БИБЛИОТЕКА ПРОГРАММИСТА

Кирупа Чиннатхамби

JAVA SCRIPT

С НУЛЯ

JavaScript

Second Edition

ABSOLUTE BEGINNER'S GUIDE

Kirupa Chinnathambi

que®

JAVA SCRIPT С НУЛЯ

Кирупа Чиннатхамби

Санкт-Петербург • Москва • Минск

2021

Кирупа Чиннатхамби

JavaScript с нуля

Серия «Библиотека программиста»

Перевел с английского Д. Акуратер

Заведующая редакцией	*Ю. Сергиенко*
Руководитель проекта	*А. Питиримов*
Ведущий редактор	*К. Тульцева*
Литературный редактор	*Н. Нефидова*
Обложка	*В. Мостипан*
Корректоры	*Н. Петрова, М. Одинокова*
Верстка	*Е. Неволайнен*

ББК 32.988.02-018

УДК 004.738.5

Кирупа Чиннатхамби

К43 JavaScript с нуля. — СПб.: Питер, 2021. — 400 с.: ил. — (Серия «Библиотека программиста»).

ISBN 978-5-4461-1701-7

JavaScript еще никогда не был так прост! Вы узнаете все возможности языка программирования без общих фраз и неясных терминов. Подробные примеры, иллюстрации и схемы будут понятны даже новичку. Легкая подача информации и живой юмор автора превратят нудное заучивание в занимательную практику по написанию кода. Дойдя до последней главы, вы настолько прокачаете свои навыки, что сможете решить практически любую задачу, будь то простое перемещение элементов на странице или даже собственная браузерная игра.

16+ (В соответствии с Федеральным законом от 29 декабря 2010 г. № 436-ФЗ.)

ISBN 978-0136502890 англ. Authorized translation from the English language edition, entitled JAVASCRIPT ABSOLUTE BEGINNER'S GUIDE, 2nd Edition by KIRUPA CHINNATHAMBI, published by Pearson Education, Inc, publishing as Que Publishing.

ISBN 978-5-4461-1701-7

Изготовлено в России. Изготовитель: ООО «Прогресс книга».
Место нахождения и фактический адрес: 194044, Россия, г. Санкт-Петербург,
Б. Сампсониевский пр., д. 29А, пом. 52. Тел.: +78127037373.
Дата изготовления: 04.2021. Наименование: книжная продукция.
Срок годности: не ограничен.
Налоговая льгота — общероссийский классификатор продукции ОК 034-2014,
58.11.12 — Книги печатные профессиональные, технические и научные.
Импортер в Беларусь: ООО «ПИТЕР М», 220020, РБ, г. Минск,
ул. Тимирязева, д. 121/3, к. 214, тел./факс: 208 80 01.
Подписано в печать 24.03.21. Формат 70x100/16. Бумага офсетная. Усл. п. л. 32,250. Тираж 1000. Заказ

КРАТКОЕ СОДЕРЖАНИЕ

ЧАСТЬ III. DOM, милый DOM

ЧАСТЬ IV. Полны событиями

ОГЛАВЛЕНИЕ

ЧАСТЬ III. DOM, милый DOM

ЧАСТЬ IV. Полны событиями

Мине!

(Той, кто до сих пор смеется над шутками из этой книги, перечитывая ее в тысячный раз.)

Благодарности

Теперь-то я знаю, что подготовка книги — дело непростое! К этому процессу подключились многие. Они были на линии фронта и в поте лица перекраивали мои нестройные рассуждения в прекрасные тексты, которые вы вот-вот прочтете. Благодарю всех коллег из издательства «Pearson», которые дали возможность этой книге появиться на свет!

Кроме того, хотел бы выразить отдельную благодарность некоторым людям. Во-первых, большое спасибо Марку Тейберу (Mark Taber) за эту возможность, Киму Спенсли (Kim Spenceley) за помощь в подготовке второго издания книги, Крису Зану (Chris Zahn) за скрупулезную проверку текста на читабельность, а также Лоретте Йейтс (Loretta Yates) за помощь в поиске нужных людей, благодаря которым удалось все это осуществить. Книгу внимательно вычитали мои старые друзья и онлайн-компаньоны — Кайл Мюррей (Kyle Murray, 1-е издание) и Тревор Маккаули (Trevor McCauley, 1-е и 2-е издания). Не устану благодарить их за подробный и полный юмора отзыв.

И наконец, хочу сказать спасибо своим родителям за то, что всегда поощряли мои хобби, такие как рисование, сочинение текстов, игра в компьютерные игры и программирование. Не будь их поддержки, из меня бы попросту не получился такой усидчивый домосед ☺.

Об авторе

Большую часть своей жизни автор, **Кирупа Чиннатхамби**, вдохновляет людей полюбить веб-разработку так, как он сам.

В 1999 году, еще до появления слова «блогер», он начал размещать обучающие материалы на форуме **kirupa.com**. С тех пор он написал сотни статей, несколько книг (конечно, ни одна из них не идет ни в какое сравнение с этой!), а также записал множество видео, которые вы можете найти на YouTube. Свободное от работы над книгой и разработок время он посвящает совершенствованию веб-сервисов в качестве менеджера по продукту в *Lightning Design System* на *SalesForce*. В выходные он, скорее всего, либо дрыхнет, либо бегает вместе с Миной за их крохотной дочуркой Акирой, защищаясь от Пикселя (он же тираннозавр рекс в теле кошки), или пишет о себе от третьего лица.

Вы можете найти Кирупу в твиттере и фейсбуке, а также во многих других уголках интернета, просто введя в поисковой строке его имя.

От издательства

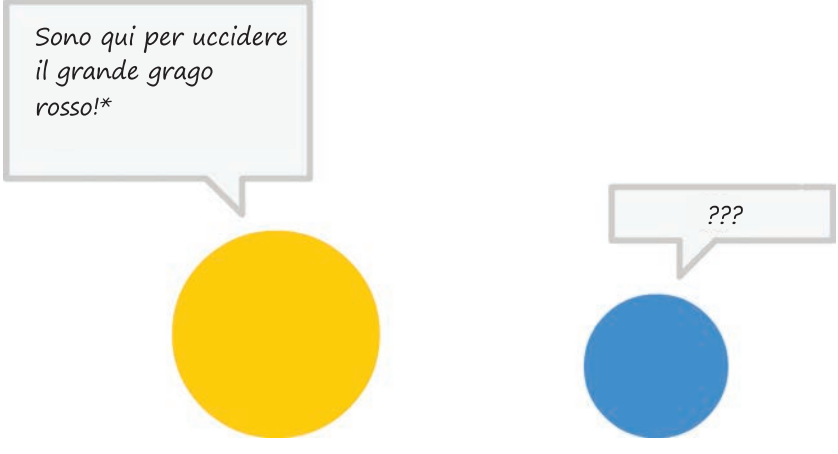
Ваши замечания, предложения, вопросы отправляйте по адресу comp@piter.com (издательство «Питер», компьютерная редакция).

Мы будем рады узнать ваше мнение!

На веб-сайте издательства www.piter.com вы найдете подробную информацию о наших книгах.

ВВЕДЕНИЕ

Вы когда-либо пытались научиться читать, писать или говорить на неродном языке? Если вы были таким же настырным, как и я, скорее всего, эти попытки поначалу выглядели как-то так:



* *Я здесь, чтобы убить большого красного дракона (итал.)!*

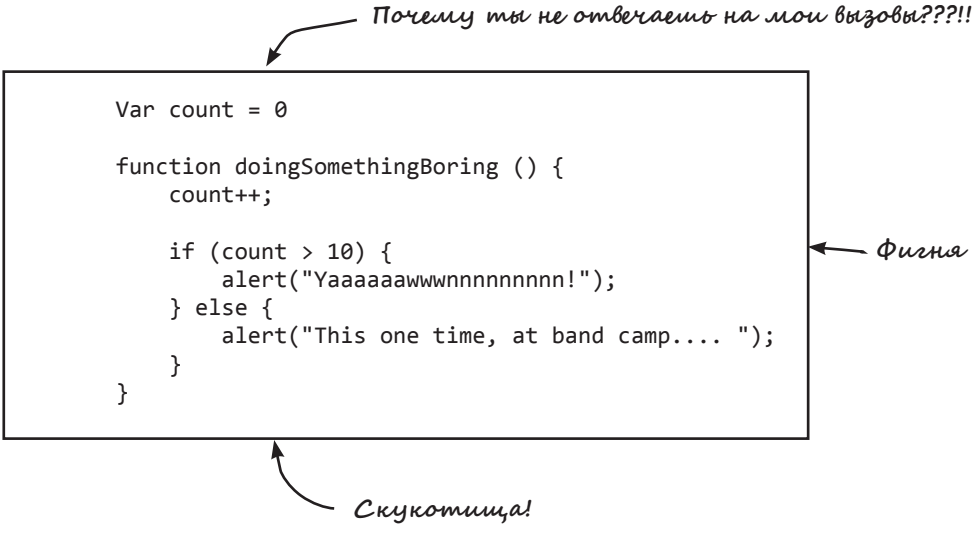
Если вы не Джейсон Борн или Роджер Федерер, вероятно, вам едва удалось выжить после освоения чужого языка. Все потому, что дело это тяжелое. Причем неважно, какой это язык по счету: ваш родной, второй или третий. На освоение языка на том уровне, на котором вы не будете звучать глупо, уходит много времени и усилий.

Все начинается с прохождения азов и требует большого количества практики и терпения. Это одна из тех областей, в которой нет коротких путей, чтобы стать профи.

Парле ву JavaScript?

Подход, который мы применяем для успешного изучения языка *программирования*, напоминает тот, которым вы пользуетесь для овладения *естественным* языком. Вначале вы беретесь за основы и, поднаторев, переходите к более продвинутым задачам. Весь этот процесс цикличен и по своей сути непрерывен. На самом деле никто из нас не перестает учиться, нужно лишь с чего-то начать. А с этим вам как раз поможет эта книга. От начала и до конца она наполнена всевозможными полезными (и хочется верить, веселыми!) знаниями, которые позволят вам освоить JavaScript.

А теперь, пусть даже я терпеть не могу плохо отзываться о языке за его спиной, я заявляю, что JavaScript весьма уныл и скучен:



Описать это иначе невозможно. Но несмотря на все это уныние, не стоит думать, что его освоение должно быть таким же (ЧТЗ: вся грамматическая белиберда тщательно разобрана по полочкам — чаще всего!). Надеюсь, что по мере изучения материала обыденные примеры языка и иллюстрации покажутся вам не только информативными, но и уморительными (инфорительными!).

Баланс обыденности и юмора нужен, чтобы разбавить процесс погружения в мир всего того интересного в JavaScript и того, что пригодится вам для продуктивной работы с этим языком.

Дойдя до последней главы, вы будете настолько прокачены, что сможете ответить практически на любой вызов от JavaScript, даже не успев вспотеть от напряжения.

Связь со мной

Если вы вдруг застрянете на каком-нибудь материале или вам просто захочется со мной поболтать, оставляйте свои сообщения на форуме:

forum.kirupa.com.

По другим вопросам вы можете писать на электронную почту (kirupa@kirupa.com), в твиттер @kirupa или фейсбук (facebook.com/kirupa). Я люблю общаться с читателями и обычно сам отвечаю на каждое сообщение.

А теперь переворачиваем страницу и поехали!

В ЭТОЙ ГЛАВЕ:

- разберемся, почему JavaScript прекрасен;
- немного попотеем, разбирая простой код;
- узнаем, что нас ждет впереди.

1

HELLO, WORLD!

Суть HTML в отображении, а CSS — в хорошей картинке. С помощью их обоих вы можете создать нечто прекрасное вроде примера с прогнозом погоды, доступного для просмотра здесь: http://bit.ly/kirupaWeather. На рис. 1.1 показано, как это выглядит.

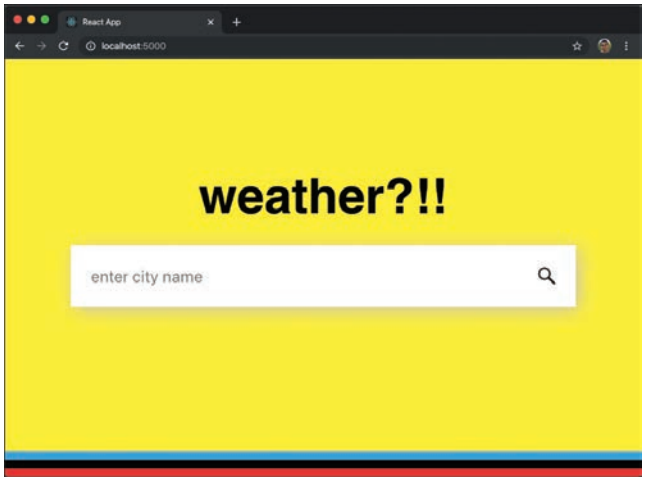


РИС. 1.1.
Приложение прогноза погоды — яркий пример дизайна страницы, выполненного с помощью CSS

Несмотря на то как изящно смотрятся сайты, выполненные с помощью CSS и HTML, они до боли статичны. Они не подстроятся и не среагируют на ваши действия. Эффект, созданный этой парочкой, напоминает непрерывный просмотр любимой серии «Друзей», что рано или поздно все равно навеет на вас скуку. Сайты, которые вы используете часто (вроде тех, что изображены на рис. 1.2), обладают некоторой степенью интерактивности и персонализации, что само по себе значительно выходит за пределы возможностей HTML и CSS.

Для оживления контента на вашем сайте вам пригодится сторонний помощник. Им вполне может стать JavaScript!

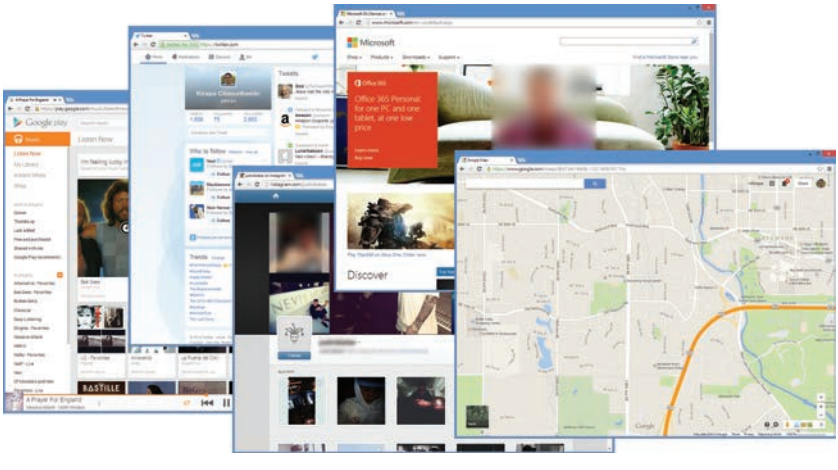


РИС. 1.2.
Примеры сайтов, функциональность которых во многом полагается на JavaScript

Что такое JavaScript?

JavaScript — это современный язык программирования и партнер HTML и CSS. Если коротко, этот язык позволяет привнести в документ интерактивность. Вот краткий перечень возможностей, которые предоставляет JavaScript:

- Прослушивание событий, будь то клик мыши или выполнение команды.
- Изменение HTML- и CSS-страниц после их загрузки.
- Задание незаурядных траекторий перемещения объектов по экрану.

- Создание увлекательнейших браузерных игр вроде Cut the Rope.
- Обмен данными между сервером и браузером.
- Взаимодействие с веб-камерой, микрофоном и другими устройствами.

А также многое другое! Написание же JavaScript кода весьма несложно. Вы комбинируете слова, большинство из которых встречается в английском языке, отдавая таким образом браузеру команды. Следующий пример — некое сочетание классики JavaScript в новой доработке:

```
let defaultName = "JavaScript";

function sayHello(name) {
  if (name == null) {
    alert("Hello, " + defaultName + "!");
  } else {
    alert("Hello, " + name + "!");
  }
}
```

Ничего страшного, если вам пока ничего не понятно. Просто обратите внимание на то, как этот код выглядит. Заметьте, что использовано много английских слов: `function`, `if`, `else`, `alert`, `name`. Помимо них есть еще странные символы и знаки из тех областей клавиатуры, куда взгляд обычно не падает. Но скоро, по мере того как их количество начнет расти на ваших глазах, вы постепенно разберетесь, за что отвечает каждый элемент приведенного кода.

Как бы там ни было, пока что этой вспомогательной информации достаточно. Хоть вы, возможно, ожидали найти здесь историю JavaScript и справку о его создателях, я не буду утомлять вас подобными скучными вещами. Вместо этого мне хотелось бы, чтобы вы немного потренировались в написании JavaScript-кода и к концу урока создали что-нибудь милое и простое, отображающее текст в браузере.

Hello, world!

Возможно, сейчас вы ощутите недостаток навыков для написания кода. Тем более если вы не знакомы с программированием даже в общих чертах. Но уже скоро вы поймете, что JavaScript не такой бесячий и сложный, каким ему хочется казаться. Итак, приступим.

ВАЖНО ЗНАТЬ ОСНОВЫ ВЕБ-ПРОГРАММИРОВАНИЯ

Чтобы начать писать на JavaScript, вам нужно знать основы создания веб-страниц, использования редактора кода и добавления HTML/CSS. Если вы не знакомы с этими вещами, призываю вас для начала прочесть статью «Создание вашей первой веб-страницы» (**https://www.kirupa.com/html5/building_your_first_web_page.htm**). Это поможет плавно настроиться на восприятие дальнейшей информации.

HTML-документ

Первое, что вам нужно, — это открыть HTML-документ. В этом документе вы будете записывать свой код на JavaScript. Затем запустите свой любимый редактор кода. Если у вас такового пока нет, рекомендую использовать Visual Studio Code. После запуска редактора перейдите к созданию нового файла. В Visual Studio Code вы увидите вкладку **Untitled**, как на рис. 1.3.

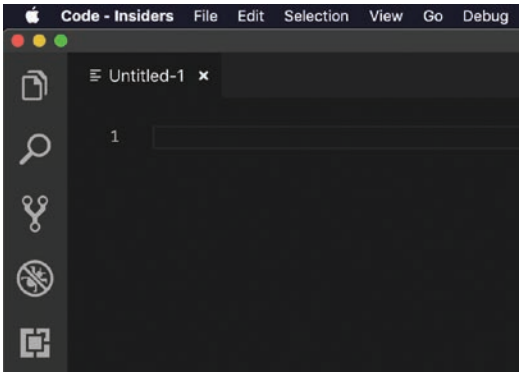


РИС. 1.3.
Вкладка Untitled-1 в Visual Studio Code

Сохраните созданный файл через меню **File | Save**. Вам нужно указать его имя и рабочий стол. После сохранения добавьте в него следующий код HTML:

```
<!DOCTYPE html>
<html>

<head>
  <meta charset="utf-8">
  <title>An Interesting Title Goes Here</title>

  <style>

  </style>
</head>

<body>
  <script>

  </script>

</body>

</html>
```

После добавления этого HTML сохраните документ для подтверждения изменений. Теперь можно посмотреть, как будет выглядеть ваша страница в браузере.

Проследуйте на рабочий стол в проводнике или поисковике и двойным щелчком откройте файл hello_world.htm. Вы увидите, как появится ваш браузер по умолчанию, который отобразит имя этого файла. На рис. 1.4 показано примерно то, что вы увидите у себя на экране.

Если все сработало, как надо, вы увидите пустую страницу, и это вполне нормально. Несмотря на то что страница имеет содержимое, на ней ничего не отображается. Скоро мы это исправим. Для этого потребуется вернуться в редактор и обратиться к тегу `<script>`, который находится в нижней части HTML:

```
<script>

</script>
```

Тег `script` выступает в роли контейнера, в который вы можете помещать любой JavaScript-код для запуска. Мы же хотим отобразить слова `hello, world!` в диалоговом окне, появляющемся при загрузке HTML-страницы. Для этого внутрь сегмента `script` добавим следующую строку:

```
<script>
  alert("hello, world!");
</script>
```

Сохраните файл HTML и запустите его в браузере. Обратите внимание на то, что увидите после загрузки страницы. Должно появиться диалоговое окно (рис. 1.5).

Если это была ваша первая попытка написания кода на JavaScript, примите мои поздравления! Теперь давайте разберем, как именно все это у вас получилось.

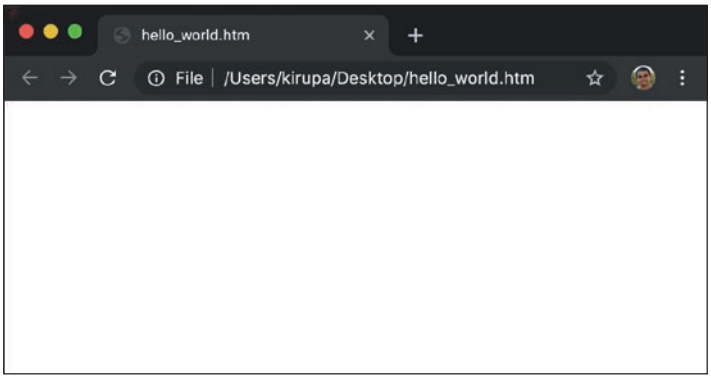


РИС. 1.4.
Проименованная вкладка в Visual Studio Code

РИС. 1.5.
Так должно выглядеть диалоговое окно hello, world

Обзор кода: инструкции и функции

Вы только что написали очень простую *инструкцию* JavaScript. Инструкция состоит из логического набора действий, которые должен выполнить браузер. В обычном приложении содержится великое множество инструкций. В нашем же случае есть только одна:

```
alert("hello, world!");
```

Понять, что перед вами инструкция, можно, обратив внимание на ее последний знак. Обычно это точка с запятой (`;`).

Внутри инструкции вы увидите всевозможный незатейливый жаргон на JavaScript. И наш код длиной в одну строку — не исключение. Мы также видим странный элемент `alert`, который задает отображение. Это пример привычного английского слова, имеющего схожее значение в мире языка JavaScript. Оно призвано привлечь ваше внимание путем отображения некоего текста.

Если быть более точным, слово `alert` — это то, что часто называют *функцией*. Вы будете использовать функции повсеместно. По своей сути они являются фрагментами переиспользуемого кода, который что-то делает. Это «что-то» может определяться вами, сторонней библиотекой или же самим фреймворком JavaScript. В нашем случае код, который дает функции `alert` магическую способность отображать диалоговое окно с переданным в нее сообщением, существует в глубинах браузера. Все, что вам на самом деле нужно знать, так это то, что для использования функции `alert` ее нужно просто вызвать и передать ей сообщение для отображения. Все остальное будет сделано за вас.

Вернемся к нашему примеру. Обратите внимание на то, как я определяю отображение текста `hello, world!`. Я заключаю эти слова в кавычки:

```
<script>
  alert("hello, world!");
</script>
```

При работе с текстом (наиболее часто используемый термин — *строка*) он всегда заключается в одинарные или двойные кавычки. И как бы странно это ни прозвучало, но у каждого языка программирования свои особенности. И эта — одна из многих, с которыми вы столкнетесь при дальнейшем знакомстве с JavaScript. Очень скоро мы рассмотрим строки более подробно, а пока просто наслаждайтесь их видом.

Сделаем еще шаг. Вместо `hello, world!` укажите свои имя и фамилию. Вот пример кода, в котором использовано мое имя:

```
<script>
  alert("Kirupa Chinnathambi!");
</script>
```

Запустите приложение, и тогда вы увидите свое имя в диалоговом окне (рис. 1.6).

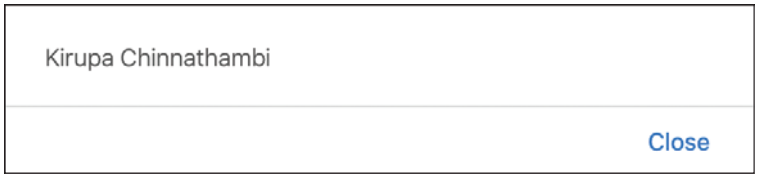


РИС. 1.6.
Теперь в диалоговом окне отображается ваше имя

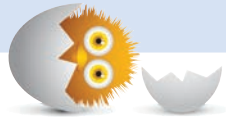
Проще простого, не так ли? Вы можете вписать в строку что угодно: имя питомца, название любимого сериала и т. д. — а JavaScript все это отобразит.

КОРОТКО О ГЛАВНОМ

В этой главе вы познакомились с написанием кода на JavaScript. В процессе этого знакомства я дал некоторые принципы и термины. Разумеется, я не жду, что вы все это сразу запомните. В следующих уроках мы возьмем наиболее интересные части пройденного материала и проработаем их более детально. В конце концов, я уверен, что с помощью JS вам хочется создавать штуки, выходящие далеко за рамки примитивного отображения текста с помощью диалогового окна.

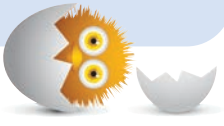
Забегая вперед, скажу, что в конце каждой главы есть набор ссылок, ведущих на ресурсы, созданные мной или другими разработчиками. Вы сможете почерпнуть больше информации, взглянуть на пройденный материал под другим углом, а также опробовать свои силы на практике, решая более сложные примеры. Рассматривайте материал из этой книги как трамплин, позволяющий допрыгнуть до более крутых штук.

Если у вас есть какие-либо вопросы относительно этой главы, оставляйте свои сообщения на форуме https://forum.kirupa.com, где вам оперативно ответит кто-то из крутых разработчиков или я.



В ЭТОЙ ГЛАВЕ:

- узнаем, как использовать значения для хранения данных;
- создадим свой код, задавая значения;
- вкратце ознакомимся с соглашениями о присвоении имен переменных.



2

ЗНАЧЕНИЯ И ПЕРЕМЕННЫЕ

Принято считать, что каждый фрагмент данных на JavaScript, предоставляемый или используемый нами, содержит значение. Из уже рассмотренного примера мы узнали, что вместо слов `hello, world!` могут быть любые слова, с помощью которых мы задаем функцию `alert`:

```
alert("hello, world!");
```

В JavaScript эти слова несут в себе определенную информацию и считаются *значениями*. Мы могли об этом особенно не задумываться, набирая их на клавиатуре, но во вселенной JavaScript каждый элемент данных, с которым мы имеем дело, считается значением.

Итак, почему это так важно понимать? А все потому, что нам предстоит много работать со значениями. И важно, чтобы эта работа не свела вас с ума. Чтобы облегчить себе жизнь, вам пригодятся умения:

- легко идентифицировать эти значения;
- повторно их использовать в ходе разработки приложения, не прибегая к лишнему дублированию самого значения.

Две вещи, на которые предстоит потратить все наше оставшееся время, — это переменные. Давайте рассмотрим их поподробнее.

Использование переменных

Переменная — это идентификатор значения. Чтобы не набирать `hello, world!` каждый раз, когда вы хотите использовать этот фрагмент для написания кода приложения, можно присвоить эту фразу переменной и использовать ее тогда, когда нужно. Еще чуть-чуть, и вам станет гораздо понятнее — я обещаю!

Есть несколько способов, как использовать переменные. В большинстве случаев лучше всего полагаться на ключевое слово `let`, после которого вы можете задать имя своей переменной:

```
let myText
```

В этой строке кода мы *объявляем* переменную `myText`. На данный момент наша переменная просто объявлена и не содержит никакого значения, то есть является пустой оболочкой.

Давайте исправим это через *инициализацию* переменной значением. К примеру, `hello, world!`.

```
let myText = "hello, world!";
```

С этого момента при выполнении кода значение `hello, world!` будет ассоциироваться с нашей переменной `myText`. Теперь соберем все части в единое выражение. Если у вас все еще открыт файл hello_world.htm, замените содержимое тега `script` следующим (или создайте новый файл, добавив в него следующий код):

```
<!DOCTYPE html>
<html>

<head>
  <meta charset="utf-8">
  <title>An Interesting Title Goes Here</title>

  <style>

  </style>
</head>

<body>
  <script>
    let myText = "hello, world!";
    alert(myText);
  </script>

</body>

</html>
```

Обратите внимание, что мы больше не передаем текст `hello, world!` в функцию `alert` напрямую. Вместо этого мы передаем в нее имя переменной `myText`. Конечный результат такой же, то есть при выполнении этой строчки отобразится функция `alert` с надписью `hello, world!`. Благодаря этому изменению нам достаточно определить `hello, world!` в одном месте кода. И если мы в дальнейшем захотим изменить `hello, world!`, к примеру, на `The dog ate my homework!` (Собака съела мою домашку!), то все, что нам понадобится, — это просто изменить фразу, определенную в переменной `myText`:

```
let myText = "The dog ate my homework!";
alert(myText);
```

Теперь во всех секциях кода при обращении к переменной `myText` мы будем видеть новый текст. Невозможно придумать ничего более удобного и значительно экономящего время, особенно при работе со сложными приложениями, что позволяет вносить изменение в одном месте, и при этом это действие отразится в остальных частях кода. Вы увидите другие, менее тривиальные примеры со значимыми переменными далее.

Еще кое-что о переменных

Материал из предыдущей главы позволяет нам сильно продвинуться вперед. По меньшей мере, в той части, которая касается знакомства с JavaScript. Сейчас мы не станем слишком углубляться в переменные, так как займемся ими в следующих главах, когда рассмотрим код с важными переменными посложнее. И прежде чем закругляться, затронем еще несколько моментов.

Именование переменных

Мы вольны именовать переменные так, как нам подходит. Игнорируя то, какие имена нам следует выбрать с точки зрения философских, культурных или стилистических предпочтений, с технической точки зрения язык JavaScript очень гибок и позволяет включать в них символы.

Однако эта гибкость не безгранична, поэтому при именовании следует помнить о следующем:

- Имя переменной может содержать один символ или столько, сколько вы хотите, — только представьте, тысячи тысяч символов!
- Переменные могут начинаться с буквы, нижнего подчеркивания или символа `$`, но не могут начинаться с числа.
- Следом за первым символом переменные могут состоять из любого сочетания букв, подчеркиваний, чисел и `$`. Мы также можем смешивать и комбинировать нижний и верхний регистры, пока не надоест.
- Пробелы не допускаются.

Ниже приведены некоторые примеры имен переменных:

```
let myText;
let $;
let r8;
let _counter;
let $field;
let thisIsALongVariableName_butItCouldBeLonger;
let $abc;
let OldSchoolNamingScheme;
```

Чтобы определить, является ли имя переменной допустимым, воспользуйтесь прекрасным сервисом по проверке имен переменных JavaScript https://mothereff.in/js-variables.

Помимо допустимых имен есть еще другие важные моменты, такие как соглашения о присвоении имен и то, сколько людей обычно именуют переменные и другие компоненты, которые вы идентифицируете с именем. Мы затронем эти темы в следующих главах.

Что еще нужно знать об объявлении и инициализации переменных

Одна из особенностей JavaScript, о которой вы скоро узнаете, состоит в том, что это всепрощающий язык, который может дать вам множество поблажек.

Объявление переменной не обязательно

Например, нам не обязательно использовать ключевое слово `let`, чтобы объявить переменную. Можно просто написать так:

```
myText = "hello, world!";
alert(myText);
```

Обратите внимание, что переменная `myText` использована, не будучи формально объявленной с помощью ключевого слова `let`. И хотя так делать и не рекомендуется, это считается вполне допустимым. В конечном счете мы получаем переменную с именем `myText`. Единственный нюанс состоит в том, что при объявлении переменной подобным образом мы делаем это глобально. Не беспокойтесь, если последнее предложение вам не понятно. Мы поговорим о значении слова «глобально» позже, когда рассмотрим область видимость переменных.

Объявлять и инициализировать переменные в отдельных строках — это круто!

Стоит упомянуть, что объявление и инициализация переменной не обязательно должны быть частью одной инструкции. Можно разделить эти действия на разные строки:

```
let myText;
myText = "hello, world!";
alert(myText);
```

На практике мы будем разделять их постоянно.

Изменение значений переменных и ключевое слово const

И наконец, отмечу, что мы можем менять значение переменной, объявленной с `let`, на что угодно и когда угодно:

```
let myText;
myText = "hello, world!";
myText = 99;
myText = 4 * 10;
myText = true;
myText = undefined;
alert(myText);
```

Если вы работали с более требовательными языками, которые не позволяют переменным хранить разные типы данных, то знайте, что эту гибкость JavaScript одни восхваляют, а другие ненавидят. Тем не менее в JavaScript есть способ запретить изменять значения переменной после инициализации. Это можно сделать с помощью ключевого слова `const`, с которым мы можем объявлять и инициализировать переменные:

```
const siteURL = "https://www.google.com";
alert(siteURL);
```

Используя `const`, мы не можем изменить значение `siteURL` на что-то иное, чем https://www.google.com. При такой попытке JavaScript начнет ругаться. Несмотря на то что в использовании этого ключевого слова есть свои подводные камни, в целом оно может оказаться очень полезным для предотвращения случайного изменения переменной. Когда придет время, мы рассмотрим подобные подводные камни более подробно.

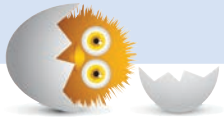
ПОЧИТАЙТЕ «ОБЛАСТЬ ВИДИМОСТИ ПЕРЕМЕННЫХ»

Теперь, когда вы знаете, как объявлять и инициализировать переменные, очень важно разобраться с их видимостью. Нужно понимать, при каких условиях можно использовать объявленную переменную. Иначе это зовется *областью видимости переменной*. Интересно? Тогда прочитайте главу 8 «Область видимости переменных».

КОРОТКО О ГЛАВНОМ

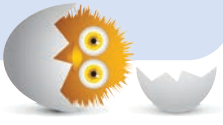
Значения — хранилища для данных, а переменные — легкий способ обратиться к этим данным. В значениях сокрыто много интересных деталей, но на данный момент они вам ни к чему. Просто знайте, что JavaScript позволяет представлять различные значения вроде текста и чисел без шума и пыли.

