окна Y
<code>width</code>	Устанавливает ширину области документа в пикселях

При успешном результате метод `window.open()` возвращает новый объект окна и открывает новое всплывающее окно или в противном случае метод просто возвращает значение `null`. Полученный результат должен быть присвоен переменной, которая может быть впоследствии проверена. Если переменная не равна `null`, она должна представлять объект всплывающего окна. Затем с помощью метода `close()` это окно можно закрыть, а используя метод `print()`, — распечатать его содержимое.

Окна также можно расположить на экране, используя координаты осей экрана X и Y в качестве аргументов метода `window.moveTo()`. Метод `window.moveBy()` перемещает окно на указанное количество пикселей относительно его текущих координат.



Из-за большого и раздражающего количества всплывающих окон разработчики браузеров добавили блокировщики, поэтому использование всплывающих окон не рекомендуется. Но для общего сведения информация о них приведена в этой книге.



popup.html



opener.html



Не добавляйте в строку списка функций пробелы, так как это может привести к сбою в работе метода `window.open()`.



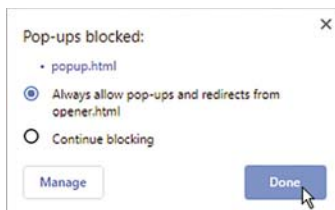
Обратите внимание, что всплывающее окно не отображает в строке заголовка указанный «фавикон» (значок веб-сайта).

- 1 Создайте HTML-документ, содержащий заголовок.
`<h1>Pop-up Window</h1>`
- 2 Создайте второй HTML-документ, содержащий заголовок.
`<h1>Main Window</h1>`
- 3 Затем во втором документе создайте самовызывающуюся функцию, которая создает объект окна.

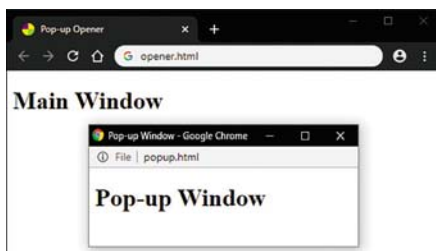
```
const popWin = window.open('popup.html', 'Popup',  
    'top=150, left=100, width=350, height=100')
```
- 4 Сохраните оба HTML-документа, затем откройте второй документ, чтобы увидеть, что его всплывающее окно заблокировано.



- 5 Откройте блокировщик всплывающих окон и разрешите их отображение с этой страницы.



- 6 Затем обновите окно браузера, чтобы увидеть всплывающее окно с указанными функциями.



Создание таймера

В JavaScript существует метод `setTimeout()`, который многократно оценивает выражение через определенное количество миллисекунд. Если указанное выражение вызывает функцию, где появляется оператор `window.setTimeout()`, создается рекурсивный цикл, в котором функция повторно выполняется по истечении указанного периода времени.

Выражение, выполняемое методом `setTimeout()`, указывается в качестве его первого аргумента, а период времени — это число, указанное в качестве второго аргумента. Время выражается в миллисекундах, где 1000 означает одну секунду.

Метод `setTimeout()` возвращает числовое значение, которое может быть присвоено переменной, чтобы однозначно идентифицировать ожидающий процесс. Это значение указывается в качестве аргумента метода `clearTimeout()` для прерывания цикла таймера в определенный момент.

Методы `setInterval()` и `clearInterval()` принимают идентичные аргументы и работают одинаково. Разница только в том, что время, указанное для `setInterval()`, определяет интервал, в котором выражение должно быть выполнено, независимо от требуемого времени. И наоборот, время, указанное в методе `setTimeout()`, определяет период времени между окончанием одного и началом следующего выполнения. Это означает, что метод `setInterval()` может попытаться выполнить перекрывающиеся выполнения, если интервал короткий, а время, необходимое для выполнения выражения, слишком велико. По этой причине рекомендуется использовать метод `setTimeout()`.

1

Создайте HTML-документ с пустым абзацем.

```
<p id="info"></p>
```



Двухминутная задача со значением времени ожидания в 10 минут запускается каждые 10 минут, но такая же задача со значением времени ожидания в 10 минут запускается каждые 12 минут (10+2).



timer.html



Дополнительную информацию о замыканиях можно найти на стр. 29.



Чтобы увидеть значение идентификатора таймера, можно к функции таймера добавить оператор `console.log(timerId)`.

2

Объявите и проинициализируйте переменную с помощью функции, которая возвращает целые числа в порядке убывания.

```
const count = ( function ( ) {
    let num = 10
    return (function( ) {return num- } )
} ) ( )
```

3

Добавьте функцию таймера. Объявите и проинициализируйте три переменные.

```
function countDown( )
{
    const info = document.getElementById( 'info' )
    let timerId = null
    let num = count( )
    // Здесь будет ваш код.
}
```

4

Затем добавьте операторы для запроса решения от пользователя и напишите ответ в текстовом поле.

```
if ( num > 0 )
{
    info.innerHTML += '<span>'+ num + '</span>'
    timerId = window.setTimeout( countDown , 1000 )
}
else
{
    info.innerHTML += '<span>Lift Off!</span>'
    window.clearTimeout( timerId )
}
```

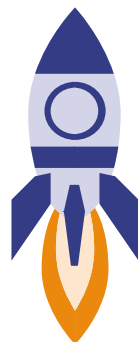
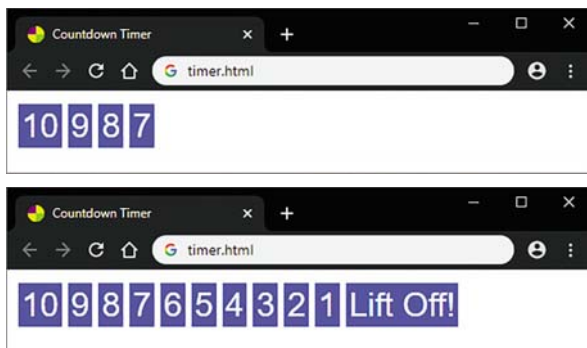
5

Добавьте после функционального блока оператор для вызова функции таймера после загрузки страницы.

```
countDown( )
```

6

Сохраните HTML-документ, затем откройте его в браузере. Проанализируйте полученные результаты — таймер отсчитывает 10 секунд.



Сведения о браузерах

Объект **window** имеет ряд дочерних объектов, у каждого из которых свои собственные свойства и методы. Например, свойство **window.navigator** содержит информацию о веб-браузере. Поскольку объект **window** — глобальный объект, его имя может быть опущено при ссылке на дочерние объекты и их свойства. Поэтому его можно просто вызывать, используя запись **alert()**. На свойство **window.onload** можно сослаться, используя запись **onload**, а на объект **window.navigator**, используя запись **navigator**.

Свойство **appName** возвращает имя браузера, свойство **appCodeName** возвращает его кодовое имя. С помощью свойства **appVersion** можно узнать версию браузера. Свойство **appCodeName** для любого браузера всегда возвращает значение «Mozilla».

При выполнении запроса к веб-серверу каждый браузер отправляет кодовое имя браузера и версию в заголовке HTTP с именем «User-Agent». Эту строку также можно получить из свойства **navigator.userAgent**. Свойство **navigator.platform** возвращает строку, представляющую платформу браузера.



Чтобы сделать очевидным происхождение методов и свойств объекта **window**, в предыдущих примерах использовался префикс **window**. Однако это не рекомендуется. Например, вместо записи **window.onload** рекомендуется использовать **onload**.



browser.html



Во всех современных веб-браузерах объект `window` включает метод `addEventListener()`. Более подробно об этом вы узнаете в главе 7.

Ранее большое внимание уделялось коду обнаружения браузера, с помощью которого пытались идентифицировать браузер, используя его свойство `navigator`. В настоящее время это считается неэффективным и рекомендуется использовать концепции обнаружения функций.

Например, запрос о том, поддерживает ли браузер важный метод `addEventListener()` и современную объектную модель документа.

- 1 Создайте HTML-документ с пустым списком.

```
<ul id="list"></ul>
```

- 2 Создайте самовызывающуюся функцию. Проинициализируйте ссылки на переменную.

```
const list = document.getElementById( 'list' )
```

- 3 Добавьте операторы для перечисления имен браузера.

```
list.innerHTML = '<li>Browser: ' + navigator.appName
list.innerHTML += '<li>Code Name: ' + navigator.
appCodeName
```

- 4 Затем добавьте операторы, чтобы извлечь сведения об используемой версии браузера и операционной системы.

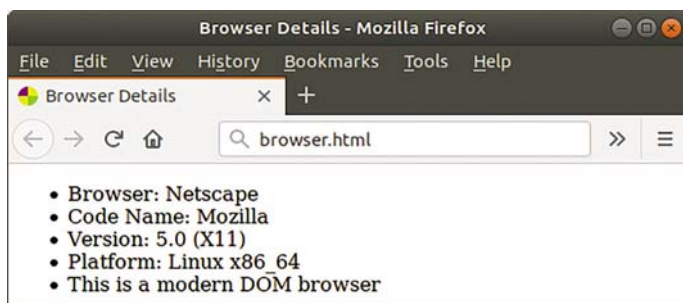
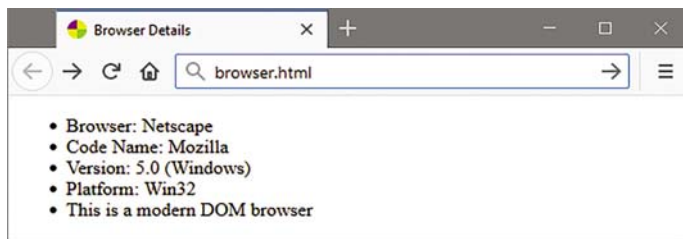
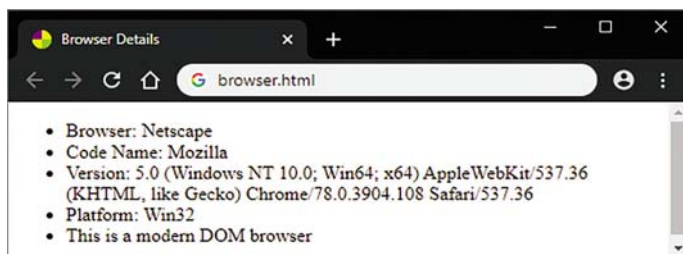
```
list.innerHTML += '<li>Version: ' + navigator.appVersion
list.innerHTML += '<li>Platform: ' + navigator.platform
```

- 5 Наконец, добавьте операторы, подтверждающие использование современного браузера.

```
if( window.addEventListener )
{
    list.innerHTML += '<li>This is a modern DOM browser'
}
```

- 6 Сохраните HTML-документ, затем откройте его в браузере. Проанализируйте полученные

результаты — информацию, содержащую название, версию и платформу браузера.



Причина, по которой Google Chrome и другие браузеры называют себя Netscape, Mozilla, относится к эпохе «браузерных войн». Браузеры должны были принимать эти названия, чтобы поддерживать все веб-страницы, загружаемые браузерами Netscape Mozilla.

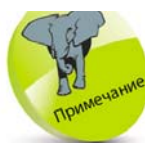
Включение/отключение функций

Дочерний объект **navigator** объекта **window** содержит метод **javaEnabled()**, возвращающий логическое значение **true**, если текущий браузер поддерживает Java.

Свойство **cookieEnabled** возвращает логическое значение **true**, если в браузере включены файлы cookie.

Также объект **navigator** содержит дочерние объекты **plugins** и **mimeTypes**. Как и в случае с другими объектами, они имеют свойство **length**, возвращающее количество элементов.





Содержимое этих элементов массива различается в зависимости от поддерживаемых браузером функций.



enabled.html

Каждый элемент массива подключаемых модулей (плагинов) имеет свойства **name** и **description**, содержащие сведения об одной установленной функции подключаемого модуля. Доступ к их элементам осуществляется с использованием порядкового номера элемента. Например, **navigator.plugins[0].name** ссылается на свойство **name** первого элемента в массиве **plugins**.

Точно так же каждый элемент массива **mimeTypes** имеет свойства **type** и **description**, содержащие сведения об одной поддерживаемой функции MIME. Доступ к их элементам осуществляется при использовании порядкового номера элемента. Например, **navigator.mimeTypes[0].type** ссылается на свойство **type** первого элемента в массиве **mimeTypes**.

- 1 Создайте HTML-документ с пустым абзацем.

```
<p id="info"></p>
```

- 2 Создайте самовызывающуюся функцию. Объявите и проинициализируйте две переменные.

```
const info = document.getElementById( 'info' )
let status = "
```

- 3 Для отображения подтверждающего сообщения в случае включения поддержки Java добавьте операторы.

```
status = (navigator.javaEnabled( ) ) ? 'Enabled' : 'Disabled'
info.innerHTML += 'Java Support is ' + status + '<hr>
```

- 4 Затем добавьте операторы для отображения подтверждающего сообщения в случае включения поддержки файлов cookie.

```
status = ( navigator.cookieEnabled )? 'Enabled' : Disabled'
info.innerHTML += 'Cookie Support is ' + status + '<hr>
```

- 5 Добавьте операторы для записи длины массива подключаемых модулей и примера элемента.

```

if ( navigator.plugins.length !== 0 )
{
    info.innerHTML += 'No. of Plugins: ' +
                      navigator.plugins.length
    info.innerHTML += '<br>Example: ' +
                      navigator.plugins[0].name
    info.innerHTML += '<br>For: ' +
                      navigator.plugins[0].description + '<hr>'
}

```

6

Наконец добавьте операторы для записи длины массива MIME и примера элемента.

```

if ( navigator.mimeTypes.length !== 0 )
{
    info.innerHTML += 'No. of MIME Types: ' +
                      navigator.mimeTypes.length
    info.innerHTML += '<br>Example: ' +
                      navigator.mimeTypes[1].type
    info.innerHTML += '<br>For: ' +
                      navigator.mimeTypes[1].description
}

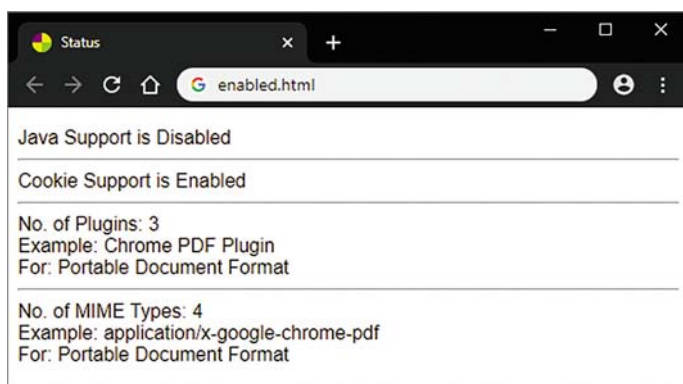
```

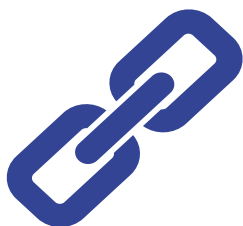
7

Сохраните HTML-документ, затем откройте его в браузере. Проанализируйте полученные результаты — отображения статуса разрешенных функций.



Вы можете использовать циклы для записи всех плагинов и содержимого элементов `mimeTypes`.





location.html



Веб-браузер может загружать файл любого поддерживаемого MIME типа — например, MIME-тип image/png для всех файлов изображений PNG.

Расположение

Объект **location** имеет пять свойств, содержащих компоненты полного URL-адреса документа, загруженного в текущий момент в окне браузера. Свойство **location.href** можно использовать, чтобы получить полный URL-адрес страницы (протокол, доменное имя, имя файла и при необходимости привязки URL-адреса). Отдельные компоненты полного адреса содержатся в протоколе **location.protocol** (http: или https:), **location.host** (доменное имя), **location.pathname** (путь к файлу) и **location.hash** (привязки URL-адреса). Назначение нового URL-адреса свойству **location** заставит браузер загружать страницу или другой ресурс по новому месту назначения.

- 1 Создайте HTML-документ с абзацем, содержащим гиперссылку.

```
<p id="info">
<a id="frag">Fragment Anchor</a>
</p>
```

- 2 Создайте самовызывающуюся функцию. Объявите и проинициализируйте две переменные.

```
const info = document.getElementById( 'info' )
let jump = confirm( 'Jump to Fragment?' )
```

- 3 Добавьте оператор для изменения местоположения окна при согласии пользователя.

```
if ( jump )
{
    location = location.href + '#frag'
}
```

- 4 Затем добавьте операторы для записи каждого компонента текущего URL-адреса.

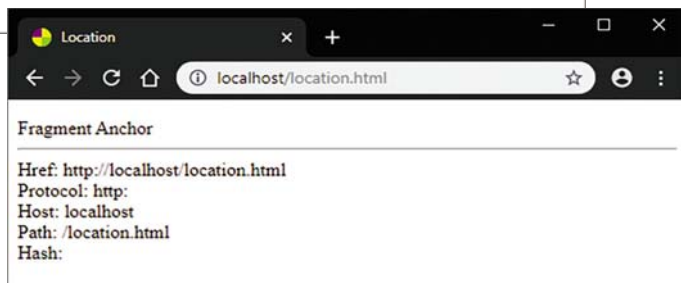
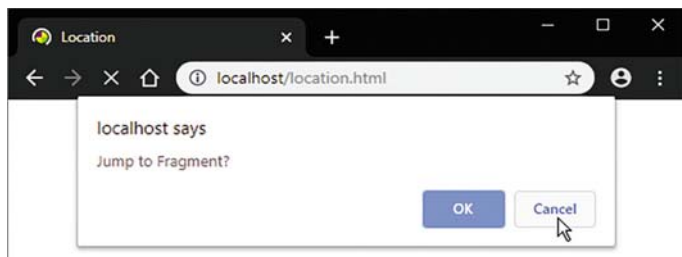
```
info.innerHTML += '<hr>Href: ' + location.href
info.innerHTML += '<br>Protocol: ' + location.protocol
info.innerHTML += '<br>Host: ' + location.host
info.innerHTML += '<br>Path: ' + location.pathname
info.innerHTML += '<br>Hash: ' + location.hash
```


5

Сохраните HTML-документ и откройте его в браузере. Проанализируйте полученные результаты — диалоговое окно с запросом на подтверждение изменения URL-адреса.

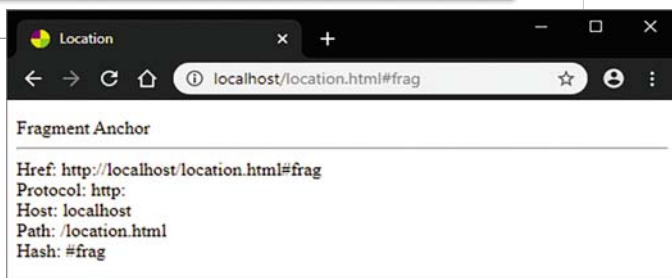
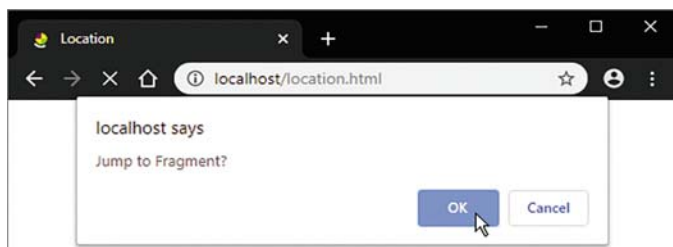
6

Нажмите кнопку Cancel (Отмена), чтобы отклонить запрос и увидеть загрузку страницы, как правило, в ее корневой папке.



7

Обновите браузер, затем нажмите кнопку ОК, чтобы принять запрос и увидеть загрузку страницы в месте расположения ее фрагмента.



Рассмотренное нами расположение — это страница, находящаяся на веб-сервере в локальной системе. Если бы страница была расположена на вашем рабочем столе, протоколом было бы **file:**, а значение **host** отсутствовало бы.




page-1.html
page-2.html
page-3.html



history.js



Чтобы очистить историю браузера, в браузере Google Chrome нажмите кнопку , затем выберите пункты **More tools** (Дополнительные инструменты), **Clear browsing data** (Очистить данные просмотра) и нажмите кнопку **Clear data** (Очистить данные).

История

Веб-браузер сохраняет историю URL-адресов текущей сессии в виде массива в дочернем объекте **history**. Как и в случае с массивами, объект **history** имеет свойство **length**, а также методы **back()** и **forward()** для перемещения между элементами. В качестве альтернативы метод **go()** объекта **history** принимает положительный или отрицательный целочисленный аргумент, указывающий, на какое количество элементов перемещаться. Например, **history.go(1)** выполняет перемещение на один элемент вперед, а **history.go(-2)** перемещает на два элемента назад.

- 1 Создайте три идентичных HTML-документа, содержащих пустой абзац, и вставьте один и тот же внешний файл сценария.

```
<p id="info" > </p>
<script src="history.js" > </script>
```

- 2 Создайте самовызывающуюся функцию. Объявите и проинициализируйте переменную.

```
const info = document.getElementById( 'info' )
```

- 3 Добавьте операторы для записи содержимого в пустые строки.