Вы объявляете переменные, чтобы значения стали более запоминающимися и годными для многократного использования. Это делается с помощью ключевого слова `let` и *имени переменной*. Если вы хотите инициализировать переменную со значением по умолчанию, поставьте после имени переменной знак равенства (=) и укажите значение, с которым хотите инициализировать эту переменную.



В ЭТОЙ ГЛАВЕ:

- узнаем, как функции помогают лучше организовать и сгруппировать код;
- поймем, как с помощью функций сделать код многократно используемым;
- узнаем о важности аргументов функции и их использовании.



3

ФУНКЦИИ

До сих пор весь написанный нами код не имел никакой структуры и был простым до безобразия:

```
alert("hello, world!");
```

С таким кодом все в порядке, особенно учитывая, что он состоит из единственной инструкции. Но в реальном мире нам не отделаться так легко и код, написанный на JavaScript под настоящие задачи в жизни, редко будет таким простым.

Предположим, что нужно отобразить расстояние, пройденное объектом (рис. 3.1).

РИС. 3.1.
Пройденное расстояние

Как вы помните еще со школы, чтобы вычислить расстояние, нужно скорость объекта умножить на время его движения (рис. 3.2).

расстояние = скорость × время

РИС. 3.2.
Вычисление расстояния

Версия этого выражения на JavaScript будет выглядеть примерно так:

```
let speed = 10;
let time = 5;
alert(speed * time);
```

У нас есть две переменные, обозначенные как скорость (`speed`) и время (`time`), каждая из которых содержит число. Функция `alert` отображает результат умножения значений, содержащихся в переменных `speed` и `time`. Это весьма буквальная интерпретация уравнения для вычисления расстояния, которое мы только что рассмотрели.

Представим, к примеру, что нам нужно рассчитать расстояние при наличии большего числа значений. Опираясь только на те знания, которые мы освоили к этому моменту, мы запишем код вот так:

```
let speed = 10;
let time = 5;
alert(speed * time);

let speed1 = 85;
let time1 = 1.5;
alert(speed1 * time1);
```

```
let speed2 = 12;
let time2 = 9;
alert(speed2 * time2);

let speed3 = 42;
let time3 = 21;
alert(speed3 * time3);
```

Не знаю, как вам, но по мне — это полная печалька. Наш код слишком многословный и повторяющийся. Мы уже видели ранее при изучении темы переменных, что повторение *не позволяет создавать легкий в сопровождении код и отнимает время*.

Подобная проблема полностью решаема с помощью того, что мы еще не раз встретим в этой книге, а именно *функций*:

```
function showDistance(speed, time) {
  alert(speed * time);
}

showDistance(10, 5);
showDistance(85, 1.5);
showDistance(12, 9);
showDistance(42, 21);
```

Не пугайтесь того, что этот код только что провернул на ваших глазах. Все просто: укороченный фрагмент кода выполнил ту же работу, что и множество строчек из рассмотренного ранее примера, но только *без побочных действий.* Мы узнаем о функциях все, в том числе как им удается так превосходно справляться со своей работой, и начнем мы прямо сейчас!

Поехали!

Что такое функция?

На самом базовом уровне функция — не более чем оболочка для некоего кода.

По сути функция:

- группирует инструкции;
- позволяет использовать код многократно.

Вы редко напишете или воспользуетесь кодом, в котором не будет функций, поэтому важно не только ознакомиться с ними, но и разобраться в тонкостях их правильной работы.

Простая функция

Лучший способ понять функции — это погрузиться в них и начать использовать. Поэтому создадим для начала самую простую функцию. Это не особо увлекательный процесс. Требуется лишь понимание некоторых особенностей синтаксиса вроде использования причудливых фигурных скобок и их собратьев.

Ниже приведен пример того, как выглядит простая функция:

```
function sayHello() {
  alert("hello!");
}
```

Однако простого определения функции еще недостаточно, ее нужно также вызвать путем добавления следующих строк в конце:

```
function sayHello() {
alert("hello!");
}
sayHello();
```

Чтобы попрактиковаться, создайте новый HTML-документ (назовите его functions_sayhello.htm):

```
<!DOCTYPE html>
<html>

<head>
  <meta charset="utf-8">
  <title>Say Hello!</title>

  <style>

  </style>
</head>
```

```
<body>
  <script>
    function sayHello() {
       alert("hello!");
    }

    sayHello();
  </script>
</body>

</html>
```

Если вы наберете этот текст полностью и просмотрите страницу в браузере, то увидите, как отобразится `hello!`. Это нужно сделать, чтобы убедиться, что наш код работает. Далее разберем, почему этот код сработал, а для этого разобьем функцию `sayHello` на несколько самостоятельных фрагментов и рассмотрим каждый из них подробнее.

Во-первых, мы видим ключевое слово `function` (рис. 3.3).

РИС. 3.3.
Ключевое слово function

Это ключевое слово сообщает движку JavaScript, который живет в глубинах вашего браузера, что весь этот блок кода нужно рассматривать как связанный с функциями.

После ключевого слова `function` мы указываем актуальное имя функции и ставим после него открывающую и закрывающую скобки `()`, как показано на рис. 3.4.

```
function sayHello() {
  alert("hello!");
}
```

РИС. 3.4.
Имя функции и скобки

Завершая процесс объявления функции, нужно поставить открывающую и закрывающую фигурные скобки, внутри которых указываются нужные инструкции (рис. 3.5).

```
function sayHello() {
  alert("hello!");
}
```

РИС. 3.5.
Открывающая и закрывающая фигурные скобки

В заключение рассмотрим содержимое функции, а именно те инструкции, которые задают функциональность (рис. 3.6).

```
function sayHello() {
  alert("hello!");
}
```

РИС. 3.6.
Содержимое функции

В нашем примере содержимым является функция `alert`, отображающая диалоговое окно со словом `hello!`.

Последнее, что осталось рассмотреть, — это вызов функции (рис. 3.7).

```
function sayHello() {
  alert("hello!");
}
sayHello();
```

РИС. 3.7.
Вызов функции

Как правило, вызов функции — это имя той функции, которую мы хотим вызвать (как всегда, со скобками в конце). Без вызова функции она ничего не будет делать. Именно с вызовом наша функция просыпается и начинает что-то делать.

Вот мы и рассмотрели простейшую функцию. Далее, с опорой на только что пройденный материал, мы ознакомимся с более жизненными примерами функций.

Создание функции, принимающей аргументы

Предыдущий пример с `sayHello` слишком прост:

```
function sayHello() {
  alert("hello!");
}
sayHello();
```

Мы вызываем функцию, и она производит определенные действия. Такое упрощение вполне нормально, потому что все функции работают одинаково. Отличия состоят лишь в особенностях того, как производится вызов функции, откуда в нее поступают данные и т. д. Первая особенность, которую мы рассмотрим, относится к функциям, принимающим *аргументы*.

Начнем с простого и знакомого примера:

```
alert("my argument");
```

Перед нами функция `alert`. Вероятно, мы уже видели ее пару-тройку (или несколько десятков) раз. Суть в том, что в эту функцию передается так называемый *аргумент*, который описывает то, что требуется отобразить при вызове. На рис. 3.8 показано, что отобразится на экране, если вызвать функцию `alert` с аргументом `my argument`.

Аргумент — это то, что находится между открывающими и закрывающими скобками. Функция `alert` — лишь одна из множества возможных функций, принимающих аргументы. Многие из функций, которые вы создадите в будущем, будут также принимать аргументы.

В этой главе мы рассматривали еще одну функцию, принимающую аргументы, а именно `showDistance`:

```
function showDistance(speed, time) {
  alert(speed * time);
}
```

РИС. 3.8.
Отображение аргумента

Понять, что функция принимает аргументы, можно, просто взглянув на ее описание (объявление):

```
function showDistance(speed, time) {
  alert(speed * time);
}
```

То, что ранее было пустыми скобками после имени функции, теперь содержит информацию о количестве аргументов, необходимых функции, а также подсказывает, какие значения заданы этим аргументам.

В случае с `showDistance` можно сделать вывод о том, что эта функция принимает два аргумента. Первый из них соответствует `speed` (скорости), а второй — `time` (времени).

Мы задаем значения аргументов в рамках вызова функции:

```
function showDistance (speed, time) {
  alert(speed * time);
}
showDistance(10, 5);
```

В нашем случае мы вызываем функцию `showDistance` и задаем значения, которые хотим в нее передать, внутри скобок (рис. 3.9).

Поскольку мы передаем больше одного аргумента, то перечисляем значение каждого через запятую. И пока я не забыл, отмечаю еще один важный момент: важен порядок, в котором вы определяете аргументы.

```
showDistance(10, 5);
```

РИС. 3.9.
Значения, которые мы хотим передать в функцию

Давайте рассмотрим этот процесс подробнее и начнем с диаграммы на рис. 3.10.

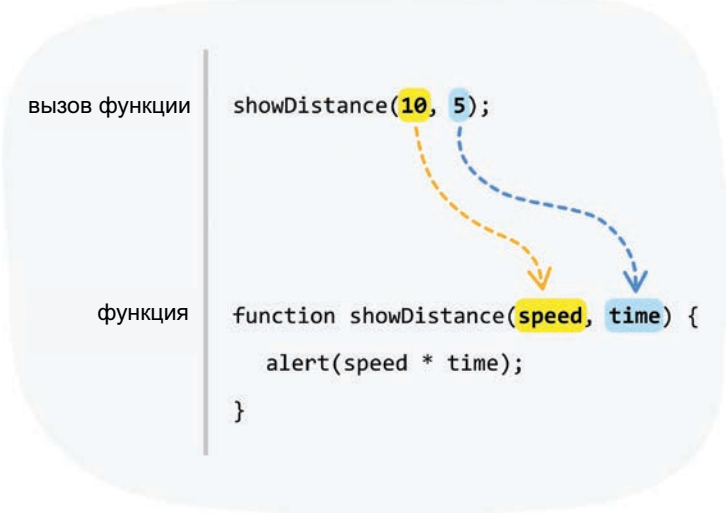


РИС. 3.10.
Диаграмма вызова функции

Для вызова в функцию `showDistance` передается `10` как аргумент для `speed` и `5` — для `time`. Изображенный на диаграмме перенос полностью основан на последовательности.

Как только передаваемые значения достигают функции, определенные для аргументов имена начинают обрабатываться как имена переменных (рис. 3.11).

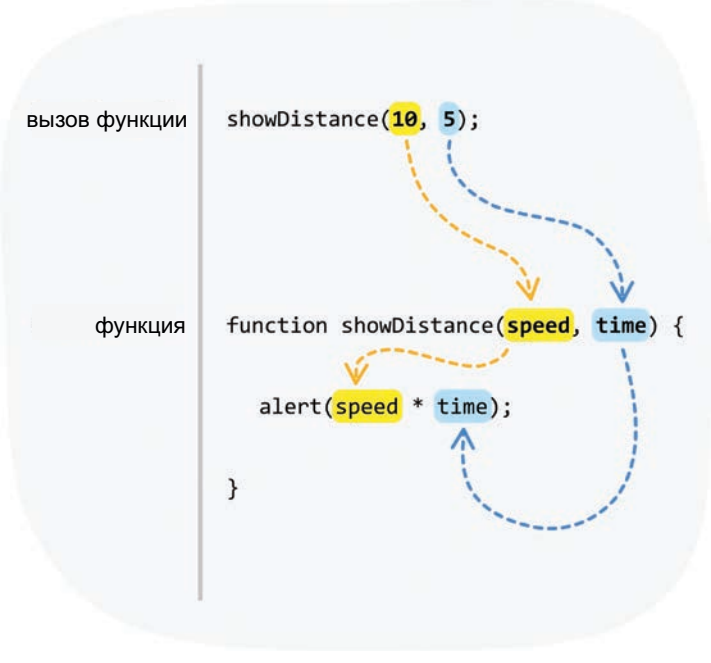


РИС. 3.11.
Имена аргументов работают как переменные

Мы можем использовать эти имена переменных, чтобы легко и без забот ссылаться на значения, содержащиеся в аргументах внутри функции.

НЕСООТВЕТСТВИЕ ЧИСЛА АРГУМЕНТОВ

Если вы не задаете аргументы при вызове или задаете меньше или больше аргументов, чем требуется функции, все по-прежнему может работать. Чтобы избежать подобных ситуаций, вы можете применять защитное программирование, и в дальнейшем мы рассмотрим этот вопрос подробнее.

В целом, чтобы создать код с более четкой структурой, необходимо передавать в функцию соответствующее число аргументов.

Создание функции, возвращающей результат

Последний вид функции, который мы рассмотрим, — это та функция, которая возвращает результат в ответ на вызов. Вот, что нам нужно сделать. У нас есть функция `showDistance`, которая, как нам прекрасно известно, выглядит так:

```
function showDistance(speed, time) {
  alert(speed * time);
}
```

Нам нужно, чтобы наша функция не вычисляла расстояние и отображала его в виде уведомления, а сохраняла полученное значение для дальнейшего использования. Мы хотим получить примерно следующее:

```
let myDistance = showDistance(10, 5);
```

Переменная `myDistance` будет содержать результат вычисления, которое выполнит функция `showDistance`.

Ключевое слово return

Возврат данных из функции производится посредством ключевого слова `return`. Давайте создадим новую функцию с именем `getDistance`, которая будет выглядеть как `showDistance`, но отличаться в том, что происходит при выполнении функции до завершения:

```
function getDistance(speed, time) {
  let distance = speed * time;
  return distance;
}
```

Обратите внимание, что мы вычисляем расстояние, умножая `speed` на `time`. Вместо отображения уведомления (`alert`) мы возвращаем значение расстояния (которое содержится в переменной `distance`).

Мы можем выполнить вызов функции `getDistance` в рамках инициализации переменной:

```
let myDistance = getDistance(10, 5);
```

Когда функция `getDistance` будет вызвана, она выполнит вычисление и вернет численное значение, которое затем будет присвоено переменной `myDistance`. И всего делов-то!

Ранний выход из функции

Как только функция доходит до ключевого слова `return`, она прекращает выполнять обработку, которую делала до этого момента, возвращает значение, заданное в вызывающей функции (caller), и производит выход:

```
function getDistance(speed, time) {
  let distance = speed * time;
  return distance;

  if (speed < 0) {
    distance *= -1;
  }
}
```

Любой код, прописанный после инструкции `return`, не будет обработан. Эта часть будет проигнорирована, как если бы ее и не было вовсе.

На практике инструкция `return` используется для завершения функции после того, как она выполнит нужные нам действия. Эта функция могла бы вернуть значение вызывающей функции, как вы видели в предыдущем примере, или просто произвести выход:

```
function doSomething() {
  let foo = "Nothing interesting";
  return;
}
```

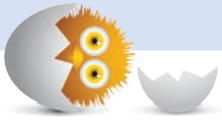
Использовать ключевое слово `return` для возвращения результата не обязательно. Оно может использоваться отдельно, как мы увидели в примере выше, просто для выхода из функции. Если значение `return` не задано функцией, возвращается значение по умолчанию, `undefined`.

КОРОТКО О ГЛАВНОМ

Функции относятся к тому небольшому числу компонентов, которые вы будете использовать практически в каждом приложении JavaScript. Они предоставляют востребованную возможность — создавать переиспользуемый код. Не важно, используете ли вы собственные функции или те, что встроены в JavaScript, — вы просто не сможете обойтись без них.

Все, что вы прочитали на данный момент, является примерами распространенного использования функций. Существуют некоторые продвинутые особенности функций, которые я не затрагивал в этой главе. Эти особенности мы рассмотрим в далеком будущем… очень далеком. Пока что изученного вами материала хватит для углубленного понимания, как использовать функций в реальной жизни.

Если у вас есть вопросы по пройденной теме — добро пожаловать на форум https://forum.kirupa.com, где я или другие смышленые веб-разработчики поможем вам.



В ЭТОЙ ГЛАВЕ:

- узнаете, как использовать инструкции `if…else` для принятия решений;
- научитесь пользоваться инструкцией `switch`.

4

УСЛОВНЫЕ ОБОЗНАЧЕНИЯ: IF…ELSE И SWITCH

Как только вы просыпаетесь, начинается осознанный или неосознанный процесс принятия решений. Выключить будильник. Включить свет. Выглянуть из окна, чтобы проверить погоду. Почистить зубы. Надеть мантию и шляпу волшебника. Посмотреть в календарь. В общем… вы меня поняли. К тому моменту, когда вы шагнете за порог, можно будет насчитать уже сотни принятых вами осознанных или неосознанных решений. Каждое из этих решений определенным образом повлияло на то, что вы в итоге станете делать.

Например, если бы на улице было холодно, вам пришлось бы решать, что надеть: худи или куртку. На рис. 4.1 показано, как формируется это решение.

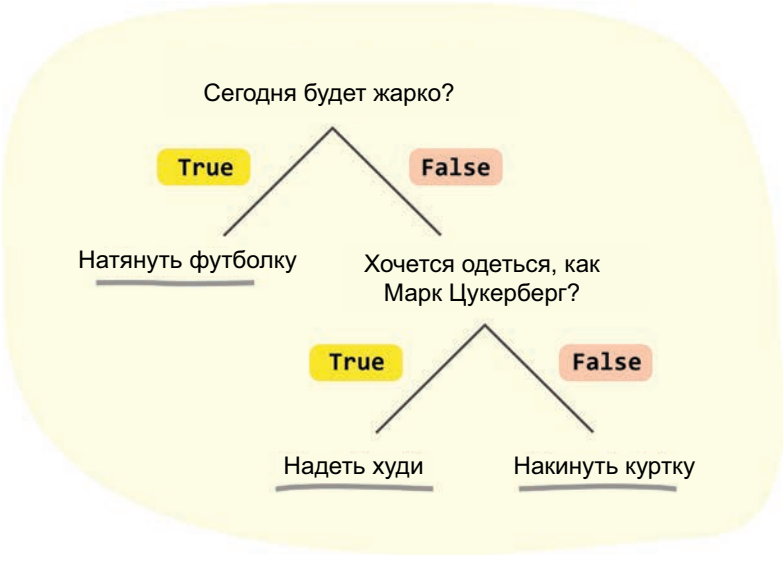


РИС. 4.1.
Моделирование решений

На каждой стадии принятия решения вы задаете себе вопрос, на который можно дать ответ `true` (верно) или `false` (неверно). Ответ на этот вопрос определяет ваш следующий шаг и то, что в итоге вы наденете: футболку, худи или куртку. В более широком смысле любое наше решение можно смоделировать в виде череды инструкций `true` и `false`. От этого может стать слегка не по себе (еще бы!), но в общем и целом именно так мы, наше окружение и большинство живых существ совершаем свой выбор.

Такое обобщение особенно применимо ко всему, что делает наш компьютер. На примере всех кодов, которые мы успели написать, это не сразу бросается в глаза, но мы скоро исправим эту ситуацию. В этом уроке мы рассмотрим *условные выражения*. Они являются цифровым эквивалентом решений в ситуациях, где код производит некоторое действие в зависимости от того, оказывается что-либо `true` или `false`.

Поехали!

Инструкция if…else

Самая распространенная условная инструкция, используемая в коде, — это *инструкция if…else*, или просто *инструкция if*. Принцип ее работы показан на рис. 4.2.

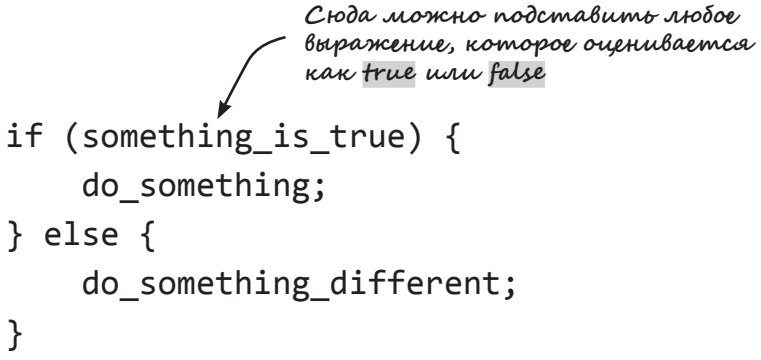


РИС. 4.2.
Как работает инструкция if

Чтобы в этом разобраться, рассмотрим инструкцию `if…else` в действии. Создайте новый HTML-документ и добавьте в него следующие разметку и код:

```
<!DOCTYPE html>
<html>

<head>
  <meta charset="utf-8">
  <title>If / Else Statements</title>
</head>

<body>
  <script>
    let safeToProceed = true;
    if (safeToProceed) {
      alert("You shall pass!");
    } else {
      alert("You shall not pass!");
    }
  </script>
</body>

</html>
```

Сохраните этот документ под именем `if_else.htm` и выполните его предпросмотр в браузере. Если все сработает как надо, вы увидите уведомление с текстом `You shall pass!` (рис. 4.3).

РИС. 4.3.
Вы увидите это уведомление

За полученный результат отвечают следующие строки кода:

```
let safeToProceed = true;

if (safeToProceed) {
  alert("You shall pass!");
} else {
  alert("You shall not pass!");
}
```

Наше *выражение* (то, что следует за ключевым словом `if` и в итоге оценивается как `true` или `false`) — это переменная `safeToProceed`. Эта переменная инициализирована как `true`, следовательно, был задействован вариант `true` инструкции `if`.

Теперь замените значение переменной `safeToProceed` с `true` на `false`:

```
let safeToProceed = true;

if (safeToProceed) {
  alert("You shall pass!");
} else {

  alert("You shall not pass!");
}
```

На этот раз при запуске кода вы увидите уведомление с текстом `You shall not pass!`, так как теперь выражение вычисляется как `false` (рис. 4.4).

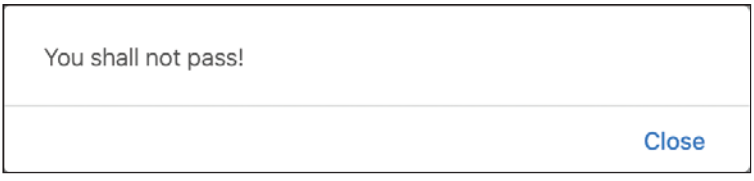


РИС. 4.4.
Уведомление, получаемое, когда выражение вычисляется как false

Пока что все это может казаться скучным, но в основном потому, что мы еще не добавляли элемент сложности, чтобы рассмотреть более жизненные сценарии. Это ждет нас на следующем этапе, когда мы глубже погрузимся в эту тему.

Встречайте: условные операторы!

В большинстве случаев наше выражение не будет простой переменной со значением `true` или `false`, подобно предыдущему примеру. В выражениях будут задействованы так называемые *условные операторы*, помогающие сравнивать два или больше двух выражений для утверждения результата `true` или `false`.

В общих чертах подобные выражения показаны на рис. 4.5.

```
if (expression operator expression) {
    do_something;
} else {
    do_something_different;
}
```

РИС. 4.5.
Общий формат выражений с условными операторами

Оператор (то есть *условный оператор*) определяет связь между выражениями. Конечная цель — вернуть результат `true` или `false`, чтобы наша инструкция `if` понимала, какой блок кода выполнять. Ключом к выполнению всех этих действий являются сами условные операторы, которые приведены в табл. 4.1.

ТАБЛ. 4.1. Операторы

Оператор	Если true
= =	Если первое выражение дает результат, равный второму выражению
>=	Если первое выражение дает результат, который больше или равен второму выражению
>	Если первое выражение дает результат больше, чем второе выражение
<=	Если первое выражение дает результат, меньший или равный второму выражению
<	Если первое выражение дает результат меньше, чем второе выражение
!=	Если первое выражение дает результат, не равный второму выражению
&&	Если и первое, и второе выражения дают результат `true`
||	Если либо первое, либо второе выражение дает результат `true`

Теперь перейдем от обобщенного понимания условных операторов к более конкретному, рассмотрев еще один пример, в котором подсвечен интересующий нас `if`-сегмент кода:

```
<!DOCTYPE html>
<html>

 <head>
  <meta charset="utf-8">
  <title>Are you speeding?</title>
</head>

 <body>
  <script>
    let speedLimit = 55;

    function amISpeeding(speed) {
      if (speed >= speedLimit) {
        alert("Yes. You are speeding.");
      } else {
        alert("No. You are not speeding. What's wrong with you?");
      }
    }

    amISpeeding(53);
    amISpeeding(72);
  </script>
</body>

</html>
```

Разберемся, что же именно здесь происходит. У нас есть переменная `speedLimit`, инициализированная как `55`. Затем есть функция `amISpeeding`, получающая аргумент `speed`. Внутри нее инструкция `if`, чье выражение проверяет, является ли полученное значение `speed` большим или равным (привет, условный оператор `>=`) значению, содержащемуся в переменной `speedLimit`:

```
function amISpeeding(speed) {
  if (speed >= speedLimit) {
    alert("Yes. You are speeding.");
  } else {
    alert("No. You are not speeding. What's wrong with you?");
  }
}
```

Последнее, что делает код, — это вызывает функцию `amISpeeding`, передавая ей два значения `speed`:

```
amISpeeding(53);
amISpeeding(72);
```

Когда мы называем эту функцию со скоростью `53`, выражение `speed >= speedLimit` вычисляется как `false`. Так происходит, потому что `53` не больше и не равно значению, сохраненному в `speedLimit`, а именно `55`. В итоге будет выводиться уведомление о том, что вы не превышаете скорость (`No. You are not speeding...`).

Противоположное происходит, когда мы вызываем `amISpeeding` со скоростью `72`. В этом случае мы превышаем скорость и выражение вычисляется как `true` с последующим появлением соответствующего уведомления.

Создание более сложных выражений

Про выражения следует знать, что они могут быть такими сложными или простыми, какими вы их сами сделаете. Они могут состоять из переменных, вызовов функций или одних значений. Они даже могут быть сделаны из сочетаний переменных, вызовов функций или голых значений, разделенных с помощью любых из описанных выше операторов. Важно сделать так, чтобы в итоге выражение вычислялось как `true` или `false`.

Привожу чуть более сложный пример:

```
let xPos = 300;
let yPos = 150;

function sendWarning(x, y) {
  if ((x < xPos) && (y < yPos)) {
    alert("Adjust the position");
  } else {
    alert("Things are fine!");
  }
}

sendWarning(500, 160);
sendWarning(100, 100);
sendWarning(201, 149);
```

Обратите внимание на то, как выглядит условие внутри инструкции `if`, принадлежащей функции `sendWarning`:

```
function sendWarning(x, y) {
        if ((x < xPos) && (y < yPos)) {
                alert("Adjust the position");
        } else {
                alert("Things are fine!");
        }
}
```

В данном случае было выполнено три сравнения. Во-первых, меньше ли `x`, чем `xPos`. Во-вторых, меньше ли `y`, чем `yPos`. И в-третьих — проверка, не расценивается ли *первая и вторая инструкции* как `true`, чтобы оператор `&&` мог также вернуть *это значение*. Можно соединять в цепочки множество условных инструкций в зависимости от того, что мы делаем. Помимо запоминания действий самих операторов может быть непросто проследить, чтобы все условия и подусловия были правильно изолированы скобками.

Весь рассматриваемый материал из текущего и предыдущего разделов подпадает под общее определение *бинарной логики*. Если вы ничего не знаете по этой теме, я рекомендую прочитать прекрасную *статью о режимах совместимости*.

Варианты инструкций if...else

Мы почти закончили с инструкцией `if`. Осталось лишь разобраться с ее «родственниками».

Одиночная инструкция if

Первый вариант — это одиночная инструкция `if` без родственника `else`:

```
if (weight > 5000) {
  alert("No free shipping for you!");
}
```

В этом случае если выражение вычисляется как `true`, то все отлично. Если как `false`, тогда код просто пропускает уведомление и переходит к выполнению следующих действий. При работе с инструкциями `if` блок `else` является опциональным. Чтобы составить контраст одиночной инструкции `if`, на помощь спешит ее родня.

Устрашающая инструкция if...else-if...else

Не все можно четко уложить в одиночную инструкцию `if` или `if…else`. В таких ситуациях можно использовать ключевое слово `else if`, чтобы создавать цепочки инструкций `if`. Не будем вдаваться в подробности, а просто посмотрим следующий пример:

```
if (position < 100) {
  alert("Do something!");
} else if ((position >= 200) && (position < 300)) {
  alert("Do something else!");
} else {
  alert("Do something even more different!");
}
```

Если первая инструкция `if` будет вычислена как `true`, тогда обработка кода пойдет по ветке первого уведомления. Если первая инструкция окажется `false`, тогда обработка переходит к вычислению инструкции `else if`, проверяя, `true` она или `false`. Это будет продолжаться, пока код не будет обработан до конца. Другими словами, обработка нашего кода — это движение вниз через все инструкции `if` и `else if`, пока одно из выражений не будет вычислено как `true`:

```
if (condition) {
  ...
} else if (condition) {
  ...
} else if (condition) {
  ...
} else if (condition) {
  ...
} else if (condition) {
  ...
```

```
} else if (condition) {
  ...
} else {
  ...
}
```

Если ни одна из инструкций не имеет выражений, вычисляемых как `true`, то выполняется обработка кода внутри блока `else` (если таковой существует). Если блока `else` нет, тогда выполнение просто перейдет к следующей части кода, находящейся за пределами инструкций `if`. С помощью более сложных выражений и инструкций `if…else if` вы можете выразить практически любое решение, которое потребуется вычислить в коде.

Фух

Теперь вы знаете все, что нужно знать об инструкции `if`. Пришло время познакомиться с совершенно иным видом условных инструкций...

Инструкция switch

В мире программирования, наполненном прекрасными инструкциями `if`, `else` *и* `else if`, потребность в ином виде взаимодействий с условными инструкциями может отсутствовать. Но суровые люди, писавшие код на машинах размером с комнату и не боявшиеся волков в заснеженных горах, не согласились бы с этим. Поэтому теперь у нас есть так называемые инструкции `switch`. И прямо сейчас мы узнаем, зачем они нужны.

Использование инструкции switch

Не будем тянуть кота за хвост и сразу посмотрим на пример. Основа структуры инструкции `switch` выглядит так:

```
switch (expression) {
  case value1:
    statement;
    break;
  case value2:
    statement;
    break;
```

```
  case value3:
    statement;
    break;
  default:
    statement;
    break;
}
```

Всегда нужно помнить, что инструкция `switch` — это условная инструкция, проверяющая, является *что-либо* `true` или `false`, и не более того. Это что-либо, в свою очередь, является вариацией того, является ли *результат вычисления выражения равным значению* `case`. Чтобы прояснить этот момент, рассмотрим более подходящий пример:

```
let color = "green";

switch (color) {
  case "yellow":
    alert("yellow color");
    break;
  case "red":
    alert("red color");
    break;
  case "blue":
    alert("blue color");
    break;
  case "green":
    alert("green color");
    break;
  case "black":
    alert("black color");
    break;
  default:
    alert("no known color specified");
    break;
}
```

Здесь у нас есть переменная `color`, которой задано значение `green`:

```
let color = "green";
```

Мы также определяем переменную `color` в качестве выражения в инструкции `switch`:

```
switch (color) {
  case "yellow":
    alert("yellow color");
    break;
  case "red":
```

```
    alert("red color");
    break;
  case "blue":
    alert("blue color");
    break;
  case "green":
    alert("green color");
    break;
  case "black":
    alert("black color");
    break;
  default:
    alert("no known color specified");
    break;
}
```

Инструкция `switch` содержит коллекцию блоков `case` (случаев). При выполнении кода лишь один из этих блоков станет избранным. Выбор конкретного блока происходит путем сопоставления значения блока `case` с результатом вычисления выражения. В нашем случае, так как выражение вычисляется со значением `green`, будет выполнен код внутри блока `case` с тем же значением `green`:

```
switch (color) {
  case "yellow":
    alert("yellow color");
    break;
  case "red":
    alert("red color");
    break;
  case "blue":
    alert("blue color");
    break;
  case "green":
    alert("green color");
    break;
  case "black":
    alert("black color");
    break;
  default:
    alert("no known color specified");
    break;
}
```

Обратите внимание, что выполняется код, содержащийся *только* внутри блока `case green`. Так происходит благодаря ключевому слову `break` в конце этого блока. Когда выполнение кода достигает `break`, происходит выход из всего блока `switch` и код продолжает свое выполнение

с участка, расположенного ниже. Если вы не указали ключевое слово `break`, то код продолжит выполняться внутри блока `case green`. Разница в том, что затем произойдет переход к следующему блоку `case` (в нашем примере `black`) и выполнению его кода. Таким же образом будут выполнены все последующие блоки `case`, если на пути не попадется другое ключевое слово `break`.

Таким образом, если вы запустите приведенный выше код, то увидите окно уведомления, как на рис. 4.6.

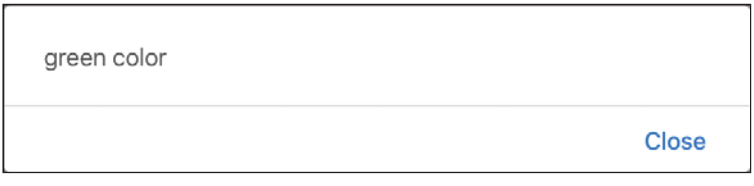


РИС. 4.6.
Окно уведомления

Вы можете менять значение переменной `color` на другие допустимые значения, чтобы посмотреть, как выполняются другие блоки `case`. В некоторых случаях ни одно их значений блоков `case` не будет совпадать с результатом вычисления выражения. В таких ситуациях инструкция `switch` просто ничего не будет делать. Если вы захотите определить для нее поведение по умолчанию, добавьте блок `default`:

```
switch (color) {
  case "yellow":
    alert("yellow color");
    break;
  case "red":
    alert("red color");
    break;
  case "blue":
    alert("blue color");
    break;
  case "green":
    alert("green color");
    break;
  case "black":
    alert("black color");
    break;
  default:
    alert("no known color specified");
    break;
}
```

Обратите внимание, что блок `default` выглядит немного иначе, чем другие инструкции `case`. Фактически в нем просто отсутствует слово `case`.

Сходство с инструкцией if...else

Мы видели, что инструкция `switch` используется для вычисления условий — так же, как и инструкция `if...else`, на которую мы потратили уйму времени. Учитывая такой финт, давайте изучим этот момент подробнее и рассмотрим, как будет выглядеть инструкция `if`, если мы буквально переведем ее в инструкцию `switch`.

Допустим, есть такая инструкция `if`:

```
let number = 20;

if (number > 10) {
  alert("yes");
} else {
  alert("nope");
}
```

Так как переменная `number` имеет значение `20`, инструкция `if` будет вычисляться как `true`. Выглядит достаточно просто. А теперь преобразуем ее в инструкцию `switch`:

```
switch (number > 10) {
  case true:
    alert("yes");
    break;
  case false:
    alert("nope");
    break;
}
```

Обратите внимание, что наше выражение — это `number > 10`. Значение `case` для блоков `case` установлено как `true` или `false`. Поскольку `number > 10` вычисляется как `true`, выполняется код внутри блока `true`. Несмотря на то что выражение в этом случае не было таким же простым, как считывание значения цвета из переменной в предыдущем разделе, на наш взгляд, принцип работы инструкции `switch` не изменился. Выражения могут быть настолько сложными, насколько вы пожелаете. Если они вычисляются во что-то, что может быть сопоставлено со значением блока `case`, тогда все в шоколаде.

Далее предлагаю рассмотреть чуть более сложный пример. Преобразуем уже рассмотренную инструкцию `switch` с переменной `color` в эквивалентные ей инструкции `if…else`. Первоначальная версия этой инструкции выглядит так:

```
let color = "green";

switch (color) {
  case "yellow":
    alert("yellow color");
    break;
  case "red":
    alert("red color");
    break;
  case "blue":
    alert("blue color");
    break;
  case "green":
    alert("green color");
    break;
  case "black":
    alert("black color");
    break;
  default:
    alert("no color specified");
    break;
}
```

Если преобразовать ее в череду инструкций `if…else`, она станет выглядеть так:

```
let color = "green";

if (color == "yellow") {
  alert("yellow color");
} else if (color == "red") {
  alert("red color");
} else if (color == "blue") {
  alert("blue color");
} else if (color == "green") {
  alert("green color");
} else if (color == "black") {
  alert("black color");
} else {
  alert("no color specified";
}
```

Как мы видим, инструкции `if...else` очень схожи с инструкциями `switch`, и наоборот. Блок `default` в этом случае становится блоком `else`, а связь между выражением и значением `case` инструкции `switch` объединена в условии `if...else` инструкции `if...else`.

Что же использовать

В предыдущем разделе мы увидели, насколько взаимозаменяемы инструкции `switch` и `if...else`. При наличии двух схожих способов выполнения каких-либо действий возникает естественное желание понять, когда лучше использовать один, а когда другой. Если кратко, то используйте тот, который вам больше нравится. В интернете много спорят о том, когда и какую из инструкций лучше использовать, но такие споры никогда не приводят к чему-то вразумительному.

Лично я предпочитаю использовать ту инструкцию, которая будет легче читаться. Если посмотреть на предыдущее сравнение инструкций `if...else` и `switch`, можно заметить, что при наличии большого количества условий инструкция `switch` выглядит немного чище. Она однозначно лаконичнее и читабельнее. Но только вам решать, какое количество условий определит ваш выбор использовать ту или иную инструкцию. Для меня обычно это четыре или пять условий.

Инструкция `switch` лучше работает, когда вы вычисляете выражение и сопоставляете результат со значением. Если вы выполняете более сложные действия с использованием странных условий, проверкой значения и т. д., вероятно, вы предпочтете использовать что-то другое. Кроме того, в ход могут пойти даже не инструкции `if...else`. Об альтернативах поговорим позже.

Подводя итог, останемся верны прежней рекомендации — использовать то, что нам больше нравится. Если вы являетесь членом команды, имеющей свои предпочтения по написанию кода, то следуйте им. В любом случае, что бы вы ни делали, будьте последовательны. Это облегчит не только вашу жизнь, но и жизнь тех, кто будет работать с вашим кодом. Кстати говоря, лично я никогда не имел дело с ситуациями, в которых мне пришлось бы использовать инструкцию `switch`. Ваш опыт все же может отличаться от моего.

КОРОТКО О ГЛАВНОМ

Несмотря на то что создание настоящего искусственного интеллекта выходит за рамки этой книги, вы можете писать код, помогающий приложениям принимать решения. Этот код почти всегда будет принимать форму инструкции `if…else`, в которой вы предоставите браузеру набор доступных для него выборов:

```
let loginStatus = false;

if (name == "Admin") {
  loginStatus = true;
}
```

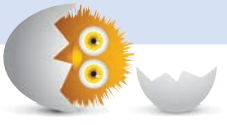
Эти выборы основываются на условиях, которые должны быть вычислены как `true` или `false`.

В этой главе мы изучили механику работы инструкций `if…else` и их кузена — инструкции `switch`. В будущих главах вы увидите, что мы будем часто взаимодействовать с этими инструкциями, как если бы они были нашими старыми друзьями. А к концу книги вы сдружитесь еще больше.

Если у вас есть какие-либо вопросы, касающиеся пройденного материала, не беспокойтесь. Обращайтесь с ними на форуме по адресу: https://forum.kirupa.com. Там вы получите быструю помощь от меня или других добрейших разработчиков.

В ЭТОЙ ГЛАВЕ:

- узнаем, как заставить код выполняться повторно;
- научимся пользоваться циклами `for`, `while` и `do...while`.



5

ЦИКЛЫ FOR, WHILE И DO... WHILE

Иногда при написании программы требуется использовать повторяющиеся действия или выполнять код несколько раз. Например, нам нужно вызвать функцию `saySomething` десять раз подряд.

Один из способов это сделать — просто перекопировать ее 10 раз и вызвать:

```
saySomething();
saySomething();
saySomething();
saySomething();
saySomething();
saySomething();
saySomething();
saySomething();
saySomething();
saySomething();
```

Такой способ сработает, и мы добьемся того, чего хотели... но так делать не стоит. Все же дублирование кода — плохая идея. Если бы нам давали пятак каждый раз, когда вы это прочтете, мы стали бы на четыре или пять монет богаче. #killing_it

Итак, даже если мы решим продублировать код несколько раз вручную, такой подход не сработает на практике. Количество повторений, которые потребуется сделать, будет варьировать в зависимости от таких внешних факторов, как число элементов в коллекции данных, результат, полученный от веб-сервиса, число букв в слове и многих других вещей, которые постоянно будут меняться. Такое количество повторов не будет всегда одинаковым, к примеру 10. Часто нам может потребоваться выполнение ОГРОМНОГО количества повторов. И нам уж точно не захочется перекопировать фрагменты кода сотни или тысячи раз, чтобы получить нужное число повторений. Это было бы ужасно.

АЛЬТЕРНАТИВА ALERT

В предыдущих главах мы использовали функцию `alert` для отображения текста на экране. В этой главе мы познакомимся с другим, не столь навязчивым способом отображения. И это будет функция `document.write`:

```
document.write("Show this on screen!");
```

С помощью этой функции можно вывести заданный текст на страницу браузера без использования диалогового окна, которое к тому же нужно постоянно закрывать. Вы поймете, почему мы предпочитаем более упрощенные структуры, когда изучите циклы и то, как выводить на экран большое количество информации.

Нужно универсальное решение для повтора кода с сохранением контроля над тем, сколько раз этот повтор будет произведен. В JavaScript такое решение представлено в виде *цикла*, который может производиться в трех вариантах:

- циклы `for`;
- циклы `while`;
- циклы `do...while`.

Каждый из вариантов позволяет определить код, который требуется повторить (то есть цикл), а также способ остановить это повторение при соблюдении заданного условия. В следующих разделах мы все это изучим.

Поехали!

Цикл for

Один из наиболее популярных способов создания цикла — это использование инструкции `for`. Цикл `for` позволяет повторять выполнение кода до тех пор, пока заданное нами выражение не вернет `false`. Разберем наглядный пример.

Если преобразовать прошлый пример с `saySomething` с помощью `for`, он будет выглядеть так:

```
for (let i = 0; i < 10; i++) {
  saySomething();
}

function saySomething() {
  document.writeln("hello!");
}
```

Если вы хотите сразу испробовать свои силы, попробуйте вписать следующий код в тег *script* внутри HTML-документа:

```
<!DOCTYPE html>
<html>

<head>
  <meta charset="utf-8">
  <title>Loops!</title>

  <style>

  </style>
</head>

<body>
  <script>
    for (let i = 0; i < 10; i++) {
      saySomething();
    }

    function saySomething() {
      document.writeln("hello!");
    }
  </script>
</body>

</html>
```

Когда документ будет готов, сохраните его и выполните предпросмотр в браузере. На рис. 5.1 показано то, что вы увидите после загрузки страницы.

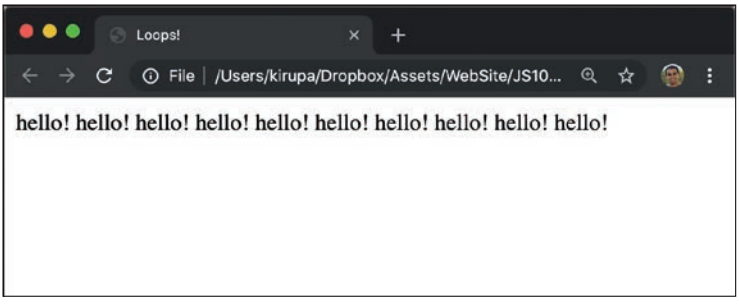


РИС. 5.1.
Слово hello! повторяется слева направо

Слово `hello!` повторится десять раз по всей странице. Это стало возможным благодаря циклу `for`. Поэтому в виде благодарности за полученный нами опыт изучим все аспекты работы цикла. А вот и наша звезда:

```
for (let i = 0; i < 10; i++) {
  saySomething();
}
```

Это цикл `for`, и он существенно отличается от тех инструкций, с которыми мы познакомились к этому моменту. Для понимания отличий представим цикл `for` в обобщенном виде, как показано на рис. 5.2.

```
for (start_point; condition; step) {
  // код для выполнения
}
```

РИС. 5.2.
Общий вид верхнего уровня цикла

Далее верхний уровень сопоставим с действительными значениями из нашего примера (рис. 5.3).

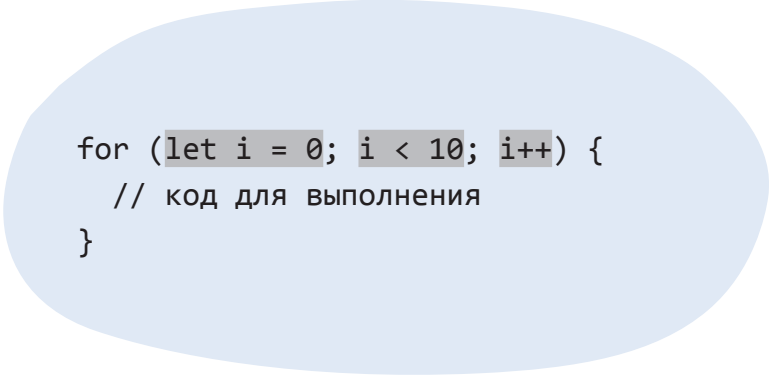


РИС. 5.3.
Действительные значения

Каждая из этих трех разноцветных секций (стадий) играет свою важную роль в процессе выполнения цикла. Чтобы использовать цикл `for` грамотно, необходимо понимать, за что отвечает каждая из секций. Рассмотрим каждую подробнее.

Стартовое значение

В этой секции мы определяем *стартовое значение* переменной-счетчика нашего цикла. Обычно сюда помещается некий код для объявления и инициализации переменной, подобно тому как мы сделали на рис. 5.4.

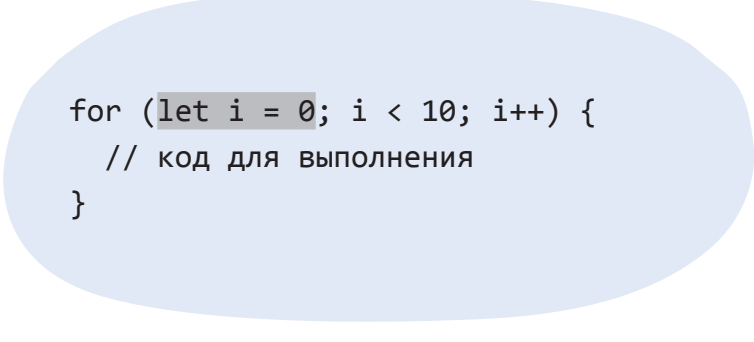


РИС. 5.4.
Объявление и инициализация переменной i

Так мы сообщаем JavaScript, что наш цикл начинается с переменной `i`, инициализированной как `0`.

Шаг

Мы пока пропустим вторую секцию и перейдем к секции *шаг* (рис. 5.5).

```
for (let i = 0; i < 10; i++) {
  // код для выполнения
}
```

РИС. 5.5.
Шаг

На этой стадии мы определяем, как будет изменяться наше стартовое значение. Например, здесь мы сообщаем, что при каждом выполнении цикла значение `i` будет увеличиваться на `1`. Это обозначается таинственной записью `i++`. Мы разберемся со значением `++` позже, когда рассмотрим принципы работы чисел и математики в JavaScript, однако иначе это можно было бы выразить как i = i + 1.

Условие, или продолжительность цикла

Возвращаясь к пропущенной нами секции, мы видим *условие*, которое определяет, когда закончится повторение цикла (рис. 5.6).

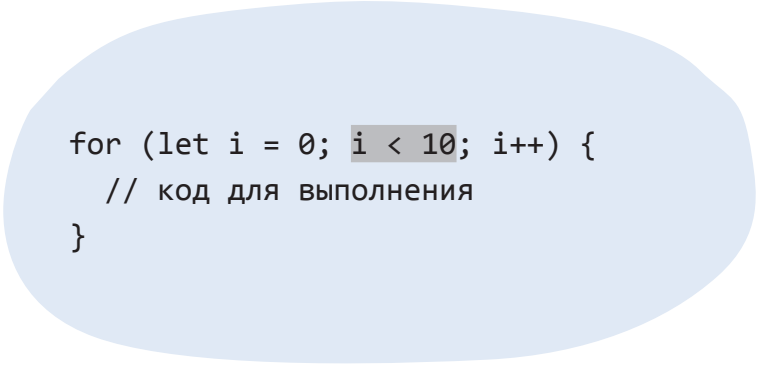


РИС. 5.6.
Часть цикла, представляющая условие

В нашем примере условие гласит, что переменная `i` должна иметь значение меньше `10`:

- если переменная `i` меньше `10`, выражение вычисляется как `true` и цикл продолжает выполнение;
- если переменная становится равна или больше `10`, то условие вычисляется как `false` и цикл прекращается.

Собирая все вместе

Теперь, когда мы изучили каждую часть цикла `for` более подробно, воспользуемся свежими знаниями, чтобы еще раз пробежаться по всему процессу от начала до конца и понять, что при этом происходит. Наш пример целиком выглядит так:

```
for (let i = 0; i < 10; i++) {
  saySomething();
}
function saySomething() {
  document.writeln("hello!");
}
```

Когда цикл `for` впервые достигает стартового значения, переменная `i` создается и инициализируется как `0`. Далее мы переходим к условию, определяющему, следует ли циклу продолжаться или нет. Условие проверяет, является ли значение `i` меньше `10`. 0 меньше `10`? Да, следовательно, условие вычисляется как `true` и код, содержащийся внутри цикла, выполняется. Как только это происходит, наступает черед шага.

На этой стадии переменная `i` увеличивается на `1` и получает значение `1`. К этому моменту наш цикл сработал один раз, обычно это называют *итерацией*. Теперь настало время следующей итерации.

При следующей итерации весь цикл начинается с начала, но переменная `i` уже не инициализируется, а просто представляет значение `1`, перешедшее из предыдущей итерации. Далее в условии вновь проверяется, меньше ли это значение, чем `10`, что оказывается верным. После этого выполняются код внутри цикла (в нашем случае функция `saySomething`) и шаг, увеличивающий значение `i` на `1`. В итоге значение `i` становится равно уже `2`, на чем текущая итерация завершается, уступая место следующей.

В этом процессе итерации сменяют друг друга, пока условие `i < 10` не будет вычислено как `false`. Поскольку мы начали цикл при `i`, равной `0`, определили, что он завершится при `i`, равном или большем `10`, а `i` увеличивается на `1` при каждой итерации, то этот цикл (и любой содержащийся в нем код) будет выполнен `10` раз до своего завершения.

Некоторые примеры цикла for

В предыдущем разделе мы разобрали простой цикл `for` и описали все его внутренние процессы. Но в отношении таких циклов и вообще всего остального в JavaScript есть один нюанс, а именно простые примеры, как правило, не охватывают все интересующие нас случаи. Лучшим решением будет рассмотреть еще несколько примеров с циклами `for`, чем мы и займемся в следующих разделах.

Прерывание цикла

Иногда возникает необходимость прервать цикл прежде, чем он завершится. Для этого мы используем ключевое слово `break`. Ниже приведен пример:

```
for (let i = 0; i < 100; i++) {
  document.writeln(i);

  if (i == 45) {
    break;
  }
}
```

Если задать `i` значение `45`, ключевое слово `break` прервет цикл. И хотя я просто взял этот пример из своей головы, отныне, если у вас возникнет необходимость прервать цикл, вы будете вооружены этим знанием.

Пропуск итерации

Кроме того, иногда могут возникать ситуации, когда нужно пропустить текущую итерацию, чтобы перейти к следующей. Ловчее всего это сделать с помощью ключевого слова `continue`:

```
let floors = 28;

for (let i = 1; i <= floors; i++) {
  if (i == 13) {
    // нет такого этажа (floor)
    continue;
  }

  document.writeln("At floor: " + i + "<br>");
}
```

В отличие от `break`, который просто прерывает цикл, `continue` как бы сообщает ему: «Остановись и переходи к следующей итерации». Чаще всего мы будем использовать ключевое слово `continue` при обработке ошибок, чтобы цикл переходил к следующему элементу.

Возврат назад

Нет никаких причин, по которым стартовое значение должно иметь переменную, инициализированную как `0`, и увеличивать ее:

```
for (let i = 25; i > 0; i--) {
  document.writeln("hello");
}
```

Можно легко начать с большего значения и затем производить его уменьшение, пока условие цикла не вернет `false`.

Вы могли слышать, что такой подход повышает производительность цикла. Дискуссия на тему того, действительно ли *уменьшение* быстрее увеличения, ведется *до сих пор*, но вы вольны экспериментировать и на своем опыте понаблюдать, так ли это.

Числа использовать необязательно

Необязательно использовать числа при заполнении цикла `for`:

```
for (let i = "a"; i != "aaaaaaaa"; i += "a") {
  document.writeln("hmm. ");
}
```

Вы можете написать все, что захотите, пока это не помешает циклу завершиться. Обратите внимание, что в этом примере в качестве единицы исчисления цикла мы используем букву `a`. При каждой итерации значение `i` увеличивается на одну `a`, а цикл останавливается, когда `i` становится равна `aaaaaaaa`.

О нет! Он не сделал этого!

О, да! Я сделал это! Побывал там, сфотографировал, запостил фотку на фейсбуке и вернулся:

```
let i = 0;
let yay = true;

for (; yay;) {
  if (i == 10) {
    yay = false;
  } else {
    i++;
    document.writeln("weird");
  }
}
```

Не обязательно заполнять все три секции цикла `for`, чтобы он заработал. До тех пор пока вы обеспечиваете выполнение условия завершения цикла, вы можете делать все, что захотите. Прямо как в примере выше.

Другие циклы

В тени его превосходительства цикла `for` живут и другие варианты циклов, а именно `while` и `do...while`. Для полного завершения темы давайте рассмотрим и их.

Цикл while

Цикл `while` повторяет заданный код, пока его условие (другое выражение) не вернет `false`. Взгляните на следующий пример:

```
let count = 0;

while (count < 10) {
  document.writeln("looping away!");

  count++;
}
```

В этом примере условие представлено выражением `count < 10`. При каждой итерации цикл увеличивает `count` на `1`:

```
let count = 0;

while (count < 10) {
  document.writeln("looping away!");

  count++;
}
```

Как только `count` станет равен `10`, цикл прекратится, так как выражение `count < 10` вернет `false`. Если вы посмотрите на все, что делает этот цикл, то увидите, что он во многом имитирует работу цикла `for`. В то время как цикл `for` требует определения стадий начала, условия и шага, цикл `while` предполагает, что вы определите все эти стадии по-своему.

Цикл do...while

А теперь пора познакомиться с Мег Гриффин[1] в семействе циклов. Цель цикла `do...while` определена еще меньше, чем в случае с `while`. Если в цикле `while` условное выражение расположено перед выполнением самого цикла, то в `do...while` оно находится в конце.

[1] Мегатрон «Мег» Гриффин — персонаж мультсериала «Гриффины», закомплексованная и неуверенная в себе девушка, над которой подшучивают семья и окружающие. — *Примеч. ред.*

Вот вам пример:

```
let count = 0;

do {
    document.writeln("I don't know what I am doing here! <br>");

    count++;
} while (count < 10);
```

Главное отличие между циклами `while` и `do…while` в том, что содержимое первого не может быть выполнено, если его условное выражение изначально вернет `false`:

```
while (false) {
  document.writeln("Can't touch this!");
}
```

В случае же с циклом `do…while`, из-за того что условное выражение вычисляется только после одной итерации, содержимое цикла будет выполнено минимум один раз:

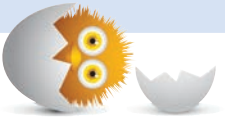
```
do {
  document.writeln("This code will run once!");
} while (false);
```

В некоторых ситуациях это может сыграть на руку. Прежде чем подвести итоги, хочу сказать еще кое-что. Инструкции `break` и `continue`, которые мы встречали ранее как часть прекрасного цикла `for`, схожим образом работают внутри циклов `while` и `do...while`.

КОРОТКО О ГЛАВНОМ

Итак, вы познакомились с циклами `for` и способами их использования и параллельно затронули их аналоги `while` и `do…while`. Пока что мы не будем часто использовать циклы. По мере погружения в более сложные ситуации вроде сбора данных, элементов DOM, управления текстом и другие процессы мы будем прибегать к их использованию все чаще. Главное — не забывать изученную в этой главе информацию.

Если у вас есть вопросы по пройденному материалу, не стесняйтесь задавать их на форуме https://forum.kirupa.com и вы получите оперативный ответ если не от меня, то от других умнейших и готовых помочь разработчиков.



В ЭТОЙ ГЛАВЕ:

- научимся комментировать код;
- разберемся, как делать это правильно.

6

КОММЕНТИРОВАНИЕ КОДА… ЧТО ЗА?!

Кажется, что все, что мы пишем в редакторе кода, предназначено исключительно для браузера:

```
let xPos = -500;

function boringComputerStuff() {
  xPos += 5;

  if (xPos > 1000) {
    xPos = -500;
  }
}
boringComputerStuff();
```

Но скоро мы убедимся, что это совсем не так. У кода есть и другая аудитория — люди.

Код часто используют или читают другие. В особенности если вы работаете в одной команде с другими разработчиками. Чтобы на выходе код выглядел максимально эффективным, нужно убедиться, что он понятен другим. Это касается и тех, кто работает независимо. Любая прекрасная функция, которая кажется логичной сегодня, может выглядеть полной чушью через неделю.

Есть множество способов решения этой проблемы. Один из лучших — это использование комментариев. В этом коротком разделе мы ответим на вопрос, что такое комментарии, узнаем, как они обозначаются в JavaScript, и рассмотрим правильные способы их использования.

Поехали!

Что такое комментарии?

Комментарии — это то, что мы пишем в виде части кода для передачи информации читающим его:

```
// Это вам за то, что не пригласили меня на день рождения!
let blah = true;

function sweetRevenge() { while (blah) {
  // Бесконечные диалоговые окна! Ха-ха-ха!!!!
    alert("Hahahaha!");
  }
}
sweetRevenge();
```

В этом примере комментарии отмечены символами `//` и дают относительно точную информацию о коде, который описывают.

О комментариях важно помнить, что они не выполняются вместе с остальным кодом. *JavaScript игнорирует комментарии*. Вы ему не нравитесь, и его не волнует, что вы хотите сказать, поэтому даже не парьтесь по поводу синтаксиса, пунктуации и всего того, что важно при написании кода. Комментарии нужны только для понимания действий с отдельными фрагментами кода.

Помимо этого, комментарии служат еще одной цели. Их можно оставлять, чтобы отмечать строки кода, выполнять которые сейчас не нужно:

```
function insecureLogin(input) {
  if (input == "password") {
    // let key = Math.random() * 100000;
    // processLogin(key);
  }
  return false;
}
```

В этом примере две нижеприведенные строки отображаются в редакторе, но не выполняются:

```
// let key = Math.random() * 100000;
// processLogin(key);
```

Мы будем часто использовать редактор в качестве черновика, и комментарии — это отличный способ отслеживать шаги, которые мы предпринимали, чтобы код заработал, при этом никак не влияя на работу приложения.

Однострочные комментарии

Есть несколько вариантов комментариев в коде. Один из них — это *однострочные комментарии*, которые обозначаются двумя наклонными чертами `//` и содержат сообщение. Такой вид комментариев мы уже видели.

Мы можем размещать их в отдельно выделенной строке:

```
// Возвращает больший из двух аргументов.
function max(a, b) {
  if (a > b) {
    return a;
  } else {
    return b;
  }
}
```

Или в одну строку с инструкцией:

```
let zorb = "Alien"; // Раздражать жителей планеты.
```

Только вам решать, где именно размещать комментарий. Выбирайте место, которое покажется наиболее подходящим.

Поскольку мне нравится быть заезженной пластинкой, еще раз повторю: комментарии не выполняются вместе с остальной частью приложения. Они видны только вам, мне и, возможно, нашим друзьям. Если последняя фраза вам непонятна, значит, вы, скорее всего, не смотрели одну из величайших комедий нашего века (к/ф «Он, я и его друзья»). В таком случае я настоятельно рекомендую отвлечься от учебы и потратить пару часов на исправление этого упущения.

КОММЕНТАРИИ С ПОМОЩЬЮ JSDOC

Когда мы пишем код и знаем, что с ним будут работать другие, наверняка захочется передать информацию наиболее простым способом и избавить от необходимости пересматривать весь исходник. Такой способ есть, и все благодаря инструменту JSDoc, который предполагает несколько иной подход к написанию комментариев:

```
/**
 * Перетасовывает содержимое массива (Array).
 *
 * @this {Array}
 * @returns {Array} Текущий массив с перетасованным содержимым.
 */
Array.prototype.shuffle = function () {
  let input = this;

  for (let i = input.length – 1; i >= 0; i--) {

    let randomIndex = Math.floor(Math.random() * (i + 1));
    let itemAtIndex = input[randomIndex];

    input[randomIndex] = input[i];
    input[i] = itemAtIndex;
  }
  return input;
}
```

Как только вы оставили комментарии к файлу, воспользуйтесь JSDoc для экспорта согласующихся частей комментариев в удобно просматриваемый набор HTML-страниц. Это позволяет тратить больше времени на написание самого кода и в то же время помогает пользователям разобраться в действиях вашего кода и понять, как использовать его отдельные части.

Если вы хотите подробнее узнать об использовании JSDoc, посетите сайт этого проекта https://jsdoc.app/.

Многострочные комментарии

Сложность с однострочными комментариями состоит в том, что приходится указывать символы `//` в начале каждой строки, к которой нужно его оставить. Это может быть очень утомительным, особенно если вы пишите большой комментарий или комментируете большой кусок кода.

Для таких случаев существует другой способ оформления комментариев, а именно с помощью символов /* и */, которые определяют начало и конец. И в результате получаются *многострочные комментарии*:

```
/*
let mouseX = 0; let mouseY = 0;

canvas.addEventListener("mousemove", setMousePosition, false);

function setMousePosition(e) {
  mouseX = e.clientX;
  mouseY = e.clientY;
}
*/
```

Вместо добавления символов // в каждую строку можно использовать символы /* и */, что сэкономит наши время и нервы.

Мы будем комбинировать однострочные и многострочные комментарии в зависимости от того, что хотим задокументировать. Поэтому нам нужно знать, как использовать оба описанных подхода.

Лучшие способы комментирования

Теперь, когда мы уже имеем достаточное представление о том, что такое комментарии, и знаем несколько способов их написания в JavaScript, давайте поговорим о том, как их правильно использовать, чтобы облегчить чтение кода.

- **Всегда комментируйте код по ходу его написания.** Создание комментариев — ужасно скучное занятие, но оно является важной частью написания кода. Вам и другим будет гораздо проще и быстрее понять код, прочитав комментарий, чем просматривать строка за строкой скучную писанину на JavaScript.
- **Не откладывайте написание комментариев на потом.** Такой подход ведет к прокрастинации, которая будет возрастать при выполнении рутинной работы. Если вы не прокомментируете код по ходу написания, то, скорее всего, этого уже не сделаете, что точно никому не пойдет на пользу.
- **Используйте обычные слова и поменьше JavaScript.** Комментарии — это одно из немногих мест, где при написании программы вы можете свободно пользоваться человеческим языком. Не услож-

няйте комментарии кодом без необходимости. Выражайтесь ясно, сжато, словами.

- **Используйте пустые пространства.** Вам нужно добиться того, чтобы при просматривании больших блоков кода комментарии выделялись и легко прослеживались. Для этого нужно активнее пользоваться пробелом и клавишами Enter-Return. Взгляните на следующий пример:

```
function selectInitialState(state) {
  let selectContent = document.querySelector("#stateList"); let
  stateIndex = null;

  /*
   Для возвращения в прежнее состояние нужно убедиться, что
   мы выбираем его в нашем UI. Это означает, что производится
   итерация по каждому состоянию выпадающего списка, пока
   не будет найдено совпадение. Когда совпадение найдено,
   нужно убедиться, чтобы оно было выбрано.
   */

  for (let i = 0; i < selectContent.length; i++) {

    let stateInSelect = selectContent.options[i].innerText;

      if (stateInSelect == state) {
        stateIndex = i;
      }
    }

    selectContent.selectedIndex = stateIndex;
  }
```

Обратите внимание, что наш комментарий грамотно выделен пробелами и его отлично видно в остальной части кода. Если ваши комментарии будут разбросаны где попало и их будет сложно опознать, это сильно замедлит чтение кода как для вас, так и для других.

- **Не комментируйте очевидные вещи.** Если строка кода говорит сама за себя, не стоит тратить время на объяснение ее действий. Это допустимо, только если в ней кроется некое едва уловимое поведение, которое вы хотите обозначить в качестве предупреждения. В остальных же случаях лучше потратить время на комментирование менее понятных участков кода.

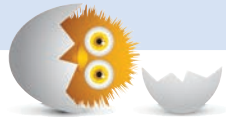
Эти рекомендации помогут вам комментировать код значительно эффективнее. Если вы работаете в крупном проекте с другими людьми,

могу вас заверить, что в вашей команде уже существует устоявшийся регламент, определяющий правильное написание комментариев. Уделите время ознакомлению с этим регламентом и следуйте ему. В итоге останутся довольными все: как вы, так и ваша команда.

КОРОТКО О ГЛАВНОМ

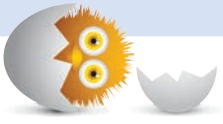
Комментарии зачастую рассматриваются как вынужденное зло. В конце концов, стали бы вы тратить несколько минут на то, чтобы прокомментировать то, что вы и так прекрасно понимаете, или предпочли бы поработать над очередной «вкусной» частью функциональности? Я предпочитаю относиться к комментариям как к «долгосрочному вложению». Значение и польза комментариев зачастую проявляются не сразу, а становятся очевидными, когда с вашим кодом начинают работать другие люди, а также если вам приходится вновь вернуться к своему коду, а вы уже забыли, что он делает и как работает. Не жертвуйте долгосрочной экономией времени в перспективе ради незначительного ускорения в работе сейчас. Делайте ставки на однострочные (`//`) и многострочные (`/*` и `*/`) комментарии уже сегодня, пока еще не слишком поздно.

И если у вас появились какие-либо вопросы по этой теме, обращайтесь за оперативной помощью ко мне и другим разработчикам на форуме https://forum.kirupa.com.



В ЭТОЙ ГЛАВЕ:

- узнаем, как делать задержки при выполнении кода;
- научимся производить повторное выполнение кода, не блокируя все приложение.



7

ТАЙМЕРЫ

По умолчанию код выполняется синхронно. Проще говоря, это значит, что выполнение инструкции происходит сразу, как только до нее доходит очередь. В этом процессе нет никаких *и*, *если* или *но*. Возможность задержки выполнения кода или откладывания его на потом не характерна для работы JavaScript по умолчанию. Мы видели нечто подобное, когда изучали циклы. Цикл выполняется со скоростью света без малейших задержек между итерациями. Это в значительной степени способствует скоростным вычислениям, но в то же время мешает, если мы хотим производить обновление в более размеренном (то есть медленнее) темпе.

Однако это вовсе не значит, что нет способа препятствовать мгновенному выполнению кода. Если немного отклониться от основной темы, можно обнаружить три функции, позволяющие это делать, — `setTimeout`, `setInterval` и `requestAnimationFrame`. В этом разделе мы рассмотрим назначение и применение каждой из них.

Поехали!