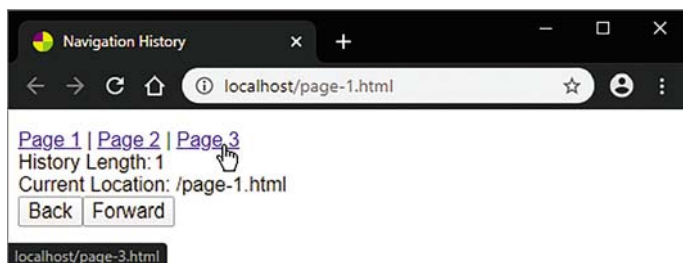
```
info.innerHTML += '<a href="page-1.html">Page 1</a> | '
info.innerHTML += '<a href="page-2.html">Page 2</a> | '
info.innerHTML += '<a href="page-3.html">Page 3</a> '
info.innerHTML += '<br>History Length: '
                                     + history.length
info.innerHTML += '<br>Current Location: ' +
                                     location.pathname + '<br>'
```

- 4 Затем добавьте операторы для создания кнопок в абзацах.

```
info.innerHTML += '<button onclick="history.
                                     back( )">Back</button>'
info.innerHTML += '<button onclick="history.
                                     forward( )">Forward</button>'
```

5 Сохраните в одной папке HTML-документ и файл сценария JavaScript, затем очистите историю браузера.

6 Откройте первую страницу, чтобы увидеть, что начальная длина истории равна 1.



URL-адреса хранятся в элементах массива объекта `history` в защищенном виде, поэтому их невозможно получить в виде строки.

7 Щелкните ссылку, чтобы загрузить третью страницу и увидеть, как длина истории увеличивается до 2.



8 Для возврата на первую страницу нажмите кнопку Back (Назад). Длина истории останется такой же — 2.



В этом примере каждый URL-адрес добавляется в массив `history` только при переходе на другую страницу и при нажатии на ссылку. Методы `back()`, `forward()` и `go()` просто выбирают элемент в `history`, поэтому свойство `length` остается неизменным.

Заключение

- Объектная модель документа (DOM) — это иерархическая структура для всех компонентов веб-страницы.
- Объект **window** — верхний уровень иерархии DOM и содержит свойства, описывающие окно браузера.
- Объект **screen** — дочерний объект по отношению к объекту **window** и содержит свойства, описывающие размеры экрана и глубину цвета.
- Объект **window** содержит методы **scrollBy()** и **scrollTo()**, а также свойства **scrollX** и **scrollY**, определяющие позицию прокрутки.
- Диалоговые сообщения могут отображаться при использовании методов **alert()**, **confirm()** и **prompt()**.
- Всплывающее окно создается с помощью метода **open()** и может быть заблокировано блокировщиком всплывающих окон.
- Метод **setTimeout()** создает таймер, который можно отменить с помощью метода **clearTimeout()**.
- Объект **navigator** — дочерний объект по отношению к объекту **window** и содержит свойства, описывающие версии браузера и серверные платформы.
- Объект **window** существует в глобальном пространстве имен, поэтому все его дочерние объекты не обязательно должны включать префикс **window**.
- Обнаружение функций используется для идентификации современной модели DOM.
- Свойства **navigator.plugins** и **navigator.mimeTypes** представляют собой массивы, содержащие сведения о поддерживаемых функциях.

- Объект **location** — дочерний объект по отношению к объекту **window** и содержит свойства, описывающие адрес загруженного документа.
- Объект **history** — дочерний объект по отношению к объекту **window**, содержащий массив посещенных адресов текущей сессии.
- Объект **history** содержит методы **back()**, **forward()** и **go()**, которые используются для перемещения по страницам в текущей сессии.

7

Методы и свойства объекта document

*В этой главе вы изучите,
как использовать
в JavaScript свойства
и методы объекта document.*

- 150 Работа с документом
- 152 Свойства интерфейса Document
- 154 Получение элементов
- 156 Работа с текстом
- 159 Управление файлами cookie
- 161 События загрузки
- 164 Ответ на события мыши
- 166 Генерация событий
- 168 Добавление переключателей
- 170 Добавление элементов выбора
- 172 Сброс формы
- 175 Проверка и отправка формы
- 177 Заключение



Работа с документом

Объект **document** — наиболее интересный из всех дочерних объектов DOM. Он обеспечивает доступ к HTML-документу.

Объект **document** определяет ряд свойств, описывающих документ и его расположение.

- Свойство `document.title` устанавливает или возвращает заголовок текущего HTML-документа.
- Свойство `location.href` устанавливает или возвращает полный URL-адрес текущей страницы.
- Свойство `document.domain` возвращает доменное имя сервера, на котором загружен текущий документ, аналогично значению `location.host`.
- HTML-документы предоставляют браузеру дату своего создания или последнего изменения в виде заголовка HTTP, чтобы браузер мог решить, использовать ли кэшированную копию документа или новый документ. Также дату и время последнего изменения текущего документа можно получить, используя свойство `document.lastModified`.
- Свойство `document.referrer` возвращает URL-адрес документа, в который загружен текущий документ. Оно используется, только если пользователь перешел по гиперссылке для загрузки страницы, а не ввел URL-адрес или использовал какой-либо другой метод.



info-1..html



info-1..html

- 1 Создайте HTML-документ, содержащий в основном разделе гиперссылку на второй (целевой) HTML-документ.

```
<p>
<a href="info-2.html" >Link to the Next Page</a>
</p>
```

- 2 Создайте целевой HTML-документ, содержащий пустой неупорядоченный список.

```
<ul id="list"></ul>
```

3

Создайте самовызывающуюся функцию, проинициализируйте ссылку на переменную.

```
const list = document.getElementById( 'list' )
```

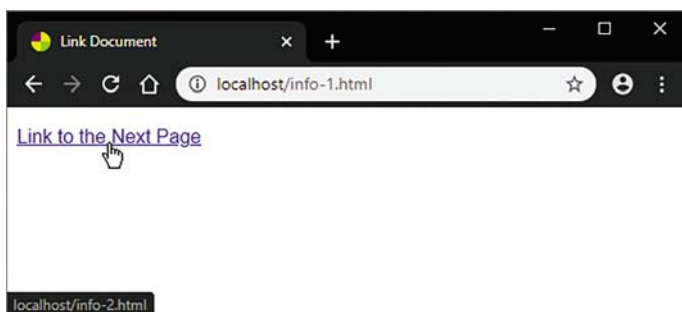
4

Для перечисления свойств документа добавьте операторы.

```
list.innerHTML = '<li>Linked From: ' + document.referrer
list.innerHTML += '<li>Title: ' + document.title
list.innerHTML += '<li>URL: ' + document.URL
list.innerHTML += '<li>Domain: ' + document.domain
list.innerHTML += '<li>Last Modified: ' +
    document.lastModified
```

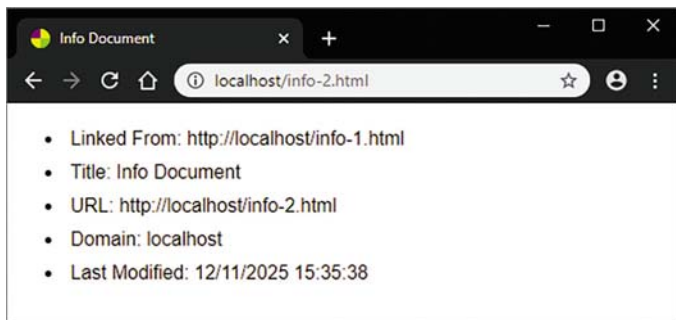
5

Сохраните HTML-документы в той же папке, затем откройте документ, содержащий ссылку, в своем браузере.



6

Щелкните гиперссылку, чтобы загрузить второй HTML-документ в браузере и проанализируйте полученную информацию о документе.



Дата, содержащаяся в свойстве `document.lastModified`, относится только к самому HTML-документу, а не к каким-либо внешним таблицам стилей или внешним файлам сценариев.



Показанный здесь домен — это домен страницы, расположенной в локальной системе на веб-сервере. В случае расположения страницы на вашем рабочем столе свойства `document.referrer` и `document.domain` не имели бы никаких значений.

Свойства интерфейса Document



Обратите внимание на стиль написания прописных букв в программировании camelCase в свойстве `styleSheets`.



components..html



user.png
64px x 64px
(серые области
прозрачны)



bg.png
24px x
100px

Объект **document** содержит следующие дочерние объекты: **forms**, **images**, **links**, **styleSheets** и **scripts**. Каждый из них — это массив, в котором всякий элемент представляет компонент документа в том же порядке, в котором они появляются в HTML-документе. Например, первое изображение, заданное тегом HTML ``, представлено как `document.images[0]`. Это означает, что на его URL-адрес можно ссылаться, используя запись `document.images[0].src`, которая определяет путь, назначенный атрибуту `src` тега HTML ``. Новый URL-адрес в сценарии будет динамически заменять старое изображение новым.

Массив **links** в HTML-документе представляет коллекцию всех элементов `<a>`; свойство **styleSheets** — `<style>`; а свойство **scripts** — `<script>`.

Массив **forms** представляет коллекцию HTML со списком всех элементов `<form>`, содержащихся в текущем документе. Однако также имеет собственные свойства — массив всех компонентов формы. Например, доступ к первому компоненту первой формы в документе HTML можно получить с помощью записи `document.forms[0].elements[0].value`. Присвоение этому компоненту нового значения в сценарии динамически заменяет старое значение.

1

Создайте HTML-документ, содержащий форму и пустой список.

```
<form>

<input type="text" name="topic" size="30"
      value="Type Your Question Here" >
<input type="button" value="Ask a Question" > <br>
<a href="formhelp.html" style="margin:5px">Help?</a>
</form>
<ul id="list"></ul>
```

2

Для стилизации шрифта и формы добавьте таблицу стилей.

```
< style>
* { font : 1em sans-serif ; }
form { width : 500px ; height : 100px ;
background: url(bg.png) ; }

</style>
```

3

Затем создайте самовывзывающуюся функцию. Объявите и проинициализируйте ссылку на переменную.

```
const list = document.getElementById( 'list' )
```

4

Добавьте в список компонентов документа следующие операторы.

```
list.innerHTML = '<li>No. Forms: ' +
document.forms.length
list.innerHTML += '<li>No. Links: ' +
document.links.length
list.innerHTML += '<li>No. Images: ' +
document.images.length
list.innerHTML += '<li>No. Style Sheets: ' + document.
styleSheets.length
list.innerHTML += '<li>No. Scripts: ' +
document.scripts.length
```

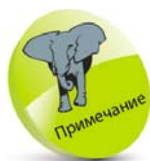
5

Наконец, добавьте операторы, чтобы перечислить два значения атрибутов.

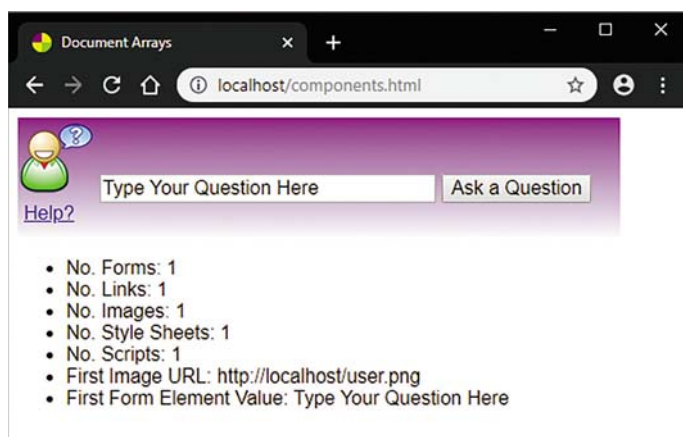
```
list.innerHTML += '<li>First Image URL: ' + document.
images[ 0 ].src
list.innerHTML += '<li>First Form Element Value: ' +
document.forms[ 0 ].elements[ 0 ].value
```

6

Сохраните HTML-документ, затем откройте его в браузере. Проанализируйте полученные результаты — отображение компонентов документа в виде списка.



Элементы массива представляют только соответствующие теги HTML.

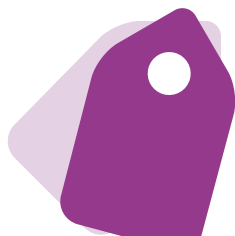


Заметьте, что изображения, добавленные в документ согласно правилам стиля, не включаются в массив **images**; в него входят только те, которые включены с помощью тегов HTML ****. Поэтому в нашем примере фоновое изображение формы (**bg.png**) не отображается в массиве **images**. Точно так же правило стиля работает с массивом **styleSheets**.

Получение элементов

Использование для ссылки на конкретный элемент точечной нотации предполагает, что разработчик подсчитает количество компонентов для вычисления каждой позиции индекса. Это особенно утомительно для работы с объемными документами, так как возникает риск появления ошибок, а при редактировании HTML-документа может измениться позиция индекса элемента. Прежде это было необходимо, но позднее к объекту **document** было решено добавить три новых метода:

- Метод **document.getElementById()**, использовавшийся в предыдущих примерах для добавления содержимого из кода JavaScript, позволяет ссылаться на любой компонент по его значению атрибута HTML **id**. Метод **getElementById()** возвращает элемент, имеющий атрибут **ID** с указанным значением.



- Метод `document.getElementsByTagName()` возвращает коллекцию всех элементов в документе с указанным именем тега в виде объекта `HTMLCollection`. Доступ к определенному элементу HTML осуществляется при использовании его порядкового номера.
- Метод `document.getElementsByClassName()` возвращает коллекцию всех элементов в документе с указанным именем класса в виде объекта `HTMLCollection`. Доступ к определенному элементу HTML осуществляется с использованием его порядкового номера.



Обратите внимание на название двух методов. В названии одного метода присутствует слово «Elements» (во множественном числе), а в имени другого метода — «Element» (в единственном числе).

- 1 Создайте HTML-документ, содержащий два списка и пустой абзац.

```
<ol>
<li class="fruit">Apple</li>
<li class="nut">Almond</li>
<li class="fruit">Apricot</li>
</ol>

<ol>
<li class="fruit">Blackberry</li>
<li id="country" class="nut">Brazil</li>
<li class="fruit">Banana</li>
</ol>
```



collection..html

- 2 Создайте самовызывающуюся функцию. Объявите и проинициализируйте пять переменных.

```
const info = document.getElementById( 'info' )
const item = document.getElementById( 'country' )
const lists = document.getElementsByTagName( 'ol' )
const fruits =
    document.getElementsByClassName( 'fruit' )

let i = 0
```

- 3 Добавьте оператор для описания объекта элемента и текста, содержащегося в его элементах.

```
info.innerHTML = item + ' Id: ' +
    item.innerText + '<br>
```



Свойство `innerText` устанавливает или возвращает содержимое указанного узла и всех его потомков. Свойство `innerHTML` устанавливает или возвращает содержимое HTML (внутренний HTML) элемента. Чтобы увидеть разницу, просто измените свойство `innerText` на `innerHTML`.

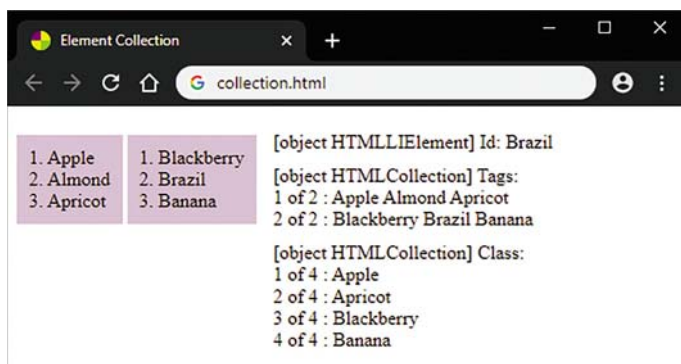
- 4 Затем добавьте операторы для описания объекта `HTMLCollection` и текста, содержащегося в его элементах.

```
info.innerHTML += '<br>' + lists + ' Tags:<br>'
for( i = 0; i < lists.length; i++ ) {
    info.innerHTML += ( i + 1 ) + ' of ' + lists.length
    info.innerHTML += ': ' + lists[ i ].innerText + '<br>'
}
```

- 5 Наконец, добавьте операторы для описания второго объекта `HTMLCollection` и текста, содержащегося в его элементах.

```
info.innerHTML += '<br>' + fruits + ' Class:<br>'
for( i = 0; i < fruits.length; i++ ) {
    info.innerHTML += ( i + 1 ) + ' of ' + fruits.length
    info.innerHTML += ': ' + fruits[ i ].innerText + '<br>'
}
```

- 6 Сохраните HTML-документ, затем откройте его в браузере. Проанализируйте результаты — значения элементов, полученные разными методами.



Работа с текстом

Как уже рассматривалось в предыдущих примерах, свойства `innerHTML` и `innerText` объекта `document` могут использоваться для написания текста в существующих элементах. Объект `document` также содержит



метод **write()**, который предоставляет еще один способ записи текста, но он автоматически вызывает метод **document.open()** для запуска нового документа, поэтому текущий документ больше не отображается.

Полезным считается то, что метод **document.createElement()** принимает имя тега в качестве аргумента и создает элемент этого типа. Затем содержимое может быть добавлено к новому элементу посредством присвоения его свойствам **innerHTML** или **innerText**. Новый элемент можно вставить на веб-страницу в существующий, указав его в качестве аргумента метода **document.appendChild()**.

Вы также можете динамически записывать атрибуты в элементы, указав имя и значение атрибута в качестве двух аргументов метода **setAttribute()**.

1

Создайте HTML-документ, содержащий заголовок и упорядоченный список из трех элементов.

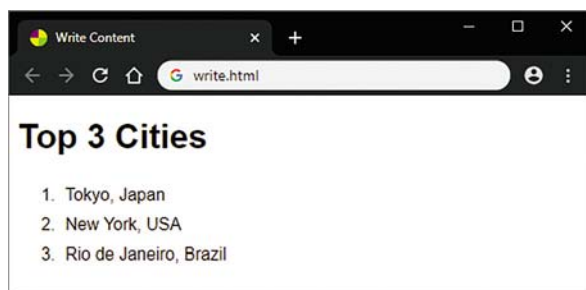
```
<h1 id="heading">Top 3 Cities</h1>
<ol id="list">
  <li>Tokyo, Japan
  <li>New York, USA
  <li>Rio de Janeiro, Brazil
</ol>
```



iwwrite.html

2

Сохраните HTML-документ и откройте его в браузере, чтобы просмотреть веб-страницу.



3

Создайте самовывзывающуюся функцию. Объявите и проинициализируйте две переменные в качестве новых элементов списка и одну ссылку на переменную.



Существующий элемент можно удалить, указав его в качестве аргумента метода `document.removeChild()`, или заменить, указав новые и старые элементы в качестве аргументов метода `document.replaceChild()`.

- 4 Присвойте двум новым элементам текстовое содержимое.

```
const itemFour = document.createElement( 'li' )
const itemFive = document.createElement( 'li' )
const heading = document.getElementById( 'heading' )
```

- 5 Вставьте на веб-страницу содержимое нового элемента в качестве дочерних элементов упорядоченного элемента списка.

```
document.getElementById( 'list' ).
    appendChild( itemFour )
document.getElementById( 'list' ).
    appendChild( itemFive )
```

- 6 Добавьте к существующему элементу заголовка атрибут, чтобы изменить его цвет шрифта.

```
heading.setAttribute( 'style', 'color: Red' )
```

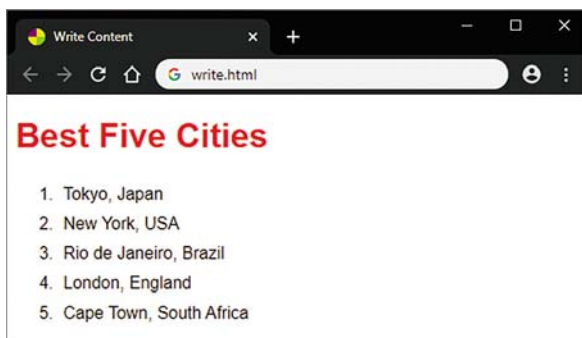
- 7 Наконец, чтобы лучше описать расширенный список, измените заголовок.

```
heading.innerHTML = 'Best Five Cities'
```

- 8 Снова сохраните HTML-документ, затем обновите страницу в браузере, чтобы увидеть новый текст.



Для удаления атрибута укажите его имя в качестве аргумента метода `removeAttribute()`.



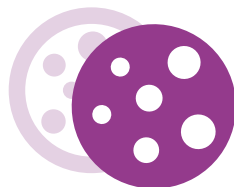
Управление файлами cookie

По соображениям безопасности в JavaScript не предусмотрена запись обычных файлов на жесткий диск пользователя, но для хранения небольшого количества данных существует запись файлов cookie. Файлы cookie ограничены по размеру и количеству — до 4 КБ и по 20 файлов соответственно на каждый веб-сервер. Как правило, данные, хранящиеся в файле cookie, идентифицируют пользователя для последующих посещений веб-сайта.