Задержка с помощью setTimeout

Функция `setTimeout` позволяет откладывать выполнение заданного кода. Вариант ее использования достаточно интересен. С помощью

этой функции мы можем обозначить, какой код выполнять и сколько миллисекунд должно пройти, прежде чем это выполнение произойдет. На практике она выглядит примерно так:

```
let timeoutID = setTimeout(someFunction, delayInMilliseconds);
```

Немного углубимся в этот пример. Допустим, нужно, чтобы функция `showAlert` была вызвана через 5 секунд. В этом случае объявление функции `setTimeout` будет выглядеть так:

```
function showAlert() {
  alert("moo!");
}

let timeoutID = setTimeout(showAlert, 5000);
```

Круто, не правда ли? Теперь поговорим о чем-то менее интересном, что позволит внести большую ясность в эту тему. Это что-то связано с переменной `timeoutID`, инициализированной как функция `setTimeout`. Эта переменная появилась не случайно. Если понадобится вновь обратиться к таймеру `setTimeout`, то потребуется способ, как сослаться на него. Самый легкий способ — это ассоциирование переменной с объявлением `setTimeout`.

Вы можете поинтересоваться, а зачем вообще повторно обращаться к таймеру? Что ж, единственная причина, которая приходит на ум, — для его отмены. Можно легко отменить функцию `setTimeout` с помощью функции `clearTimeout`, в которую в качестве аргумента нужно передать ID таймаута (`timeoutID`):

```
clearTimeout(timeoutID);
```

Если вы вообще не планируете отменять установленный таймер, можете использовать `setTimeout` напрямую, не прибегая к инициализации переменной.

Давайте поговорим о том, когда это применяется на практике, а именно при *разработке UI* (пользовательского интерфейса). При его разработке откладывание некоторых действий на определенное время достаточно распространено.

Например, когда меню разворачивается и сворачивается обратно через несколько секунд, если пользователь с ним не взаимодействует.

Или в случае, когда есть некая операция, которая выполняется слишком долго и не может завершиться, а функция `setTimeout` прерывает эту операцию и возвращает контроль пользователю.

Мой любимый пример (которому я также посвятил целый урок) — это использование функции `setTimeout`, чтобы определить, активен ли пользователь или отсутствует.

Если вы поищите информацию о `setTimeout` на указанном сайте или в Google, то найдете множество реальных примеров, где она может очень пригодиться.

Выполнение циклов с помощью setInterval

Следующая функция-таймер, которую мы рассмотрим, — это `setInterval`. Она похожа на `setTimeout` в том, что позволяет выполнять код спустя определенное время. Отличие же ее в том, что она не просто выполняет код один раз, а продолжает его повторное выполнение бесконечно.

Вот как она используется:

```
let intervalID = setInterval(someFunction, delayInMilliseconds);
```

За исключением имени, способ использования функции `setInterval` идентичен способу применения `setTimeout`. Первый аргумент определяет встроенный код или функцию, которую нужно выполнить. Второй аргумент определяет время задержки (интервал) между выполнениями. При желании вы также можете инициализировать функцию `setInterval` как переменную для хранения ID интервала, который позднее сможете использовать, например, для прекращения выполнения.

Отлично! Теперь, когда вы уже знакомы с процессом, рассмотрите пример кода, выполняющего функцию `drawText` с интервалом между циклами в 2 секунды:

```
<!DOCTYPE html>
<html>

<head>
  <meta charset="utf-8">
  <title>Show me some text!</title>
</head>

<body>
  <script>
    let thingToPrint = "";

    function drawText() {
      thingToPrint += "#";
      document.writeln(thingToPrint);
    }
```

```
    setInterval(drawText, 2000);
  </script>
</body>

</html>
```

Для прекращения выполнения цикла мы можем использовать функцию `clearInterval` с соответствующим аргументом:

```
clearInterval(intervalID);
```

Используется она в точности как `clearTimeout` — мы передаем ей ID таймера `setInterval`, который при желании можем задать в момент его создания.

В реальной жизни функция `setInterval` долгое время оставалась основной функцией для создания анимации в JavaScript. Например, если потребуется получить 30 или 60 кадров в секунду, вы будете «играть» со значением задержки времени примерно так:

```
// 1000 разделить на 60 – это значение в миллисекундах для 60fps
setInterval(moveCircles, 1000 / 60);
```

Чтобы увидеть работу `setInterval` и в других реалистичных примерах на самом сайте, загляните в конец статьи «Создание симпатичного слайдера контента» (https://www.kirupa.com/html5/creating_a_sweet_content_slider.htm) или прочтите материал «Создание аналоговых часов» (https://www.kirupa.com/html5/create_an_analog_clock_using_the_canvas.htm). И там и там вы найдете достаточно яркие примеры использования `setInterval`.

Плавная анимация с помощью requestAnimationFrame

Настало время перейти к моей любимой функции: `requestAnimaton-Frame`. Эта функция создана для синхронизации кода с событием перерисовки содержимого браузера. Поясняю подробнее: в любой момент браузер занят обработкой миллиона различных элементов. Среди них: манипуляции с макетом, реакция на прокрутку страниц, прослушивание кликов мыши, отображение результатов нажатий клавиатуры, исполнение кода, загрузки ресурсов и т. д. Одновременно со всем этим он также перерисовывает экран со скоростью 60 кадров в секунду, ну или пытается это делать.

Если у вас есть код, запускающий какую-либо анимацию на экране, скорее всего, вам захочется, чтобы эта анимация выполнялась кор-

ректно и не затерялась в остальной мешанине действий, выполняемых браузером. В таких случаях использование ранее отмеченной техники `setInterval` не гарантирует, что скорость обновления не упадет, когда браузер будет занят оптимизацией других элементов. Чтобы браузер не приравнивал код, отвечающий за анимацию, любому другому JavaScript-коду, существует функция `requestAnimationFrame`. Браузеры воспринимают эту функцию особым образом, что позволяет задавать время ее выполнения наиболее оптимально, избегая провала частоты кадров, а также ненужных действий и прочих побочных эффектов, мешающих применению других решений, связанных с выполнением циклов.

Запуск этой функции почти такой же, как в случае с `setTimeout` / `setInterval`:

```
let requestID = requestAnimationFrame(someFunction);
```

Единственное заметное отличие состоит в том, что мы не задаем значение продолжительности. Это значение вычисляется автоматически в зависимости от: текущей скорости обновления кадров, активна ли текущая вкладка или нет, работает ли устройство от батареи или сети и многих других факторов, выходящих далеко за пределы нашего контроля и понимания.

Тем не менее подобное использование функции `requestAnimationFrame` — лишь учебный пример. В реальности вам вряд ли придется вызывать `requestAnimationFrame` всего лишь раз, как в нашем примере. Ключом ко всем анимациям, создаваемым в JavaScript, является анимационный цикл, и именно к этому циклу нам нужно применить `requestAnimationFrame`. Результат такого применения выглядит следующим образом:

```
function animationLoop() {
  // код, отвечающий за анимацию

  requestAnimationFrame(animationLoop)
}

// Начать выполнение цикла анимации!
animationLoop();
```

Обратите внимание, что наша функция `requestAnimationFrame` определяет выполнение вызова функции `animationLoop` каждый раз, когда браузер производит обновление. Выглядит это так, как будто функция `requestAnaimationFrame` вызывает `animationLoop` напрямую, что не совсем так. И это вовсе не ошибка в коде. Несмотря на то что такой

вид циклической зависимости на практике гарантированно привел бы к зависанию браузера, реализация `requestAnimationFrame` этого избегает. Наоборот, она обеспечивает вызов `animationLoop` именно такое число раз, чтобы гарантированно отобразить плавную и текущую анимацию. При этом не возникает проблем с функционированием остальных приложений.

Чтобы узнать подробнее о функции `requestAnimationFrame` и ее основном использовании для создания крутых анимаций, ознакомьтесь со статьей «Анимации в JavaScript» (https://www.kirupa.com/html5/animating_in_code_using_javascript.htm). В этом разделе также более глубоко освещена тема использования `requestAnimationFrame` и рассмотрены аспекты, не затронутые в текущей главе.

КОРОТКО О ГЛАВНОМ

Если вы считаете, что таймеры применяются в более узкой сфере, чем такие важные элементы, как инструкции `if...else` и циклы, пройденные нами ранее, то вы, скорее всего, правы. Вы можете создать множество прекрасных приложений, не прибегая к использованию `setTimeout`, `setInterval` или `requestAnimationFrame`. Тем не менее это не отменяет того, что знать их обязательно. Однажды наступит тот час, когда вам потребуется отложить выполнения кода, продолжительное выполнение цикла или создать приятную анимацию с помощью JavaScript. И когда этот момент наступит, вы будете готовы! Ну или по крайней мере будете знать, что гуглить.

Чтобы увидеть, как применяются эти функции-таймеры в рабочей среде, ознакомьтесь с приведенными ниже статьями и примерами, которые помогут лучше усвоить материал:

- Создание анимаций с помощью requestAnimationFrame (http://bit.ly/kirupaAnimationsJS).
- Создание симпатичного слайдера контента (http://bit.ly/sliderTutorial).
- Создание аналоговых часов (http://bit.ly/kirupaAnalogClock).
- Генератор припадка (http://bit.ly/kirupaSeizureGenerator). (Спойлер: анимация на этом сайте бешено мерцает.)

Я уже неоднократно отмечал, что JavaScript может вызвать ступор, в особенности когда дело доходит до таймеров. Если у вас возникнут сложности, я и другие опытные разработчики с радостью вам поможем. Обращайтесь на форум https://forum.kirupa.com, и мы выведем вас из ступора.

В ЭТОЙ ГЛАВЕ:

- разберем, что такое глобальная область;
- познакомимся с различными техниками использования локальной области;
- узнаем некоторые особенности, из-за которых код может вести себя непредсказуемо.

8

ОБЛАСТЬ ВИДИМОСТИ ПЕРЕМЕННЫХ

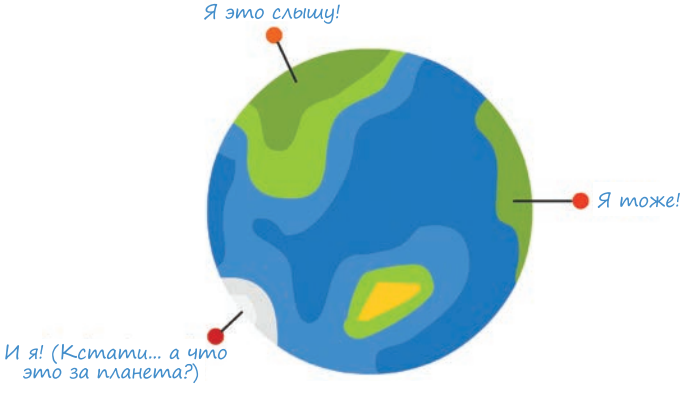
Давайте вспомним материал о переменных, пройденный несколько глав назад. Каждая объявленная нами переменная имеет конкретный уровень видимости, который определяет то, когда мы можем ее использовать. Выражаясь человеческим языком, это значит, что простое объявление переменной еще не дает нам возможности вызывать ее из любой части кода. Поэтому нужно понять несколько простых аспектов *области видимости переменных*.

В этом уроке я объясню области видимости переменных на примере распространенных случаев, большинство из которых мы уже встречали. Вообще, это достаточно обширная тема, но мы пробежимся по верхам. Время от времени эта тема будет всплывать в последующих уроках, так что мы сможем пополнить знания, полученные здесь.

Поехали!

Глобальная область видимости

Мы начнем с изучения самой обширной области, которая известна как *глобальная область видимости*. В реальной жизни, когда мы говорим, что что-то может быть услышано глобально, то имеем в виду, что услышим это «что-либо», находясь в любой точке мира.



В JavaScript все работает аналогично. Например, если мы говорим, что переменная доступна глобально, то имеем в виду, что у любого кода на нашей странице есть доступ для ее считывания или изменения. Если мы хотим что-либо определить глобально, то делаем это в коде, находящемся исключительно за пределами какой-либо функции.

Рассмотрим пример:

```
<!DOCTYPE html>
<html>

<head>
  <meta charset="utf-8">
  <title>Variable Scope</title>
</head>

<body>
  <script>
    let counter = 0;

    alert(counter);
  </script>
</body>

</html>
```

Здесь мы просто объявляем переменную `counter` и инициализируем ее как `0`. Поскольку эта переменная объявлена напрямую в теге `script` и не помещена в функцию, она считается *глобальной*. Это означает, что к этой переменной `counter` может обратиться любой код, находящийся в документе.

В следующем примере выделен этот аспект:

```
<!DOCTYPE html>
<html>

<head>
  <meta charset="utf-8">
  <title>Variable Scope</title>
</head>

<body>
  <script>
    let counter = 0;

function returnCount() {
  return counter;
}

    alert(returnCount());
  </script>
</body>

</html>
```

Мы видим, что переменная `counter` объявлена вне функции `returnCount`. Но, несмотря на это, у функции есть полный доступ к ней. При выполнении кода функция `alert` вызывает функцию `returnCount`, которая возвращает значение переменной `counter`.

Все это время мы использовали глобальные переменные, не придавая этому значения. В данной ситуации я всего лишь официально представил вам гостя, который тусуется на вашей вечеринке уже какое-то время.

Локальная область видимости

Становится еще интереснее, когда мы смотрим на элементы, не объявленные глобально. С этого момента понимание, что такое области

видимости, начинает приносить свои плоды. Как мы видели ранее, к переменной, объявленной глобально, внутри функции также есть доступ:

```
let counter = 0;

function returnCount() {
  return counter;
}
```

Но не наоборот. Переменная, объявленная внутри функции, не будет доступна извне:

```
function setState() {
  let state = "on";
}
setState();

alert(state) // undefined
```

В этом примере переменная `state` объявлена внутри функции `setState` и обращение к ней извне не работает. Причина в том, что область видимости переменной `state` является локальной относительно самой функции `setState`. В более общей форме это можно описать, сказав, что переменная `state` *локальна*.

ИСПОЛЬЗОВАНИЕ ПЕРЕМЕННЫХ БЕЗ ОБЪЯВЛЕНИЯ

Если мы инициализируем переменную `state`, не объявляя ее формально, область ее видимости будет определяться совсем иначе:

```
function setState() {
  state = "on";
}
setState(); alert(state) // "on"
```

Хотя в этом случае переменная `state` и появляется внутри функции `setState`, сам факт того, что мы объявили ее без `let`, `const` (или `var` — устаревший способ объявления переменных), делает ее глобальной. По большому счету, вам не понадобится объявлять переменные подобным образом. Поэтому всегда используйте `let` или `const`.

Особенности областей видимости

Поскольку сейчас мы разбираем JavaScript, было бы слишком просто останавливаться на достигнутом понимании областей видимости. В последующих разделах я обозначу некоторые важные особенности.

Области блоков

Наш код состоит из блоков. Множества блоков! И вообще, что такое блок? Блок — это набор инструкций JavaScript, почти всегда заключенный в фигурные скобки. Например, посмотрим на следующий код:

```
let safeToProceed = false;

function isItSafe() {
  if (safeToProceed) {
    alert("You shall pass!");
  } else {
    alert("You shall not pass!");
  }
}

isItSafe();
```

Если посчитать количество пар фигурных скобок, станет ясно, что здесь три блока. Один из них — это область, содержащаяся в самой функции `isItSafe`:

```
let safeToProceed = false;

function isItSafe()
  if (safeToProceed) {
    alert("You shall pass!");
  } else {
    alert("You shall not pass!");
  }
}

isItSafe();
```

Второй блок — это область инструкции `if`:

```
let safeToProceed = false;

function isItSafe() {
  if (safeToProceed) {
    alert("You shall pass!");
  } else {
    alert("You shall not pass!");
  }
}
```

Третий блок — это область, охваченная инструкцией `else`:

```
let safeToProceed = false;

function isItSafe() {
  if (safeToProceed) {
    alert("You shall pass!");
  } else {
    alert("You shall not pass!");
  }
}
```

Любая переменная, объявленная внутри блока через `let` или `const`, является локальной для этого блока и всех содержащихся в нем дочерних блоков. Чтобы лучше это понять, посмотрите на следующий код — вариацию функции `isItSafe`:

```
function isThePriceRight(cost) {
  let total = cost + 1;

  if (total > 3) {
    alert(total);
  } else {
    alert("Not enough!");
  }
}

isThePriceRight(4);
```

Мы объявляем переменную `total` как часть блока функции, затем обращаемся к ней внутри блока `if`. Как вы думаете, к чему это приведет? Переменная `total` полностью (ха-ха!) доступна отсюда, так как блок `if` является дочерним блоком блока функции. Выражаясь корректно, скажем, что переменная `total` находится *в области* функции `alert`.

А что насчет следующей ситуации?

```
function isThePriceRight(cost) {
  let total = cost + 1;

  if (total > 3) {
    let warning = true;
    alert(total);
  } else {
    alert("Not enough!");
  }

  alert(warning);
}

isThePriceRight(4);
```

У нас есть переменная `warning`, объявленная внутри блока `if`, а еще — функция `alert`, которая пытается вывести значение `warning`. В этом случае, так как мы пытаемся обратиться к переменной `warning` в блоке, находящемся вне блока, в котором она была объявлена, функция `alert` не будет отображать значение `true`. Учитывая то, где находится функция `alert`, переменная `warning` находится *вне области видимости*.

ОБЪЯВЛЕНИЕ ПЕРЕМЕННЫХ С КЛЮЧЕВЫМ СЛОВОМ VAR

Несколько абзацев выше я упомянул вскользь, что переменные когда-то объявлялись с помощью ключевого слова `var`. Ключевые слова `let` (и `const`) были введены позднее для облегчения объявления переменных. Если вы до сих пор используете `var`, пора переходить на `let`. Мы еще не обсуждали, почему `let` — более предпочтительный вариант, и решили, что обсудим это позже, когда начнем рассматривать области переменных подробнее. Что ж, настал этот момент!

Переменные, объявленные с `var`, распространяются на область функции. Они не распространяются на блоки, подобные инструкциям `if` и `else`. Если мы изменим последний пример, объявив в нем переменную `warning` с `var` вместо `let`, то код будет выглядеть так:

```
function isThePriceRight(cost) {
  let total = cost + 1;

  if (total > 3) {
    var warning = true;
    alert(total);
  } else {
    alert("Not enough!");
  }
```

```
  alert(warning);
}

isThePriceRight(4);
```

В изначальном варианте этого кода функция `alert` в отношении переменной `warning` ничего бы не отобразила, так как эта переменная, будучи объявленной с `let`, оказывалась вне области видимости. При использовании же `var` ситуация меняется и вы увидите отображение `true`. В этом и есть отличие между `var` и `let`. Область видимости переменных, объявленных с `var`, ограничивается уровнем функций, поэтому если переменная объявляется в любом месте внутри функции, то считается, что она находится в области видимости. Область же видимости переменных, объявленных с `let`, как мы увидели ранее, определяется уровнем блока.

Степень свободы, с которой переменная `var` определяет область видимости, слишком большая, и поэтому легко допустить ошибку, связанную с переменными. По этой причине я ратую за использование `let`, когда дело доходит до объявления переменных.

Как JavaScript обрабатывает переменные

Если вам показалось, что логика областей блока из предыдущего раздела выглядит странно, значит, вы еще не видели следующий пример. Взгляните на этот код:

```
let foo = "Hello!";
alert(foo);
```

Глядя на него, мы можем обоснованно утверждать, что отобразится значение `Hello!`, и в целом будем правы. А что, если переместить объявление и инициализацию переменной в конец?

```
alert(foo);
let foo = "Hello!";
```

В этом случае код выдаст ошибку. Доступ к переменной `foo` осуществляется без обращения к ней. А теперь давайте заменим `let` на `var`, получив следующее:

```
alert(foo);
var foo = "Hello!";
```

При выполнении этого варианта кода его поведение будет отличаться от предыдущего и вы увидите отображение `undefined`. Что конкретно здесь происходит?

Когда JavaScript при выполнении достигает определенной области (глобальной, функции и т. д.), то первое, что он делает, — полностью сканирует тело кода в поиске объявленных переменных. Если он встречает переменные, объявленные с `var`, то по умолчанию инициализирует их как `undefined`. Если же они объявлены с `let` или `const`, он оставляет их *полностью неинициализированными*. При завершении он перемещает все встреченные переменные в верхнюю часть соответствующей им области, которой в случае с `let` и `const` является ближайший блок, а в случае с `var` — ближайшая функция.

Давайте подробнее рассмотрим, что это значит. Изначально наш код выглядит так:

```
alert(foo);
let foo = "Hello!";
```

Когда JavaScript приступает к его обработке, код принимает следующий вид:

```
let foo;
alert(foo);
foo = "Hello!";
```

Несмотря на то что переменная `foo` была объявлена в нижней части кода, она смещается вверх. Формально это называется *поднятием переменной*. Особенность `let` и `const` в том, что при поднятии переменные остаются неинициализированными. Если вы попробуете обратиться к неинициализированной переменной, то код выдаст ошибку и прекратит выполнение. Если мы изменим предыдущий пример, используя `var`, то в итоге для JavaScript код будет выглядеть так:

```
var foo = undefined;
alert(foo);
foo = "Hello!";
```

Переменная по-прежнему поднимается, но при этом инициализируется как `undefined`, благодаря чему код продолжает выполнение.

Главная мысль из всего этого: *пожалуйста, объявляйте и инициализируйте переменные прежде, чем их использовать*. Несмотря на то что JavaScript способна в некоторой степени справиться с работой в тех

случаях, когда мы пренебрегаем данными действиями, это лишь добавит еще больше путаницы.

Замыкания

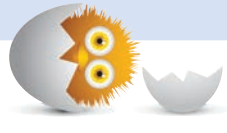
Ни одно обсуждение области переменных не будет полным, если не рассмотреть замыкания. Сейчас я не стану объяснять эту тему, приберегем ее для главы 9.

Прежде чем вы продолжите читать дальше, убедитесь, что весь пройденный материал вам понятен. Если у вас есть какие-то вопросы, обращайтесь на форум https://forum.kirupa.com, где я и другие разработчики с радостью помогут вам.

КОРОТКО О ГЛАВНОМ

Место размещения переменных в коде имеет огромное влияние на то, где они могут использоваться. Переменные, объявленные глобально, доступны по всему приложению. Переменные, объявленные локально, — только внутри области, в которой расположены. Внутри диапазона глобальных и локальных переменных у JavaScript есть огромное количество действий в запасе.

Эта глава представила обзор аспектов влияния области переменных на ваш код. В ближайшем будущем вам предстоит встретиться с некоторыми наиболее актуальными из этих аспектов.



В ЭТОЙ ГЛАВЕ:

- узнаем, что такое замыкания;
- свяжем воедино все полученные знания о функциях, переменных и области видимости.

9

ЗАМЫКАНИЯ

Вероятно, что к этому моменту вы уже знаете все о функциях и обо всех их забавных особенностях. Важная часть работы с функциями, JavaScript и (возможно) жизнью в целом — это понимание *замыканий*. На рис. 9.1 замыкание обозначено серой областью, в которой пересекаются области функций и переменных.

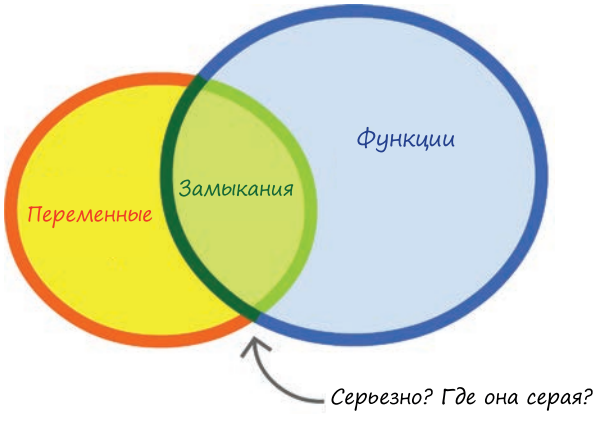


РИС. 9.1.
Замыкания

Больше ничего не буду говорить о замыканиях — лучше всего за них скажет сам код. Как бы я ни пытался сейчас их описать, это только еще больше вас запутает. В последующих разделах мы начнем со знакомого нам материала и будем постепенно продвигаться вглубь вражеской территории, населенной замыканиями.

Поехали!

Функции внутри функций

Первое, что мы сделаем, — проясним, что происходит, когда вы используете функции внутри функций, причем внутренняя функция возвращается. Для начала рассмотрим короткий пример.

Взгляните на этот код:

```
function calculateRectangleArea(length, width) {
  return length * width;
}

let roomArea = calculateRectangleArea(10, 10);
alert(roomArea);
```

Функция `calculateRectangleArea` получает два аргумента и возвращает результат их умножения туда, откуда пришел вызов. В данном примере роль *стороны, направившей вызов*, играет переменная `roomArea`.

После выполнения этого кода переменная `roomArea` содержит результат умножения 10 на 10, равный 100 (рис. 9.2).

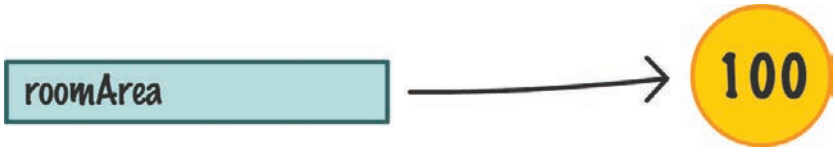


РИС. 9.2.
Результат roomArea

Как вы знаете, функция может вернуть практически что угодно. В данном случае мы вернули число. Вы так же легко можете вернуть текст (то есть *строку*), значение `undefined`, *пользовательский объект* и т. д.

До тех пор пока код, вызывающий функцию, знает, как поступить с возвращаемым ей результатом, вы можете делать практически все, что пожелаете. Вы даже можете вернуть другую функцию. На этом моменте остановимся поподробнее.

Ниже представлен простейший пример того, что я имею в виду:

```
function youSayGoodBye() {

  alert("Good Bye!");

  function andISayHello() {
    alert("Hello!");
  }

  return andISayHello;
}
```

У нас могут быть функции, содержащие в себе другие функции. В данном примере есть функция `youSayGoodbye`, которая содержит `alert` и функцию `andISayHello` (рис. 9.3).

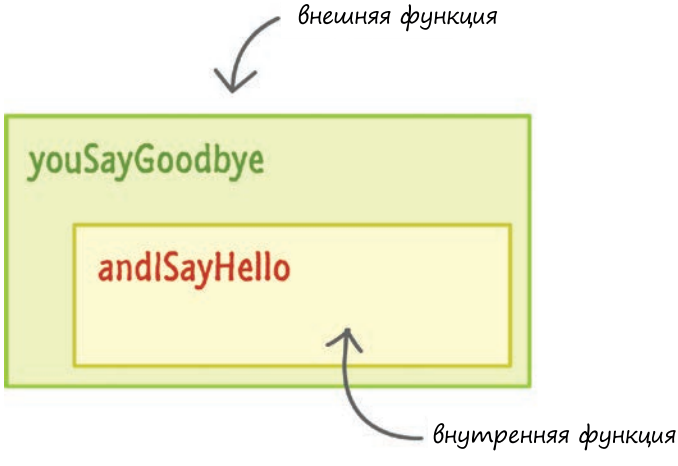


РИС. 9.3.
Функция внутри функции

В этом примере интересно то, что возвращает функция `youSayGoodbye`, когда ее вызывают. А возвращает она функцию `andISayHello`:

```
function youSayGoodBye() {

  alert("Good Bye!");

  function andISayHello() {
    alert("Hello!");
  }

  return andISayHello;
}
```

Попрактикуемся на таком примере. Для вызова функции инициализируем переменную, указывающую на `youSayGoodBye`:

```
let something = youSayGoodBye();
```

В момент выполнения этой строки кода будет также выполнен *весь код* внутри функции `youSayGoodBye`. Это значит, что вы увидите диалоговое окно (благодаря `alert`), говорящее нам `Good Bye!` (рис. 9.4).

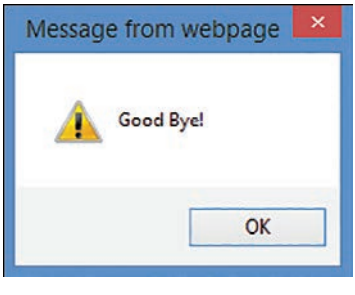


РИС. 9.4.
Диалоговое окно Good Bye!

Как часть процесса выполнения до завершения функция `andISayHello` будет также создана и затем возвращена. В этот момент наша переменная — `something`, способная обратиться только к одному элементу, а именно к функции `andISayHello` (рис. 9.5).

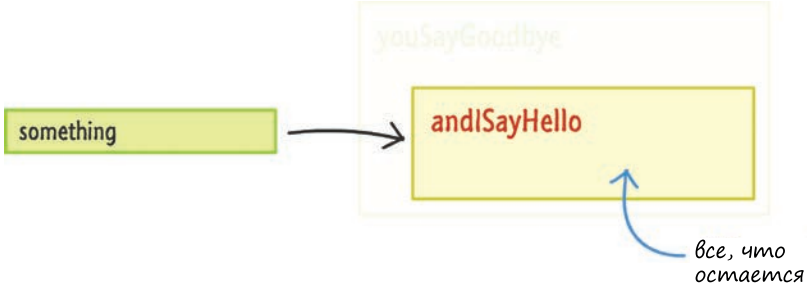


РИС. 9.5.
Переменная something и функция andISayHello

С точки зрения переменной `something` внешняя функция `youSayGoodBye` просто исчезает. Так как теперь переменная `something` указывает на функцию, вы можете активировать эту функцию, вызвав ее, как обычно, с помощью открывающих и закрывающих скобок:

```
let something = youSayGoodBye();
something();
```

Когда вы это сделаете, произойдет выполнение внутренней возвращенной функции (то есть `andISayHello`). Как и раньше, ждите появления диалогового окна, но теперь оно скажет `Hello!` (рис. 9.6), что определено в `alert` внутри этой функции.

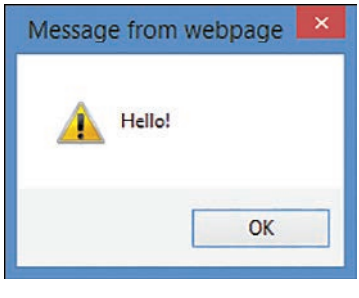


РИС. 9.6.
Hello!

Все это скорее похоже на обзор. Единственное новое, о чем вы могли узнать, — это то, что как только функция возвращает значение, она уходит из поля действия. Остается только возвращенное значение.

Хорошо. А теперь, как я и обещал, мы подобрались к границе вражеской территории. В следующем разделе дополним только что пройденный материал, рассмотрев схожий пример, но с некоторой особенностью.

Когда внутренние функции независимы

В предыдущем примере внутренняя функция `andISayHello` была самостоятельной и не опиралась ни на какие переменные или состояние внешней функции:

```
function youSayGoodBye() {

  alert("Good Bye!");

  function andISayHello() {
    alert("Hello!");
  }

  return andISayHello;
}
```

На практике мы будем сталкиваться с подобной ситуацией очень редко. Зачастую у нас будут переменные и данные, используемые совместно как внешней, так и внутренней функцией. Для наглядности посмотрим на такой пример:

```
function stopWatch() {
  let startTime = Date.now();

  function getDelay() {
    let elapsedTime = Date.now() – startTime;
    alert(elapsedTime);
  }

  return getDelay;
}
```

Здесь показан очень простой способ измерения времени, необходимого для какого-либо действия. Внутри функции `stopWatch` мы видим переменную, для которой установлено значение `Date.now()`:

```
function stopWatch() {
  let startTime = Date.now();

  function getDelay() {
    let elapsedTime = Date.now() – startTime;
    alert(elapsedTime);
  }

  return getDelay;
}
```

У нас также есть внутренняя функция `getDelay`:

```
function stopWatch() {
  let startTime = Date.now();

  function getDelay() {
    let elapsedTime = Date.now() – startTime;
    alert(elapsedTime);
  }

  return getDelay;
}
```

Функция `getDelay` отображает диалоговое окно, содержащее разницу во времени между новым вызовом `Date.now()` и ранее объявленной переменной `startTime`.

Что касается функции `stopWatch`, то последнее, что она делает перед завершением, — это возврат функции `getDelay`. Как мы можем увидеть, этот код очень похож на код из предыдущего примера. У нас есть внешняя и внутренняя функции, а также внешняя функция, возвращающая внутреннюю.

Теперь, чтобы увидеть `stopWatch` в действии, добавьте следующие строки кода:

```
let timer = stopWatch();

// Сделать что-нибудь за некоторое время.
for (let i = 0; i < 1000000; i++) {
  let foo = Math.random() * 10000;
}

// Вызвать возвращаемую функцию.
timer();
```

Полностью разметка и код выглядят так:

```
<!DOCTYPE html>
<html>

<head>
  <meta charset="utf-8">
  <title>Closures</title>

  <style>

  </style>
</head>

  <body>
    <script>
      function stopWatch() {
        var startTime = Date.now();

        function getDelay() {
          var elapsedTime = Date.now() – startTime;
          alert(elapsedTime);
        }

        return getDelay;
      }

      let timer = stopWatch();

      // Сделать что-нибудь за некоторое время.
      for (let i = 0; i < 1000000; i++) {
        let foo = Math.random() * 10000;
      }

      // Вызвать возвращаемую функцию.
    timer();
  </script>
</body>

</html>
```

Если вы запустите этот код, то увидите диалоговое окно, отображающее, сколько миллисекунд прошло между инициализацией переменной `timer`, выполнением цикла `for` до завершения и вызовом переменной `timer` в качестве функции (рис. 9.7).

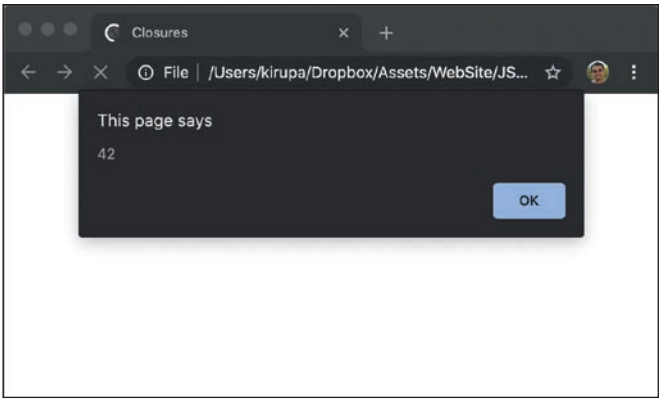


РИС. 9.7.
Переменная timer, вызванная в качестве функции

Если объяснить по-другому, то мы вызываем функцию `stopWatch`, затем выполняем длительную операцию и вызываем эту функцию повторно, чтобы узнать продолжительность этой длительной операции.

Теперь, когда мы видим, что наш пример работает, вернемся к функции `stopWatch` и посмотрим, что именно происходит. Как я уже отмечал чуть выше, многое из того, что мы видим, схоже с примером `youSayGoodBye` / `andISayHello`. Но есть одна особенность, которая привносит отличие в текущий пример, и важно обратить внимание на то, что происходит, когда функция `getDelay` возвращается в переменную `timer`.

На рис. 9.8 мы видим незавершенную визуализацию этого процесса:

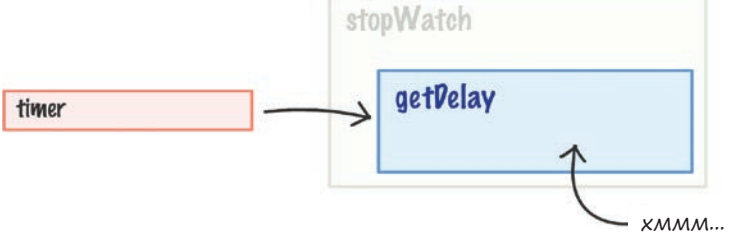


РИС. 9.8.
Внешняя функция stopWatch больше не действует, и переменная timer становится привязанной к функции getDelay

Внешняя функция `stopWatch` вышла из игры, и переменная `timer` стала привязанной к функции `getDelay`. А теперь укажем на эту особенность. Функция `getDelay` опирается на переменную `startTime`, существующую в контексте внешней функции `stopWatch`:

```
function stopWatch() {
  let startTime = Date.now();

  function getDelay() {
    let elapsedTime = Date.now() – startTime;
    alert(elapsedTime);
  }

  return getDelay;
}
```

Когда внешняя функция `stopWatch` перестает действовать и `getDelay` возвращается в переменную `timer`, что происходит на следующей строке?

```
function getDelay() {
  let elapsedTime = Date.now() – startTime;
  alert(elapsedTime);
}
```

В текущем контексте показалось бы логичным, если бы переменная `startTime` не была определена, верно? Но пример сработал, а значит, дело в чем-то еще, а именно в скромном и загадочном замыкании. Теперь остается пояснить, что должно произойти, чтобы переменная `startTime` сохранила значение, а не оставалась неопределенной.

Рабочая среда JavaScript, отслеживающая все переменные, использование памяти, ссылок и т. д., действительно умна. В нашем примере ею обнаружено, что внутренняя функция (`getDelay`) опирается на переменные из внешней функции (`stopWatch`). Когда это происходит, рабочая среда обеспечивает, чтобы любые нужные переменные из внешней функции были доступны внутренней функции, *даже если внешняя функция перестает действовать*.

Для наглядности посмотрим, как выглядит переменная `timer`, на рис. 9.9.

Она по-прежнему ссылается на функцию `getDelay`, но `getDelay` при этом также имеет доступ к переменной `startTime`, которая существовала во внешней функции `stopWatch`. Так как внутренняя функция *замкнула*

связанные с ней переменные внешней функции в своей области, мы называем ее *замыканием* (рис. 9.10).

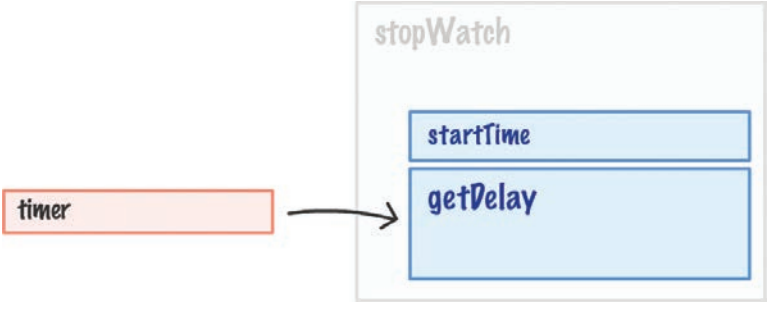


РИС. 9.9.
Переменная timer

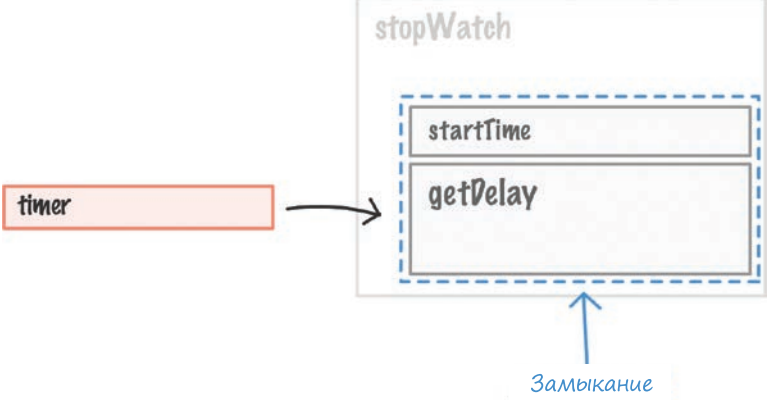


РИС. 9.10.
Схематичное изображение замыкания

Формально замыкание можно определить как вновь созданную функцию, которая также содержит свой переменный контекст (рис. 9.11).

В нашем примере это описано так: переменная `startTime` получает значение `Date.now` в момент инициализации переменной `timer` и начинает выполнение функции `stopWatch`. Затем функция `stopWatch` возвращает внутреннюю функцию `getDelay` и прекращает действие, оставляя при этом те из своих переменных, на которые опирается внутренняя функция. Внутренняя же функция, в свою очередь, замыкает эти переменные.

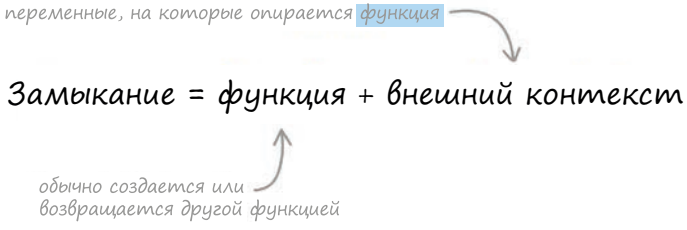


РИС. 9.11.
Более формальное определение замыкания

КОРОТКО О ГЛАВНОМ

Разбор замыканий на примерах позволил обойтись без множества скучных определений, теорий и жестикуляций. На самом деле замыкания — обычная для JavaScript тема. Вы будете иметь с ними дело в любых мудреных и менее сложных ситуациях.

Если из всего этого нужно было бы запомнить что-то одно, то вот оно: самое важное, что делают замыкания, — это позволяют функциям работать, даже когда их среда существенно изменяется или исчезает. Любые переменные, находившиеся в области при создании функции, замыкаются и защищаются, чтобы обеспечить продолжение работы функции. Подобное поведение очень важно для таких динамических языков, как JavaScript, где вам часто приходится создавать, изменять и уничтожать что-либо на ходу. Удачи!

В этой главе мы затронули много тем. Если у вас есть какие-либо вопросы касательно пройденного, пожалуйста, пишите мне на форуме **https://forum.kirupa.com**, и вы получите ответ в кратчайшие сроки.

В ЭТОЙ ГЛАВЕ:

- узнаем, где и как можно размещать код;
- разберем плюсы и минусы различных подходов.

10

ГДЕ МОЖНО РАЗМЕЩАТЬ КОД?

Ну что ж, немного отвлечемся от программирования по расписанию (ха!). Пока что весь написанный нами код в полном объеме «жил» внутри HTML-документа:

```
<!DOCTYPE html>
<html>

<head>
  <meta charset="utf-8">
  <title>An Interesting Title Goes Here</title>

  <style>
    body {
      background-color: #EEE;
    }

    h1 {
      font-family: sans-serif;
      font-size: 36px;
    }

    p {
      font-family: sans-serif;
    }
  </style>
</head>
```

```
<body>
  <h1>Are you ready for this?</h1>
  <p>Are you ready for seeing (or already having seen!) the most
  amazing dialog ever?</p>

  <script>
    alert("hello, world!");
  </script>
</body>

</html>
```

Попробуем отмотать немного назад и поразмыслить вот над чем: целесообразно ли использовать элементы HTML, CSS и JS в одном документе? Для простоты нашего толкования структуры документа попробуем прибегнуть к творческому подходу и изменить представление кода, например отобразив его в виде таких вот чудных рамок (рис. 10.1):

РИС. 10.1.
Наше представление веб-страницы

В примере ниже HTML-документ отделен от JavaScript лишь парой тегов `script`. Но на самом деле JavaScript-код не обязательно селить внутри HTML-документа. Его можно написать в отдельном файле (рис. 10.2).

При этом подходе мы больше не пишем JavaScript-код в HTML-документе. У нас по-прежнему есть тег `script`, но он *лишь указывает на файл JavaScript*, а не содержит все строки его кода.

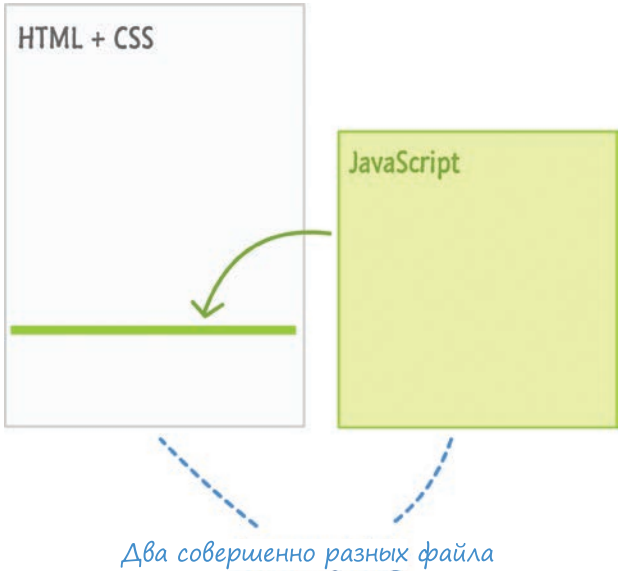


РИС. 10.2.
Теперь код JS живет в своем собственном файле!

Стоит заметить, что оба подхода не исключают друг друга, то есть мы можем совмещать их в документе HTML, применяя гибридный подход, при котором у нас будут как внешний файл JavaScript, так и строки JS-кода, содержащиеся внутри документа (рис. 10.3).

РИС. 10.3.
Смешанный подход, при котором код JS существует в нескольких местах

Что еще интереснее, так это то, что у каждого из этих подходов могут быть свои дополнительные вариации. Например, создание нескольких секций `script` в HTML-документе или нескольких JS-файлов и т. д. В последующих разделах мы рассмотрим подробнее оба подхода и обсудим, когда и какой из них лучше выбирать.

В конце у вас сложится представление обо всех плюсах и минусах, что позволит вам грамотно взаимодействовать с JavaScript при разработке веб-страниц и приложений.

Поехали!

Подход № 1: весь код в HTML-документе

Первым мы рассмотрим подход, который использовали все это время. Он предполагает, что весь код JS находится внутри тега `script` рядом с остальной частью HTML-документа:

```
<!DOCTYPE html>
<html>

<body>

  <h1>Example</h1>

  <script>
    function showDistance(speed, time) {
      alert(speed * time);
    }

    showDistance(10, 5);
    showDistance(85, 1.5);
    showDistance(12, 9);
    showDistance(42, 21);
  </script>
</body>

</html>
```

Когда ваш браузер загружает страницу, он поочередно считывает каждую строку кода в HTML-документе сверху вниз. Дойдя до тега

`script`, он также обрабатывает все строки JavaScript-кода. Закончив выполнение, он продолжает считывать оставшуюся часть документа. Это значит, что местоположение тега `script` очень важно. Мы обсудим этот момент чуть позже, при рассмотрении темы выполнения кода в подходящее время.

Подход № 2: код существует в отдельном файле

В этом случае наш основной HTML-документ не содержит никакого JS-кода. Вместо этого весь код располагается в отдельном документе. Такой подход разделен на две части. Первая относится к самому файлу JavaScript, а вторая отвечает за связь этого файла с HTML. Давайте рассмотрим обе эти части подробнее.

JS-файл

Основой описываемого подхода является отдельный файл, содержащий код JavaScript. Не имеет значения, как вы назовете этот файл, но его расширение, как правило, будет **.js**. Например, мой файл JS назван **example.js**.

Внутри этого файла будет размещен только JavaScript:

```
function showDistance(speed, time) {
  alert(speed * time);
}

showDistance(10, 5);
showDistance(85, 1.5);
showDistance(12, 9);
showDistance(42, 21);
```

Все, что мы обычно поместили бы в тег `script`, попадет сюда. Кроме этого, мы не будем ничего добавлять, так как появление в этом файле других элементов вроде произвольно выбранных частей HTML и CSS не допускается, а в ответ на подобные действия браузер заругается.

Ссылка на JavaScript-файл

Как только мы создали файл JS, второй (и заключительный) шаг — это связать его со страницей HTML. За это отвечает тег `script`, а конкретнее — его атрибут `src`, который указывает на расположение JavaScript-файла:

```
<!DOCTYPE html>
<html>

<body>
  <h1>Example</h1>

  <script src="example.js"></script>
</body>

</html>
```

В этом примере, если файл JavaScript расположен в одном каталоге с HTML, мы можем использовать относительный путь и сослаться напрямую на имя файла. А если этот файл находится в другом каталоге, нам нужно будет соответственно изменить путь:

```
<!DOCTYPE html>
<html>

<body>
  <h1>Example</h1>

  <script src="some/other/folder/example.js"></script>
</body>

</html>
```

В этом случае наш файл сценария вложен в три уровня каталогов с именами `some`, `other` и `folder`. Кроме того, мы также можем прибегнуть к использованию абсолютного пути вместо относительного:

```
<!DOCTYPE html>
<html>

<body>
  <h1>Example</h1>

  <script src="https://www.kirupa.com/js/example.js"></script>
</body>

</html>
```

Не важно, если пути относительные или абсолютные, — они будут прекрасно работать. В ситуациях же, когда путь между HTML-страницей и сценарием, на который мы ссылаемся, отличается (например, внутри шаблона, серверного включения, сторонней библиотеки и т. д.), безопаснее использовать абсолютный путь.

Итак, какой подход использовать?

У нас есть два основных подхода, определяющих, как и где следует размещать код (рис. 10.4).

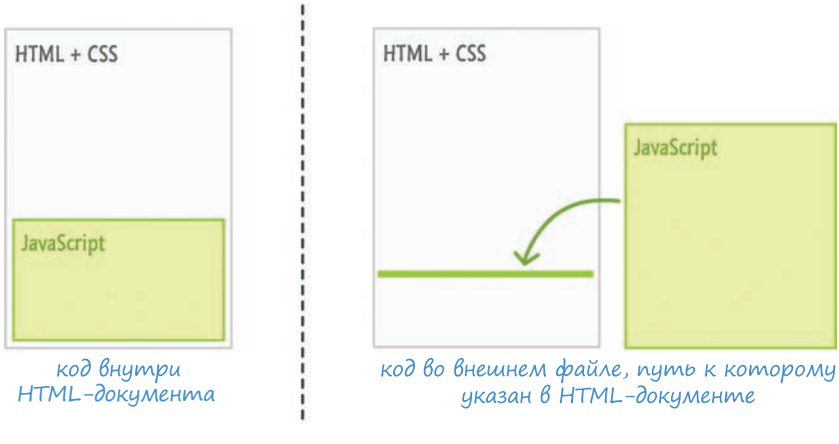


РИС. 10.4.
Два основных подхода, используемых для работы с кодом JavaScript

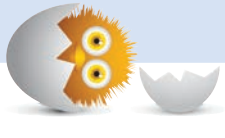
Конечный выбор подхода зависит от ответа на следующий вопрос: «Планируется ли использование одного и того же кода в нескольких HTML-документах?»

СЧИТЫВАНИЕ И РАСПОЛОЖЕНИЕ СЦЕНАРИЕВ В ДОКУМЕНТЕ

Несколькими разделами ранее я описывал выполнения сценариев. Браузер считывает HTML-страницу, начиная сверху и построчно перемещаясь вниз. Когда он достигает тега `script`, то начинает выполнять содержащийся в нем код. Это выполнение также происходит построчно, начинаясь с верхней части. Все остальные возможные действия страницы на время этого выполнения приостанавливаются. Если тег `script` ссылается на внешний JS-файл, то прежде, чем начать выполнение этого файла, браузер его загрузит.

При таком линейном считывании документа браузером наблюдается интересные нюансы, влияющие на то, где лучше размещать теги `script`. Технически их можно разместить в любой части HTML-документа. Тем не менее для этого есть предпочтительное место. Учитывая то, как браузер считывает страницу, и то, что он все блокирует во время выполнения сценариев, *следует размещать теги* `script` *в нижней части HTML-документа, после всех его элементов*.

Если тег `script` окажется в верхней части документа, то на время его выполнения браузер заблокирует все остальное. В результате если вы начнете загружать большой файл сценария или будете выполнять сценарий, то пользователь сможет увидеть частично загруженную и не отвечающую HTML страницу. До тех пор пока у вас не возникнет реальной потребности в выполнении JavaScript до полного считывания документа, размещайте теги `script` в конце, как показано в большинстве предыдущих примеров. Есть и еще одно преимущество в расположении скриптов внизу страницы, но о нем я расскажу значительно позднее, когда мы будем говорить о DOM (объектная модель документа) и о том, что происходит в процессе загрузки страницы.



Да, мой код будет использоваться в нескольких документах!

Если ответ *да*, тогда наверняка лучше будет поместить код в отдельный файл и сделать на него ссылку во всех HTML-страницах, которые должны его выполнять. Первая причина, по которой это стоит сделать, — вы сможете избежать повторения кода на всех этих страницах (рис. 10.5).

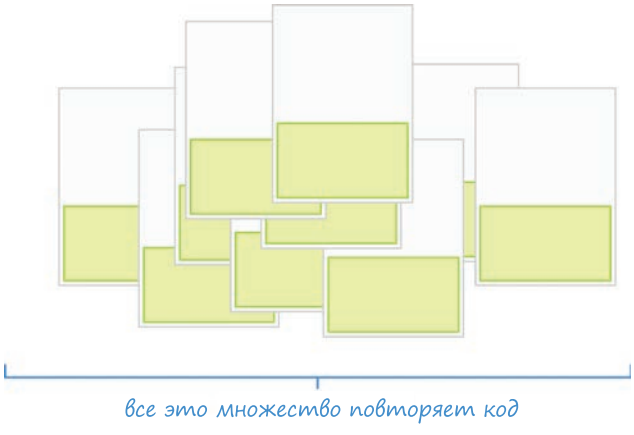


РИС. 10.5.
Повторение кода — это проблема!

Сопровождение повторяющегося кода — это кошмар, при котором изменение сценария потребует внесения изменений в каждый HTML-документ. Если вы используете некий шаблон или логику SSI, где ваш сценарий содержится только в одном HTML-фрагменте, тогда проблема сопровождения окажется не столь велика.

Вторая причина заключается в размере файла. Если вы дублируете сценарий во множестве HTML-страниц, то каждый раз, когда пользователь загружает одну из этих страниц, он заново загружает весь сценарий. Это не такая уж и проблема, когда речь идет о небольших сценариях, но, как только в коде появляется нескольких сотен строк, размер начинает существенно расти.

Если вы размещаете весь код в одном файле, то все только что упомянутые проблемы у вас уже не возникнут (рис. 10.6).

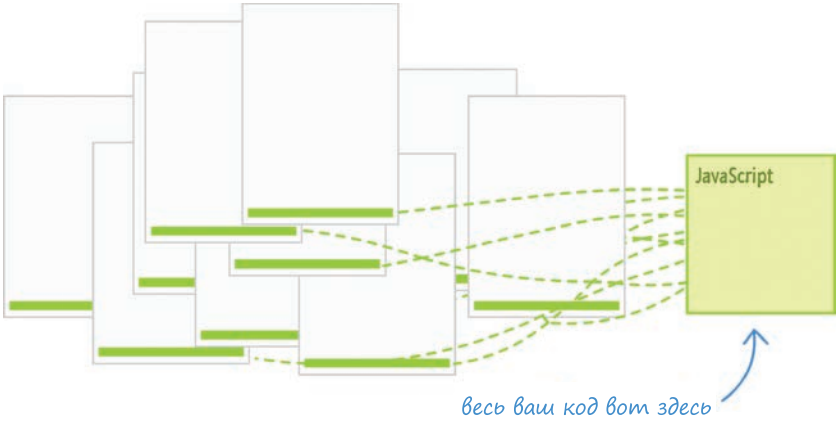


РИС. 10.6.
Весь код в одном месте

Сопровождение кода облегчается, так как его обновление производится только в одном файле. Любой HTML-документ, ссылающийся на этот JS-файл, при загрузке автоматически получает его последнюю версию. Это также позволит браузеру загружать код всего один раз и *предоставлять его сохраненную в кэше версию для последующих обращений*.

Нет, мой код используется только в одном HTML-документе

Если ваш ответ *нет*, то можете поступать по своему усмотрению. Вы по-прежнему можете разместить код в отдельном файле и сослаться на него из HTML-документа, но это принесет меньше выгоды, чем в случае с несколькими документами, который мы рассмотрели ранее.

В такой ситуации вполне подойдет размещение всего кода полностью в HTML-файле. В этом аспекте код из большинства примеров написан внутри HTML-документа. Примеры рассматриваемых кодов не предполагают их использования для множественных страниц и также большие размеры, при которых для повышения читаемости может потребоваться размещение кода в отдельном файле.

КОРОТКО О ГЛАВНОМ

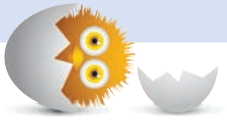
Теперь вы увидели, что решение даже такого простого вопроса, как размещение кода, требует многостраничных объяснений. Добро пожаловать в мир HTML и JavaScript, где нет четкого деления на то, что такое хорошо и что такое плохо. Возвращаясь к сути этого урока, стоит отметить, что типичный HTML-документ будет содержать много файлов сценариев, загружаемых из внешних источников. Некоторые из этих файлов будут вашими, некоторые же будут взяты из сторонних ресурсов.

Напомню и о смешанном подходе, когда в вашем HTML-документе содержится не только ссылка на отдельный JS-файл, но и сам JS-код. Такой подход тоже достаточно распространен. В конце концов, только вам решать, какой именно подход использовать. Надеюсь, что эта глава предоставила вам достаточно информации, чтобы вы могли выбирать подходы осознанно. В главе 35 «События загрузки страницы и прочее» мы еще больше углубимся в пройденную в этой главе тему, рассмотрев на примерах связанные с загрузкой страниц и некоторые особые атрибуты, которые привносят в этот процесс свои сложности. Но пока не будем о них думать.

Если у вас появились вопросы, уверенно задавайте их на форуме **https://forum.kirupa.com**. Я и другие разработчики будем рады вам помочь!

В ЭТОЙ ГЛАВЕ:

- изучим альтернативы alert для отображения результатов;
- поймем, как работает консоль;
- изучим разные удобные варианты журналирования.



11

ВЫВОД СООБЩЕНИЙ В КОНСОЛЬ

При написании кода возможны две ситуации. В первой вас будет терзать любопытство: а запустится ли только что написанный вами код? Во второй вы будете знать, что код выполняется, но при этом выполняется некорректно. Что-то не так... *Но где же...*

В обеих ситуациях вам понадобится дополнительное ви́дение действий кода. Неизменным подходом для получения такого ви́дения является использование метода `alert`:

```
let myButton = document.querySelector("#myButton");
myButton.addEventListener("click", doSomething, false);

function doSomething(e) {
  alert("Is this working?");
}
```

Использование метода `alert`, в общем-то, неплохой вариант, и он прекрасно работает в простых ситуациях. Но когда код усложняется, такой подход теряет свою эффективность. Как новичка вас, скорее всего, взбесят постоянные закрытия огромного количества диалоговых окон, появляющихся в процессе выполнения кода. Кроме того, потребуется удобный способ сохранять получаемые сообщения, а динамичная смена диалоговых окон `alert` изрядно затрудняет любые попытки журналирования.

В текущем уроке мы рассмотрим одно из величайших изобретений, помогающее нам лучше понять, что именно делает наш код. Если говорить точнее, мы приступаем к изучению *консоли*.

Поехали!

Знакомство с консолью

Даже если вы считаете, что пишете идеальный JS-код, то все равно будете проводить много времени в так называемой *консоли*. Если ранее вам не приходилось ею пользоваться, поясню, что она относится к инструментам разработчика и служит для вывода на экран текста и прочих данных, с которыми (иногда) можно взаимодействовать.

Выглядит она примерно так, как на рис. 11.1.

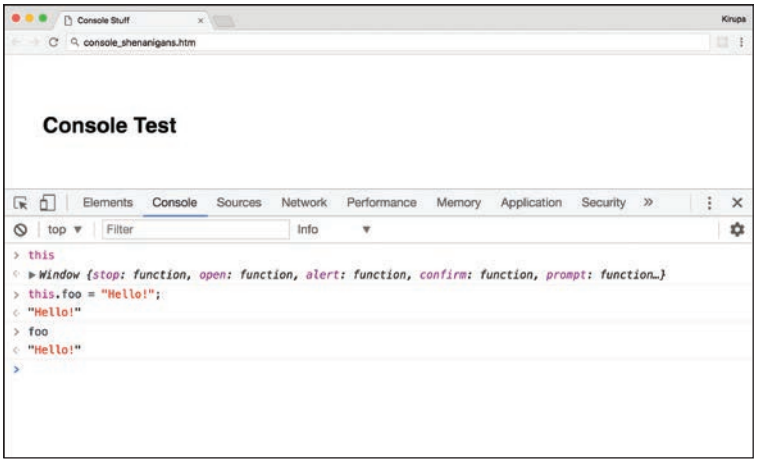


РИС. 11.1.
Знакомство с консолью

Консоль помогает решать многие задачи на очень высоком уровне:

- читать сообщения, задав в коде их журналирование и вывод на экран;
- модифицировать состояние приложения, устанавливая (или переписывая) переменные и значения;
- просматривать значение элемента DOM, используемый стиль или доступный код, находящийся в области видимости;
- использовать ее как виртуальный редактор кода и записывать/выполнять определенный код просто ради интереса.

В этом разделе мы не будем заострять внимание на всех возможностях консоли. Вместо этого плавно, шаг за шагом, научимся использовать ее для простого отображения сообщений. Остальные невероятные возможности консоли мы успеем рассмотреть позже, поэтому не переживайте.

Отображение консоли

Первое, что мы сделаем, — запустим саму консоль. Она относится к браузерным инструментам разработки, доступ к которым вы можете получить либо через меню браузера, либо с помощью горячих клавиш. Находясь в браузере, нажмите комбинацию Ctrl + Shift + I, если работаете в Windows, или Cmd + Alt + I, работая на Mac.

В зависимости от платформы браузера каждый из инструментов разработки будет выглядеть несколько иначе. Нам же важно найти вкладку консоли и открыть ее.

На рис. 11.2. показана консоль Chrome.

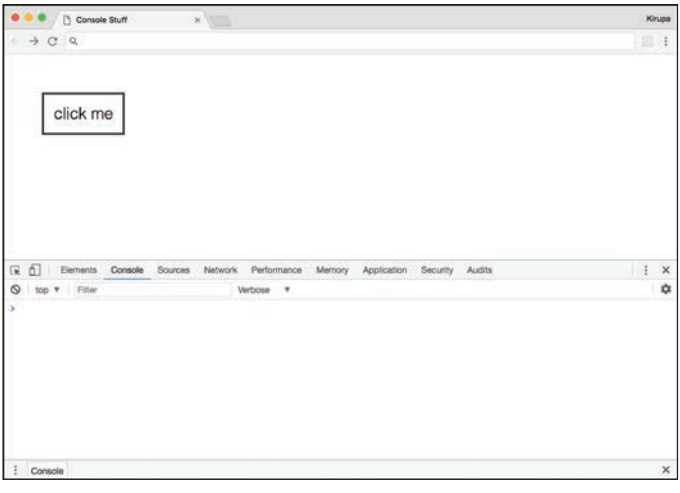


РИС. 11.2.
Консоль Chrome

В Safari консоль будет выглядеть, как на рис. 11.3.

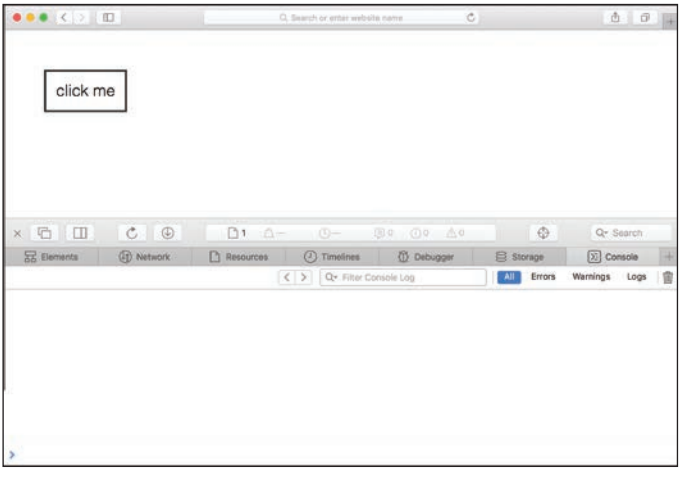


РИС. 11.3.
Консоль Safari

Консоль Firefox изображена на рис. 11.4.

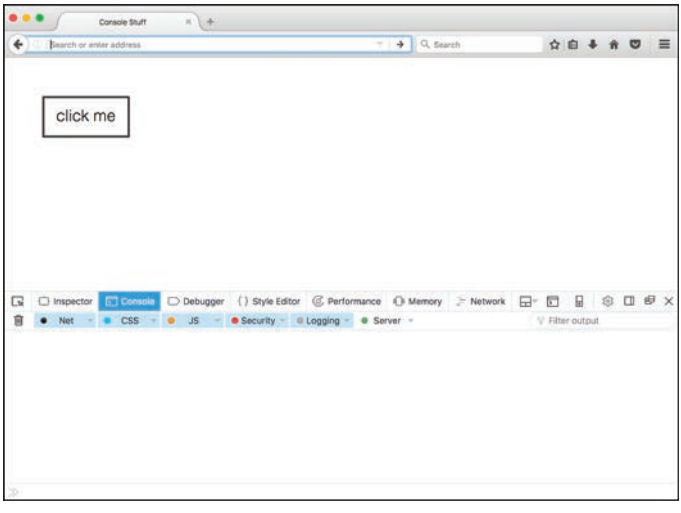


РИС. 11.4.
Консоль Firefox

Консоль Microsoft Edge показана на рис. 11.5.

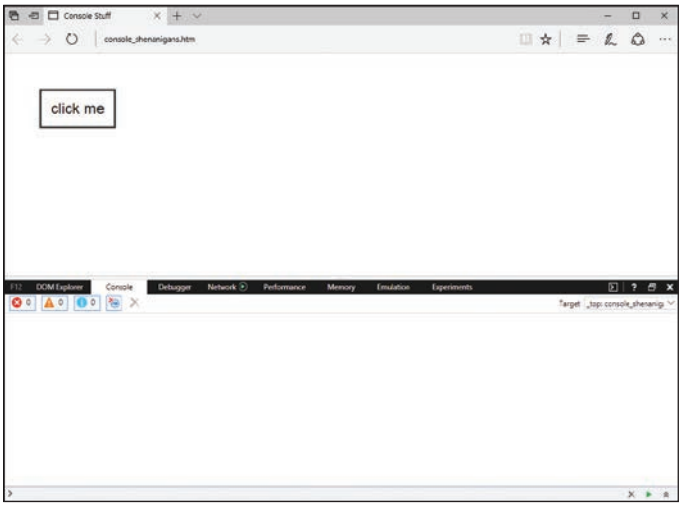


РИС. 11.5.
Консоль Edge

Хочу отметить, что не имеет значения, какой браузер вы используете. Консоль выглядит и работает почти одинаково в любом из них. Просто вызовите ее в своем любимом браузере и приготовьтесь использовать, пока читаете следующие разделы.

Для тех, кому важны детали

Сейчас вы можете просто прочесть несколько следующих разделов и изучить целую кучу информации, и пальцем не пошевельнув. Если такой вариант вам по душе, тогда пропустите этот раздел и переходите к следующему.

Но если вы хотите немного потренироваться и увидеть некоторые из особенностей консоли, создайте HTML-документ и добавьте в него следующий код HTML, CSS и JavaScript:

```
<!DOCTYPE html>
<html>

<head>
  <title>Console Stuff</title>

  <style> #container {
    padding: 50px;
  }

  #myButton {
    font-family: sans-serif;
    font-size: 24px;
    font-weight: lighter;
    background-color: #FFF;
    border: 3px #333 solid;
    padding: 15px;
  }

  #myButton:hover {
   background-color: aliceblue;
  }
  </style>
</head>

<body>
  <div id="container">
    <button id="myButton">click me</button>
  </div>
```

```
  <script>

    let myButton = document.querySelector("#myButton");
    myButton.addEventListener("click", doSomething, false);

    function doSomething(e) {
      alert("Is this working?");
    }
  </script>
</body>

</html>
```

Перед нами очень простая HTML-страница с кнопкой, по которой вы можете кликнуть. Когда вы это сделаете, появится диалоговое окно (такое же было описано ранее). Далее мы изменим этот пример, чтобы опробовать некоторые из возможностей консоли.

Журналирование в консоли

Первое, что мы сделаем, — прикажем нашей консоли вывести информацию на экран. Это аналогично тому, что мы ранее делали с помощью инструкции `alert`, и осуществляется почти так же легко. Ключом же в данном случае выступает *API консоли*. Этот протокол содержит множество свойств и методов, предоставляющих различные способы вывода данных на экран. Основным и, вероятно, наиболее популярным является метод `log`.