Данные cookie хранятся в свойстве **cookie** объекта **document** в виде одной или нескольких пар **key=value** (атрибут=значение), оканчивающихся символом;. Значение не может содержать пробелов, запятых или точек с запятой, если оно не передано в качестве аргумента встроенной функции **encodeURIComponent()**, которая кодирует строку в формате Unicode — например,%20.

По умолчанию, если дата истечения срока действия не указана, то он ограничен текущей сессией браузера. Пара **expires=date** (срок действия=дата) устанавливает дату истечения срока хранения файлов cookie. Метод **toUTCString()** преобразует дату в строке, используя часовой пояс UTC. Установка даты истечения срока действия существующего файла cookie на прошлую дату приведет к его удалению.

Чтобы получить данные из файла cookie, необходимо выполнить некоторые действия со строкой для возврата Unicode в обычный текст с использованием встроенной функции **decodeURI()**. Поэтому%20 снова становится пробелом и будет разделять элементы данных имени и значения. В строке cookie несколько пар можно разделить, указав символ; в качестве аргумента метода **split()**. Точно так же разделяются атрибуты и значения — указывается символ = в качестве аргумента метода **split()**. Аналогично если значение представляет собой список элементов, разделенных запятыми. Чтобы разделить список



Дата истечения срока действия cookie, как правило, не может быть прочитана JavaScript. При необходимости добавьте ее в список значений cookie.



Длинное число, используемое для установки срока действия, — это количество миллисекунд одного дня.

элементов, подобно элементам массива, символ запятой, также можно указать в качестве аргумента.

Полезно создать внешний файл JavaScript, содержащий служебные функции «сеттер» — для записи и «геттер» — для чтения, которые можно легко вызвать для хранения и извлечения данных cookie.



cookie.js

- 1 Создайте файл JavaScript с помощью функции установки, которая содержит параметры для атрибутов, значений и аргументов срока действия.

```
function setCookie( key, value, days ) {
    const d = new Date( )
    d.setTime( d.getTime( ) + ( days * 86400000 ) )
    document.cookie = key + '=' + encodeURIComponent( value ) +
        '; expires=' + d.toUTCString( ) + ';'
}
```

- 2 Чтобы принять ключевой аргумент, добавьте функцию «геттер».

```
function getCookie( key ) {
    if( document.cookie )
    {
        const pairs =
            decodeURI( document.cookie ).split( ';' )

        let i, name, value
        for( i = 0; i < pairs.length; i++ )
        {
            name = ( pairs[ i ].split( '=' )[ 0 ] ).trim( )
            if( name === key ) { value =
                pairs[ i ].split( '=' )[ 1 ] }
        }
        return value
    }
}
```



Обратите внимание, что в нашем примере для удаления пробелов используется метод `trim()`.



cookie.html

- 3 Создайте HTML-документ, содержащий пустой список, и импортируйте внешний файл JavaScript.

```
<ul id="list"></ul> <script src="cookie.js"></script>
```

4

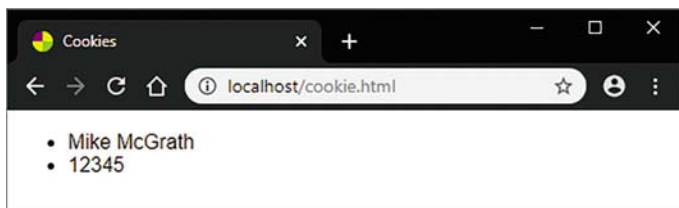
В другом элементе скрипта создайте самовызывающуюся функцию, которая устанавливает файл cookie, а затем получает его значения.

```
setCookie( 'User','Mike McGrath,12345', 7 )
```

```
const list = document.getElementById( 'list' )
let i, value = getCookie( 'User' )
if( value.indexOf( ',' ) )
{
    value = value.split( ',' )
}
for( i = 0; i < value.length; i++ )
{
    list.innerHTML += '<li>' + value[ i ]
}
}
```

5

Сохраните HTML-документ и файл JavaScript в одной папке на веб-сервере, затем откройте веб-страницу в своем браузере и проанализируйте результаты — полученные значения файлов cookie.

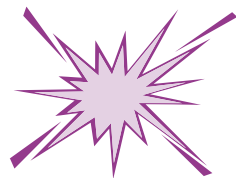


Файл cookie можно удалить, установив дату истечения срока его действия равной дате, предшествующей текущей фактической дате.

События загрузки

Модель DOM позволяет JavaScript реагировать на «события», которые происходят на веб-странице посредством действий разработчиков, предоставляя функции, которые будут выполняться при наступлении определенного события. Эти функции известны как «обработчики событий»:

- **load** — срабатывает, когда страница загружается в браузере.





Также существует событие выгрузки, которое запускается, когда пользователь покидает страницу. Его обработчик событий может быть присвоен свойству `window.onunload` или может быть указан в методе `addEventListener()`.



load..html

- **click** — срабатывает, когда пользователь нажимает кнопку мыши.
- **keydown** — срабатывает, когда пользователь нажимает клавишу клавиатуры.
- **change** — срабатывает, когда пользователь редактирует поле ввода.
- **submit** — срабатывает, когда пользователь отправляет HTML-форму.

Чтобы отреагировать на событие загрузки, обработчик событий может быть привязан к свойству **onload** объекта **window**, используя следующий синтаксис:

onload=имя-функции

В качестве альтернативы имя события и имя функции обработчика событий можно указать в качестве аргументов метода **addEventListener()** объекта **window**. Имя события должно быть заключено в кавычки, например:

addEventListener('load' , имя-функции)

Обработчик события загрузки используется для проверки функций браузера и данных cookie.

- 1 Создайте HTML-документ, который содержит абзац со ссылкой на пример файлов cookie, приведенный на стр. 162, пустой абзац, а также импортирует внешний файл JavaScript из предыдущего примера.


```
<p><a href="cookie.html">Link</a></p>
<p id="info"></p>
<script src="cookie.js"></script>
```
- 2 Затем добавьте обработчик событий, который будет выполняться при загрузке страницы.


```
addEventListener( 'load', welcome )
```
- 3 Чтобы приветствовать пользователя, добавьте следующий обработчик событий.

```
function welcome() {
  const info = document.getElementById( 'info' )
  if( getCookie( 'Name' ) )
  {
    info.innerHTML = 'Welcome Back, ' +
                      getCookie( 'Name' )
  }
  else
  {
    let name = prompt( 'Please Enter Your Name, 'User' )
    setCookie( 'Name', name, 7 )
    info.innerHTML = 'Welcome, ' + name
  }
}
```



Второй аргумент
метода `prompt()` —
это входное значе-
ние по умолчанию.

4

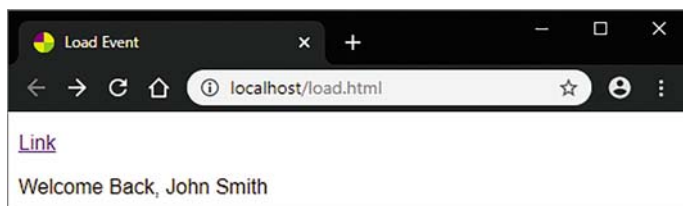
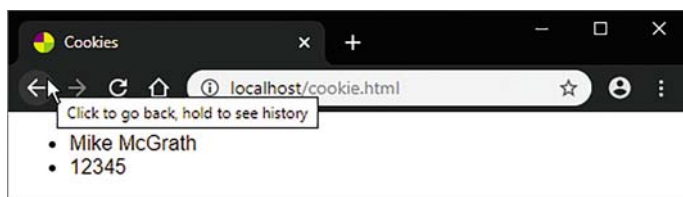
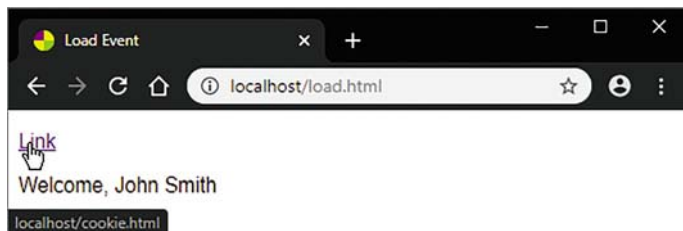
Сохраните HTML-документ и откройте его
в браузере — введите свое имя, перейдите
по ссылке и нажмите кнопку возврата.

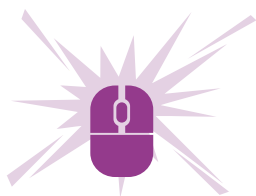
localhost says

Please Enter Your Name

John Smith

OK Cancel





Любой видимый объект в HTML-документе содержит обработчик событий, поэтому может быть реализован как кнопка. Большинство разработчиков предпочитают привязать обработчик событий, используя метод `addEventListener()`, а не свойство `onclick`.



mouse.html

Ответ на события мыши

Обработчики событий, которые выполняются при нажатии пользователем на определенный объект в HTML-документе, назначаются путем присвоения имени функции свойствам объекта **onclick** и **ondblclick**. Эти свойства реагируют на событие «click», которое срабатывает, когда пользователь щелкает кнопку мыши один раз, и событие «dblclick» при двойном щелчке.

Кроме того, свойства объекта **onmousedown** и **onmouseup** могут привязать обработчики событий, которые будут выполняться при нажатии кнопки мыши, вызывая событие «mousedown», а при отпускании — событие «mouseup».

Свойства объекта **onmouseover** и **onmouseout** привязывают обработчики событий, которые будут выполняться при перемещении указателя мыши на элемент, вызывая событие «mouseover», а при перемещении указателя мыши за пределы изображения — «mouseout». Такие события обычно используются для создания эффекта от наведения курсора, например, для изменения значения цвета свойства **style.background** на другое.

В качестве альтернативы имя события и имя функции обработчика событий можно указать в качестве аргументов метода **addEventListener()**. Такой метод используется аналогично методу **addEventListener()** объекта **window**, чтобы привязать функцию по имени или указать в качестве второго аргумента.

Когда выполняется событие, объект **event** передается встроенной функции обработчика событий. Существует свойство **type**, которое идентифицирует имя этого события.

1

Создайте HTML-документ, содержащий два абзаца и кнопку.

```
<p id="box">Target</p>
<p id="info">Place Mouse Over Target</p>
<button id="btn">Click Me</button>
```

2

Создайте самовызывающуюся функцию. Объявите и проинициализируйте две переменные.

```
( function () {
  const box = document.getElementById( 'box' )
  const btn = document.getElementById( 'btn' )
  // Здесь будет ваш код.
} )()
```

3

Добавьте операторы, чтобы привязать обработчики событий, которые будут передавать аргументы второй функции.

```
box.addEventListener( 'mouseover',
  function ( event ) { reactTo( event, 'Red' ) } )
box.addEventListener( 'mouseout',
  function ( event ) { reactTo( event, 'Purple' ) } )
box.addEventListener( 'mousedown',
  function ( event ) { reactTo( event, 'Green' ) } )
box.addEventListener( 'mouseup',
  function ( event ) { reactTo( event, 'Blue' ) } )
btn.addEventListener( 'click',
  function ( event ) { reactTo( event, 'Orange' ) } )
```

4

Наконец, добавьте вторую функцию для отображения типа события. Затем измените цвет фона первого абзаца.

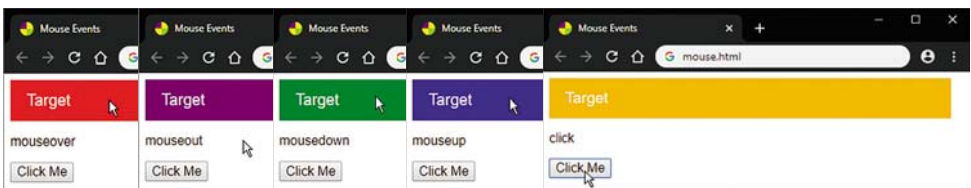
```
function reactTo( event, color ) {
  document.getElementById( 'box' ).style. background =
  color
  document.getElementById( 'info' ).innerText = event.
  type
}
```

5

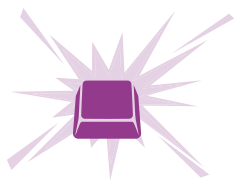
Сохраните HTML-документ и откройте его в браузере. Проанализируйте полученные результаты — события и реакции на них.



Во избежание утечки памяти при удалении объекта, к которому привязаны обработчики событий, следует также удалить его обработчики событий.



Генерация событий



Значения Unicode для общих символов такие же, как в коде ASCII, где значения строчных букв a–z равны 65–90, а значения прописных букв A–Z равны 97–122.



values.html

В дополнение к событиям мыши, описанным на стр. 164–165, существует событие «mousemove», которое передает объект события в функцию-обработчик со свойствами **x** и **y**, определяющими текущие координаты указателя мыши.

Также существует событие «keydown», срабатывающее при первом нажатии на клавишу, событие «keypress», срабатывающее, когда клавиша нажата, и событие «keyup», срабатывающее при отпускании клавиши. Каждое из них передает объект события в функцию-обработчик со свойством **type**, которое идентифицирует имя события, и свойством **keyCode**, в котором хранится числовое значение последней нажатой клавиши.

Числовое значение атрибута — это его значение Unicode, которое можно указать в качестве аргумента метода **String.fromCharCode()** для преобразования его в символьное значение.

Обработчики событий могут быть привязаны с помощью свойств **onmousemove**, **onkeydown**, **onkeypress** и **onkeyup** или указаны в качестве аргументов метода **addEventListener()**.

- 1 Создайте HTML-документ, содержащий пустой абзац.

```
<p id="info"></p>
```

- 2 Создайте самовывзывающуюся функцию, которая привязывает обработчик для трех событий и передает аргумент события этой функции.

```
(function () {
  document.addEventListener( 'keydown',
    function ( event ){ reactTo( event ) } )
  document.addEventListener( 'keyup',
    function ( event ){ reactTo( event ) } )
  document.addEventListener( 'mousemove',
    function ( event ){ reactTo( event ) } )
})()
```

3

В функции обработчика событий объявите и проинициализируйте переменную.

```
function reactTo( event ) {
  const info = document.getElementById( 'info' )
  // Здесь будет ваш код.
}
```

4

Затем добавьте операторы для отображения значений координат текущего положения указателя мыши.

```
if( event.type === 'mousemove' )
{
  info.innerHTML =
    'Mouse pointer is at X:' + event.x + ' Y:' + event.y
}
```

5

Добавьте операторы для отображения значения Unicode клавиши клавиатуры при ее нажатии.

```
if( event.type === 'keydown' )
{
  info.innerHTML += '<hr>' + event.type
  info.innerHTML += ':' + event.keyCode
}
```

6

Наконец, добавьте операторы для отображения символа этой же клавиши клавиатуры при ее отпускании.

```
if( event.type === 'keyup' )
{
  info.innerHTML += '<br>' + event.type + ':' +
    String.fromCharCode( event.keyCode ) + '<hr>'
}
```

7

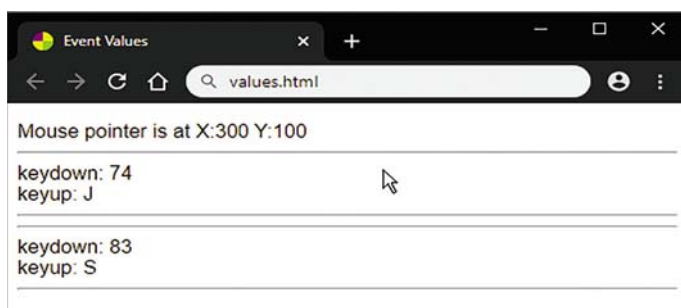
Сохраните HTML-документ и откройте его в браузере. Наведите указатель мыши на окно и проанализируйте, как меняются значения координат при перемещении указателя мыши.



События `keydown` и `keyup` работают со всеми клавишами клавиатуры. Событие `keypress` работает только с буквенно-цифровыми клавишами.



Значения координат всегда привязаны к положению окна, даже если обработчик событий привязан к объекту, отличному от объекта document.



- 8 Нажмите любую клавишу на клавиатуре, чтобы увидеть ее значение Unicode.

Добавление переключателей

Переключатели с зависимой фиксацией или «радиокнопки» представляют собой группы кнопок, из которых можно выбрать только одну. Атрибут **name** должен иметь одно и то же значение. Имя группы переключателей — это имя массива. Доступ к каждому переключателю осуществляется при использовании значения индекса массива.

В отличие от групп переключателей с зависимой фиксацией, группы независимых кнопок или «флажков» позволяют пользователю выбрать один или несколько вариантов. Однако, как и в случае с группами переключателей, каждый атрибут **name** имеет одно и то же значение. В программном коде имя группы — это имя массива, в котором доступ к каждой кнопке осуществляется с помощью значения индекса массива.

Переключатели, как и флажки, имеют свойство **checked**, которое принимает значение **true**, если переключатель или флажок выбран. В противном случае свойство **checked** принимает значение **false**. Чтобы проверить свойство **checked** или какие переключатели или флажки выбраны (отмечены), используется цикл по массиву переключателей или флажков. Свойство **checked** принимает значение **true**, если переключатель или флажок выбран.



1

Создайте HTML-документ, содержащий форму с группой из трех флажков и кнопкой отправки.

```
<form id="pizza" action="echo.pl" method="POST">
<fieldset>
<legend>Select Pizza Toppings</legend>
<input type="checkbox" name="Top"
value="Cheese">Cheese
<input type="checkbox" name="Top" value="Ham">Ham
<input type="checkbox" name="Top"
value="Peppers">Peppers
</fieldset>
<input type="submit">
</form>
```



checkbox.html

2

Создайте самовывзывающуюся функцию, которая назначает обработчик событий для события отправки формы и проверяет один флажок.

```
(function ( ) {
const form = document.getElementById( 'pizza' )
form.addEventListener( 'submit',
function ( event ) { reactTo( form, event ) } )
form.Top[ 0 ].checked = true
}) ( )
```

3

Создайте функцию обработчика событий. Объявите и проинициализируйте три переменные и создайте цикл, чтобы определить отмеченные флажки.

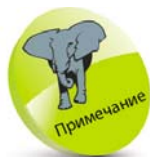
```
function reactTo( form, event ) {
let i, ok, summary = ""
for( i = 0; i < form.Top.length ; i++ )
{
if( form.Top[ i ].checked )
{
summary += form.Top[ i ].value + ' '
}
}
// Здесь будет ваш код.
}
```



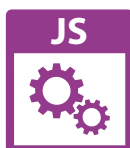
Если в группе отмечено несколько флажков, их значения представляются в виде списка, разделенного запятыми.



При нажатии кнопки генерируется действие по умолчанию для формы, которое заключается в отправке данных формы на веб-сервер. Вызов метода `preventDefault()` останавливает их отправку.



Этот пример запускается на локальном веб-сервере, который поддерживает серверный скрипт PERL. Он обрабатывает отправку формы и предоставляет ответ веб-браузеру.

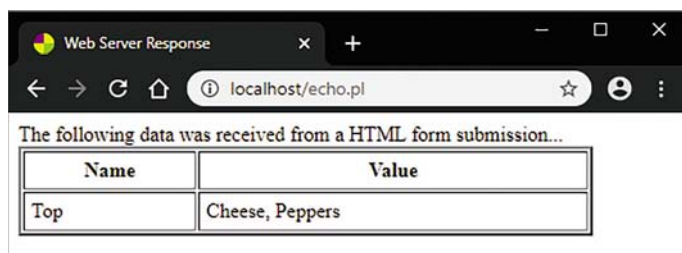
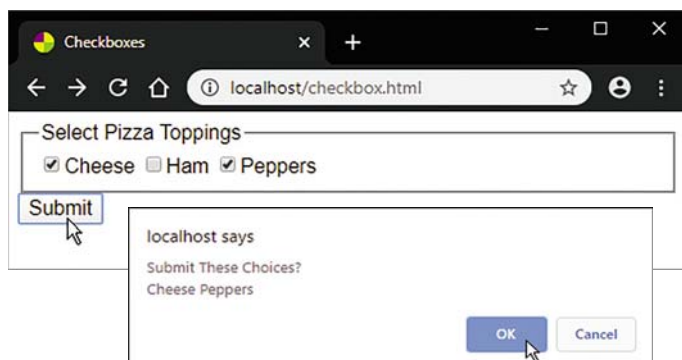


echo.pl

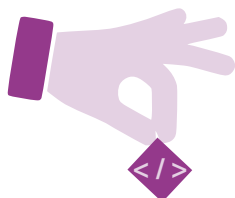
- Добавьте операторы для подтверждения выбора и отправьте их на веб-сервер или отмените отправку.

```
ok = confirm( 'Submit These Choices?\n' + summary )
if( !ok ) { event.preventDefault( ) }
```

- Сохраните HTML-документ на веб-сервере, затем откройте его в браузере. Сделайте свой выбор и подтвердите отправку.



Добавление элементов выбора



Параметры, представленные в объекте раскрывающегося HTML-списка `<select>`, определены массивом `options[]`, в котором каждый элемент содержит параметр, указанный тегом HTML `<option>`. После отправки на веб-сервер значение, присвоенное атрибуту `name` тега `<select>`, и значение, присвоенное атрибуту `value` тега `<option>`, отправляются как пара `name=value`.

Важно отметить, что объект списка выбора имеет свойство **selectedIndex**, содержащее номер индекса выбранного элемента массива **options[]**. Этот номер также можно использовать для получения значения выбранного параметра.

При изменении пользователем выбранного параметра запускается событие «change». Свойство объекта списка **onchange** привязывает обработчик событий, который выполняется при изменении выбранного параметра. Как вариант, имя события и имя функции обработчика событий можно указать в качестве аргументов метода **addEventListener()**.

1

Создайте HTML-документ, содержащий список выбора, кнопку отправки и пустой абзац.

```
<form action="echo.pl" method="POST">
<select id="list" name="City">
<option value="Rome" selected>Rome</option>
<option value="London">London</option>
<option value="New York">New York</option>
</select>
<input type="submit">
</form>
<p id="info"></p>
```



options.html

2

Создайте самовызывающуюся функцию, привяжите обработчик событий для отправки формы и загрузки окна.

```
(function () {
  const list = document.getElementById( 'list' )
  list.addEventListener( 'change',
    function () { reactTo( list, event ) } )
  addEventListener( 'load',
    function () { reactTo( list, event ) } )
})()
```



Обратите внимание, что атрибут HTML **selected** осуществляет выбор первого элемента **option**, доступ к которому определен как **options[0]**.

3

Добавьте обработчик событий, который будет отображать выбранное событие и текущий список в абзаце.

```
function reactTo( list, event ) {
  const info = document.getElementById( 'info' )
```

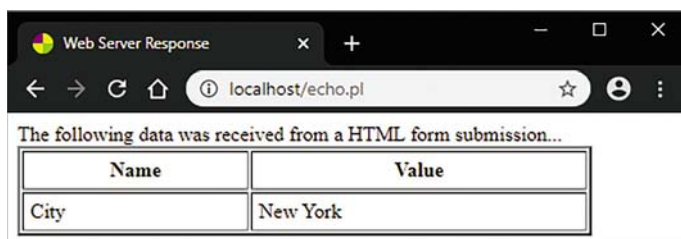
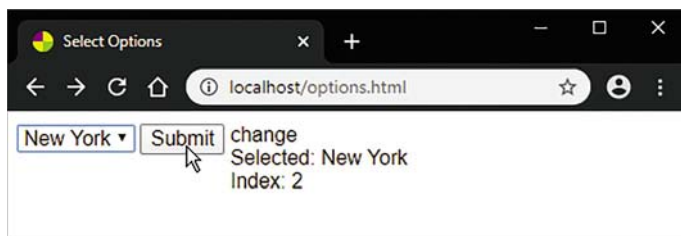
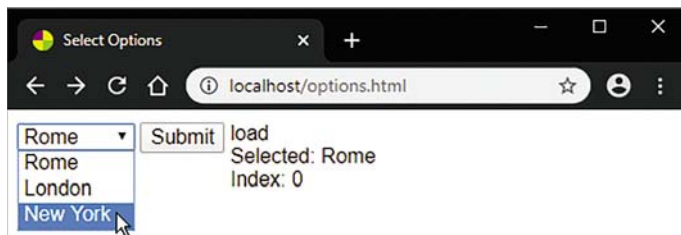


Доступ к выбранной опции осуществляется следующим образом: `document.forms[0].elements[0].options[1].value`.

```
let index = list.options.selectedIndex
let city = list.options[ index ].value
info.innerHTML = event.type + '<br>Selected: '
info.innerHTML += city + '<br>Index: ' + index
}
```

4

Сохраните HTML-документ на веб-сервере и откройте его в браузере. Затем выберите вариант и отправьте форму.



echo.pl

Сброс формы

Обработчик событий может быть привязан к свойствам **onfocus** и **onfocusout**. Событие «focus» запускается, когда пользователь выбирает текстовое поле, а событие «focusout» активируется при покидании этого текстового поля.

Обработчик событий может быть привязан к свойству **onreset**. Событие **onreset** происходит при сбросе формы.

Как и в случае с другими объектами, имя события и имя функции обработчика событий указывается в качестве аргументов для текстового поля и метода объекта формы `addEventListener()`.

1

Создайте HTML-документ, содержащий форму с текстовым полем, кнопкой сброса и пустым абзацем.

```
<form id="code" >
<input id="lang" name="Language" type="text" >
<input type="reset">
<input type="submit">
</form>
<p id="info"></p>
```



reset..html

2

Создайте самовызывающуюся функцию, привяжите обработчик для событий текстового поля «focus» и «focusout», события формы «reset» и события окна «load».

```
(function () {
  const form = document.getElementById( 'code' )
  const lang = document.getElementById( 'lang' )
  const info = document.getElementById( 'info' )

  lang.addEventListener( 'focus',
    function ( event ) { reactTo( event, info ) } )
  lang.addEventListener( 'focusout',
    function ( event ) { reactTo( event, info ) } )
  form.addEventListener( 'reset',
    function () { defaultMessage( info ) } )
  addEventListener( 'load',
    function () { defaultMessage( info ) } )
})()
```



Текстовые поля также имеют свойство `onselect`, которому может быть привязан обработчик событий для ответа на событие «select», срабатывающее, когда пользователь выбирает фрагмент текста.

3

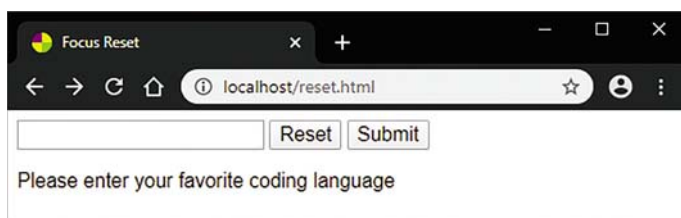
Чтобы отобразить тип события, привяжите обработчик для событий текстового поля «focus» и «focusout».

```
function reactTo( event, info ) {
  info.innerHTML = event.type
}
```

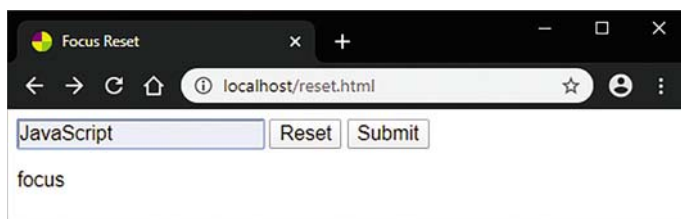
- 4 Для вывода сообщения привяжите событиям загрузки окна и сброса формы следующий обработчик событий.

```
function defaultMessage( info ) {  
    info.innerHTML =  
        'Please enter your favorite coding language'  
}
```

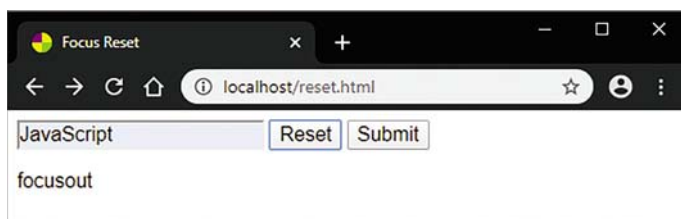
- 5 Сохраните HTML-документ, затем откройте его в браузере. Проанализируйте результат — полученное сообщение.



- 6 Выберите текстовое поле и проанализируйте, как работает событие «focus». Затем введите название вашего любимого языка программирования.

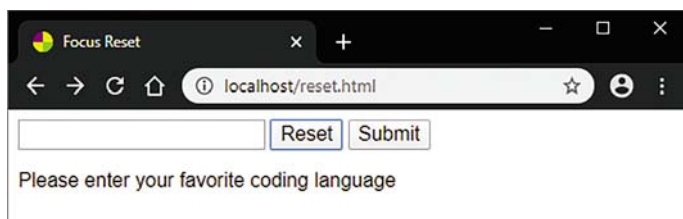


- 7 Нажмите клавишу Tab, чтобы переместить фокус на кнопку Reset (Сброс). Проанализируйте, как работает событие «focusout».



8

Нажмите клавишу Enter (нажмется кнопка Reset (Сброс)), чтобы очистить текстовое поле и снова увидеть полученное сообщение.



Проверка и отправка формы

При отправке формы срабатывает событие **onsubmit**. В качестве альтернативы имя события «submit» и имя функции обработчика событий указываются как аргументы метода **addEventListener()** объекта формы.

Самый простой уровень проверки формы заключается в проверке наличия текста в поле ввода, чтобы убедиться в том, что пользователь действительно его ввел. Если значение равно пустой строке, то есть запись не производилась, функция проверки вызовет метод **event.preventDefault()**, чтобы предотвратить отправку формы.

Более высокий уровень проверки формы заключается в проверке введенной пользователем строки, чтобы убедиться в том, что она соответствует требованиям. Например, при необходимости ввода адреса электронной почты строка должна содержать символ «@» и как минимум один символ «.». Если хотя бы один из них отсутствует, строка не будет соответствовать допустимому формату адреса электронной почты, поэтому функция проверки предотвратит отправку формы.

Доступ к элементу формы осуществляется с помощью указания значения атрибута **name** в скобках массива **elements[]**.





validate.html

1

Создайте HTML-документ, содержащий форму с двумя текстовыми полями и кнопкой отправки.

```
<form id="contact" action="echo.pl" method="POST">
<fieldset>
<legend>Please Enter Your Details</legend>
Name: <input type="text" name="Name" value="">
<br><br>
Email: <input type="text" name="Email" value="">
</fieldset>
<input type="submit">
</form>
```

2

Создайте самовызывающуюся функцию. Для события отправки формы определите следующий обработчик событий.

```
( function () {
  const form = document.getElementById( 'contact' )
  form.addEventListener( 'submit' ,
    function ( event ) { validate( form, event ) } )
} )()
```

3

Для проверки введенного текста добавьте обработчик событий.

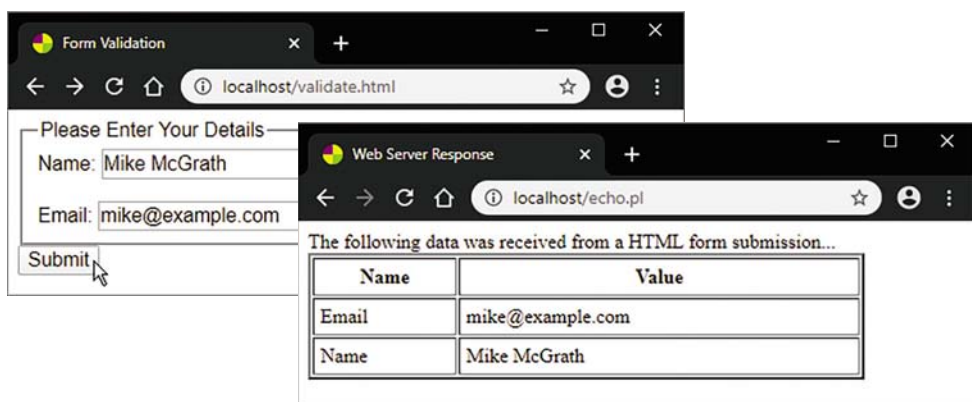
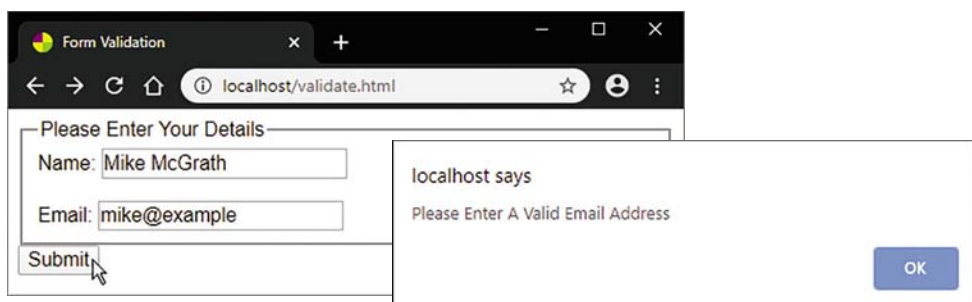
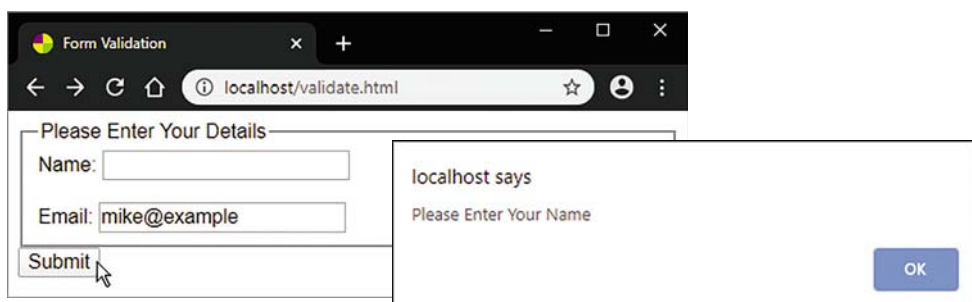
```
function validate( form, event ) {
  let value = form.elements[ 'Name' ].value
  if( value === "" ) {
    alert( 'Please Enter Your Name' )
    event.preventDefault() ; return }
  value = form.elements[ 'Email' ].value
  if( ( value === "" ) || ( value.indexOf( '@' ) === -1 ) ||
    ( value.indexOf( '.' ) === -1 ) ) {
    alert( 'Please Enter A Valid Email Address' )
    event.preventDefault() }
}
```



Метод `indexOf()` возвращает индекс первого вхождения указанного значения в строковый объект или значение `-1`, если символ не найден. Более подробную информацию можно найти на стр. 119.

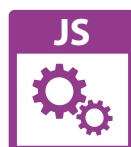
4

Сохраните HTML-документ на веб-сервере, затем откройте его в браузере. Введите свои данные и отправьте форму.



Заклучение

- Объект **document** содержит следующие свойства, описывающие документ: **title**, **URL**, **domain**, **lastModified** и **referrer**.
- Объект **document** содержит дочерние объекты **forms**, **images**, **links**, **styleSheets** и **scripts**. Каждый из них представляет собой массив компонентов документа.



echo.pl

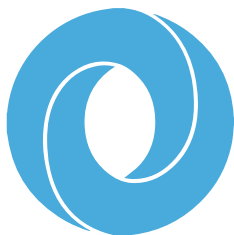
- Массив **forms** — это коллекция HTML со списком всех элементов **<form>**, содержащихся в текущем документе.
- Методы **getElementById()**, **getElementsByTagName()** и **getElementsByClassName()** позволяют ссылаться на компоненты HTML-документа.
- Свойства **innerHTML** и **innerText** используются для записи содержимого в существующие элементы.
- Методы **createElement()**, **appendChild()** и **setAttribute()** позволяют добавлять содержимое в документ.
- Свойство **cookie** содержит пары «key=value» (атрибут=значение), которые могут хранить небольшой объем данных в системе пользователя.
- Функции **encodeURIComponent()**, **decodeURIComponent()**, **encodeURIComponent()** и **split()** используются для обработки строк с данными cookie.
- DOM позволяет JavaScript реагировать на такие события, как **load**, **click**, **keydown**, **change** и **submit**.
- Функции обработчика событий могут быть привязаны к свойству объекта или определены методом **addEventListener()**.
- Объект **event** передается в обработчик событий, а событие идентифицируется по его свойству **event.type**.
- Событие «mousemove» передает объект события в функцию-обработчик со свойствами **x** и **y**, определяющими текущие координаты указателя мыши.
- Переключатели и флажки имеют свойство **checked**, которое принимает значение **true**, если переключатель или флажок выбран.
- Объект списка выбора имеет свойство **selectedIndex**, содержащее номер индекса выбранного элемента массива **options[]**.
- При нажатии кнопки генерируется действие по умолчанию для формы, которое заключается в отправке данных формы на веб-сервер. Вызов метода **preventDefault()** останавливает их отправку.

8

Разработка веб-приложений

*В этой главе вы узнаете, как
создавать веб-приложение,
которое извлекает данные
из онлайн-ресурса.*

- 180 Введение в JSON
- 182 Промисы
- 185 Получение данных
- 187 Разработка интерфейса
- 190 Заполнение ячеек
в таблице
- 192 Заполненная таблица
- 194 Обновление приложений
- 197 Заключение



Введение в JSON

JSON (Нотация объектов JavaScript) — это популярный текстовый формат для хранения и транспортировки данных. Это подмножество JavaScript. Данные JSON записываются в виде пар **key: value** (атрибут: значение), разделенных запятыми. Все атрибуты должны иметь тип данных **String** и заключены в двойные кавычки. Связанные с этими атрибутами значения должны принадлежать только к одному из следующих типов данных:

- **String** (строка) — строки в JSON должны быть заключены в двойные кавычки.
- **Number** (число) — целое число или число с плавающей запятой.
- **Object** (объект) — объект JSON.
- **Array** (массив) — массив.
- **Boolean** (логическое значение) — **true** (истина) или **false** (ложь).
- **null** — значение **null**.

Пары **key: value** (атрибут: значение) должны быть заключены в фигурные скобки, например:

```
{"name":"Alice","age":21,"city":"New York"}
```

Также можно легко преобразовать объект JavaScript в объект JSON, указав его в качестве аргумента метода **JSON.stringify()**. И наоборот, можно преобразовать объект JSON в объект JavaScript, указав его в качестве аргумента метода **JSON.parse()**.

Обычно объекты JSON хранят данные как онлайн-ресурс в текстовом файле с расширением файла **.json**. Следовательно, с веб-сервера сценарий веб-страницы получает данные в формате JSON.



Бесплатный валидатор объектов JSON вы можете найти на сайте jsonlint.com.

Метод **JSON.parse()** используется для преобразования объекта JSON в объект JavaScript. Доступ к данным осуществляется с использованием точечной нотации или с помощью скобок.

Как JSON, так и XML (расширяемый язык разметки) можно использовать для получения данных с веб-сервера. Однако предпочтительнее обратиться к JSON, поскольку вам необходимо перебирать элементы для извлечения данных из формата XML, тогда как метод **JSON.parse()** просто возвращает строку всех данных.

Важной особенностью считается, что JavaScript извлекает данные из онлайн-ресурсов JSON для использования в веб-приложениях.

1

Создайте самовызывающуюся функцию, которая создает объект JavaScript, содержащий строку и массив.

```
let obj = { category: 'Fashion',  
  models: [{ name: 'Alice', age: 21, city: 'New York' },  
    { name: 'Kelly', age: 23, city: 'Las Vegas' } ] }
```



json.html

2

Создайте объект JSON и распечатайте его.

```
let json_obj = JSON.stringify( obj )  
console.log( json_obj )
```

3

Для сравнения создайте объект JavaScript и распечатайте его.

```
let new_obj = JSON.parse( json_obj )  
console.log( new_obj )
```

4

Наконец, распечатайте выбранные значения, используя точечную нотацию и квадратные скобки.

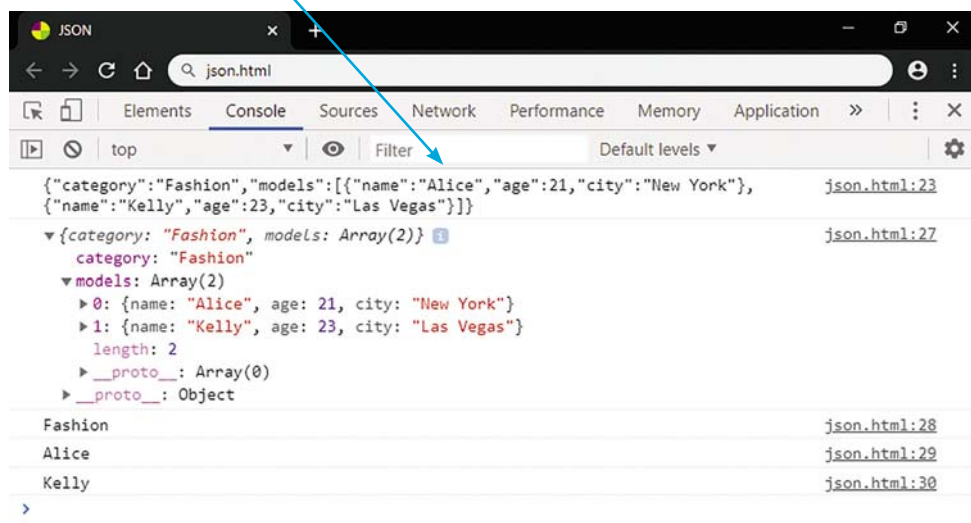
```
console.log( new_obj[ 'category' ] )  
console.log( new_obj.models[ 0 ].name )  
console.log( new_obj[ 'models' ][ 1 ][ 'name' ] )
```



Все значения **String** объекта JSON заключены в двойные кавычки!

5

Сохраните HTML-документ, затем откройте его в веб-браузере. Запустите консоль, чтобы проанализировать сравниваемые объекты. Для этого выберите пункт Developer Tools (Инструменты разработчика) => Console (Консоль).