Знакомство с методом log

В самом базовом варианте использование этого метода выглядит так:

```
console.log("Look, ma! I'm logging stuff.")
```

Вы вызываете его через объект `console` и передаете ему текст, который хотите отобразить. Чтобы увидеть этот процесс в действии, можно заменить `alert` в нашем примере на следующее:

```
function doSomething(e) {
  console.log("Is this working?");
}
```

После запуска кода щелкните по кнопке click me и загляните в консоль. Если все сработало как надо, вы увидите в ней текст «Is this working?» («Работает?»), как показано на рис. 11.6:

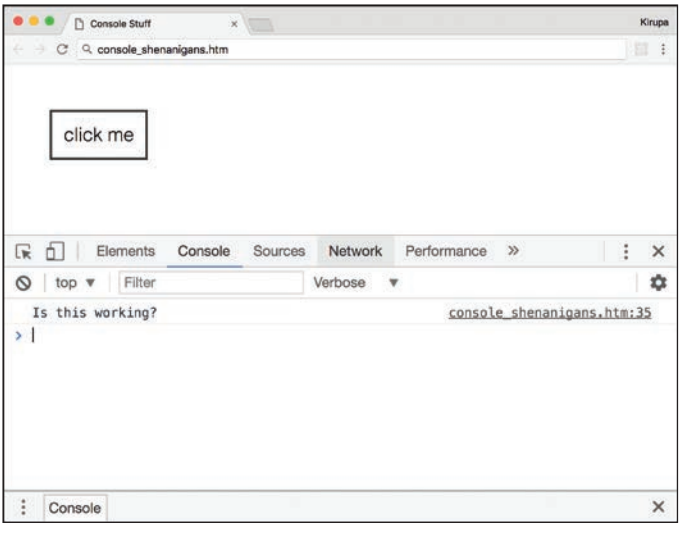


РИС. 11.6.
Кнопка click me выведена на экран

Если вы продолжите щелкать по кнопке, то увидите, что количество экземпляров «Is this working?» увеличивается, как показано на рис. 11.7.

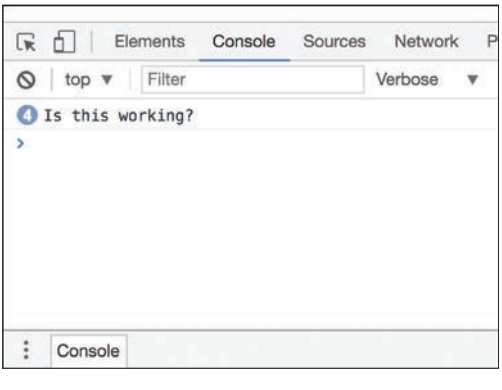


РИС. 11.7.
Каждое нажатие на кнопку отображается в консоли

То, как это выглядит, будет зависеть от используемых инструментов разработки. Вы можете просто увидеть счетчик слева от сообщения, который будет увеличиваться, как показано на скриншоте. Вы также увидите повторение текста «Is this working?» в каждой строке. Не переживайте, если то, что вы видите в своей консоли, немного отличается от скриншотов. Важно то, что ваш вызов `console.log` работает и журналирует сообщения, которые вы видите в консоли. Более того, эти сообщения не только для чтения. Вы можете их выбирать, копировать и даже распечатать и повесить в рамочке на стене.

Предопределенный текст — не предел

Теперь, когда вы познакомились с основами, углубимся немного в тему. При использовании консоли ваши возможности не ограничены выводом только предопределенного текста. Например, распространенный случай — это вывод на экран значения чего-либо существующего только в качестве вычисления выражения или обращения к значению. Чтобы наглядно увидеть, что я имею в виду, внесите следующие изменения в вашу функцию `doSomething`:

```
function doSomething(e) {
  console.log("We clicked on: " + e.target.id);
}
```

Тем самым мы даем команду консоли отобразить текст «We clicked on» в дополнение к значению `id` элемента, по которому мы щелкнули. Если вы выполните предпросмотр внесенных изменений в браузере и щелкните по кнопке click me, то в консоли увидите результат, соответствующий рис. 11.8.

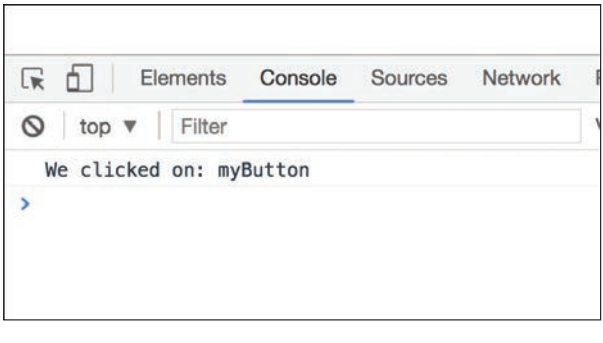


РИС. 11.8.
Отображен id кнопки, по которой мы кликнули

Значение `id` кнопки отображается в дополнение к предопределенному тексту. Конечно, вывод значения `id` элемента не самая потрясающая идея, но на деле вы вольны выводить практически все, что может быть представлено в виде текста. Так что это мощная возможность!

Отображение предупреждений и ошибок

Настало время рассмотреть метод `log`. Объект `console` предоставляет в наше распоряжение методы `warn` и `error`, которые позволяют выводить сообщения в формате предупреждений и ошибок, как это показано на рис. 11.9 соответственно.

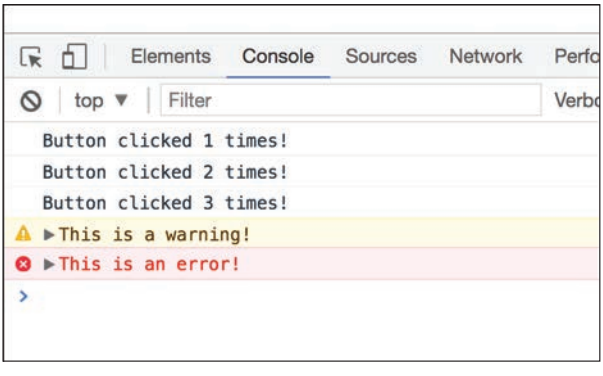


РИС. 11.9.
Мы можем делать предупреждения и указывать на ошибки… как босс!

Способ использования этих двух методов отличается от того, как вы использовали метод `log`. Просто передайте в них то, что нужно вывести на экран. В следующем листинге приведен пример их использования:

```
let counter = 0;

function doSomething(e) {
  counter++;

  console.log("Button clicked " + counter + " times!");

  if (counter == 3) {
    showMore();
  }
}
```

```
function showMore() {
  console.warn("This is a warning!");
  console.error("This is an error!");
}
```

Когда этот код выполняется и известная нам кнопка нажимается трижды, происходит вызов функции `showMore`. В этой функции мы расположили только сообщения консоли о предупреждении и об ошибке:

```
function showMore() {
  console.warn("This is a warning!");
  console.error("This is an error!");
}
```

Ошибки и предупреждения предоставляют еще одну крутую возможность, выходящую за пределы простого отображения и очень отличающую их от скучного аналога `log`. Вы можете развернуть их в консоли и просмотреть полную трассировку стека функций, выполненных кодом, до момента встречи с ними (рис. 11.10).

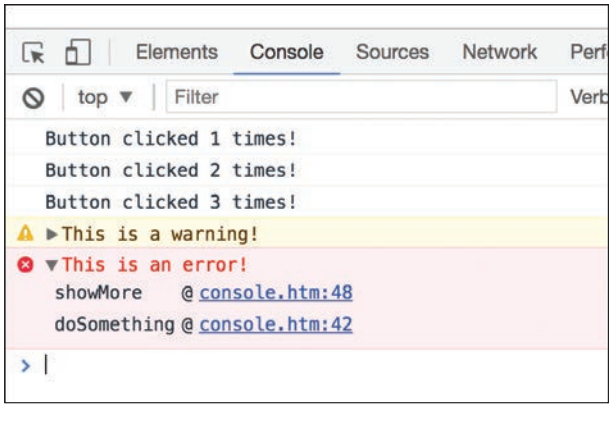


РИС. 11.10.
Просмотр деталей ошибок

Это полезно при работе с большими частями кода, где есть множество ответвлений. Методы `warn` и `error` являются прекрасным способом понять те извилистые пути, которые прошел код, чтобы оказаться в том или ином конечном состоянии.

КОРОТКО О ГЛАВНОМ

Консоль предоставляет вам один из лучших инструментов для понимания действий кода. Отображение сообщений — это только часть предлагаемых консолью возможностей. Помимо рассмотрения одного лишь примера с выводом сообщений нам предстоит освоить еще больше информации. Мы также узнаем о других возможностях консоли, но даже эти несколько техник, изученных к настоящему моменту, позволят вам значительно продвинуться при поиске и устранении багов.

Если у вас появились любые вопросы по этой теме, задавайте их на форуме **https://forum.kirupa.com**. Мы с остальными разработчиками с удовольствием поможем их решить!

В ЭТОЙ ГЛАВЕ:

- разберем суть и особенности объектов;
- познакомимся с базовыми типами в JavaScript;
- выясним, что пицца обладает не только прекрасным вкусом, но и образовательной ценностью.

12

О ПИЦЦЕ, ТИПАХ, ПРИМИТИВАХ И ОБЪЕКТАХ

Пора заняться серьезными делами. Суперсерьезными! В последних нескольких главах мы изучили разные значения, в том числе: строки (текст), числа, логические значения (`true` и `false`), функции и другие встроенные элементы JavaScript.

Вот некоторые примеры, чтобы освежить память:

```
let someText = "hello, world!";
let count = 50;
let isActive = true;
```

В отличие от других языков, JavaScript упрощает определение и использование этих встроенных элементов. Нам даже не требуется составлять план их будущего использования. Но несмотря на всю простоту, существует множество скрытых деталей. И их знание важно, так как не только облегчает понимание кода, но и ускоряет выявление причин его неисправностей.

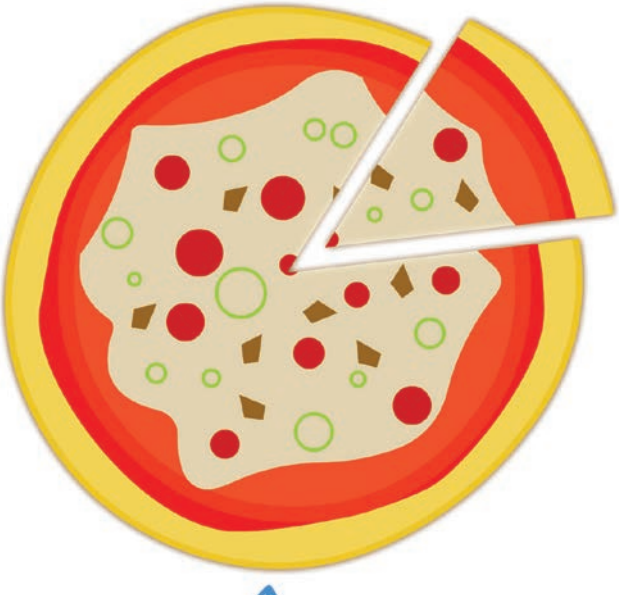
Как вы могли предположить, *встроенные элементы* — не самый удачный способ описания различных значений, используемых в JS. Существует более официальное имя для таких значений, а именно *типы*. В этой главе мы начнем плавное знакомство с их сутью и назначением.

Поехали!

Сначала поговорим о пицце

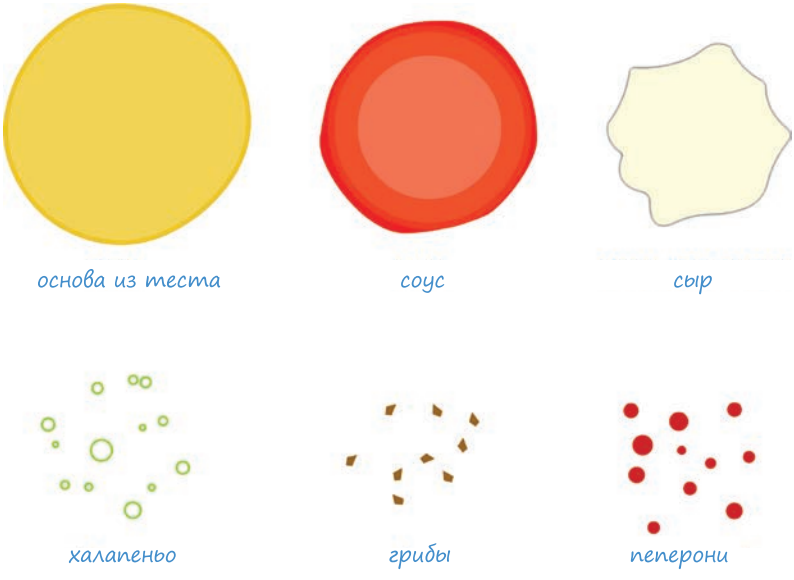
Поскольку я постоянно что-нибудь ем (или думаю, что бы съесть), то постараюсь объяснить загадочный мир типов на более простом примере — мире пиццы.

Если вы давненько ее не ели, то напомню, как она выглядит:



Если ваша пицца выглядит по-другому, верните ее™

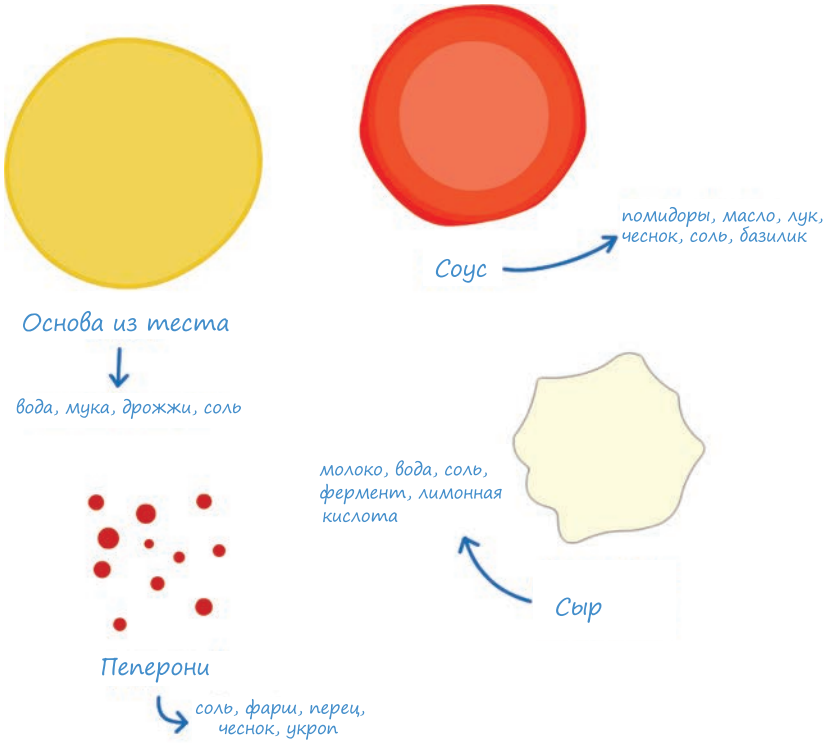
Конечно же, пицца не появляется в таком виде из ниоткуда. Она создается из простых и сложных ингредиентов:



Простые ингредиенты легко выявить. Это *грибы* и *халапеньо*. Причина, по которой мы называет их простыми, в том, что их нельзя разложить на составные части:

Они не изготавливаются и не составляются из других компонентов.

К сложным же ингредиентам относятся *сыр*, *соус*, *основа из теста* и *пеперони*. Сложными их делает то, что они сделаны из *других ингредиентов*:



К сожалению, такие ингредиенты, как сыр и пеперони, не бывают простыми. Для их приготовления нужно смешивать, жарить и добавлять различные компоненты. Кроме того, их получение не ограничивается смешиванием простых ингредиентов, но может также требовать совмещения сложных.

От пиццы к JavaScript

Все, что мы узнали о пицце в предыдущем разделе, было неспроста. Описание простых и сложных ингредиентов вполне применимо к типам

в JavaScript. Каждый отдельно взятый ингредиент можно рассматривать как аналог типа, который вы можете использовать (рис. 12.1).

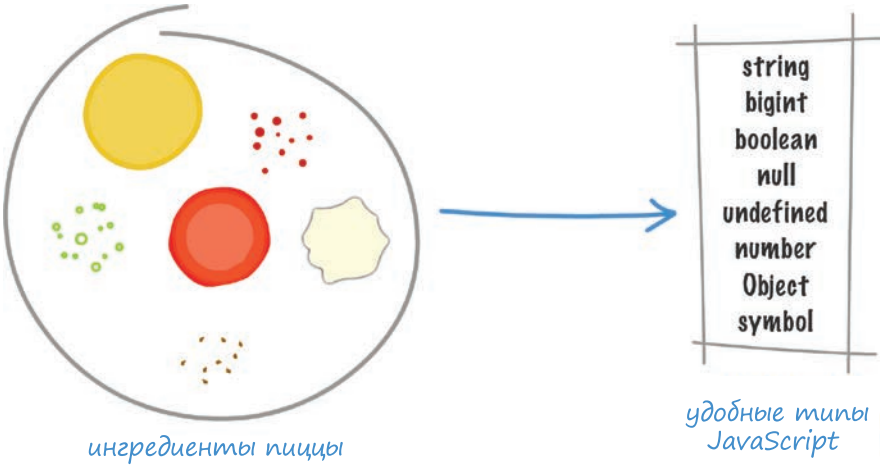


РИС. 12.1.
Список простых типов JavaScript

Подобно сыру, соусу, пеперони, грибам и бекону в нашей пицце, типами в JavaScript являются `string` (строка), `number` (число), `boolean` (логическое значение), `null` (пустой), `undefined` (не определен), `bigint` (целочисленные значения), `symbol` (символы) и `Object` (объект). С некоторыми из этих типов вы уже можете быть знакомы, с некоторыми — нет. Подробнее мы будем рассматривать их в дальнейшем, сейчас же в табл. 12.1 вы можете посмотреть краткое описание их назначения.

Как мы видим, каждый тип имеет свое уникальное назначение. При этом они, аналогично ингредиентам пиццы, также подразделяются на простые и сложные. Только в терминологии JavaScript простые и сложные типы называются *примитивами* (примитивные типы) и *объектами* (объектные типы) соответственно.

К примитивным типам относятся `string`, `number`, `boolean`, `null`, `bigint`, `symbol` и `undefined`. Любые значения, попадающие в их юрисдикцию, не подлежат делению на части. Они являются халапеньо и грибами в мире JavaScript. Примитивы достаточно легко определять и оформлять в понятные элементы. В них нет глубины, и при встрече с ними мы, как правило, получаем то, что видим изначально.

ТАБЛ. 12.1. Типы

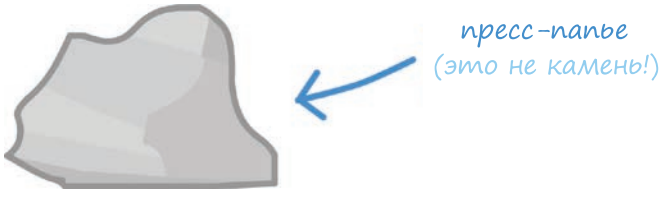
Тип	Назначение
`string`	Основная структура для работы с текстом
`number`	Позволяет работать с числами
`boolean`	Используется там, где нужно получить `true` или `false`
`null`	Предстает цифровым эквивалентом ничего
`undefined`	Похожий по смыслу на `null`. Возвращается, когда значение подразумевается, но на деле отсутствует. Например, если вы объявляете переменную, но ничего ей не присваиваете
`bigint`	Позволяет работать с крайне большими и малыми числами, выходящими за пределы возможностей обычного типа `number`
`symbol`	Нечто уникальное и неизменяемое, то, что при желании можно использовать как идентификатор свойств объекта
`Object`	Выступает в роли оболочки для других типов, включая другие объекты

Объектные же типы, представленные как `Object` в вышеприведенной таблице, оказываются более загадочными. Поэтому, прежде чем перейти к описанию деталей всех перечисленных типов, стоит отдельно рассмотреть, чем именно являются объекты.

Что такое объект?

Принцип объектов в таких языках программирования, как JavaScript, прекрасно отражает их аналогию из реальной жизни, в которой мы все буквально окружены объектами. К ним относятся ваш компьютер, книга на полке, картошка (спорно), будильник, плакат, заказанный на eBay, и т. д. Продолжать можно бесконечно.

Некоторые объекты вроде пресс-папье малофункциональны и могут долго бездействовать.



Другие объекты, вроде телевизора, уже выходят за рамки простого существования и выполняют множество задач:

Обычный телевизор получает сигнал, позволяет вам включать его и выключать, щелкать каналы, регулировать громкость и прочее.

Здесь важно понять, что объекты имеют разную форму, размер и назначение. Несмотря на эти отличия, на верхнем уровне они все одинаковы и представляют собой *абстракцию*. Они дают возможность пользоваться ими, не задаваясь вопросом об их внутреннем устройстве. Даже простейшие объекты скрывают в себе определенный уровень сложности, о котором можно не париться.

Например, не важно, что именно происходит внутри телека, как спаяны провода или какой клей использовался для соединения деталей. Все это не имеет значения. Все, что вас интересует, так это чтобы телевизор выполнял свое предназначение. Он должен исправно переключать каналы, позволять регулировать громкость и пр. Остальное — лишние заморочки.

В принципе, объект можно рассматривать как черный ящик. Существует ряд предопределенных/описанных действий, которые он совершает. Увидеть же, как он это делает, достаточно непросто. На деле вас это и не интересует до тех пор, пока он делает все как надо. Мы изменим свое представление об этом позже, когда поучимся создавать внутренности объекта, а пока насладимся простотой этого мира.

Предопределенные объекты в JavaScript

Помимо встроенных типов, перечисленных ранее, в JS также изначально присутствуют предопределенные объекты. Эти объекты позволяют работать с чем угодно, включая наборы данных, даты, текст и числа. В табл. 12.2 приводится аналогичный предыдущему список, описывающий их назначения:

ТАБЛ. 12.2. Объекты

Тип	Назначение
`Array`	Помогает хранить, извлекать и манипулировать наборами данных
`Boolean`	Служит оболочкой примитива `Boolean`, а также работает посредством значений `true` и `false`
`Date`	Упрощает работу с датами и их представление
`Function`	Позволяет вызывать заданный код
`Math`	Умник среди типов, расширяющий возможности работы с числами
`Number`	Служит оболочкой примитива `number`
`RegExp`	Предоставляет богатые возможности сопоставления текстовых шаблонов
`String`	Служит оболочкой примитива `string`

Использование встроенных объектов несколько отличается от использования примитивов. Каждый объект в этом плане по-своему особенный. Подробное пояснение всех этих особенностей использования я отложу на потом, а здесь приведу короткий фрагмент кода с комментарием, который покажет возможные варианты:

```
// массив
let names = ["Jerry", "Elaine", "George", "Kramer"];
let alsoNames = new Array("Dennis", "Frank", "Dee", "Mac");

// округленное число
let roundNumber = Math.round("3.14");

// текущая дата
let today = new Date();

// объект boolean
let booleanObject = new Boolean(true);
```

```
// бесконечность
let unquantifiablyBigNumber = Number.POSITIVE_INFINITY;

// объект string
let hello = new String("Hello!");
```

Вас может несколько озадачить то, что примитивы `string`, `boolean`, `symbol`, `bigint` и `number` могут существовать и в форме объектов. Внешне эта объектная форма выглядит очень похожей на примитивную. Вот пример:

```
let movie = "Pulp Fiction";
let movieObj = new String("Pulp Fiction");

console.log(movie);
console.log(movieObj);
```

При выводе обоих вариантах вы увидите одинаковый результат. Тем не менее внутренне `movie` и `movieObj` весьма различны. Первый буквально является примитивом типа `string`, а второй имеет тип `Object`. Это ведет к интересному (а иногда и непонятному) поведению, о котором я постепенно расскажу в процессе изучения встроенных типов.

КОРОТКО О ГЛАВНОМ

Если вам кажется, что все оборвалось на самом интересном месте, то это вполне нормально. Главный вывод здесь в том, что примитивы составляют большинство основных типов, которые вы будете использовать. Объекты несколько сложнее и состоят из примитивов или других объектов. Мы узнаем обо всем этом больше, когда начнем углубляться в тему. Помимо прочего, мы также узнали имена встроенных типов и некоторые присущие им особенности.

В последующих главах мы глубже изучим все эти типы, а также связанные с их использованием нюансы. Рассматривайте эту главу как плавный разгон, после которого вы резко влетите на рельсы безумных американских горок.

В ЭТОЙ ГЛАВЕ:

- используем массивы для обработки списков данных;
- научимся выполнять распространенные задачи с помощью различных свойств массива.

13

МАССИВЫ

Давайте представим, что вы хотите составить список на листке бумаги. Назовем его *продукты*. Теперь запишите в нем пронумерованный список, начинающийся с нуля, и перечислите все, что вам нужно (рис. 13.1).

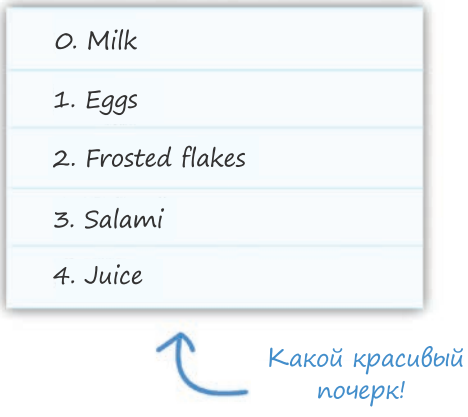


РИС. 13.1.
Список продуктов

Написав простой список, вы получили пример массива из реальной жизни. Листок бумаги, проименованный как *продукты*, это и есть ваш массив. Предметы же, которые вы хотите купить, — это значения массива.

В этом уроке вы не только узнаете, какие продукты я предпочитаю покупать, но и познакомитесь с очень распространенным типом — *массивом*.

Поехали!

Создание массива

Сейчас для создания массивов крутые чуваки используют открывающиеся и закрывающиеся квадратные скобки. Ниже приведена переменная `groceries` (продукты), инициализированная как пустой массив:

```
let groceries = [];
```

Такой скобочный способ создания массива больше известен как *литеральная нотация массива*.

Как правило, вы будете создавать массив, изначально содержащий определенные элементы. Для этого просто поместите нужные элементы в скобки и разделите их запятыми:

```
let groceries = ["Milk", "Eggs", "Frosted Flakes", "Salami", "Juice"];
```

Обратите внимание, что теперь массив содержит `Milk` (молоко), `Eggs` (яйца), `Frosted Flakes` (глазированные хлопья), `Salami` (салями) и `Juice` (сок). Считаю необходимым напомнить о важности запятых.

Теперь, когда вы научились объявлять массив, давайте взглянем на то, как его можно использовать для хранения данных и работы с ними.

Обращение к значениям массива

Одна из прелестей массивов в том, что вы имеете легкий доступ не только к ним самим, но и к их значениям, аналогично выделению одного из продуктов в вашем списке (рис. 13.2).

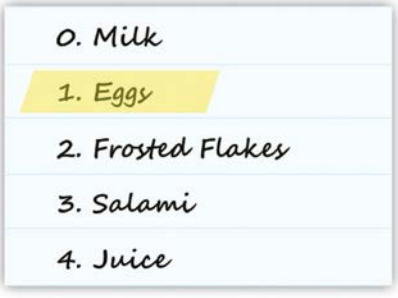


РИС. 13.2.
Массивы позволяют выборочно обращаться к отдельным элементам

Для этого вам достаточно знать простую процедуру обращения к отдельному элементу.

Внутри массива каждому элементу присвоен номер, начиная с нуля. На рис. 13.2 `Milk` имеет значение `0`, `Eggs` — `1`, `Frosted Flakes` соответствует значение `2` и т. д. Формально эти номера называются значением индекса (индексами).

В данном случае наш массив `groceries` объявлен следующим образом:

```
let groceries = ["Milk", "Eggs", "Frosted Flakes", "Salami", "Juice"];
```

Если мне понадобится обратиться к одному из элементов, то все, что потребуется, — это передать значение его индекса:

```
groceries[1]
```

Значение индекса передается массиву внутри квадратных скобок. В текущем примере мы обращаемся к значению `Eggs`, так как именно этому элементу соответствует позиция индекса `1`. Если передать `2`, то вернется `Frosted Flakes`. Вы можете продолжать передавать значения индекса, пока они не закончатся.

Диапазон чисел, которые вы можете использовать в качестве значений индекса, на одно меньше, чем длина самого массива. Причина в том, что индексы начинаются с `0`. Если в массиве есть пять элементов, то попытка отобразить `grocery[6]` или `grocery[5]` приведет к появлению сообщения `undefined`.

Идем дальше. В большинстве реальных сценариев вам понадобится программно перебирать весь массив вместо обращения к каждому элементу отдельно.

Для осуществления этого вы можете использовать цикл `for`:

```
for (let i = 0; i < groceries.length; i++) {
  let item = groceries[i];
}
```

Помните, что диапазон цикла начинается с `0` и заканчивается на одно значение раньше полной длины массива (возвращаемой как свойство `length`). Все работает именно так по уже описанной мной причине — значения индекса начинаются с `0` и заканчиваются на одно значение раньше, чем возвращаемая длина массива. При этом свойство `length` возвращает точное число элементов.

Добавление элементов

Ваши массивы будут редко сохранять свое изначальное состояние, так как вы, скорее всего, будете добавлять в них элементы. Для этого используется метод `push`:

```
groceries.push("Cookies");
```

Метод `push` вызывается непосредственно для массива, при этом в него передаются добавляемые данные. В итоге вновь добавленные элементы всегда оказываются в конце массива.

Например, если выполнить этот код для изначального массива, вы увидите, что элемент `Cookies` (печенье) добавлен в его конец (рис. 13.3).

РИС. 13.3.
Теперь массив расширен добавленным в конец элементом Cookies

Если же вы хотите добавить данные в начало, используйте метод `unshift`:

```
groceries.unshift("Bananas");
```

При добавлении данных в начало массива значение индекса каждого из существующих в нем элементов увеличивается с учетом вновь появившихся данных (рис. 13.4).

РИС. 13.4.
Только что добавленный элемент вставлен в начало

Причина в том, что первый элемент массива всегда будет иметь значение индекса `0`. Поэтому элемент, изначально занимающий позицию значения `0`, вынужденно смещается, смещая и все следующие за ним элементы, освобождая тем самым место для добавляемых данных.

При использовании методы `push` и `unshift` помимо добавления элементов также возвращают новую длину массива:

```
console.log(groceries.push("Cookies")); // возвращает 6
```

Я не уверен, полезно ли это, но на всякий случай имейте это в виду.

Удаление элементов

Для удаления можно использовать методы `pop` и `shift`. `Pop` удаляет последний элемент и возвращает его:

```
let lastItem = groceries.pop();
```

Метод `shift` делает то же самое, но с обратной стороны массива, то есть вместо удаления и возвращения последнего элемента он проделывает это с первым:

```
let firstItem = groceries.shift();
```

При удалении элемента из начала массива позиции индексов остальных уменьшаются на `1`, заполняя тем самым появившийся пропуск (рис. 13.5).

РИС. 13.5.
Что происходит при удалении элементов из массива

Обратите внимание, что при добавлении элементов с помощью `unshift` или `push` значение, возвращаемое при вызове этих методов, является новой длиной массива. Но при использовании методов `pop` или `shift` происходит не то же самое. В данном случае при удалении элементов значение, возвращаемое при вызове метода, является самим удаляемым элементом.

Поиск элементов в массиве

Для поиска элементов внутри массива существует несколько методов: `indexOf`, `lastIndexOf`, `includes`, `find`, `findIndex` и `filter`. Во избежание усложнения мы пока что сконцентрируемся на `indexOf` и `lastIndexOf`.

Работа этих двух индексов заключается в сканировании массива и возвращении индекса совпадающего элемента.

Метод `indexOf` возвращает первый найденный индекс искомого элемента:

```
let groceries =["Milk", "Eggs", "Frosted Flakes", "Salami", "Juice"];
let resultIndex = groceries.indexOf("Eggs",0);

console.log(resultIndex); // 1
```

Обратите внимание, что переменная `resultIndex` содержит результат вызова `indexOf` для массива `groceries`. Для использования `indexOf` я передаю ему искомый элемент вместе с индексом, с которого следует начать:

```
groceries.indexOf("Eggs", 0);
```

В данном случае `indexOf` вернет значение `1`.

Метод `lastIndexOf` похож на `indexOf` в использовании, но отличается тем, *что* возвращает при обнаружении элемента. Если `indexOf` находит первый индекс искомого элемента, то `lastIndexOf` находит и возвращает последний индекс этого элемента.

Если же искомый элемент в массиве отсутствует, оба этих метода возвращают `-1`.

Слияние массивов

Последнее, что мы рассмотрим, — это слияние двух раздельных массивов для создания нового. Предположим, у вас есть два массива `good` (хорошие) и `bad` (плохие):

```
let good = ["Mario", "Luigi", "Kirby", "Yoshi"];
let bad = ["Bowser", "Koopa Troopa", "Goomba"];
```

Чтобы совместить их, используйте метод `concat` для массива, который вы хотите расширить, и передайте в него второй массив в виде аргумента. В итоге будет возвращен новый массив, содержащий и `good`, и `bad`:

```
let goodAndBad = good.concat(bad);
console.log(goodAndBad);
```

В этом примере метод `concat` возвращает новый массив, поэтому переменная `goodAndBad` становится массивом, содержащим результат произведенной конкатенации. Первыми в новом массиве идут элементы `good`, а затем элементы `bad`.