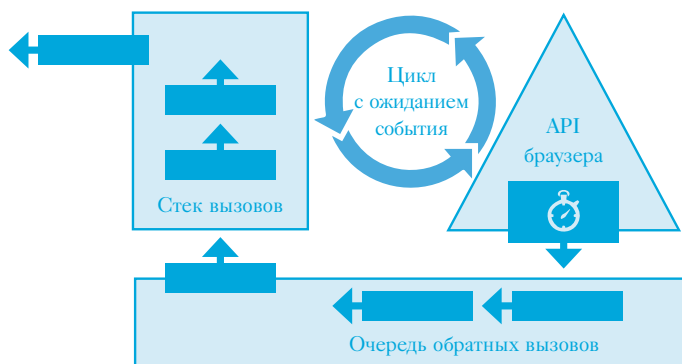
Промисы

JavaScript — это однопоточный «синхронный» язык. Значит, в любой момент времени может выполняться только одна операция. По мере выполнения сценария каждая операция добавляется в «стек вызовов», выполняется (в порядке сверху вниз), а затем удаляется из стека.



Веб-браузеры имеют несколько API, включая Fetch API, который предоставляет интерфейс для получения ресурсов по сети.

Также существуют функции, которые обрабатываются API браузера («Интерфейс прикладного программирования»), а не движком JavaScript. Например, метод **setTimeout()** обрабатывается браузером, поэтому другие операции выполняются после завершения работы таймера. Когда таймер заканчивает работу, он передает операцию в «очередь обратных вызовов», а затем (когда она достигает начала очереди) выполняется. Этот процесс контролируется «циклом с ожиданием события», который постоянно отслеживает состояние стека вызовов и очереди обратных вызовов.



В JavaScript объект **Promise** (промис) представляет окончательное завершение или сбой асинхронной операции и ее результирующее значение. Первый аргумент (*resolve*) вызывает успешное исполнение промиса, второй (*reject*) отклоняет его.

Объект **Promise** создается при помощи ключевого слова **new** и конструктора **Promise()**. В качестве аргумента он принимает функцию с двумя аргументами, которые вызывают успешное выполнение промиса либо отклоняют его.

Каждый объект **Promise** содержит методы **then()** и **catch()**, которые можно применять, используя точечную запись. Метод **then()** обрабатывает результирующее значение синхронной операции, а метод **catch()** обрабатывает ошибки, если они отклонены.

1

Создайте самовызывающуюся функцию, которая создает объект **Promise** и будет выполнять одну из двух функций с задержкой в одну секунду.

```
const promise = new Promise( function( resolve, reject )
{
  let random = Math.round(Math.random() * 10 )
  if ( random % 2 === 0 )
  { setTimeout( function() { resolve( random ) }, 1000 ) }
  else
  { setTimeout( function() { reject ( random ) }, 1000 ) }
})
```



Вы можете связать с промисом несколько вызовов метода **then()**, но только один из них вызовет успешное исполнение.



json.html



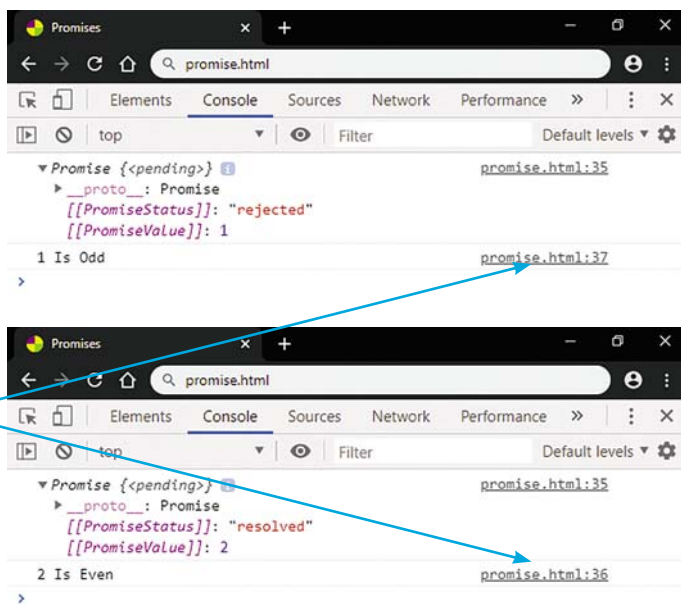
Обратите внимание, как консоль предоставляет номер строки промиса и функции, обработавшей результат.

- 2 Добавьте оператор с методами, отображающими состояние промиса и возвращенные значения.

promise

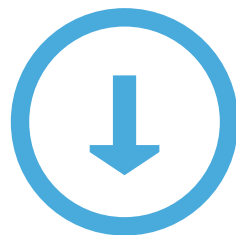
```
.then(console.log( promise ))  
.then(function( res ) { console.log( res + ' Is Even' ) })  
.catch(function( err ) { console.log( err + ' Is Odd' ) })
```

- 3 Сохраните HTML-документ, затем откройте его в браузере. Запустите консоль, чтобы проанализировать выполнение асинхронных операций. Для этого выберите пункт Developer Tools (Инструменты разработчика) => Console (Консоль).



Получение данных

Веб-браузеры поддерживают Fetch API, которые предоставляют интерфейс для получения ресурсов по сети. Метод `fetch()` принимает единственный аргумент — URL-адрес ресурса, который вам необходимо получить.



Метод `fetch()` асинхронный, поэтому другие операции могут выполняться в ожидании получения ресурса. По завершении он возвращает объект **Promise**, содержащий ответ (объект **HTTPResponse**).

Как правило, метод `fetch()` получает ресурс JSON, и проанализировать его можно с помощью метода `json()` объекта **HTTPResponse**. Затем возвращенные данные JSON передаются в качестве аргумента следующему связанному методу промиса, который, в свою очередь, передает данные JSON в функцию-обработчик. Процесс выглядит следующим образом:

```
fetch( url )
.then( function( response ) { return response.json( ) } )
.then( function( data ) { return обработчик( data ) } )
.catch( function( err ) { return console.log( err ) } )
```

Стрелочные функции

Стрелочные функции представляют собой сокращенную версию обычных функций. Они образуются с помощью знака `=>`. Это позволяет опустить ключевое слово функции, например:

```
.then( ( response ) => { return response.json( ) } )
```

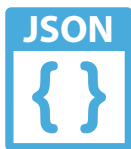
Если тело функции содержит только один оператор, и этот оператор возвращает значение, вы также можете опустить фигурные скобки и ключевое слово **return**, например:

```
.then( ( response ) => response.json( ) )
```

Параметры могут отображаться в виде списка, разделенного запятыми и заключенного в круглые скобки `()`. Если функция принимает один параметр, то скобки вокруг него можно опустить, например:



В стрелочных функциях ключевое слово **this** представляет объект в исходном контексте, тогда как в обычных ключевое слово **this** представляет объект, который вызывает функцию.

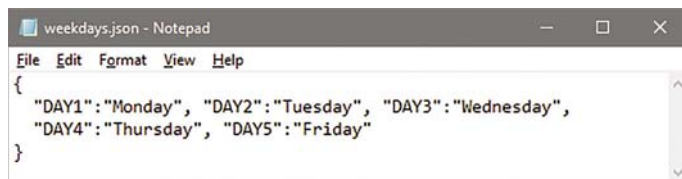


weekdays.json

```
fetch( url )
.then( response => response.json( ) )
.then( data => обработчик( data ) )
.catch( err => console.log( err ) )
```

Обратите внимание, что стрелочные функции обрабатывают ключевое слово **this** иначе, чем обычные.

- 1 Откройте текстовый редактор, например, в Windows приложение Notepad (Блокнот). Создайте документ JSON с объектом, содержащим пять пар key: value (атрибут: значение).



- 2 Сохраните документ JSON на веб-сервере в папке «htdocs», чтобы он был доступен по сети.
- 3 Создайте асинхронный HTTP-запрос, создав промис, разрешенный посредством получения данных JSON.

```
fetch( 'http://localhost/weekdays.json' )
.then( response => response.json( ) )
.then( data => list( data ) )
.catch( err => console.log( err ) )
```

- 4 Создайте функцию для распечатки данных.

```
function list( data ) {
  const values = Object.values( data )
  let i = 0
  while( i < values.length ) { console.log( values[ i ] ); i++ }
}
```

- 5 Сохраните HTML-документ и документ JSON. Затем через HTTP откройте веб-страницу и запустите консоль, чтобы просмотреть полученные данные. Для этого выберите пункт Developer Tools (Инструменты разработчика) => Console (Консоль).



fetch.html

Monday	fetch.html:44
Tuesday	fetch.html:44
Wednesday	fetch.html:44
Thursday	fetch.html:44
Friday	fetch.html:44

Разработка интерфейса

Теперь приступим к созданию веб-приложения, которое позволит динамически обновлять данные на веб-странице без ее перезагрузки. Интерфейсом предусмотрена форма ввода и таблица с несколькими пустыми ячейками. Некоторые ячейки изначально заполняются данными JSON, а другие вычисляются.

1

Создайте HTML-документ с таблицей в основном разделе документа.

```
<div id = "databox">
<table>
<tr>
<td></td>
<td class="col">1</td>
<td class="col">2</td>
<td class="col">3</td>
<td class="col">4</td>
<td class="col">5</td>
<td class="tot">Total</td>
</tr>

<tr>
<td class="row">1</td>
<td id="n0"></td>
<td id="n1"></td>
<td id="n2"></td>
<td id="n3"></td>
<td id="n4"></td>
<td id="rt1" class="tot"></td>
</tr>
```



webapp.html



Пустые белые ячейки, расположенные в теле таблицы, имеют идентификаторы `n0–n14`, итоговые ячейки строки имеют идентификаторы `rt1–rt3`, итоговые ячейки столбца имеют идентификаторы `ct1–ct3`, а общая итоговая ячейка в правом нижнем углу имеет идентификатор `gt`.

```
<tr>
<td class="row">2</td>
<td id="n5"></td>
<td id="n6"></td>
<td id="n7"></td>
<td id="n8"></td>
<td id="n9"></td>
<td id="rt2" class="tot"></td>
</tr>
```

```
<tr>
<td class="row">3</td>
<td id="n10"></td>
<td id="n11"></td>
<td id="n12"></td>
<td id="n13"></td>
<td id="n14"></td>
<td id="rt3" class="tot"></td>
</tr>
```

```
<tr>
<td class="tot">Total</td>
<td id="ct1" class="tot"></td>
<td id="ct2" class="tot"></td>
<td id="ct3" class="tot"></td>
<td id="ct4" class="tot"></td>
<td id="ct5" class="tot"></td>
<td id="gt" class="gto"></td>
</tr>
</table>
</div>
```

2

После разделения таблицы добавьте форму, содержащую два поля выбора, поле ввода текста и кнопку.

```
<form action="#"> <fieldset id="editbox" >
<legend id="legend" >Cell Editor</legend> <div>
<select id="rownum" >
    <option selected="selected" >Row</option>
    <option>1</option>
    <option>2</option>
    <option>3</option>
</select>
```

```

<select id="colnum" >
    <option selected="selected">Column</option>
    <option>1</option>
    <option>2</option>
    <option>3</option>
    <option>4</option>
    <option>5</option>
</select>
New Value:
<input id="newval" type="text" size="5" value="" >
<input type="button" value="Update" onclick="update( )" >
</div> </fieldset> </form>

```

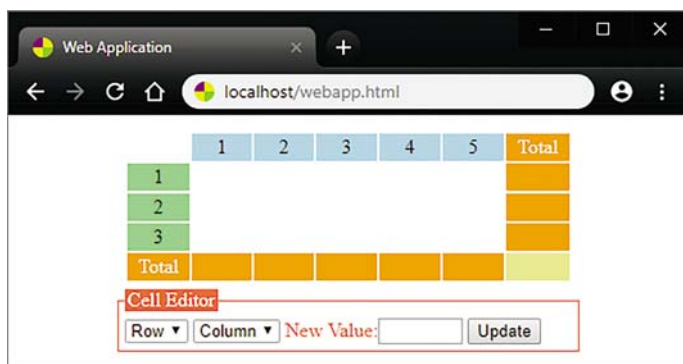
3

Добавьте таблицу стилей в заголовок HTML-документа. Затем через HTTP откройте веб-страницу и проанализируйте полученные результаты — стили, примененные к таблице и форме.

```

<style>
legend { background: Tomato ; color: White ; }
table { text-align: center ; }
td     { width: 50px ; height: 20px ; }
td.col { background: LightBlue ; }
td.row { background: LightGreen ; }
td.tot { background: DarkOrange ; color: White ; }
td.gto { background: Khaki ; }
#databox{ width: 360px ; padding: 5px ; margin: auto ; }
#editbox{ width: 360px ; padding: 5px ; margin: auto ;
          border: 1px solid Tomato ; color: Tomato ; }
</style>

```



В этом примере файлы JSON и HTML расположены на веб-сервере в папке «htdocs», поэтому URL-адрес домена в HTTP — «localhost».

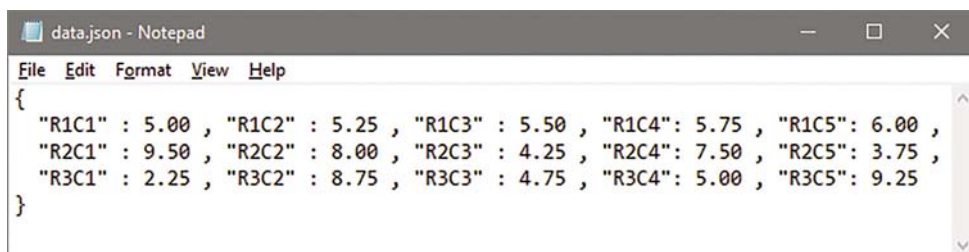
Заполнение ячеек в таблице

После создания таблицы и формы можно создать файл данных JSON и написать JavaScript-код, с помощью которого будем заполнять таблицу.



data.json

- 1 Откройте текстовый редактор, например, в Windows приложение Notepad (Блокнот). Создайте документ JSON с объектом, содержащим пятнадцать пар key: value (атрибут: значение).



webapp.html
(продолжение)

- 2 Сохраните документ JSON и HTML-документ на веб-сервере в папке «htdocs», чтобы он был доступен по сети.
- 3 Добавьте в HTML-документ элемент `<script> </script>` непосредственно перед закрывающим тегом `</body>`.
- 4 Затем в элемент `<script>` вставьте функцию для выполнения асинхронного HTTP-запроса, создав промис, разрешенный посредством получения данных JSON.

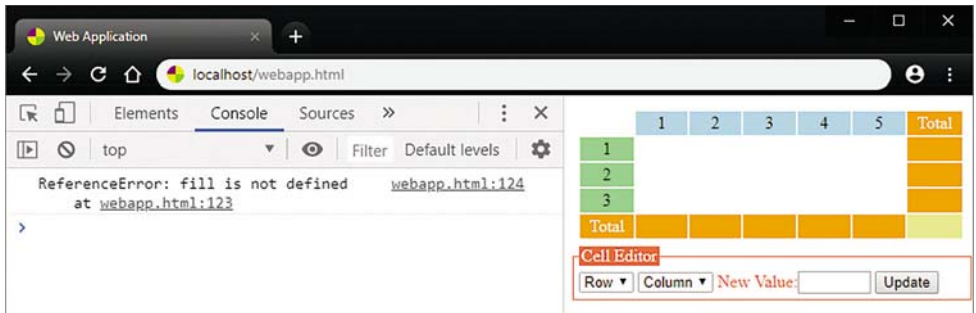
```
fetch( 'http://localhost/data.json' )  
  .then( response => response.json() )
```

- 5 Свяжите следующие два оператора с промисом, который передаст данные JSON следующему связанному методу. В противном случае, если запрос не может быть разрешен, выведите сообщение об ошибке.

```
.then( cells => fill( cells ) )
.catch( err => console.log( err ) )
```

6

Сохраните HTML-документ. Затем через HTTP перезагрузите веб-страницу и откройте консоль, выбрав пункт Developer Tools (Инструменты разработчика) => Console (Консоль). Проанализируйте полученные результаты — сообщение об ошибке, поскольку функция для получения данных JSON еще не создана.



7

Вернитесь в HTML-документе к элементу `<script>` и создайте функцию для получения данных JSON и вставьте все ее значения в ячейки таблицы.

```
function fill( cells ) {

  const values = Object.values( cells )
  let i = 0

  while( i < values.length )
  {
    document.getElementById( 'r' + i ).innerText =
      values[ i ].toFixed( 2 )
    i++
  }
}
```

8

Снова сохраните HTML-документ. Затем через HTTP перезагрузите веб-страницу, чтобы увидеть отображаемые в таблице значения.

	1	2	3	4	5	Total
1	5.00	5.25	5.50	5.75	6.00	
2	9.50	8.00	4.25	7.50	3.75	
3	2.25	8.75	4.75	5.00	9.25	
Total						

Cell Editor
 Row ▼ Column ▼ New Value: Update

Заполненная таблица

После того как пустые белые ячейки таблицы будут заполнены данными в соответствии с инструкциями, описанными на стр. 192–193, можно рассчитать и заполнить строки и столбцы таблицы окончательными значениями, а также записать общее итоговое значение.



webapp.html
(продолжение)



Этот цикл перебирает ячейки строки, увеличивая и добавляя номер индекса к символу «n», чтобы выбрать значения идентификатора элемента n0, n1, n2, n3, n4. Затем цикл перебирает ячейки следующей строки и т. д.

- 1 В конце функционального блока `fill(cells)`, непосредственно перед последним символом `}`, вставьте вызов функции, вычисляющей итоговые значения.

`total()`

- 2 Создайте функцию расчета итоговых значений. Объявите и проинициализируйте четыре переменные.

```
function total() {
  let i = 0
  let sum = 0
  let rownum = 1
  let colnum = 0
  // Здесь будет ваш код.
}
```

- 3 Затем в функциональном блоке вставьте цикл для записи суммы каждой строки в ячейке, расположенной в конце каждой строки.

```

for ( i = 0 ; i < 15 ; i++ )
{
    sum += parseFloat( document.getElementById( 'n' + i )
                                                                .innerText )

    if ( ( i + 1 ) % 5 === 0 )
    {
        document.getElementById(
            'rt' + rownum ).innerText = sum.toFixed( 2 )
        rownum++
        sum = 0
    }
}

```

4

В функциональном блоке вставьте цикл для записи суммы каждого столбца в ячейку, расположенную внизу каждого столбца.

```

while ( colnum !== 5 )
{
    for ( i = 0; i < 15; i++ )
    {
        if ( i % 5 === 0 ) sum += parseFloat( document
            .getElementById( 'n' + ( i + colnum ) ).innerText )
    }
    colnum++
    document.getElementById( 'ct' + colnum )
        .innerText = sum.toFixed( 2 )

    sum = 0
}

```

5

В функциональном блоке вставьте цикл для записи общего количества всех белых ячеек в ячейку, расположенную в правом нижнем углу таблицы.

```

for ( i = 0; i < 15; i++ )
{
    sum += parseFloat( document.getElementById( 'n' + i )
                                                                .innerText )
}
document.getElementById( 'gt' ).innerText =
    sum.toFixed( 2 )

```



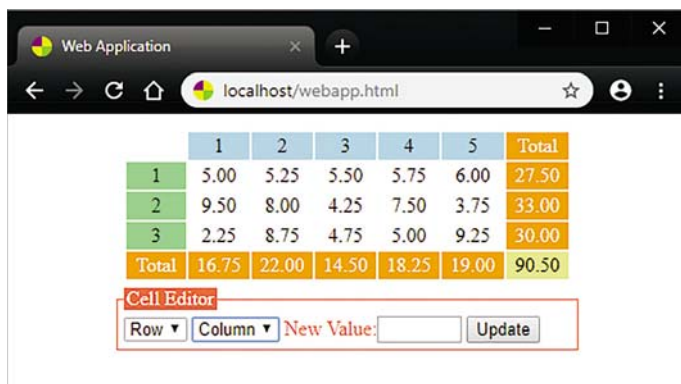
Этот цикл перебирает ячейки столбца, увеличивая и добавляя номер индекса, затем добавляет номер увеличивающегося столбца к символу «n», чтобы выбрать значения идентификатора элемента n0, n5, n10. Потом цикл перебирает ячейки следующего столбца и т. д.



Обратите внимание, что метод toFixed(2) гарантирует, что значения ячеек будут отображаться с двумя десятичными знаками.

6

Сохраните изменения и через HTTP перезагрузите веб-страницу. Проанализируйте полученные результаты — итоговые значения, записанные в таблицу.



	1	2	3	4	5	Total
1	5.00	5.25	5.50	5.75	6.00	27.50
2	9.50	8.00	4.25	7.50	3.75	33.00
3	2.25	8.75	4.75	5.00	9.25	30.00
Total	16.75	22.00	14.50	18.25	19.00	90.50

Cell Editor
 Row ▼ Column ▼ New Value: Update



webapp.html
(продолжение)



В нашем примере для выполнения арифметических операций надо использовать функцию `parseFloat()`, так как необходимо преобразовать вводимые строковые значения `String` в числовой тип данных.

Обновление приложений

После заполнения всех ячеек в соответствии с инструкциями, описанными на стр. 194–195, можно к кнопке формы привязать обработчик событий, позволяющий пользователю обновлять таблицу.