Отображение, фильтрация и сокращение массивов

До сих пор мы рассматривали различные способы добавления элементов, их удаления и другие счетные операции. Помимо этого, массивы предлагают простые способы управления содержащимися в них данными. Эти простые способы представлены методами `map` (отображение), `reduce` (сокращение) и `filter` (фильтрация).

Консервативный способ

Прежде чем говорить о `map`, `reduce` и `filter` и предлагаемом ими удобстве обращения с данными, давайте рассмотрим не самый удобный подход. При этом подходе вы традиционно используете цикл `for` и отслеживаете свое местонахождение в массиве, испытывая при этом, мягко говоря, не самые приятные чувства.

Для наглядности давайте рассмотрим следующий массив имен:

```
let names = ["marge", "homer", "bart", "lisa", "maggie"];
```

Этот соответствующим образом названный массив `names` содержит список имен, написанных в нижнем регистре. Мы же хотим исправить это, сделав первую букву каждого из них заглавной. С помощью цикла `for` это можно сделать так:

```
let names = ["marge", "homer", "bart", "lisa", "maggie"];

let newNames = [];

for (let i = 0; i < names.length; i++) {
  let name = names[i];
  let firstLetter = name.charAt(0).toUpperCase();
```

```
  newNames.push(firstLetter + name.slice(1));
}

console.log(newNames);
```

Обратите внимание, что мы перебираем каждый элемент, делаем первую букву заглавной и добавляем исправленное имя в новый массив `newNames`. Здесь нет ничего магического или сложного, но вы будете часто брать элементы массива, изменять их (или обращаться к ним) и возвращать новый массив с измененными данными. Это достаточно тривиальная задача, где задействуется много рутинного повторяющегося кода. В больших кодовых базах разбор происходящего в цикле добавляет ненужные хлопоты. Вот почему были введены методы `map`, `filter` и `reduce`. С их помощью вы получаете все возможности цикла `for` без ненужных побочных эффектов и лишнего кода. Кому бы это не понравилось?

Изменение каждого элемента с помощью map

Начнем с метода `map`, который мы используем для модификации всех элементов массива во что-либо другое, представленное в виде нового массива (рис. 13.6).

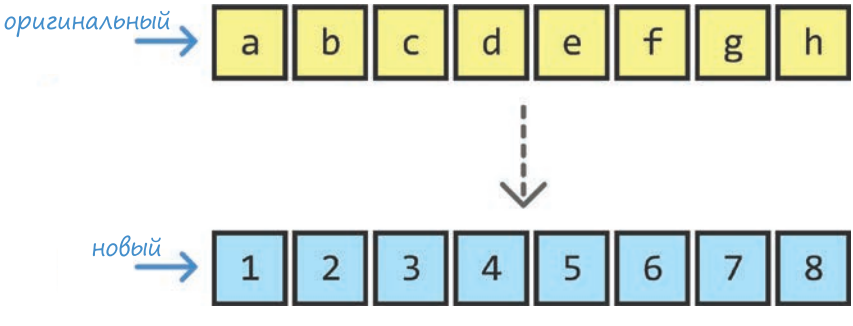


РИС. 13.6.
Оригинальный и новый массивы

Используется `map` следующим образом:

```
let newArray = originalArray.map(someFunction);
```

Эта единственная строка выглядит приятно и располагающе, но изнутри является весьма сложной. Давайте с этим разберемся. Работает метод `map` так: вы вызываете его для массива, на который хотите воздействовать (`originalArray`), и передаете ему функцию (`someFunction`) в качестве аргумента. Функция будет выполняться для каждого элемента массива, то есть вы изначально сможете написать код для изменения всех этих элементов на ваше усмотрение. В конечном итоге вы получите новый массив, содержащий данные, полученные после выполнения функции `someFunction` для элементов оригинального массива. Звучит просто, не правда ли?

Теперь, вооружившись `map`, давайте вернемся к нашей предыдущей задаче по изменению первых букв имен в массиве на заглавные. Сначала взглянем на весь код целиком, а затем рассмотрим важные детали.

```
let names = ["marge", "homer", "bart", "lisa", "maggie"];

function capitalizeItUp(item) {
  let firstLetter = item.charAt(0).toUpperCase();
  return firstLetter + item.slice(1);
}

let newNames = names.map(capitalizeItUp);
console.log(newNames);
```

Разберемся, как этот код работает. Нас интересует функция `capitalizeItUp`, переданная в виде аргумента методу `map`. Эта функция выполняется для каждого элемента, и стоит обратить внимание, что текущий элемент передается ей в качестве аргумента. Для ссылки на аргумент текущего элемента вы можете использовать любое имя на ваш выбор. Мы ссылаемся на этот аргумент с помощью банального `item`:

```
function capitalizeItUp(item) {
  let firstLetter = item.charAt(0).toUpperCase();
  return firstLetter + item.slice(1);
}
```

Внутри этой функции мы можем написать любой код для нужного изменения текущего элемента массива. Единственное, что остается сделать, — это вернуть значение элемента нового массива:

```
function capitalizeItUp(item) {
  let firstLetter = item.charAt(0).toUpperCase();
  return firstLetter + item.slice(1);
}
```

Вот и все. После выполнения этого кода `map` возвращает новый массив, в котором все элементы имеют заглавные буквы и расположены на соответствующих местах. Исходный массив остается неизмененным, имейте это в виду.

ФУНКЦИИ ОБРАТНЫХ ВЫЗОВОВ

Наша функция `capitalizeItUp` также известна как *функция обратного вызова*. Такие функции подразумевают два действия:

- передачу в качестве аргумента другой функции;
- вызов из другой функции.

Вы будете встречать ссылки на функции обратных вызовов постоянно. Например, когда мы вскоре начнем рассматривать методы `filter` и `reduce`. Если вы слышите о них впервые, то теперь будете иметь о них лучшее представление. Если же вы были знакомы с этими функциями ранее, тем лучше для вас.

Фильтрация элементов

При использовании массивов вы будете часто фильтровать (то есть удалять) элементы на основе заданного критерия (рис. 13.7).

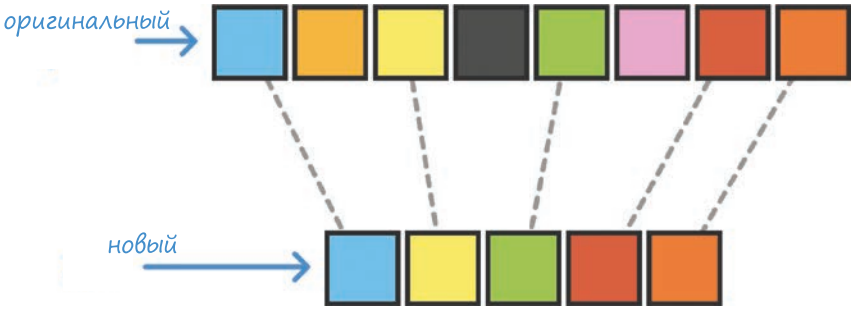


РИС. 13.7.
Уменьшение количества элементов

Например, у нас есть массив чисел:

```
let numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12];
```

В данный момент в нем есть и четные, и нечетные числа. Предположим, что нам надо проигнорировать все нечетные и просмотреть только четные. Этого можно добиться с помощью метода `filter`, отфильтровав все нечетные числа, чтобы остались только нужные нам четные.

Используется метод `filter` аналогично методу `map`. Он получает один аргумент — функцию обратного вызова, а эта функция, в свою очередь, определяет, какие элементы массива отфильтровать. Это легче понять, взглянув на код:

```
let numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12];

let evenNumbers = numbers.filter(function (item) {
  return (item % 2 == 0);
});

console.log(evenNumbers);
```

Мы создаем новый массив `evenNumbers`, который будет содержать результат выполнения метода `filter` для массива `numbers`. Содержимым этого массива будут четные числа благодаря нашей функции обратного вызова, проверяющей каждый элемент, чтобы узнать, будет ли результат `item % 2` (то есть будет ли остаток при делении на 2) равен `0`. Если функция вернет `true`, то элемент будет отправлен в отфильтрованный массив. Если же вернется `false`, элемент будет проигнорирован.

Здесь стоит заметить, что наша функция обратного вызова не является явно именованной, как функция `capitalizeItUp` в предыдущем примере. Она является анонимной, но это не мешает ей выполнять свою работу. Вы будете часто встречать функции обратного вызова в анонимной форме, поэтому стоит знать такой способ их определения.

Получение одного значения из массива элементов

Последним мы рассмотрим метод `reduce`. Он достаточно странный. В случаях с методами `map` и `filter` мы начинали с массива, имеющего один набор значений, а заканчивали другим массивом с другим набором значений. Используя метод `reduce`, мы по-прежнему будем начинать с массива. А вот в конце будем получать всего одно значение (рис. 13.8).

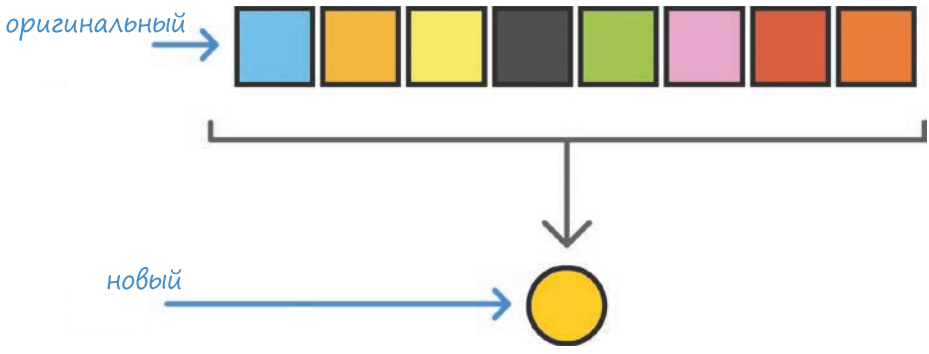


РИС. 13.8.
От множества к одному

Здесь для прояснения происходящего необходим пример.

Давайте еще раз используем массив чисел из предыдущего раздела:

```
let numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12];
```

Мы хотим сложить все значения. В этом и есть смысл метода `reduce`, когда все значения массива сужаются в один элемент. Взгляните на этот код:

```
let total = numbers.reduce(function(total, current) {
  return total + current;
}, 0);

console.log(total);
```

Мы вызываем `reduce` для массива чисел и передаем в него два аргумента:

- функцию обратного вызова;
- начальное значение.

Начинаем суммирование с начального значения `0`, а функция отвечает за добавление каждого элемента массива. В отличие от предыдущих примеров, где функции получали только текущий элемент массива, функция для метода `reduce` задействуется в большей степени. В данном случае ей приходится иметь дело с *двумя* аргументами:

- первый содержит итоговое значение, полученное в результате всех произведенных на этот момент действий;
- второй — это текущий элемент массива.

Используя эти два аргумента, вы можете легко создавать различные сценарии, задействующие отслеживание чего-либо. В нашем примере, поскольку нам просто нужна сумма всех элементов массива, мы складываем `total` со значением `current`. Итоговым результатом будет `31`.

Подробнее об аргументах функций обратных вызовов

Для методов `map` и `filter` в наших функциях обратных вызовов мы определяли только один аргумент, представляющий текущий элемент массива. Для метода `reduce` мы определяли два аргумента, представлявших итоговое значение и текущий элемент. Помимо этого, функции обратных вызовов имеют два опциональных аргумента, которые вы также можете определить:

- индекс текущего элемента массива;
- массив, для которого вызывается `map`, `filter` или `reduce`.

Для методов `map` и `filter` эти аргументы стали бы вторым и третьим. Для `reduce` они бы оказались третьим и четвертым. Вы можете никогда не столкнуться с необходимостью определять эти опциональные аргументы, но если они все же вам понадобятся, знайте, где их искать.

Мы почти закончили. Давайте взглянем на пример, где показана работа метода `reduce` с нечисленными значениями:

```
let words = ["Where", "do", "you", "want", "to", "go", "today?"];

let phrase = words.reduce(function (total, current, index) {
  if (index == 0) {
    return current;
  } else {
    return total + " " + current;
  }
}, "");

console.log(phrase);
```

Здесь совмещается текстовое содержимое массива `words` (слов), чтобы создать значение, которое будет выглядеть как `Where do you want to go today?` (Куда ты хочешь пойти сегодня?) Обратите внимание, что происходит в функции обратного вызова. Помимо совмещения каждого элемента в одну фразу мы определяем опциональный третий аргумент, представляющий индекс нашего текущего элемента. Мы используем этот индекс для отдельного случая с первым словом, чтобы определить, нужен перед ним пробел или нет.

Экскурс в функциональное программирование

Как показали последние несколько разделов, методы `map`, `filter` и `reduce` существенно упрощают работу с массивами. Но они проявляют себя и еще в одной огромной области, которая известна как *функциональное программирование*. Функциональное программирование — это способ написания кода, где вы используете функции, которые:

- могут работать внутри других функций;
- избегают совместного использования и изменения состояния;
- возвращают один и тот же вывод для одного и того же ввода.

Есть и другие мелочи, которые можно было бы здесь перечислить, но для начала хватит и этого. Вы уже видели работу принципов функционального программирования в функциях обратных вызовов. Эти функции идеально подходят под три перечисленных критерия, поскольку могут быть добавлены в любую ситуацию или исключены из нее до тех пор, пока аргументы будут работать. Они не изменяют никакое состояние и полноценно работают внутри методов `map`, `filter` и `reduce`. Функциональное программирование — это занятная тема, требующая гораздо более тщательного рассмотрения. Поэтому пока оставим все как есть, а подробным изучением этой темы займемся позднее.

КОРОТКО О ГЛАВНОМ

Пожалуй, это все, что следует знать про массивы, так как именно для этого вы их и будете использовать чаще всего. По крайней мере, теперь вы точно сможете создать с их помощью список продуктов.

Дополнительные ресурсы и примеры:

- Перемешивание массива: http://bit.ly/kirupaArrayShuffle
- Выбор произвольного элемента массива: http://bit.ly/kirupaRandomItemArray
- Удаление повторов из массива: http://bit.ly/kirupaRemoveDuplicates
- Хеш-таблицы против массивов: http://bit.ly/kirupaHvA

В ЭТОЙ ГЛАВЕ:

- разберем обработку текста в JavaScript;
- научимся выполнять основные операции со строками;
- рассмотрим различные свойства строк.

14

СТРОКИ

Будучи людьми, мы постоянно взаимодействуем со словами — произносим, пишем и также прибегаем к их использованию при написании программ. Так уж вышло, что JavaScript тоже привязан к словам. Буквы и всяческие символы, составляющие наш с вами язык, имеют в JS свое официальное имя — *строки*. Строки в JavaScript — не что иное, как наборы знаков. Несмотря на то что звучит это занудно, умение обращаться к этим знакам и манипулировать ими является необходимым навыком. Этому и будет посвящен этот урок.

Поехали!

Основы

Работать со строками в коде легко. При этом нам просто нужно заключать их в одинарные или двойные кавычки. Вот некоторые примеры:

```
let text = "this is some text";
let moreText = 'I am in single quotes!';

console.log("this is some more text");
```

Помимо простого перечисления строк мы нередко будем их совмещать. Это легко делается с помощью оператора +:

```
let initial = "hello";
console.log(initial + " world!");

console.log("I can also " + "do this!");
```

Во всех этих примерах мы видим строку. Единственная причина, по которой я указываю на столь очевидный факт, в том, что когда мы видим содержимое строк так же буквально, как сейчас, эти строки правильнее называть *строчные литералы*. Итоговая структура при этом не меняется и по-прежнему является примитивным типом *строка* (как один из простых ингредиентов пиццы в предыдущей главе).

Рисунок 14.1 показывает визуальное представление строк `text` и `moreText`.

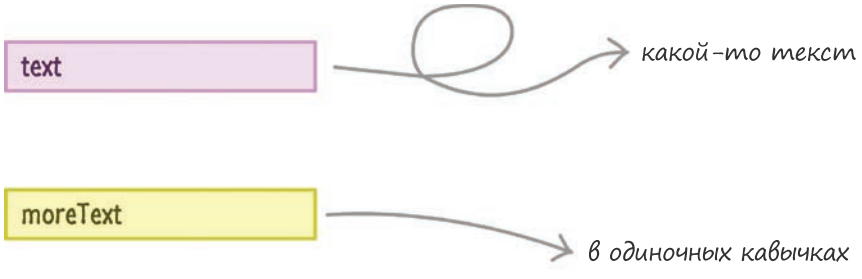


РИС. 14.1.
Визуализация строк

У нас есть всего две переменные, указывающие на литеральные куски текста. Больше здесь ничего не происходит. Если вам интересно, зачем я вообще представил визуализацию столь очевидного, то стоит отметить, что визуализации существенно усложнятся, когда мы перейдем на территорию `Object`. В этой главе вы это отчасти увидите.

Тем не менее все это не так уж важно... пока что. Единственное, что важно помнить, — это то, что необходимо заключать строчные литералы в кавычки (") или апострофы ('), тем самым обособляя их как отдельный участок текста. Если этого не сделать, то ваш код, скорее всего, просто не запустится.

На этом основы заканчиваются. Самое же интересное начнется при использовании всего спектра функциональности JS для работы со строками. Мы перейдем к рассмотрению этой и другой информации в ближайших разделах.

Свойства и методы строк

Когда мы работаем со строками, то реализация объекта `String` предполагает множество свойств, обычно упрощающих работу с текстом. В последующих разделах вместо скучного изучения каждого из этих свойств я сфокусируюсь на наиболее важных, которые будут актуальны в контексте ваших задач.

Обращение к отдельным символам

Несмотря на то что строка выглядит как единый элемент, на деле она состоит из набора знаков. Мы можем обращаться к каждому из них несколькими способами. Наиболее распространено использование массивоподобной скобочной нотации, в которую передается число, соответствующее индексу знака:

```
let vowels = "aeiou";
console.log(vowels[2]);
```

В этом примере мы увидим знак `i`, так как именно этот элемент находится под индексом `2`. Для наглядного представления происходящего рассмотрите рис. 14.2.

РИС. 14.2.
Гласные отображены в виде индексов

Когда дело касается *индекса*, стоит помнить, что его позиция в JavaScript начинается с `0`. Именно поэтому позиция нашего индекса `2`, но порядковый номер действительной позиции элемента — `3`. Вы к этому привыкнете в процессе работы с JavaScript и прочими языками, в чьих названиях не содержатся слова Visual и Basic, что подразумевало бы начало отсчета индексов с `1`.

Переходим далее. Мы можем обращаться ко всем знакам строки с помощью цикла, перебирающего их индексы. Началом цикла будет `0`, а завершение будет определяться длиной самой строки. Длина (то есть количество знаков) возвращается свойством `length`.

Рассмотрим это на примере из предыдущего раздела:

```
let vowels = "aeiou";

for (let i = 0; i < vowels.length; i++) {
  console.log(vowels[i]);
}
```

Несмотря на то что мы можем редко использовать циклы для строк, достаточно распространено использование свойства `length` для получения числа знаков строки.

Также существует альтернатива массиво-скобочной нотации, известная как метод `charAt`. Он возвращает знак согласно определенному индексу:

```
let vowels = "aeiou";
console.log(vowels.charAt(2));
```

В итоге мы получаем то же, что и в случае с описанной ранее массивоподобной нотацией. Я бы рекомендовал использовать этот метод, только если вы работаете для устаревших версий браузеров вроде Internet Explorer 7.

ПОГОДИТЕ, ЧТО?!

Если вам интересно, где строковые примитивы имеют возможность обратиться к свойствам, доступным только для строковых объектов, потерпите до следующей главы. Там мы рассмотрим это более подробно.

Совмещение (конкатенация) строк

Для совмещения двух строк мы можем просто использовать операторы + или +=, складывая их как обычный набор чисел:

```
let stringA = "I am a simple string.";
let stringB = "I am a simple string, too!";

console.log(stringA + " " + stringB);
```

Обратите внимание, что в третьей строке мы складываем `stringA` со `stringB`. Между ними мы определяем пустое пространство (`" "`), чтобы обеспечить разделяющий их пробел. Вы можете смешивать и сопоставлять строчные литералы со строчными примитивами и строчными объектами, получая при этом все тот же совмещенный текст.

К примеру, вот рабочий вариант:

```
let textA = "Please";
let textB = new String("stop!");
let combined = textA + " make it " + textB;

console.log(combined);
```

Несмотря на все это смешение, тип переменной `combined` будет простым *строчным* примитивом.

Для совмещения строк мы также можем использовать метод `concat`. Его можно вызывать из любой строки, определив порядок строчных примитивов, литералов и объектов, которые мы хотим склеить в единую мегастроку:

```
let foo = "I really";
let blah = "why anybody would";
let blarg = "do this";

let result = foo.concat(" don't know", " ", blah, " ", blarg);

console.log(result);
```

В большинстве случаев для этой цели можете просто использовать подход с + и +=. Он быстрее, чем метод `concat`. А если нет разницы, то почему бы не повысить скорость кода?

Получение подстрок из строк

Иногда мы заинтересованы в определенном отрезке знаков отдельной части строки. Для удовлетворения такого интереса служат два метода — `slice` и `substr`. Предположим, у нас есть следующая строка:

```
let theBigString = "Pulp Fiction is an awesome movie!";
```

Давайте с ней немного поработаем.

Метод slice

Метод `slice` позволяет нам определять начальную и конечную позиции интересующей части строки, которую мы хотим извлечь:

```
let theBigString = "Pulp Fiction is an awesome movie!";
console.log(theBigString.slice(5, 12));
```

В этом примере мы извлекаем знаки между индексами `5` и `12`. В итоге будет возвращено слово `Fiction`.

Значения начальной и конечной позиций не обязательно должны быть положительными. Если вы определите отрицательное значение конечной точки, то она будет вычислена обратным отсчетом от конца строки:

```
let theBigString = "Pulp Fiction is an awesome movie!";
console.log(theBigString.slice(0, -6));
```

То же касается и определения начальной точки, которая при отрицательном значении вычисляется также с конца строки:

```
let theBigString = "Pulp Fiction is an awesome movie!";
console.log(theBigString.slice(-14, -7));
```

Мы только что рассмотрели три варианта использования метода `slice`. Я всегда использовал только первый способ с положительными значениями начала и конца нужного отрезка, и вы, вероятно, последуете тем же путем.

Метод substr

Следующий подход для разделения строк — метод `substr`. Он также работает с двумя аргументами:

```
let newString = substr(start, length);
```

Первый из них является числом, определяющим стартовую позицию, а второй представляет число, задающее длину итоговой подстроки. Станет понятнее, если взглянуть на следующие примеры:

```
let theBigString = "Pulp Fiction is an awesome movie!";
console.log(theBigString.substr(0, 4)); // Pulp
```

Наша подстрока начинается с нулевой позиции и отсчитывает четыре знака вперед. Поэтому возвращается `Pulp`. Если мы захотим извлечь слово `Fiction`, то код будет выглядеть так:

```
let theBigString = "Pulp Fiction is an awesome movie!";
console.log(theBigString.substr(5, 7)); // Fiction
```

Если мы не определим длину, возвращаемая подстрока будет содержать знаки от стартовой позиции и до конца:

```
let theBigString = "Pulp Fiction is an awesome movie!";
console.log(theBigString.substr(5)); // Fiction is an awesome movie!
```

Есть еще несколько вариаций передачи значений в метод `substr`, но эти являются основными.

Разделение строки с помощью split

То, что вы можете объединить, можно и разделить на части. Я уверен, что это изречение одного из мудрецов. Еще одним способом разделения строк является метод `split`. Вызов этого метода для строки возвращает массив подстрок. Точки же разделения изначальной строки на подстроки мы определяем знаком или регулярным выражением (RegExp).

Давайте взглянем на простой пример:

```
let inspirationalQuote = "That which you can concatenate, you can
                          also split apart.";
let splitWords = inspirationalQuote.split(" ");

console.log(splitWords.length); // 10
```

Здесь мы разделяем текст `inspirationalQuote` в местах пробелов. При каждой встрече со знаком пробела оставшаяся позади часть массива удаляется и становится элементом возвращаемого массива.

Вот другой пример:

```
let days = "Monday,Tuesday,Wednesday,Thursday,Friday, Saturday,Sunday";
let splitWords = days.split(",");

console.log(splitWords[6]); // Sunday
```

У нас есть переменная `days`, содержащая строку дней, разделенных запятыми. Если мы хотим отделить каждый день, то можем использовать метод `split` с запятой в качестве разделителя. Конечным результатом будет массив из семи элементов, каждый из которых будет представлять день недели из оригинальной строки.

Вас удивит, как часто вы будете использовать метод `split` для разрыва последовательности знаков, которая может быть как простым предложением, так и сложными данными, возвращаемыми веб-службой.

Поиск по строке

Если нам вдруг понадобится найти знак(и) в строке, мы можем использовать методы `indexOf`, `lastIndexOf` и `match`. Сперва рассмотрим `indexOf`.

Этот метод получает искомый нами знак(и) в качестве аргумента. Если он его (их) находит, то возвращает позицию индекса строки, где происходит первое включение. Если совпадений не обнаруживается, `indexOf` возвращает `-1`. Посмотрим на пример:

```
let question = "I wonder what the pigs did to make these birds so
               angry?";
console.log(question.indexOf("pigs")); // 18
```

Мы пытаемся выяснить, есть ли `pigs` (свиньи) в нашей строке. Так как искомый элемент существует, метод `indexOf` сообщает нам, что первое включение этого слова встречается в 18-м индексе. Если же мы ищем что-либо несуществующее вроде буквы `z` в следующем примере, возвращается значение `-1`:

```
let question = "I wonder what the pigs did to make these birds so
               angry?";
console.log(question.indexOf("z")); // -1
```

Метод `lastIndexOf` очень похож на `indexOf`. Как можно догадаться по его имени, он возвращает индекс последнего включения искомого элемента:

```
let question = "How much wood could a woodchuck chuck if
               a woodchuck could chuck wood?";
console.log(question.lastIndexOf("wood")); // 65
```

Вы можете определить еще один аргумент для описанных методов `indexOf` и `lastIndexOf`. Помимо указания искомых знаков вы можете также определить индекс строки, с которого нужно начать поиск:

```
let question = "How much wood could a woodchuck chuck if
               a woodchuck could chuck wood?";
console.log(question.indexOf("wood", 30)); // 43
```

Последнее, что стоит упомянуть об `indexOf` и `lastIndexOf`, — это то, что вы можете сопоставлять любой экземпляр знаков, существующих в строке. При этом не важно, делаете вы это для целых слов или того, что ищете в виде более крупного набора знаков. Обязательно это учитывайте.

Прежде чем подвести итог, давайте рассмотрим метод `match`. В его случае у вас уже меньше контроля. Этот метод в качестве аргумента получает `regexp`:

```
let phrase = "There are 3 little pigs.";
let regexp = /[0-9]/;

let numbers = phrase.match(regexp);

console.log(numbers[0]); // 3
```

Здесь также возвращается массив совпадающих подстрок, поэтому будет уместно применить свои навыки работы с массивами, чтобы налегке продолжить работу с результатом. Работу с регулярными выражениями мы с вами затронем позже.

Строки в нижнем и верхнем регистрах

Под конец давайте рассмотрим то, что не потребует сложных объяснений. Для изменения регистра строки мы можем использовать методы с соответствующими именами, а именно `toUpperCase` для подъема в верхний регистр и `toLowerCase` для приведения в нижний. Взгляните на пример:

```
let phrase = "My name is Bond. James Bond.";

console.log(phrase.toUpperCase()); // MY NAME IS BOND. JAMES BOND.
console.log(phrase.toLowerCase()); // my name is bond. james bond.
```

Я же говорил, что это очень легко!

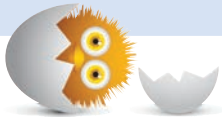
КОРОТКО О ГЛАВНОМ

Строки — это один из немногих основных типов данных в JavaScript, и вы только что видели хороший обзор многих возможностей, которые они предоставляют. Без ответа я оставил всего один случай, когда примитивы загадочным образом получают свойства, обычно характерные исключительно для объектов. Мы рассмотрим этот вопрос в следующей главе.

Некоторые дополнительные ресурсы и примеры:

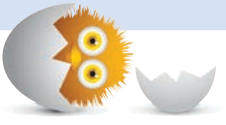
- `Devowelizer` (функция, исключающая гласные буквы): http://bit.ly/kirupaDeVowelize
- Капитализация первой буквы строки: http://bit.ly/kirupaCapLetter
- 10 способов развернуть строку: http://bit.ly/kirupaWaysToReverseString

Если у вас есть какие-либо вопросы, касающиеся строк… жизни или JavaScript в целом, обращайтесь за ответами на форум https://forum.kirupa.com.



В ЭТОЙ ГЛАВЕ:

- лучше поймем принципы работы примитивов и объектов;
- узнаем, что даже у примитивов есть черты объектов;
- выясним, почему JavaScript стал настолько популярен.



15

КОГДА ПРИМИТИВЫ ВЕДУТ СЕБЯ КАК ОБЪЕКТЫ

В предыдущей главе «Строки» и отчасти в главе «О пицце, типах, примитивах и объектах» мы мельком затронули нечто сбивающее с толку. Я несколько раз отмечал, что примитивы очень просты и понятны. В отличие от объектов, они не содержат свойств, которые позволяют обыгрывать значения интересными (и не очень) способами. Действительно, при наглядном рассмотрении всех возможностей использования строк кажется, что наши примитивы таят в себе некую темную сторону:

```
let greeting = "Hi, everybody!!!";
let shout = greeting.toUpperCase(); // Откуда появился toUpperCase?
```

Как видно из приведенного фрагмента, переменная `greeting`, содержащая примитивное значение в форме текста, судя по всему, имеет доступ к методу `toUpperCase`. Как такое вообще возможно? Откуда появился этот метод? Почему мы здесь? Ответы на подобные непростые вопросы и составят львиную долю информации этой главы.

Поехали!

Строки — это не единственная проблема

Так как строки весьма интересны и в некотором смысле игривы (прямо как золотистый ретривер), их легко выбрать в качестве главного виновника этой путаницы с примитивами и объектами. Но как в итоге выясняется, в их банду также входят и многие другие примитивные типы. Таблица 15.1 показывает популярные встроенные типы `Object`, включая *большинство* виновников (`Symbol` и `BigInt` отсиживаются в стороне), которые, помимо прочего, замешаны и в связях с примитивами:

ТАБЛ. 15.1. Объектные типы, включая те, что представляют примитивы

Тип	Назначение
`Array`	Помогает хранить, извлекать и управлять наборами данных
`Boolean`	Выступает в роли обертки для примитива `boolean`; а также работает с помощью `true` и `false`
`Date`	Упрощает представление дат и работу с ними
`Function`	Позволяет вызывать заданный код
`Math`	Умник среди типов, расширяющий возможности работы с числами
`Number`	Выступает в качестве обертки для примитива `number`
`RegExp`	Предоставляет множество возможностей для сопоставления шаблонов в тексте
`String`	Выступает в качестве обертки для примитива `string`

Всегда при работе с логическими значениями, числами или строчными примитивами у нас есть доступ к свойствам, представленным их объектными эквивалентами. В ближайших разделах вы увидите, что конкретно при этом происходит.

Давайте все-таки выберем строки

Как и говорилось в предыдущих главах, обычно мы используем строки в форме литералов:

```
let primitiveText = "Homer Simpson";
```

Как видно из таблицы, строки тоже могут быть использованы как объекты. Есть несколько способов создания нового объекта, но в случае создания объекта для типа вроде строки чаще всего используется ключевое слово *new*, сопровождаемое `String`:

```
let name = new String("Batman");
```

`String` в данной ситуации не просто обычное слово. Оно представляет собой так называемую *функцию-конструктор*, которая используется исключительно для создания новых объектов. Аналогично наличию нескольких способов создания объектов есть несколько способов создания объектов `String`. Я же считаю, что достаточно знать один способ, который *не следует* использовать для их создания.

Как бы то ни было, главное отличие между примитивной и объектной формами строки — это существенное количество лишнего багажа, присущего объекту. На рис. 15.1 — визуальное представление нашего объекта `String` с именем `name`.

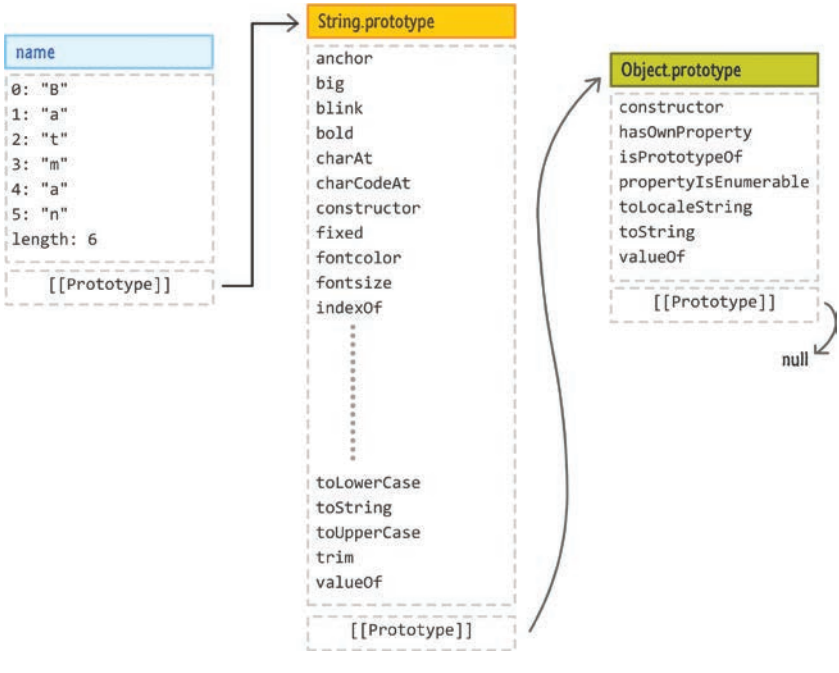


РИС. 15.1.
Углубленный вид объекта String

Переменная `name` содержит указатель на текст `"Homer Simpson"`. Нам также доступно все множество свойств и методов, присущих объекту `String`, включая те, что вы уже использовали ранее (`indexOf`, `toUpperCase` и пр.).