1

Создайте функцию, объявите и проинициализируйте шесть переменных.

```
function update() {
  let row = document.getElementById( 'rownum' )
                                     .options.selectedIndex
  let col = document.getElementById( 'colnum' )
                                     .options.selectedIndex
  let newval = parseFloat( document
                           .getElementById( 'newval' ).value )
  let legend = document.getElementById( 'legend' )
  let target = null
  // Здесь будет ваш код.
}
```

2

Добавьте в функциональном блоке следующие операторы, проверяющие данные формы. В противном случае, если оно недействительно, выведите информационное сообщение.

```
if( row === 0 ) { legend.innerText = 'Select a row'; return }
if( col === 0 ) { legend.innerText = 'Select a column' ;
                                                         return }
if( !newval ) { legend.innerText = 'Enter a value'; return }
if(isNaN( newval ) ) { legend.innerText = 'Enter a number';
                                                         return }
```

3

В функциональном блоке в выбранную ячейку введите допустимое значение, затем вычислите и выведите новые итоговые значения.

```
target = ( ( ( row - 1 ) * 5 ) + col ) - 1
          document.getElementById( 'n' + target )
              .innerText = newval.toFixed( 2 )
total( )
```

4

Наконец, в функциональном блоке сбросьте форму для дальнейшего ввода.

```
document.getElementById( 'rownum' )
    .options[ 0 ].selected = true
document.getElementById( 'colnum' )
    .options[ 0 ].selected = true
document.getElementById( 'newval' ).value = ''
legend.innerText = 'Cell Editor'
```

5

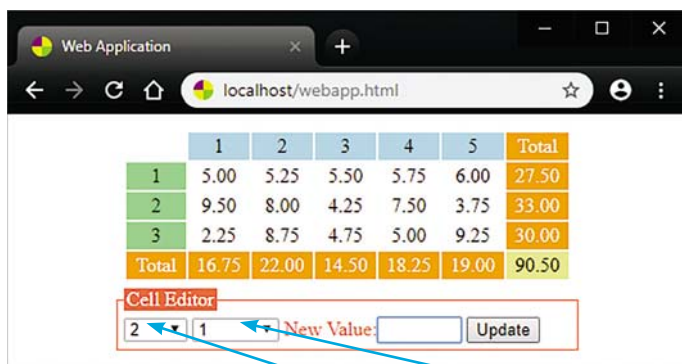
Сохраните HTML-документ. Теперь через HTTP откройте веб-страницу, отредактируйте значение ячейки и проанализируйте, как обновляются итоговые значения.



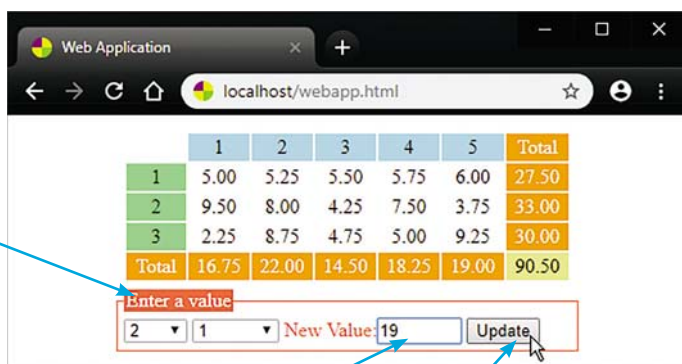
Пока ввод не завершен, вы можете нажать кнопку, чтобы увидеть сообщения проверки.



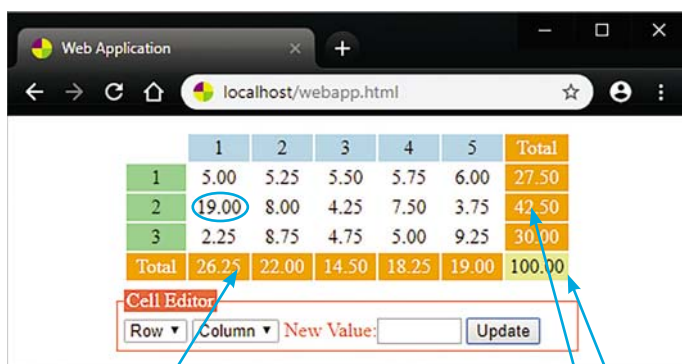
Функция `toFixed(2)` отображает целые числа с двумя десятичными знаками.



Например: выберите 2-ю строку и 1-й столбец.



Введите **новое значение** и нажмите кнопку.



Проанализируйте, как обновляются значения ячейки, итоговое значение в столбце, итоговое значение в строке и общее итоговое значение, и как выглядит вид самой формы для дальнейшего редактирования.

Заключение

- JSON — это текстовый формат, который используется для хранения и обмена данными. Кроме того, это подмножество языка JavaScript.
- Объекты **JSON** содержат список пар `key: value` (атрибут: значение), разделенных запятыми и заключенных в фигурные скобки.
- Все атрибуты объекта **JSON** должны иметь тип данных **String**.
- Все значения объекта **JSON** должны принадлежать только к одному из следующих типов данных: **String**, **Number**, **Object**, **Array**, **Boolean** или **null**.
- Объекты JavaScript можно преобразовать в объекты JSON, используя метод **JSON.stringify()**.
- Объекты **JSON** можно преобразовать в объекты JavaScript при помощи метода **JSON.parse()**.
- JavaScript — это однопоточный синхронный язык. Однако интерфейсы API браузера поддерживают некоторые асинхронные операции.
- Цикл событий постоянно отслеживает состояние стека вызовов и очереди обратного вызова.
- Объект **Promise** представляет собой окончательное завершение или сбой асинхронной операции и ее результирующее значение.
- Методы **then()** и **catch()** могут быть связаны с объектом **Promise** для обработки результатов асинхронной операции.
- Fetch API используется в JavaScript для асинхронного получения ресурсов по сети.
- По завершении метод **fetch()** возвращает объект **Promise**, содержащий объект **HTTPResponse**.

- Объект **HTTPResponse** содержит метод **json()**, который можно использовать для анализа объекта **JSON**.
- Стрелочные функции представляют собой сокращенную версию обычных функций и образуются с помощью знака **=>**.
- Стрелочные функции обрабатывают ключевое слово **this** иначе, чем обычные.
- Данные **JSON** можно использовать для заполнения ячеек таблицы веб-приложения.

9

Написание скриптов

<i>В этой главе</i>	200	Запрос данных
<i>демонстрируются</i>	202	Встраиваемая векторная графика
<i>некоторые полезные</i>		
<i>возможности различных</i>		
<i>API.</i>	205	Работа с холстами
	207	Хранение данных
	209	Перемещение элементов
	212	Связь между объектами window
	214	Местоположение пользователей
	217	Заключение



На каждом этапе запроса свойству `readyState` присваивается следующее значение:

- 0:** Не инициализирован
- 1:** Сервер подключен
- 2:** Запрос получен
- 3:** Обработка запроса
- 4:** Операция завершена успешно

По завершении свойству `status` присваивается следующий код:

- 200:** "OK" (OK)
- 403:** "Forbidden" (Запрещено)
- 404:** "Page Not Found" (Страница не найдена)



books.xml

Запрос данных

Асинхронный JavaScript и XML («AJAX») позволяют обновлять веб-страницы, запрашивая данные с веб-сервера и не прерывая другие операции.

Чтобы использовать AJAX, сначала необходимо в JavaScript создать объект `XMLHttpRequest`, используя ключевое слово `new` и конструктор `XMLHttpRequest()`. Запрос указывается в виде трех аргументов метода `open()`, в которые вводится метод поиска, URL-адрес и логическое значение `true` для асинхронного выполнения запроса.

Затем, используя метод объекта `send()`, запрос можно отправить. Для обработки ответа от веб-сервера необходимо свойству `onreadystatechange` объекта `XMLHttpRequest` привязать обработчик событий. В первую очередь, чтобы убедиться, что ответ действительно полный и успешный, необходимо проверить, что свойство объекта `readyState` возвращает значение `4`, а свойство объекта `status` — `200`.

Текст, полученный посредством объекта `XMLHttpRequest`, автоматически сохраняется в его свойстве `responseText`. Модель DOM представляет элементы XML-документа как «узлы», поэтому для доступа к их содержимому необходимо выполнить дополнительные действия. Например, все узлы элементов с определенным именем тега могут быть присвоены переменной массива при помощи метода `getElementsByTagName()` свойства `responseXML`.

Каждый узел элемента имеет свойство `firstChild`, которое возвращает первый дочерний узел как узел элемента, текстовый узел или узел комментария. Таким образом, свойство `firstChild.data` показывает текст в элементе XML.

1

Создайте XML-документ с 10 элементами `<book>`, содержащими внутренние (дочерние) элементы `<title>`.

```
books.xml - Notepad
File Edit Format View Help
<?xml version="1.0" encoding="UTF-8" ?>
<catalog>
  <book>
    <id>978-1-84078-840-2</id>
    <title>C Programming in easy steps, 5th Edition</title>
  </book>
  <book>
    <id>978-1-84078-757-3</id>
    <title>C++ Programming in easy steps, 5th Edition</title>
  </book>
</catalog>
```

2

Создайте HTML-документ, содержащий элемент кнопки и пустой элемент упорядоченного списка.

```
<button onclick="loadXML( )">Get Books</button> <ol
id="list"></ol>
```

3

Создайте функцию для обработки запроса данных XML.

```
function loadXML( ) {
  const xmlhttp = new XMLHttpRequest( )
  xmlhttp.open( 'GET', 'books.xml', true )
  xmlhttp.send( )
  xmlhttp.onreadystatechange = function( ) {
    if(this.readyState == 4 && this.status == 200 )
      getData( this ) }
}
```

4

Затем добавьте функцию для обработки ответа и записи XML-данных в пустой элемент списка.

```
function getData( xml ) {
  const xmlDoc = xml.responseXML
  const tags = xmlDoc.getElementsByTagName( 'book' )
  let list = "", i = 0
  for ( i = 0 ; i < tags.length ; i++ ) {
    list += '<li>' + tags[ i ].getElementsByTagName( 'title' )
    [ 0 ].firstChild.data }
    document.getElementById( 'list' ).innerHTML = list
  }
}
```

5

Сохраните документы XML и HTML на веб-сервере в папке «htdocs». Затем для



На рисунке показаны только 2 из 10 элементов `<book>`, представленные в файле исходного кода, который можно бесплатно загрузить с сайта www.ineasysteps.com/resource-centre/downloads.



xmlhttp.html

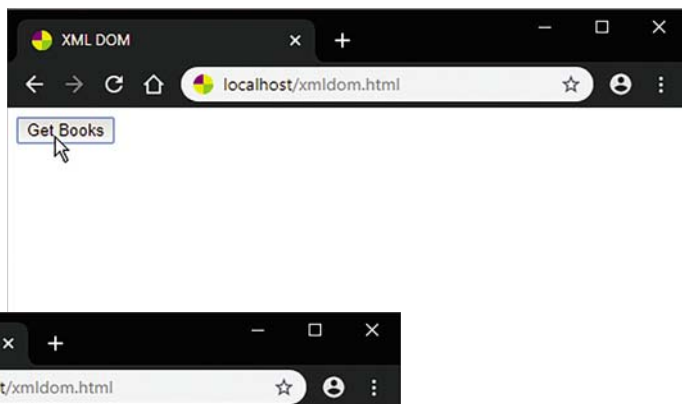


В этом примере каждая итерация цикла получает «book.title.data». Используя CSS, элементы списка с помощью селектора `li:nth-child(even)` получают чередующиеся цвета фона.



Как и HTML-документы, документы XML также можно проверить с помощью онлайн-инструмента проверки W3C (validator.w3.org).

запроса данных XML откройте веб-страницу и нажмите кнопку.



Встраиваемая векторная графика



SVG-графика, встроенная в документ HTML, имеет объектную модель (DOM). DOM в SVG, как и в HTML, содержит модель событий. Это значит, что при использовании этой технологии для реализации интерактивных действий со стороны разработчика требуется меньше усилий, поскольку события привязываются непосредственно к элементам DOM.

Для создания ключа доступа к встроенной SVG DOM необходимо сначала с помощью метода `getSVGDocument()` создать ссылочный объект на документ SVG. Затем элементы, атрибуты и текст в нем могут быть адресованы через объект `SVGDocument`:

1

Создайте SVG-графику, содержащую прямоугольник, круг и текст.

```
<?xml version="1.0" encoding="UTF-8"?>
<svg xmlns="http://www.w3.org/2000/svg"
  version="1.1" width="100%" height="100%"
  viewBox="0 0 500 70" preserveAspectRatio="none" >

  <rect width="100%" height="100%"
    style="fill: bisque; stroke-width:2; stroke: tomato" />
  <text id="svgTxt" x="10" y="50"
    font-family="sans-serif"
    font-size="30" fill="tomato">SVG Text</text>
  <circle id="svgBtn" cx="460" cy="35" r="30"
    fill="tomato" cursor="pointer" />
</svg>
```



banner.svg

2

Создайте HTML-документ, включающий SVG-графику, поле ввода текста и кнопку.

```
<embed id="svgDoc" src="banner.svg"
  type="image/svg+xml" width="500" height="70" >
<br>
<input id="htmTxt" size="50" >
<button id="htmBtn">
  Send To SVG Document</button>
```



svgdom.html

3

Создайте ссылку на документ SVG и элементы в обоих документах.

```
function loadSVG( ) {
  const svgDoc = document.getElementById( 'svgDoc' )
    .getSVGDocument( )
  const svgTxt = svgDoc.getElementById( 'svgTxt' )
  const svgBtn = svgDoc.getElementById( 'svgBtn' )
  const htmTxt = document.getElementById( 'htmTxt' )
  const htmBtn = document.getElementById( 'htmBtn' )
  // Здесь будет ваш код.
}
onload = loadSVG
```



Обратите внимание, что этот пример инициализируется событием window.onload.



Обратите внимание, что в документе SVG текст содержится в свойстве `wholeText` узла `lastChild` элемента SVG `<text>`.



Более подробную информацию об SVG можно узнать на сайте w3.org/TR/SVG11/svgdom.html.

4

Добавьте операторы, чтобы привязать обработчик событий для кнопки HTML.

```
htmlBtn.addEventListener( 'click', function ( ) {
    svgTxt.lastChild.replaceWith( htmlTxt.value )
    htmlTxt.value = ' ' } )
```

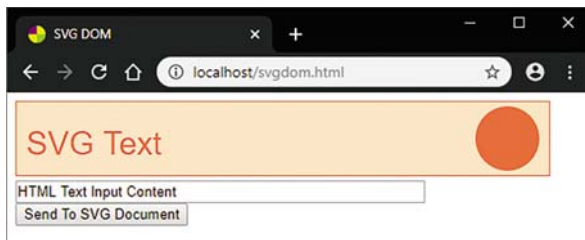
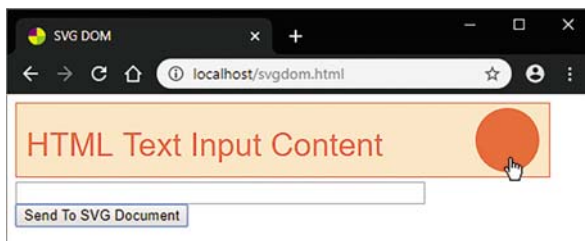
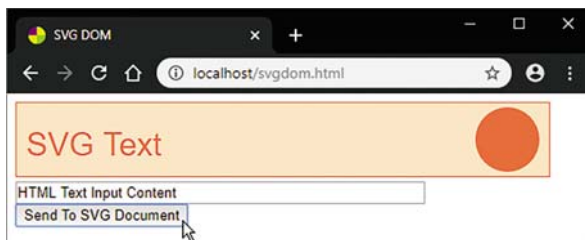
5

Теперь добавьте операторы, чтобы указать функцию обработчика событий для круга SVG.

```
svgBtn.addEventListener( 'click', function ( ) {
    htmlTxt.value = svgTxt.lastChild.wholeText
    svgTxt.lastChild.replaceWith( 'SVG Text' ) } )
```

6

Сохраните документы HTML и SVG на веб-сервере в папке «htdocs». Затем откройте веб-страницу в браузере и введите текст. Теперь нажмите кнопку и обведите, чтобы увидеть, как ваш текст перемещается между HTML и SVG.



Работа с холстами

Элемент HTML `<canvas>` создает на странице область холста растрового изображения, в которой с помощью JavaScript можно создавать фигуры и текст, используя методы и свойства API Canvas2D. Анимации легко создать на холсте с помощью многократного очищения текущего холста, а затем перекрашивая его формами и изменяя их положение быстрее, чем это может обнаружить человеческий глаз. Для создания анимации необходимо выполнить следующие действия:

- Инициализируйте объект Context и сформируйте начальные позиции.
- Очистите холст, затем нарисуйте фигуры.
- Рассчитайте новые положения формы для следующей перерисовки.

1

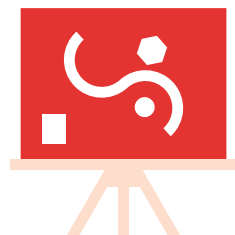
Создайте HTML-документ, включающий в основной части цветную область холста.

```
<canvas id="canvas" width="500" height="150"
  style="background: Bisque; border: 2px solid
  Tomato"> [Canvas Area]
</canvas>
```

2

Создайте функцию. После загрузки содержимого HTML проинициализируйте свойство объекта Context. Затем для определения начальной позиции «мяча» и шага объявите и проинициализируйте переменные, обозначающие размер холста с координатами X и Y.

```
function init() {
  const canvas = document.getElementById( 'canvas' )
  if ( canvas.getContext )
  {
    const context = canvas.getContext( '2d' )
    const cw = canvas.width
    const ch = canvas.height
    let x = 5 , y = 44 , dx = 5 , dy = 5
```



canvas.html



Более подробную информацию об API Canvas2D можно узнать на сайте w3.org/TR/2dcontext.



Чтобы рисовать на холсте, сценарий должен сначала создать объект `CanvasRenderingContext2D`, содержащий методы и свойства рисования. В нашем примере объект присваивается переменной с именем `context`. Обратите внимание, что функция инициализации вызывается при возникновении события `DOMContentLoaded`.



Обратите внимание, как направление меняется на противоположное, когда мяч сталкивается с периметром, поэтому мяч от холста не отскакивает.



Во избежание повторного раскрашивания создайте статический фон и границы в виде стилей.

```
context.fillStyle = 'Red'
// Здесь будет ваш код.
}
}
document.addEventListener( 'DOMContentLoaded', init )
```

3

Добавьте операторы для вычисления новых координат X и Y и вызовите функцию, чтобы рисовать мяч на холсте каждые 25 миллисекунд.

```
setInterval( function( ) {
    if ( ( x + dx > cw-30 ) || ( x + dx < 10 ) ) dx = -dx
    if ( ( y + dy > ch-30 ) || ( y + dy < 10 ) ) dy = -dy
    x += dx
    y += dy
    paint( context, cw, ch, x, y ), 25 )
```

4

Наконец, для рисования мяча на холсте добавьте следующую функцию.

```
function paint( context, cw, ch, x, y ) {
    context.clearRect( 0, 0, cw, ch )
    context.beginPath( )
    context.arc( x, y, 30, 0, ( Math.PI * 2 ), true )
    context.fill( )
}
```

5

Сохраните HTML-документ и программный код. Затем откройте в своем браузере веб-страницу и проанализируйте, каким образом мяч подпрыгивает на холсте.



Хранение данных

Объекты веб-хранилища **localStorage** и **sessionStorage** позволяют хранить данные в браузере. Объект **localStorage** хранит данные без срока годности и не удаляет после закрытия браузера. Объект **sessionStorage** хранит данные для одной сессии и при закрытии вкладки браузера данные теряются. Объекты хранилища **localStorage** и **sessionStorage** предоставляют одинаковые методы и свойства. В методе **setItem()** указываются два аргумента, которые должны быть сохранены. Например, **localStorage.setItem("Name", "Mike")**. Также сохраненные данные можно получить, указав атрибут в качестве единственного аргумента метода **getItem()**, или удалить, указав атрибут в качестве единственного аргумента метода **removeItem()**. Кроме того, все сохраненные данные легко удалить с помощью метода **clear()**.



Все данные методов **localStorage** и **sessionStorage** хранятся в виде строковых значений, поэтому полученные значения 5 и 7 в результате будут объединены как 57.

1

Создайте HTML-документ, содержащий поля для ввода текста и три кнопки.

```
<fieldset> <legend id="legend" >Enter Data</legend>
<input id="data" type="text" >
<button onclick="store()">Store Data</button>
<button onclick="recall()">Recall Data</button>
<button onclick="remove()">Reset</button>
</fieldset>
```



webstorage.html

2

Создайте для первой кнопки обработчик событий, чтобы сохранить данные в локальном хранилище, если они действительны.

```
function store() {
    let data = document.getElementById( 'data' ).value
    if(data === " ") {return false} else {
        localStorage.setItem( 'ls_data', data )
        document.getElementById( 'data' ).value = "
        document.getElementById( 'legend' ).innerText =
        localStorage.getItem( 'ls_data' ) + ' - Is Stored' )
    }
}
```




Чтобы использовать веб-хранилище, страницы должны обслуживаться через HTTP (с веб-сервера). HTTP предлагает альтернативу хранилищу файлов cookie и гораздо больший объем памяти, по крайней мере, 5 МБ.



Сокращенная форма записи позволяет вам просто указать атрибут в качестве имени объекта. Например, метод `localStorage.setItem('A', '1')` может быть записан как `localStorage.A='1'`, а метод `localStorage.getItem('A')` — как `localStorage.A`.

3

Затем привяжите обработчик событий ко второй кнопке, чтобы получить и отобразить данные в локальном хранилище.

```
function recall() {
    if (localStorage.getItem( 'ls_data' ) === null ) {
        document.getElementById( 'legend' ).innerText =
            'Enter Data' ; return false } else {
        document.getElementById( 'data' ).value = "
        document.getElementById( 'legend' ).innerText =
            'Stored Data: ' + localStorage.getItem( 'ls_data' )
        }
}
```

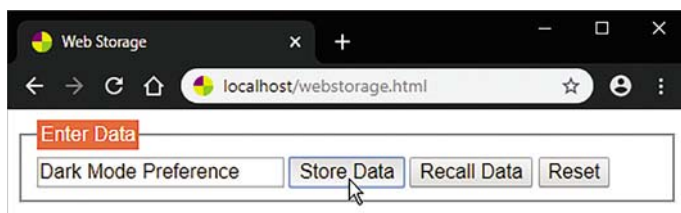
4

Наконец, создайте обработчик событий для третьей кнопки, чтобы удалить данные из локального хранилища.

```
function remove() {
    if ( localStorage.getItem( 'ls_data' ) === null ) {
        document.getElementById( 'legend' ).innerText
            = 'Enter Data'; return false } else {
        document.getElementById( 'legend' ).innerText =
            localStorage.getItem( 'ls_data' ) + ' Is Removed'
        localStorage.remove( 'ls_data' )
        document.getElementById( 'data' ).value = ""
    }
}
```

5

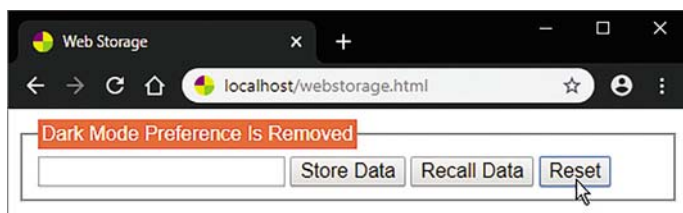
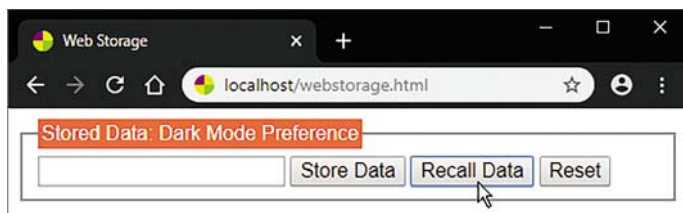
Сохраните HTML-документ на веб-сервере в папке «htdocs». Затем в своем браузере откройте веб-страницу, введите текст и нажмите кнопку Store Data (Сохранить данные).



6

Перезагрузите браузер и снова откройте свою веб-страницу. Нажмите кнопку Recall

Data (Вызвать данные), чтобы увидеть, что данные были сохранены. Для удаления данных нажмите кнопку Remove Data (Удалить данные).

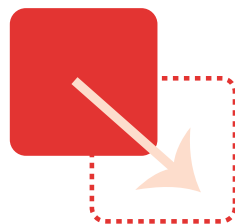


Более подробную информацию о хранении данных вы можете найти на сайте www.w3.org/TR/webstorage

Перемещение элементов

Перемещение пользователем различных элементов страницы поддерживается технологией Drag-and-Drop. Она определяет множество событий, которые запускаются, когда пользователь перетаскивает объект. Разберем наиболее важные и необходимые события для создания Drag-and-Drop: «dragstart», «dragover» и «drop». Обработчики событий должны быть написаны для каждого из следующих событий:

- **ondragstart** — возникает, когда пользователь начинает перетаскивать элемент или выделенный текст.
- **ondragover** — возникает, когда перетаскиваемый элемент или выделенный текст перетаскивается через допустимую цель перетаскивания.
- **ondrop** — возникает, когда перетаскиваемый элемент или выделенный текст отбрасывается на допустимую цель перетаскивания.





dragndrop.html

Кроме того, обработчик событий **ondrop** должен гарантировать, что цель не может быть отброшена сама на себя.

- 1 Создайте HTML-документ с абзацем, содержащим изображения и пустой список.

```
<p>




</p>
<fieldset>
<legend>Folders Dropped:</legend>
<ol id="list" ></ol>
</fieldset>
```



Folders Dropped: _____

- 2 После полной загрузки DOM добавьте метод `document.addEventListener`, который привязывает обработчик событий.

```
document.addEventListener ( 'DOMContentLoaded',
dragNdDrop )
```

- 3 Создайте функцию для получения ссылок на изображения и элементы списка.

```
function dragNdDrop( ) {
  const bin = document.getElementById( 'bin' )
  const folders = document. getElementsByClassName(
'folder' )
  const list = document.getElementById( 'list' )
  // Здесь будет ваш код.
}
```

- 4 Добавьте операторы, чтобы привязать обработчики событий к каждому изображению

папки и отменить по умолчанию действия с корзиной.

```
let i = 0
for( i = 0 ; i < folders.length; i++ ) {
    folders[ i ].ondragstart = function( event ) {
        event.dataTransfer.setData( 'Text', this.id ) }
    }
    bin.ondragover = function( event ) { return false }
```

5

Наконец добавьте операторы, чтобы получить идентификатор и ссылку на элемент удаляемой папки. Затем (если это не корзина) запишите в список идентификатор и удалите ее изображение.

```
bin.ondrop = function( event ) {
    const did = event.dataTransfer.getData( 'Text' )
    const tag = document.getElementById( did )
    if ( did === 'bin' ) { return false }
    else { list.innerHTML += '<li>' + did }
    tag.parentNode.removeChild( tag )
}
```

6

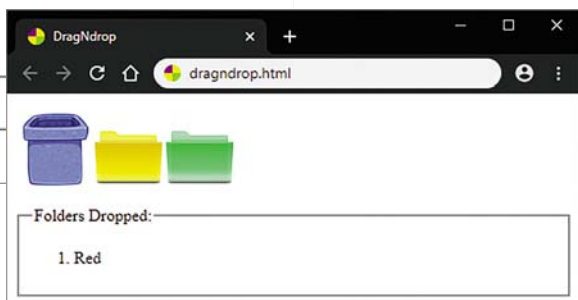
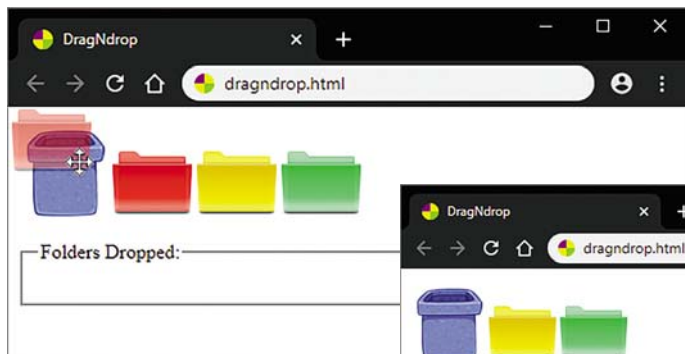
Сохраните HTML-документ, откройте его в браузере и переместите папки в корзину.



Обратите внимание, что аргументы метода `dataTransfer.setData()` события `dragstart` определяют формат данных и фактические данные. В нашем случае это формат «Text» и идентификатор перемещаемого элемента.



Более подробную информацию о технологии Drag-and-Drop вы можете найти на сайте dev.w3.org/html5/spec/dnd.html.





Чтобы использовать API Messaging, страницы должны обслуживаться через HTTP (с веб-сервера). В целях безопасности обмен сообщениями между документами будет успешным только в том случае, если отправитель корректно указывает домен получателя в методе `postMessage()`, а получатель проверяет домен отправителя в свойстве `event.origin`.

Связь между объектами window

API Messaging поддерживает безопасную передачу текстовых сообщений между документами. Это возможно даже в случае размещения документов на разных доменах. Например, документ, размещенный на локальном домене, включает `IFrame`, содержащий документ из другого домена, который может безопасно отправлять сообщения каждому из них.

Чтобы отправить сообщение в другой документ, отправляющему документу необходимо сначала получить ссылку на окно, где находится принимающий. Свойство `contentWindow` возвращает объект `window`, созданный элементом `IFrame`. Далее метод `postMessage()` обеспечивает безопасную связь между объектами `window` из разных источников. Метод `postMessage()` требует указания двух аргументов: отправляемого сообщения и целевого домена документа. Например, `otherWindow.postMessage('Hello', 'http://example.com')`.

Чтобы получить сообщение, отправленное из другого документа, сначала необходимо добавить к принимающему документу «прослушиватель» сообщения. Для этого необходимо, чтобы в методе `addEventListener()` этого окна были указаны три аргумента: аргумент, который должен прослушивать тип «сообщения»; функция обработчика событий для обработки сообщения и логическое значение `false`, указывающее, что дальнейшая обработка не требуется.

Сообщение передается обработчику событий как «событие», которое имеет свойство `origin`, содержащее домен отправляющего документа, и свойство `data` с текстом.



sender.html

1

Создайте HTML-документ, содержащий заголовков, абзац для отображения домена документа, `IFrame` для удаленного домена и кнопку Send Message (Отправить сообщение).

```
<h1>Sender</h1>
<p id="host" >Main Page Domain: </p>
<iframe id="cage" width = "450" height = "120"
src="http://example.com/receiver.html" > </iframe>
<button onclick="sendMsg( )" >Send Message</ button>
```

- 2 Создайте код для отображения домена, на котором размещен этот HTML-документ.

```
document.getElementById( 'host' ).innerText +=
document.domain
```

- 3 Добавьте функцию для отправки сообщения.

```
function sendMsg( ) {
const cage = document.getElementById( 'cage' )
.contentWindow
cage.postMessage( 'Message Received from: ' +
document.domain, 'http://example.com' )
}
```

- 4 Создайте другой HTML-документ, включающий заголовок, абзац для отображения домена документа и пустой абзац, в котором появится полученное сообщение.



receiver.html

```
<h1>Receiver</h1>
<p id="host" >Iframe Page Domain: </p>
<p id="msg"></p>
```

- 5 Создайте код для отображения домена, где размещен второй HTML-документ. Затем добавьте метод document.addEventListener, который присоединяет к документу обработчик событий.

```
document.getElementById( 'host' ).innerText +=
document.domain
window.addEventListener( 'message', readMsg )
```

- 6 Добавьте функцию для записи полученного сообщения.

```
function readMsg( event ) {
if ( event.origin === 'http://localhost' )
```

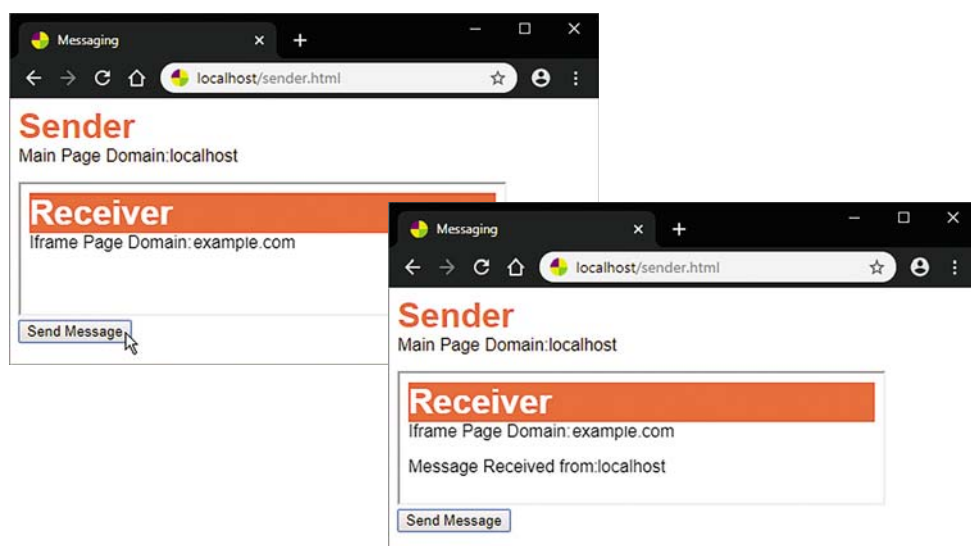


Более подробную информацию о связи между объектами Window вы можете найти на сайте dev.w3.org/html5/webstorage.

```
document.getElementById( 'msg' ).innerText =
    event.data
}
```

7

Сохраните документы в разных доменах и нажмите кнопку для отправки сообщений между документами.



Местоположение пользователей



Возможность точно определять географическое положение пользователя поддерживается API геолокации HTML. Из соображений конфиденциальности у пользователя запрашивается разрешение на предоставление сведений о местоположении для отправки информации о ближайших точках беспроводного доступа и IP-адресе компьютера, например, в службы геолокации Google. Эта служба возвращает приблизительные координаты широты и долготы местонахождения пользователя. Успешно полученные координаты могут быть отображены на странице и предоставлены службе Google Maps для указания на карте.

1

Создайте HTML-документ, содержащий в основной части два абзаца фиксированного размера.

```
<p id = "msg" style = "width:450px; height:50px" ></p>
<p id = "map" style = "width:450px; height:200px"></p>
```



geolocation.html

2

Добавьте в раздел заголовка документа элемент, чтобы получить ключ API для Google Maps.

```
<script src = "//maps.googleapis.com/maps/ api/
js?key=API_KEY " ></script>
```

3

В конце раздела тела документа создайте функцию, которая после загрузки HTML-документа ищет местоположение пользователя.

```
function init( ) {
    if (navigator.geolocation) {
        document.getElementById( 'msg' ).innerText =
            'Geolocation service is trying to find you...'
        navigator.geolocation.getCurrentPosition(
            success, fail )
    } else {
        document.getElementById( 'msg' ).innerText =
            'Your browser does not support Geolocation
            service'
    }
}
document.addEventListener( 'DOMContentLoaded', init )
```



В этом примере объект `navigator.geolocation` предоставляет метод `getCurrentPosition()`, который используется для получения информации о текущем положении пользователя. Возвращаются широта и долгота его позиции или сведения об ошибке.

4

Если попытка не удалась, добавьте функцию для отображения соответствующего сообщения.

```
function fail( position ) {
    document.getElementById( 'msg' ).innerText =
        'Geolocation service cannot find you at this time'
}
```



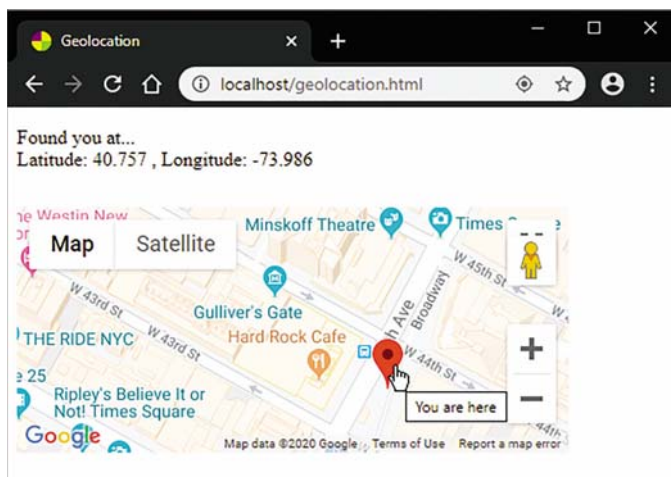
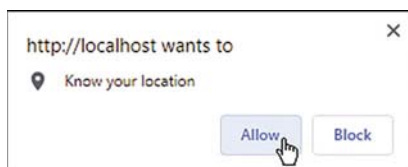

В первую очередь необходимо на сайте cloud.google.com войти в облачную платформу Google и создать новый проект. Затем включите в проекте Maps JavaScript API и получите ключ API. Чтобы запустить код, замените API_KEY на шаге 2 своим собственным ключом.

- 5 Добавьте функцию для отображения полученных координат и карты при успешном выполнении.

```
function success( position ) {
    const lat = position.coords.latitude
    const lng = position.coords.longitude
    document.getElementById( 'msg' ).innerHTML =
        'Found you at...<br>Latitude: ' + lat + ',
        Longitude: ' + lng

    const latLng = new google.maps.LatLng( lat, lng )
    const options = { zoom: 18 , center : latLng ,
        mapTypeId: google.maps.MapTypeId.ROADMAP}
    const map = new google.maps.Map ( document.
        getElementById( 'map' ) , options )
    const marker = new google.maps.Marker( { position:
        latLng , map: map , title: 'You are here' } )
}
```

- 6 Сохраните HTML-документ на локальном веб-сервере в папке «htdocs». Затем откройте веб-страницу и дайте разрешение на определение вашего местоположения.



Более подробную информацию об API геолокации вы можете узнать на сайте dev.w3.org/geo/api.

Заключение

- Технология AJAX позволяет обновлять веб-страницы, запрашивая данные с веб-сервера без прерывания других операций.
- Текст, полученный посредством объекта XMLHttpRequest, автоматически сохраняется в его свойстве responseText.
- SVG-графика, встроенная в документ HTML, имеет объектную модель (DOM). DOM в SVG, как и в HTML, содержит модель событий.
- Метод getSVGDocument() создает ссылающийся объект на документ SVG. Элементы, атрибуты и текст в нем могут быть адресованы через объект SVGDocument.
- Элемент HTML <canvas> создает на странице область холста растрового изображения, в которой с помощью JavaScript можно создавать фигуры и текст, используя методы и свойства API Canvas2D.
- Метод getContext() создает ссылающийся объект области холста.
- Объекты веб-хранилища localStorage (для долгосрочного хранения) и sessionStorage (для временного хранения) позволяют хранить данные в браузере.
- Объекты хранилища localStorage и sessionStorage предоставляют одинаковые методы и свойства. В методе setItem() указываются два аргумента, которые должны быть сохранены.
- Перемещение пользователем различных элементов страницы поддерживается технологией Drag-and-Drop.
- Наиболее важные и необходимые события для создания Drag-and-Drop — это «dragstart», «dragover» и «drop».

- API Messaging поддерживает возможность безопасной передачи текстовых сообщений между документами. Это реально даже в случае размещения документов в двух разных доменах.
- Чтобы получить сообщение, принимающий документ должен правильно идентифицировать домен, на котором размещен отправляющий документ.
- Возможность точно определять географическое положение пользователя поддерживается API геолокации HTML. Из соображений конфиденциальности у пользователя запрашивается разрешение на предоставление сведений о местоположении.
- Службы геолокации Google возвращают приблизительные координаты широты и долготы пользователя.
- Успешно полученные координаты могут быть отображены на странице и предоставлены службе Google Maps для указания на карте.

Предметный указатель

A–Z

API (Интерфейс прикладного программирования, Application Programming Interface)	158
браузера	158
геолокации	184
dataTransfer.getData(), метод перемещения объектов	181
Dbclick, событие при нажатии кнопки мыши	142
ECMAScript	8
Fetch API	160
IEEE (Институт инженеров по электротехнике и электронике)	94
Java	8
JavaScript	8
JavaScript Object Notation (JSON) (Нотация объектов JavaScript)	8, 156
JScript	8
LSB (младший значащий бит)	40
Messaging API	182
MSB (старший значащий бит)	40
Ondragover, атрибут	180
Ondragstart, атрибут	180
Ondrop, атрибут	180
URL-адрес	124
Web Storage API	178
XML (Расширяемый язык разметки)	156
XML, Объектная модель документа (DOM)	
Метод getSVGDocument()	174

A

Анимация (холст)	176
Анонимные функции	20
Аргументы функций	18
Арифметические операторы	
+ сложение	13, 30
-- декремент	30
/ деление	30
** возведение в степень	30
++ инкремент	30
% остаток от деления	30
* умножение	30
- вычитание	13, 30
Асинхронные операции	158
Асинхронный JavaScript и XML («AJAX»)	172
Атрибут	
id	10
объекта	64

Б

Блок функции	12, 18
--------------	--------

В

Веб-приложение	162, 164, 166, 168
Внешние файлы JavaScript	9
Всплывающие окна	116

Встроенные элементы, iframe	182
Выражения 13	
Ключевые слова	12
Операторы	12
Значения	12
Выход из цикла	58
Вычисление итоговых значений	166

Д

Данные, свойства управления событиями	182
Двоичное число	40
Диалоговые окна	112
Документ JSON	161, 164

З

Замыкания	24, 118
Запрос HTTP	161, 164
Значение null	102

И

Иерархия, DOM	108
Инструкции (операторы)	
Инструменты разработчика	10
Интерфейс	162

К

Ключевое слово	14–15
break	
в структуре цикла	58
в структуре оператора switch	50
case	50
catch	60
const	16, 22
continue	58
default	50
do	56

else	48
finally	60
for	52
if	46
in	66
let	13, 22
new	78, 158, 172
return	18
switch	50
this	64, 160
throw	60
try	60
typeof	17
var	16
while	54
Коды символов ASCII	144
Комментарии // и /* */	13
Конкатенация строк	30
Консоль JavaScript	10
Конфликт переменных	22
Координаты широты и долготы	184

Л

Лексическая область видимости	22
Литеральные значения	13
Логические операторы	
&& логическое И	36
! логическое НЕ	36
логическое ИЛИ	36
Логический тип данных	
Значение false (ложь)	16
Значение true (истина)	16
Логическое свойство checked	146

М

Массив	
elements[]	152
options[]	148
Метод	
join()	74
pop()	74

push()	74
reverse()	74, 76
shift()	74
slice()	74
sort()	74, 76
splice()	74
unshift()	74
Свойство length	72
Метасимволы	86
Метод	18
addEventListener()	120, 140, 182
alert()	10, 112
clear()	178
clearInterval()	118
clearTimeout()	118
confirm()	112
decodeURI()	138
encodeURI()	138
fetch()	160, 164
getCurrentPosition()	184
getElementById()	134
getElementsByClassName()	134
getElementsByTagName()	134, 172
getItem()	178
indexOf()	102
isNaN()	28
json()	160
json()	160, 164
lastIndexOf()	102
log()	10
parentNode.removeChild()	181
parseFloat()	28
parseInt()	28
postMessage()	182
preventDefault()	147, 152
prompt()	112
removeAttribute()	137
removeItem()	178
scrollBy()	114, 128
scrollTo()	114
setAttribute()	136
setInterval()	118
setItem()	178
setTimeout()	118, 158
split()	138

String()	28
toLocaleString()	68
toString()	28, 84
toUTCString()	138

H

Независимые кнопки	
или «флажки»	146
Немедленно вызываемые	
функции (IIFE)	20
Номера индексов начинаются	
с нуля	70

O

Область видимости	
Замыкания	25
Глобальная	22, 25
Локальная	22, 25
Обнаружение функции	120
Обновление таблицы	168
Обработка ошибок	60
Объект	68
Array	68
Boolean	64
Console	
Метод log()	10
Context	176
Date 78	
Конструктор Date()	78, 84
Метод	
getDate()	80
getDay()	80
getFullYear()	80
getHours()	82
getMilliseconds()	82
getMinutes()	82
getMonth()	80
getSeconds()	82
getTime()	78
getTimezoneOffset()	78
setDate()	84

setFullYear()	84	ReferenceError	60
setHours()	84	SyntaxError	60
setMilliseconds()	84	TypeError	60
setMinutes()	84	URIError	60
setMonth()	84	event	
setSeconds()	84	Метод preventDefault()	152
toDateString()	84	Свойство	
toLocaleString()	84	keyCode	144
toString()	78, 84	type	144
toTimeString()	84	x	144
toUTCString()	78, 84	y	144
document	130	form	
Дочерний объект		Массив elements[]	152
anchors	108	history	126
forms	132	Метод	
Метод		back()	126
appendChild()	136	forward()	126
createElement()	136	go()	126
Свойство		Свойство length	126
cookie	138	HTMLCollection	134
domain	130	HTTPResponse	
elements	108	JSON	156
Дочерний объект		Метод	
images	132	JSON.parse()	156
links	132	JSON.stringify()	156
scripts	132	localStorage	178
styleSheets	132	location	
Метод		Свойство	
getElementById()	134	hash	124
getElementsByClassName()	134	host	124
getElementsByTagName()	134	href	124
open()	136	pathname	124
removeChild()	137	protocol	124
replaceChild()	137	Math	90
write()	136	Константа	
Свойство		E	90
lastModified	130	LN2, константа LN10	90
referrer	130	LOG2E, константа LOG10E	90
title	130	PI	90
URL	130	SQRT1_2, константа SQRT2	90
error		Метод	
Error	60	abs()	92
EvalError	60	acos()	92
InternalError	60	asin()	92
RangeError	60	atan()	92

atan2()	92	String	98
ceil()	92, 94, 96	Метод	
cos()	92	charAt()	102
exp()	92	charCodeAt()	102
floor()	92, 94	concat()	98
log()	92	fromCharCode()	102, 144
max()	92	indexOf()	102
min()	92	lastIndexOf()	102
pow()	92	match()	102
random()	92, 96	replace()	102
round()	92, 94	search()	102
sin()	92	slice()	100
sqrt()	92	split()	100, 106
tan()	92	substring()	100
navigator	120, 122	substr()	100
Дочерний объект		toLowerCase()	98
geolocation	184	toUpperCase()	98
mimeType	122	Свойство length	98
plugins	122	SVGDocument	
Метод javaEnabled()	122	Метод getSVGDocument()	174
Свойство		Узел lastChild	
appName	120	Метод replaceWith()	175
appName	120	Свойство wholeText	175
appVersion	120	window	108, 140
cookieEnabled	122	Метод	
platform	120	addEventListener()	182
userAgent	120	alert()	10
Promise	158, 160	close()	116
Метод		moveBy()	116
catch()	158, 160, 164	moveTo()	116
then()	158, 160, 164	open()	116
RegExp	86	print()	116
Метод		XMLHttpRequest	172
exec()	86	Метод	
test()	86	open()	172
screen		send()	172
Дочерний объект orientation		Свойство	
Свойство		firstChild	172
availHeight	110	onreadystatechange	172
availWidth	110	readyState	172
colorDepth	110	responseText	172
height	110	responseXML	172
type	110	создание	68
width	110	Объектная модель документа	
sessionStorage	178	(DOM)	8, 108

Окно консоли	10
Раскрывающийся список	
constructor	25
Округление чисел	94
Оператор .	42
вызова ()	20
группировки выражений ()	42
доступа к массиву	42
конкатенации +	17
побитовый	
& ~ ^ << >> >>>	40
присваивания	
+= присваивание	
со сложением	32
= присваивание	13, 32
/= присваивание	
с делением	32
**= присваивание	
и возведение в степень	32
%= присваивание по модулю	32
*= присваивание	
с умножением	32
-= присваивание	
с вычитанием	32
сравнения	
=== строго равно	34
> больше	34
>= больше или равно	34
!== строго не равно	34
< меньше	34
<= меньше или равно	34
умножения *	19
Очередь обратных вызовов	158

П

Папка htdocs	161
Пара key:value (атрибут:	
значение)	156, 170
Переключатели с зависимой	
фиксацией или «радиокнопки»	146
Переменные	13, 16
Объявление	16
Соглашение по именованию	14

Поведение	64
Подстрока	100
Пользовательские объекты	64, 66
Преобразование значений	28
Примитивные значения	68
Приоритет операций	42
Пробел	12, 104
Проверка	86
Проверка электронного	
адреса	87, 152
Прототип свойства	25

Р

Разрешение экрана	110
Раскрывающийся список	148
Расширенные объекты	66

С

Самовывзывающаяся функция	20, 24
Свойства, объект window	108
Свойство	
API Canvas2D	176
clientHeight	115
clientWidth	115
contentWindow (iframe)	182
description	122
firstChild	
data	172
innerHTML	136
innerText	10, 136
length	72
message, объект error	60
name, массив plugins	122
name, объект error	60
NaN	28
onchange	148
onclick	142
ondblclick	142
onfocus	150
onfocusout	150
onkeydown	144

Ц

Цикл	
for in	66, 108-109
событий	158

Ч

Частные переменные	25
--------------------	----

Числовой тип данных	16
Чувствительность к регистру	13

Э

Элементы массива	70
Эпоха, 1 января 1970 года	
00:00:00 GMT	78

Основные функции JavaScript

// Script Author.	Комментарий – однострочный
// Разработчик сценария (кода).	
/* Script Description */	Комментарий – однострочный
/* Описание сценария (кода) */	или многострочный
console.log(...)	Вывод – в консоль JavaScript
alert(...)	Вывод – в диалоговое окно
let greeting='Hello World!'	Переменная – можно переназначить
const pi=3.142	Постоянная – нельзя переназначить
let chars=['A','B','C']	Переменная массива – где char[0] – A
function fname(){...}	Функция – без определенных параметров
function add (a,b){return a+b}	Функция – возвращает сумму двух аргументов
(function){...}()	Функция – анонимная и самовывзывающаяся
+ - * / % ++ -- **	Операторы – арифметические
= += -= *= /= %= **=	Операторы – присваивания
=== != > < >= <=	Операторы – сравнения
&& !	Операторы – логические
? :	Оператор – условный (тернарный)
& ~ ^ << >> >>>	Операторы – побитовые
if (condition){...}else{...}	Условное ветвление
switch (expr){case val-ue...;break}	Сравнение через оператор switch
for (i=0;i<n;i++){...}	Цикл – код тела цикла выполняется, если i меньше n
while(i<n){...}	Цикл – код тела цикла выполняется, пока условие истинно
do{...}while(i<n)	Цикл – цикл сначала выполнит тело, а затем проверит условие
try{...}catch{...}finally{...}	Обработка ошибок – с помощью данной конструкции исключение (ошибку) можно перехватить и обработать
let user={name:'Mike';title:'Author'}	Объект – user.name – Mike
user.who=function(){return this.name}	Метод объекта – user.who() получает значение name
const now=new Date()	Объект Date – создает объект Date с текущей датой и временем
const pattern=/.+@.+\./	Регулярные выражения – основной формат электронной почты
setTimeout('fname',1000)	Метод – задержка выполнения функции – 1 с
addEventListener('click',fname)	Метод – указывает функцию обработчика событий
const info=document. getElementById('info')	Ссылка на элемент – по значению атрибута id
info.innerHTML+=...'	Свойство элемента – добавляется к содержанию элемента

Все права защищены. Книга или любая ее часть не может быть скопирована, воспроизведена в электронной или механической форме, в виде фотокопии, записи в память ЭВМ, репродукции или каким-либо иным способом, а также использована в любой информационной системе без получения разрешения от издателя. Копирование, воспроизведение и иное использование книги или ее части без согласия издателя является незаконным и влечет уголовную, административную и гражданскую ответственность.

Научно-популярное издание

МИРОВОЙ КОМПЬЮТЕРНЫЙ БЕСТСЕЛЛЕР

МакГрат Майк

JAVASCRIPT ДЛЯ НАЧИНАЮЩИХ

Главный редактор *Р. Фасхутдинов*
Руководитель направления *В. Обручев*
Ответственный редактор *Е. Истомина*
Литературный редактор *Н. Домнина*
Младший редактор *Д. Данилова*
Художественный редактор *А. Шуклин*
Компьютерная верстка *Б. Руссо*
Корректор *Л. Макарова*

Страна происхождения: Российская Федерация
Шығарылған елі: Ресей Федерациясы

ООО «Издательство «Эксмо»

123308, Россия, город Москва, улица Зорге, дом 1, строение 1, этаж 20, каб. 2013.
Тел.: 8 (495) 411-68-86.

Home page: www.eksmo.ru E-mail: info@eksmo.ru

Өндіруші: «ЭКСМО» АҚБ Баспасы,

123308, Ресей, қала Мәскеу, Зорге көшесі, 1 үй, 1 ғимарат, 20 қабат, офис 2013 ж.
Тел.: 8 (495) 411-68-86.

Home page: www.eksmo.ru E-mail: info@eksmo.ru

Тауар белгісі: «Эксмо»

Интернет-магазин : www.book24.ru

Интернет-магазин : www.book24.kz

Интернет-дүкен : www.book24.kz

Импортёр в Республику Казахстан ТОО «РДЦ-Алматы».

Қазақстан Республикасындағы импорттаушы «РДЦ-Алматы» ЖШС.

Дистрибьютор и представитель по приему претензий на продукцию,

в Республике Казахстан: ТОО «РДЦ-Алматы»

Қазақстан Республикасында дистрибьютор және өнім бойынша арыз-талаптарды

қабылдаушының өкілі «РДЦ-Алматы» ЖШС.

Алматы қ., Домбровский көш., 3-а, литер Б, офис 1.

Тел.: 8 (727) 251-59-90/91/92; E-mail: RDC-Almaty@eksmo.kz

Өнімнің жарамдылық мерзімі шектелмеген.

Сертификация туралы ақпарат сайтта: www.eksmo.ru/certification

Сведения о подтверждении соответствия издания согласно законодательству РФ
о техническом регулировании можно получить на сайте Издательства «Эксмо»:

www.eksmo.ru/certification

Өндірген мемлекет: Ресей. Сертификация қарастырылмаған

Дата изготовления / Подписано в печать 08.08.2022.

Формат 70х100^{1/16}. Печать офсетная. Усл. печ. л. 18,8.

Тираж экз. Заказ

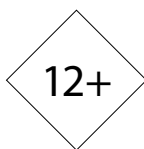
 **БОМБОРА**
ИЗДАТЕЛЬСТВО

БОМБОРА – лидер на рынке полезных и вдохновляющих книг.
Мы любим книги и создаем их, чтобы вы могли творить, открывать
мир, пробовать новое, расти. Быть счастливыми. Быть на волне.

 bombora.ru

 [bomborabooks](https://t.me/bomborabooks)

 [bombora](https://www.instagram.com/bombora)



ISBN 978-5-04-121621-4



9 785041 216214 >

В электронном виде книги издательства вы можете
купить на www.litres.ru

ЛитРес:
одна клик до книг



Москва. ООО «Торговый Дом «Эксмо»

Адрес: 123308, г. Москва, ул. Зорге, д.1, строение 1.

Телефон: +7 (495) 411-50-74. **E-mail:** reception@eksmo-sale.ru

По вопросам приобретения книг «Эксмо» зарубежными оптовыми
покупателями обращаться в отдел зарубежных продаж ТД «Эксмо»

E-mail: international@eksmo-sale.ru

*International Sales: International wholesale customers should contact
Foreign Sales Department of Trading House «Eksmo» for their orders.
international@eksmo-sale.ru*

По вопросам заказа книг корпоративным клиентам, в том числе в специальном
оформлении, обращаться по тел.: +7 (495) 411-68-59, доб. 2261.

E-mail: ivanova.ey@eksmo.ru

Оптовая торговля бумажно-беловыми

и канцелярскими товарами для школы и офиса «Канц-Эксмо»:

Компания «Канц-Эксмо»: 142702, Московская обл., Ленинский р-н, г. Видное-2,
Белокаменное ш., д. 1, а/я 5. Тел./факс: +7 (495) 745-28-87 (многоканальный).

e-mail: kanc@eksmo-sale.ru, сайт: www.kanc-eksmo.ru

Филиал «Торгового Дома «Эксмо» в Нижнем Новгороде

Адрес: 603094, г. Нижний Новгород, улица Карпинского, д. 29, бизнес-парк «Грин Плаза»

Телефон: +7 (831) 216-15-91 (92, 93, 94). **E-mail: reception@eksmonn.ru**

Филиал ООО «Издательство «Эксмо» в г. Санкт-Петербурге

Адрес: 192029, г. Санкт-Петербург, пр. Обуховской обороны, д. 84, лит. «Е»

Телефон: +7 (812) 365-46-03 / 04. **E-mail: server@szko.ru**

Филиал ООО «Издательство «Эксмо» в г. Екатеринбург

Адрес: 620024, г. Екатеринбург, ул. Новинская, д. 2щ

Телефон: +7 (343) 272-72-01 (02/03/04/05/06/08)

Филиал ООО «Издательство «Эксмо» в г. Самаре

Адрес: 443052, г. Самара, пр-т Кирова, д. 75/1, лит. «Е»

Телефон: +7 (846) 207-55-50. **E-mail: RDC-samara@mail.ru**

Филиал ООО «Издательство «Эксмо» в г. Ростове-на-Дону

Адрес: 344023, г. Ростов-на-Дону, ул. Страны Советов, 44А

Телефон: +7(863) 303-62-10. **E-mail: info@rnd.eksmo.ru**

Филиал ООО «Издательство «Эксмо» в г. Новосибирске

Адрес: 630015, г. Новосибирск, Комбинатский пер., д. 3

Телефон: +7(383) 289-91-42. **E-mail: eksmo-nsk@yandex.ru**

Обособленное подразделение в г. Хабаровске

Фактический адрес: 680000, г. Хабаровск, ул. Фрунзе, 22, оф. 703

Почтовый адрес: 680020, г. Хабаровск, А/Я 1006

Телефон: (4212) 910-120, 910-211. **E-mail: eksmo-khv@mail.ru**

Республика Беларусь: ООО «ЭКМО АСТ Си энд Си»

Центр оптово-розничных продаж Cash&Carry в г. Минск

Адрес: 220014, Республика Беларусь, г. Минск, проспект Жукова, 44, пом. 1-17, ТЦ «Outleto»

Телефон: +375 17 251-40-23; +375 44 581-81-92

Режим работы: с 10.00 до 22.00. **E-mail: exmoast@yandex.by**

Казахстан: «РДЦ Алматы»

Адрес: 050039, г. Алматы, ул. Домбровского, 3А

Телефон: +7 (727) 251-58-12, 251-59-90 (91,92,99). **E-mail: RDC-Almaty@eksmo.kz**

**Полный ассортимент продукции ООО «Издательство «Эксмо» можно приобрести в книжных
магазинах «Читай-город» и заказать в интернет-магазине: www.chitai-gorod.ru.**

Телефон единой справочной службы: 8 (800) 444-8-444. Звонок по России бесплатный.

Интернет-магазин ООО «Издательство «Эксмо»

www.book24.ru

Розничная продажа книг с доставкой по всему миру.

Тел.: +7 (495) 745-89-14. **E-mail: imarket@eksmo-sale.ru**



book 24.ru

Официальный
интернет-магазин
издательской группы
«ЭКМО-АСТ»

ЛУЧШИЕ КНИГИ О БИЗНЕСЕ С ЛОГОТИПОМ ВАШЕЙ КОМПАНИИ? ЛЕГКО!

Удивить своих клиентов, бизнес-партнеров, сделать памятный подарок сотрудникам и рассказать о своей компании читателям бизнес-литературы? Приглашаем стать партнерами выпуска актуальных и популярных книг. О вашей компании узнает наиболее активная аудитория.

ПАРТНЕРСКИЕ ОПЦИИ:

- Специальный тираж уже существующих книг с логотипом вашей компании.
- Размещение логотипа на супер-обложке для малых тиражей (от 30 штук).
- Поддержка выхода новинки, которая ранее не была доступна читателям (50 книг в подарок).

ПАРТНЕРСКИЕ ВОЗМОЖНОСТИ:

- Рекламная полоса о вашей компании внутри книги.
- Вступительное слово в книге от первых лиц компании-партнера.
- Обращение первых лиц на суперобложке.
- Отзыв на обороте обложки вложение информационных материалов о вашей компании (закладки, листовки, мини-буклеты).



У вас есть возможность обсудить свои пожелания с менеджерами корпоративных продаж. Как?

Звоните:

+7 495 411 68 59, доб. 2261

Заходите на сайт:

eksmo.ru/b2b



ПОСЛЕ ПРОЧТЕНИЯ ЭТОЙ КНИГИ ВЫ ВЫУЧИТЕ JAVASCRIPT

**ЦВЕТНЫЕ ПРИМЕРЫ СДЕЛАЮТ КОД ПОНЯТНЕЕ,
И ВЫ БЕЗ ТРУДА ОСВОИТЕ:**

- **ОСНОВЫ ЯЗЫКА – СИНТАКСИС, КЛЮЧЕВЫЕ СЛОВА,
ОПЕРАТОРЫ, СТРУКТУРУ
И ВСТРОЕННЫЕ ОБЪЕКТЫ**
- **ФУНКЦИОНАЛЬНОСТЬ ВЕБ-СТРАНИЦ**
- **ВЕБ-ПРИЛОЖЕНИЯ**

С помощью этих значков
ваше обучение станет проще
и понятнее



ПОЛЕЗНЫЕ СОВЕТЫ



ВЫДЕЛЯЕТСЯ ЦВЕТОМ
ТО, ЧТО НЕОБХОДИМО
ЗАПОМНИТЬ



ПРЕДУПРЕЖДАЕТ ВАС
О ПОТЕНЦИАЛЬНОЙ
ОПАСНОСТИ!

Вы получите отличное представление о JavaScript и сможете добавлять интересные динамические сценарии на свои веб-страницы. Вам не нужно знать какой-либо язык сценариев, поэтому данный самоучитель идеально подходит для новичков в JavaScript.



**СКАЧАЙТЕ ПРИМЕРЫ КОДА БЕСПЛАТНО
И ПРИСТУПАЙТЕ К РАБОТЕ СРАЗУ ЖЕ!**

ISBN 978-5-04-121621-4



9 785041 216214 >

БОМБОРА
издательство

БОМБОРА – лидер на рынке полезных и вдохновляющих книг. Мы любим книги и создаем их, чтобы вы могли творить, открывать мир, пробовать новое, расти. Быть счастливыми. Быть на волне.

bombora.ru [bomborabooks](https://www.bomborabooks.ru) [bombora](https://www.bombora.ru)