Мы сделаем обширный обзор этого визуального представления позднее, когда будем рассматривать объекты подробнее. Пока не беспокойтесь, если не совсем понимаете ее суть. Просто знайте, что объектная форма любого примитива несет в себе богатый функционал.

Почему это важно

Давайте вернемся к сбивающему с толку моменту. Наша строка — это примитив. Как может примитивный тип позволить обращаться к его свойствам? Дело в том, что JavaScript — весьма странный язык. Предположим, у нас есть следующая строка:

```
let game = "Dragon Age: Origins";
```

Очевидно, что переменная `game` — это строковый примитив, присвоенный конкретному тексту. Если мы захотим обратиться к `length` этого текста, то сделаем следующее:

```
let game = "Dragon Age: Origins";
console.log(game.length);
```

Как часть вычисления `game.length` JavaScript преобразует строковый примитив в объект. На короткое время наш приземленный примитив станет прекрасным объектом, чтобы мы могли выяснить его длину. Здесь следует помнить, что все это временно. Так как этот временный объект не имеет основы и ни к чему не привязан, то после выполнения своей миссии он удаляется и остается лишь результат вычисления `length` (число), а переменная `game` по-прежнему является строковым примитивом.

Такая трансформация происходит только с примитивами. Если мы создадим объект `String`, то он так и останется навсегда объектом. Представим следующий пример:

```
let gameObject = new String("Dragon Age:Origins");
```

В данном случае переменная `gameObject` очень четко указывает на что-то имеющее тип `Object`. Эта переменная продолжит указывать на тип `Object`, пока вы не измените строку или сделаете что-нибудь, что приведет к изменению ссылки. Способность примитива трансформи-

роваться в объект, а затем обратно в примитив является уникальной. Объекты в такой глупости не участвуют.

Вы можете легко убедиться в сказанном мной, проверив тип ваших собственных данных. Это можно сделать с помощью ключевого слова `typeof`. В следующем примере я использую его для подтверждения всего, что только что сказал:

```
let game = "Dragon Age: Origins";
console.log("Length is: " + game.length);

let gameObject = new String("Dragon Age:Origins");

console.log(typeof game); // строка
console.log(typeof game.length); // число
console.log(typeof gameObject); // объект
```

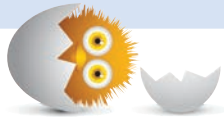
Думаю, вы по достоинству оцените эти знания.

КОРОТКО О ГЛАВНОМ

Надеюсь, это краткое объяснение поможет вам осознать, почему примитивы при необходимости ведут себя как объекты. Здесь может возникнуть другой вопрос, а именно: «Зачем вообще кто-то мог решить разработать язык с такими странностями?» В конце концов, если примитив превращается в объект при необходимости, то почему бы ему так и не оставаться им навсегда? Ответ на этот вопрос будет связан с потреблением памяти.

Как я уже недавно упоминал, объектная форма объекта примитива несет на себе существенно больше багажа, чем обычный примитив. В итоге это требует дополнительных ресурсов для поддержания функциональности. Решением в этом случае послужил компромисс. Все литеральные значения вроде текста, чисел и логических значений хранятся в виде примитивов, если изначально таковыми создаются и/или используются. Только при необходимости они преобразовываются в соответствующие им формы `Object`. Чтобы обеспечить минимальное потребление памяти приложением, эти преобразованные объекты быстро удаляются (*сборщиком мусора*), как только выполнят свою задачу.

Есть вопросы? Задавайте их на форуме https://forum.kirupa.com и получайте развернутые оперативные ответы от единомышленников.



В ЭТОЙ ГЛАВЕ:

- разберемся с числами;
- узнаем, с какими численными значениями вам предстоит работать;
- познакомимся с объектом Math и тем, что он делает.

16

ЧИСЛА

В JavaScript приходится часто иметь дело с числами. Даже если вы не будете работать непосредственно с числами, то будете косвенно сталкиваться с ними во многих базовых и не только задачах вроде ведения подсчета чего-либо, работы с массивами и т. д.

В текущей главе я представлю вам числа на примере их использования для выполнения многих привычных задач. Наряду с этим мы несколько выйдем за рамки основ, чтобы обширнее представить себе интересные и полезные возможности, связанными с ними.

Поехали!

Использование чисел

Используются числа очень просто. Ниже приведен простой пример, в котором я объявляю переменную `stooges`, инициализированную как число `3`:

```
let stooges = 3;
```

Вот и все. Ничего сложного. Если вы захотите использовать более сложные числа, то просто используйте их, как обычно:

```
let pi = 3.14159;
let color = 0xFF;
let massOfEarth = 5.9742e+24;
```

В этом примере вы видите десятичное, шестнадцатеричное, а также очень большое значение, в котором используется экспонента. В итоге ваш браузер автоматически сделает то, что потребуется. Имейте в виду, что при этом также могут быть использованы и отрицательные значения. Для этого достаточно добавить знак минуса (-) перед числом:

```
let temperature = -42;
```

В этом кратком разделе вы познакомились с тем, как чаще всего будете использовать числа. В течение следующих разделов мы углубимся в тему и рассмотрим некоторые интересные возможности, применимые к ним.

ЧИСЛА В JAVASCRIPT

Вам любопытно, почему работать с числами так легко? Отвечаю! JavaScript не богат на численные типы. Вам не придется объявлять числа с типами `int`, `double`, `byte`, `float` и пр., как это делается в некоторых других языках. Единственное исключение представлено в виде типа `BigInt`, который вы будете использовать, если вам понадобится действительно огромное или малое число. Об этом типе мы поговорим позже.

Отмечу еще, что в JavaScript все числа конвертируются в 64-битные числа с плавающей точкой.

Операторы

Ни одно введение в тему чисел нельзя считать полноценным, не показав, как используются математические операторы для реализации задач первого класса школьной программы по математике.

В текущем разделе мы рассмотрим распространенные операторы.

Простые математические действия

В JavaScript вы можете создавать простые математические выражения, используя `+`, `-`. `*`, `/` и `%` для сложения, вычитания, умножения, деления и нахождения остатка (модуля) чисел соответственно. Если вы умеете пользоваться калькулятором, то сможете производить простые вычисления и в JavaScript.

Вот некоторые примеры с применением перечисленных операторов:

```
let total = 4 + 26;
let average = total / 2;
let doublePi = 2*3.14159;
let subtractItem = 50 – 25;
let remainder = total % 7;
let more = (1 + average * 10) / 5;
```

Обратите внимание, что в последней строке я определяю фиксированный порядок выполнения операций, заключая в скобки выражение, которое хочу вычислить как группу. Опять же это все уровень калькулятора.

JavaScript производит вычисление выражений в следующем порядке:

1. Скобки.
2. Экспоненты.
3. Умножение.
4. Деление.
5. Сложение.
6. Вычитание.

Для запоминания этого порядка иногда используют соответствующие мнемонические схемы. В начальных классах меня научили вот такому: Please Excuse My Dear Aunt Sally[1].

[1] Аббревиатура PEMDAS (*Parentheses, Exponents, Multiplication, Division, Addition, Subtraction*) переводится как: «Простите мою дорогую тетушку Салли». По-русски PEMDAS будет ССУДСВ (скобки, степень, умножение, деление, сложение, вычитание). Читатели могу придумать собственные мнемонические уловки. — *Примеч. ред.*

Увеличение и уменьшение

Нередко в отношении чисел вы будете производить увеличение и уменьшение переменной на определенную величину. Ниже представлен пример увеличения переменной `i` на 1:

```
let i = 4;
i = i + 1;
```

Вам не обязательно увеличивать или уменьшать именно на 1. Вы можете использовать произвольное число:

```
let i = 100;
i = i – 2;
```

При этом также не обязательно использовать именно сложение или вычитание. Вы можете выполнять и другие операции:

```
let i = 100;
i = i / 2;
```

Здесь стоит разглядеть шаблон. Независимо от того, какой оператор используете, вы заметите, что всегда изменяете переменную `i`. В связи с частым использованием этого шаблона существуют специальные операторы для упрощения процесса (табл. 16.1).

ТАБЛ. 16.1. Операторы, упрощающие увеличение и уменьшение

Выражение	Действие
i++	Увеличивает *i* на 1 (*i* = *i* + 1)
i--	Уменьшает *i* на 1 (*i* = *i* – 1)
i += *n*	Увеличивает *i* на *n* (*i* = *i* + *n*)
i-= *n*	Уменьшает *i* на *n* (*i* = *i* – *n*)
i *= *n*	Умножает *i* на *n* (*i* = *i* * *n*)
i /= *n*	Делит *i* на *n* (*i* = *i* / *n*)
i %= *n*	Находит остаток *i* при делении на *n* (*i* = *i* % *n*)
i **= *n*	Экспоненциальный оператор, где *i* возводится в степень *n*

Если я использую эти операторы для трех примеров, приведенных ранее, то код будет выглядеть так:

```
i++;
i -= 2;
i /= 2;
```

Прежде чем мы здесь закончим, есть одна хитрость, о которой вам следуем знать. Она касается операторов `--` и `++` для увеличения и уменьшения значения на 1. Тут важно определить оператор перед переменной или после нее.

Рассмотрим пример:

```
let i = 4;
let j = i++;
```

После выполнения этих двух строк значением `i` будет `5`, как вы и могли ожидать. Значением `j` будет `4`. Обратите внимание, что в этом примере оператор используется после переменной.

Если же мы расположим его перед ней, то результат будет несколько иным:

```
let i = 4;
let j = ++i;
```

В этом случае значением `i` по-прежнему будет `5`. Но при этом удивительным образом значением `j` теперь также будет `5`.

Эти два примера отличаются лишь расположением оператора, которое определяет, будет ли возвращено увеличенное значение или значение, имевшееся до увеличения.

Шестнадцатеричные и восьмеричные значения

Помимо обычных десятичных значений вы можете использовать шестнадцатеричные (основание 16) и восьмеричные (основание 8). При работе с восьмеричными обязательно начинайте числа с `0`:

```
let leet = 0o2471;
```

При использовании шестнадцатеричных начинайте с `0x`:

```
let leet = 0x539;
```

Во многих ситуациях придется взаимодействовать с этими значениями в форме строк. В связи с этим вы уже не сможете манипулировать ими, как обычными числами. Для начала потребуется преобразовывать эти строки в числа.

Делается это с помощью функции `parseInt`:

```
let hexValue = parseInt('FFFFFF', 16);
let octalValue = parseInt('011', 8);
```

Функция `parseInt` получает шестнадцатеричное или восьмеричное значение, сопровождаемое основанием, из которого вы производите преобразование.

Особые значения — Infinity и NaN

Последним, что мы рассмотрим, будут два глобальных свойства, с которыми вам предстоит сталкиваться и которые не являются числовыми значениями. Это `Infinity` (бесконечность) и `NaN` (*не число*):

Infinity

Вы можете использовать значения `Infinity` и `-Infinity` для определения бесконечно больших и бесконечно малых чисел:

```
let myLoveForYou = Infinity * 2;
```

На деле вы вряд ли будете часто использовать `Infinity`. Чаще такие значения могут быть возвращены в результате выполнения кодом каких-то задач. Например, если вы разделите на 0, то в качестве результата будет возвращено именно значение `Infinity`.

NaN

Ключевое слово `NaN` обозначает «не число» и возвращается, когда вы пытаетесь произвести недопустимую вычислительную операцию. Например:

```
let nope = 1920 / "blah";
```

В данном случае будет возвращено `NaN`, так как нельзя делить число на строку. Существуют простые случаи, в которых это будет происходить, и некоторые из них мы рассмотрим позднее.

Получение числа из строки

Иногда у вас будут числа, заключенные внутри строк. Чтобы подробно ознакомиться с этой темой, прочтите статью «Получение числа из строки» (https://www.kirupa.com/html5/going_from_a_string_to_a_number.htm).

Объект Math

Числа используются во множестве математических выражений, которые зачастую выходят за рамки простого сложения, вычитания, умножения и деления. Если бы в курсе математики читали только перечисленное выше, все было бы проще. Для упрощения выполнения сложных операций с числами как раз и служит объект `Math`. Он предлагает множество удобных функций и констант, мы же вкратце рассмотрим, на что он способен.

Скукота!

Буду с вами честен. Разбор всех предлагаемых объектом `Math` возможностей был бы скучен. Если вы не фанат этой темы, то я предлагаю пробежаться по следующим разделам и возвращаться к ним уже по мере необходимости. Объект `Math` никуда не уйдет — друзей у него нет, поэтому он будет преданно ждать вас и никуда не денется.

Константы

Чтобы избавить вас от необходимости определять такие математические постоянные, как число π, постоянная Эйлера, натуральный логарифм и т. д., объект `Math` определяет большинство распространенных констант за вас (табл. 16.2).

ТАБЛ. 16.2. Константы

Использование	Что обозначает
`Math.E`	Постоянная Эйлера
`Math.LN2`	Натуральный логарифм 2
`Math.LN10`	Натуральный логарифм 10
`Math.LOG2E`	Log E по основанию 2
`Math.LOG10E`	Log E по основанию 10
`Math.PI`	3,14159 (это все, что я помню, и мне лень искать остальное!)
`Math.SQRT1_2`	$\sqrt{\frac{1}{2}}$
`Math.SQRT2`	$\sqrt{2}$

Из всех этих констант я чаще всего использовал `Math.PI`:

Прошу прощения, что запостил эту картинку

Ее вы будете использовать везде, от рисования кругов на экране и до определения тригонометрических выражений. Честно говоря, даже и не помню, использовал ли я вообще другие константы. Вот пример функции, возвращающей длину окружности по заданному радиусу:

```
function getCircumference(radius) {
  return 2 * Math.PI * radius;
}

console.log(getCircumference(2));
```

Используется `Math.PI` и все прочие константы так же, как и любая переменная с именем.

Округление чисел

Ваши числа часто будут содержать ненужную точность:

```
let position = getPositionFromCursor(); // 159.3634493939
```

Чтобы облегчить округление таких чисел до разумного целочисленного значения, используются функции `Math.round()`, `Math.ceil()` и `Math.floor()`, в которые число передается в виде аргумента (табл. 16.3).

ТАБЛ. 16.3. Функции округления

Функция	Действие
`Math.round()`	Возвращает число, округленное до ближайшего целого числа. При этом округление происходит вверх, если аргумент больше или равен 0,5. Если аргумент меньше 0,5, округление производится до текущего целого числа
`Math.ceil()`	Возвращает число, которое больше или равно вашему аргументу
`Math.floor()`	Возвращает число, которое меньше или равно вашему аргументу

Легче всего понять эту таблицу, посмотрев функции в действии:

```
Math.floor(.5); // 0
Math.ceil(.5); // 1
Math.round(.5); // 1

Math.floor(3.14); // 3
Math.round(3.14); // 3
Math.ceil(3.14); // 4
```

```
Math.floor(5.9); // 5
Math.round(5.9); // 6
Math.ceil(5.9); // 6
```

Эти функции всегда округляют до целого числа. Если вы хотите произвести округление до точного набора цифр, то ознакомьтесь со второй половиной статьи «Округление чисел в JavaScript» (https://www.kirupa.com/html5/rounding_numbers_in_javascript.htm).

Тригонометрические функции

Больше всего мне нравится, что объект `Math` дает удобный способ обращаться почти что к любым тригонометрическим функциям, которые могут понадобиться (табл. 16.4).

Для их использования просто передайте число в качестве аргумента:

```
Math.cos(0); // 1
Math.sin(0); // 0
Math.tan(Math.PI / 4); // 1
Math.cos(Math.PI); // 1
Math.cos(4 * Math.PI); // 1
```

ТАБЛ. 16.4. Тригонометрические функции

Функция	Действие
`Math.cos()`	Вычисляет косинус аргумента
`Math.sin()`	Вычисляет синус аргумента
`Math.tan()`	Вычисляет тангенс аргумента
`Math.acos ()`	Вычисляет арккосинус аргумента (крутое название, да?)
`Math.asin()`	Вычисляет арксинус аргумента
`Math.atan ()`	Вычисляет арктангенс аргумента

Эти функции получают значения в виде радиан. Если же ваши числа представлены в виде градусов, то сначала преобразуйте их в радианы.

Степени и квадратные корни

В табл. 16.5 даны еще несколько функций, присущих объекту `Math`, а именно `Math.pow()`, `Math.exp()` и `Math.sqrt()`.

ТАБЛ. 16.5. Функции для вычисления степеней и квадратных корней

Функция	Действие
`Math.pow()`	Возводит число в заданную степень
`Math.exp()`	Возводит постоянную Эйлера в заданную степень
`Math.sqrt()`	Возвращает квадратный корень заданного аргумента

Теперь взглянем на несколько примеров:

```
Math.pow(2, 4); //эквивалент 2^4 (или 2 * 2 * 2 * 2)
Math.exp(3); //эквивалент Math.E^3
Math.sqrt(16); //4
```

Обратите внимание, что `Math.pow()` получает два аргумента. Это, вероятно, первая рассмотренная нами встроенная функция, получающая два аргумента, что даже несколько вдохновляет.

Получение абсолютного значения

Если вам понадобится получить абсолютное значение, просто используйте функцию `Math.abs()`:

```
Math.abs(37); //37
Math.abs(-6); //6
```

На этом все.

Случайные числа

Для генерации случайных чисел между 0 и чуть меньше, чем 1, можно использовать функцию `Math.random()`. Эта функция не получает аргументы, но вы можете легко использовать ее как часть выражения:

```
let randomNumber = Math.random() * 100;
```

При каждом вызове этой функции вы увидите случайное число, возвращаемое для `Math.random()`. Все подробности ее использования для генерации случайных чисел вы можете найти в статье «Случайные числа в JS» (https://www.kirupa.com/html5/random_numbers_js.htm).

КОРОТКО О ГЛАВНОМ

На этом ознакомительная глава, посвященная числам и объекту `Math` в JavaScript, окончена. Как вы видите, легче уже некуда. JS предоставляет максимально простой подход для работы с этими элементами, а эта глава лишь мельком показала горизонты их возможностей на случай, если вы решите направиться к ним.

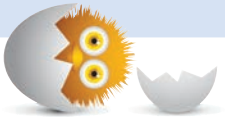
Ниже представлены дополнительные ресурсы с примерами, которые помогут вам лучше понять возможности использования чисел в JavaScript:

- Получение числа из строки: http://bit.ly/kirupaStrToNum
- Случайные числа в JS: http://bit.ly/kirupaRandom
- Продвинутые случайные числа в JS: http://bit.ly/AdvRandom
- Почему мои числа не складываются: http://bit.ly/kirupaFPG
- Случайные цвета в JS: http://bit.ly/kirupaRandomColors

Числа в JavaScript — это занятная тема, которая местами может быть запутывающей. Если у вас вдруг возникнут трудности, то прояснить ситуацию вы можете, обратившись на форум https://forum.kirupa.com.

В ЭТОЙ ГЛАВЕ:

- поймем разницу между свойствами данных и свойствами-аксессорами;
- познакомимся с методами получения и изменения данных (геттеры и сеттеры);
- узнаем, когда использовать свойство доступа, а когда свойства-аксессоры.



17

МЕТОДЫ ПОЛУЧЕНИЯ И ИЗМЕНЕНИЯ ДАННЫХ

Свойства, с которыми мы работали до сих пор, известны как *свойства данных*. Для этих свойств мы задаем имя и присваиваем им значение:

```
let foo = {
  a: "Hello",
  b: "Monday";
}
```

Для считывания свойства нужно просто обратиться к нему напрямую:

```
console.log(foo.a);
```

Записываются же значения в свойства вполне ожидаемым способом:

```
foo.a = "Manic";
```

Помимо чтения и записи значения, мы больше ничего не можем сделать. Такова горькая правда о свойствах данных. Продолжая тему чтения и записи свойств, что, если бы мы могли следующее:

- использовать существующий синтаксис для чтения и записи значений свойств;
- получать возможность выполнять пользовательский код на фоне?

Это было бы неплохо, как считаете? Скажу больше: все это нам доступно. Такие возможности предоставляют дружественные и трудолюбивые *свойства-аксессоры*. В текущем разделе мы все о них узнаем и познакомимся с великими рок-звездами — загадочными геттерами и сеттерами.

Поехали!

История двух свойств

Внешне свойства-аксессоры и свойства данных очень схожи. Для свойств данных вы можете производить чтение и запись свойства:

```
theObj.storedValue = "Unique snowflake!"; // запись
console.log(theObj.storedValue); // чтение
```

С помощью свойств-аксессоров вы можете, в принципе, то же самое:

```
myObj.storedValue = "Also a unique snowflake!"; // запись
console.log(myObj.storedValue); // чтение
```

Глядя на само использование свойства, мы не можем сказать, является ли оно свойством данных или свойством-аксессором. Чтобы обнаружить отличие, нам нужно посмотреть туда, где свойство фактически определено. Взгляните на следующий код, в котором внутри объекта `zorb` определено несколько свойств:

```
let zorb = {
  message: "Blah",

  get greeting() {
    return this.message;
  },

  set greeting(value) {
    this.message = value;
  }
};
```

Первое сверху — это `message`, стандартное свойство данных:

```
let zorb = {
  message: "Blah",

  get greeting() {
    return this.message;
  },

  set greeting(value) {
    this.message = value;
  }
};
```

Мы узнаем, что это свойство данных, так как в нем присутствует только имя свойства и значение. А вот дальше все немного интереснее. Следующее свойство — это `greeting`, которое не похоже ни на одно из свойств, встреченных нами ранее:

```
let zorb = {
  message: "Blah",

  get greeting() {
    return this.message;
  },

  set greeting(value) {
    this.message = value;
  }

};
```

Вместо того чтобы обходиться именем и значением, как `message`, свойство `greeting` разделено на две функции, которым предшествует ключевое слово `get` или `set`:

```
let zorb = {
  message: "Blah",

  get greeting() {
    return this.message;
  },

  set greeting(value) {
    this.message = value;
  }
};
```

Эти ключевые слова и пары функций известны как *геттеры* и *сеттеры* соответственно. Особенными их делает то, что мы не обращаемся к `greeting` как к функции, а делаем это так же, как и с обычным свойством:

```
zorb.greeting = "Hola!";
console.log(zorb.greeting);
```

Самое же интересное происходит на уровне геттеров и сеттеров, поэтому мы рассмотрим их глубже.

Знакомство с геттерами и сеттерами

На данный момент мы знаем лишь, что *геттер* и *сеттер* — это модные названия функций, которые ведут себя как свойства. Когда мы пытаемся считать свойство-аксессор (`zorb.greeting`), вызывается функция геттер:

```
let zorb = {
  message: "Blah",

  get greeting() {
    return this.message;
  },

  set greeting(value) {
    this.message = value;
  }
};
```

Аналогичным образом, когда мы задаем новое значение свойству-аксессору (`zorb.greeting = "Hola!"`), вызывается функция сеттер:

```
let zorb = {
  message: "Blah",

  get greeting() {
    return this.message;
  },

  set greeting(value) {
    this.message = value;
  }

};
```

Основной потенциал геттеров и сеттеров лежит в коде, который мы можем выполнять, когда считываем или записываем свойство. *Так как мы*

имеем дело с функциями под прикрытием, то можем выполнять любой нужный нам код. В примере с `zorb` мы использовали геттер и сеттер `greeting`, чтобы приблизительно повторить поведение свойств данных. Мы можем назначить значение, а затем считать его. Скучновато, не правда ли? Но это не должно происходить именно так, и следующие примеры привнесут больше интереса в этот процесс.

Генератор крика

Вот пример, в котором любое определяемое нами сообщение будет преобразовано в верхний регистр:

```
var shout = {
  _message: "HELLO!",

  get message() {
    return this._message;
  },

  set message(value) {
    this._message = value.toUpperCase();
  }
};

shout.message = "This is sparta!";
console.log(shout.message);
```

Обратите внимание, что как часть определения значения свойства `message` мы храним введенное значение в верхнем регистре благодаря методу `toUpperCase`, который передается всем строковым объектам. Все это гарантирует, что при попытке считать сохраненное сообщение мы увидим полностью заглавную версию того, что введем.

Регистрирование действий

В следующем примере у нас есть объект `superSecureTerminal`, регистрирующий имена всех пользователей:

```
var superSecureTerminal = {
  allUserNames: [],
   _username: "",

  showHistory() {
    console.log(this.allUserNames);
  },
```

```
  get username() {
    return this._username;
  },

  set username(name) {
    this._username = name;
    this.allUserNames.push(name);
  }
}
```

Это регистрирование обрабатывается внутри сеттера `username`, где каждое предоставляемое имя пользователя сохраняется в массиве `allUserNames`, а функция `showHistory` выводит сохраненные имена пользователей на экран. Прежде чем продолжить, давайте протестируем этот код. Мы попробуем обратиться к `superSecureTerminal` не так, как делали это до сих пор. Для этого мы используем кое-какие знания о создании объектов и сделаем следующее:

```
var myTerminal = Object.create(superSecureTerminal);
myTerminal.username = "Michael Gary Scott";
myTerminal.username = "Dwight K. Schrute";
myTerminal.username = "Creed Bratton";
myTerminal.username = "Pam Beasley";

myTerminal.showHistory();
```

Мы создаем новый объект `myTerminal`, основанный на объекте `superSecureTerminal`. С этого момента мы можем делать с `myTerminal` все, что угодно, в привычном режиме.

Проверка значения свойства

Последним мы рассмотрим пример, в котором сеттеры производят проверку переданных им значений:

```
let person = {
  _name: "",
  _age: "",

  get name() {
    return this._name;
  },

  set name(value) {
    if (value.length > 2) {
    this._name = value;
  } else {
```

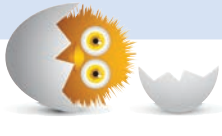
```
    console.log("Name is too short!");
  }
},

  get age() {
    return this._age;
  },

  set age(value) {
    if (value < 5) {
    console.log("Too young!");
  } else {
    this._age = value;
  }
},

  get details() {
    return "Name: " + this.name + ", Age: " + this.age;
  }
}
```

Обратите внимание, что мы производим проверку допустимого ввода для обоих свойств `name` и `age`. Если введенное имя короче двух знаков, выводится соответствующее уведомление. Если указан возраст меньше пяти, то также выскакивает уведомление. Возможность проверять, является ли присваиваемое свойству значение подходящим, вероятно, одна из лучших возможностей, предлагаемых геттерами и сеттерами.

КОРОТКО О ГЛАВНОМ

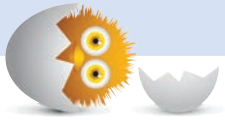
Стоит ли прекращать создавать стандартные свойства данных и использовать эти модные свойства-аксессоры? На самом деле нет. Все зависит от текущих потребностей и будущих нужд. Если вы уверены, что свойству никогда не потребуется дополнительная гибкость, предлагаемая геттерами и сеттерами, то можете просто оставить его в виде свойства данных. Если вам когда-нибудь понадобится к нему вернуться, то изменение свойства данных на свойство-аксессор полностью происходит за кадром. Мы можем делать это, не влияя на итоговое применение самого свойства. Круто, не правда ли?

Если у вас возникнут сложности в этой теме, то обращайтесь за помощью на форум **https://forum.kirupa.com**.



В ЭТОЙ ГЛАВЕ:

- подробно разберем принципы работы объектов;
- научимся создавать пользовательские объекты;
- раскроем таинственное свойство прототипа;
- произведем наследование.



18

ОБ ОБЪЕКТАХ ПОДРОБНЕЕ

Знакомясь с объектами в главе 12 «О пицце, типах, примитивах и объектах», мы произвели очень поверхностный обзор того, чем являются объекты в JavaScript и как их воспринимать. На тот момент этого было достаточно, чтобы рассмотреть основы некоторых встроенных типов, но теперь пора двигаться дальше. В этой главе увидим, что вся предыдущая информация была лишь вершиной айсберга.

Здесь мы уже подробнее пересмотрим объекты и затронем некоторые наиболее продвинутые темы вроде объекта `Object`, создания пользовательских объектов, наследования, прототипов и ключевого слова `this`. Если все перечисленное кажется вам совершенно непонятным, то я обещаю, что к завершению главы мы это исправим.

Поехали!

Знакомство с объектом

В самом низу пищевой цепочки есть тип `Object`, который закладывает основу как для пользовательских объектов, так и для встроенных типов вроде `Function`, `Array` и `RegExp`. Практически все, за исключением `null`

и `undefined`, непосредственно связано с `Object` или может стать им при необходимости.

Как мы уже видели, функциональность, которую предоставляет `Object`, весьма мала. Он позволяет определять множество именованных пар ключ — значение, которые мы с любовью называем *свойствами*. Это не особо отличается от того, что мы видим в других языках, использующих хэш-таблицы, ассоциативные массивы и словари.

Как бы то ни было, все это скучно. Мы же собираемся изучать объекты на практике.

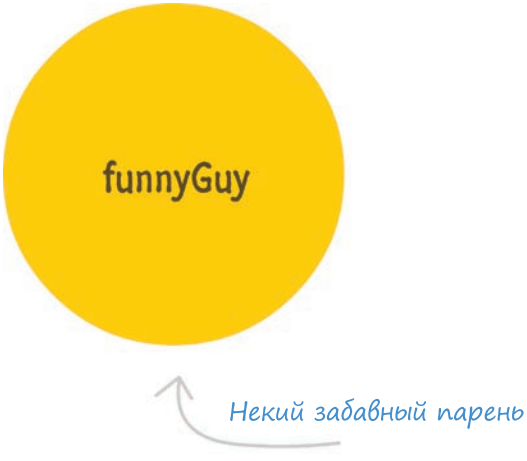
Создание объектов

Первое, что мы рассмотрим, — это создание объекта. Для этого существует несколько способов, но все крутые ребята создают их с помощью забавного (но компактного) *синтаксиса объектного литерала*:

```
let funnyGuy = {};
```

Все верно. Вместо написания `new Object()`, как это делали еще ваши деды, мы можем просто инициализировать наш объект, используя `{}`.

По завершении выполнения этой строки мы получим созданный объект `funnyGuy` с типом `Object`:



Создание объектов имеет еще кое-какие особенности кроме только что рассмотренного нами синтаксиса объектного литерала, но их мы рассмотрим в более подходящее время.

Добавление свойств

Как только у нас появился объект, мы можем использовать один из ряда путей для добавления к нему свойств. Возьмем простой и производительный вариант, который задействует подобную массиву скобочную нотацию, где имя свойства будет указано в виде индекса.

Продолжим с нашего объекта `funnyGuy`:

```
let funnyGuy = {};
```

Предположим, мы хотим добавить свойство `firstName` и задать ему значение `Conan`. Добавление свойства в данном случае производится с помощью синтаксиса записи через точку:

```
funnyGuy.firstName = "Conan";
```

Вот и все. После добавления свойства мы можем обращаться к нему посредством того же синтаксиса:

```
let funnyFirstName = funnyGuy.firstName;
```

АЛЬТЕРНАТИВА ЗАПИСИ ЧЕРЕЗ ТОЧКУ

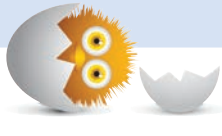
Для определения считывания свойств мы использовали подход, называемый *записью через точку*. Но у него есть альтернатива, использующая вместо точки скобки:

```
let funnyGuy = {};

funnyGuy["firstName"] = "Conan";
funnyGuy["lastName"] = "O'Brien";
```

Какой из этих подходов использовать, решать только вам (или команде), но есть определенные случаи, для которых предназначены именно скобки. Имеются в виду случаи, когда мы работаем со свойствами, чьи имена нам нужно генерировать динамически. В примере же с `firstName` и `lastName` мы прописали их статично. Взгляните на следующий фрагмент кода:

```
let myObject = {};

for (let i = 0; i < 5; i++) {
  let propertyName = "data" + i;

  myObject[propertyName] = Math.random() * 100;
}
```

Мы имеем объект `myObject` — обратите внимание на то, как мы устанавливаем его свойства. У нас нет статичного списка имен свойств, вместо этого мы создаем имя свойства, опираясь на значение индекса массива. Когда мы выясняем имя свойства, то используем эти данные для создания свойства в `myObject`. Генерируемые именами свойств будут `data0`, `data1`, `data2`, `data3` и `data4`. Эта возможность динамического определения имени свойства в процессе изменения или чтения объекта оказывается доступной благодаря именно скобочному синтаксису.



Теперь, прежде чем продолжить, давайте добавим еще одно свойство, назовем его `lastName` и присвоим ему значение `O'Brien`:

```
funnyGuy.lastName = "O'Brien";
```

К этому моменту мы уже в хорошей форме, а наш полный код `funnyGuy` выглядит следующим образом:

```
let funnyGuy = {};

funnyGuy.firstName = "Conan";
funnyGuy.lastName = "O'Brien";
```

При его выполнении будет создан объект `funnyGuy`, и в нем будут определены два свойства — `firstName` и `lastName`.

Мы только что рассмотрели, как *пошагово* создавать объект и устанавливать для него свойства. Если же мы изначально знаем, какие свойства должны быть в объекте, то можем объединить некоторые шаги:

```
let funnyGuy = {
  firstName: "Conan",
  lastName: "O'Brien"
};
```

Конечный результат в таком случае будет идентичен предыдущему, в котором мы сперва создали объект `funnyGuy` и лишь затем определили в нем свойства.

Есть и еще одна деталь, касающаяся добавления свойств, на которую стоит обратить внимание. К текущему моменту мы рассмотрели различные объекты, имеющие свойства, чьи значения состоят из чисел, строк и т. д. А вы знали, что свойством объекта может являться другой объект? Это вполне возможно! Взгляните на следующий объект `colors`, чье свойство `content` содержит объект:

```
let colors = {
  header: "blue",
  footer: "gray",
  content: {
    title: "black",
    body: "darkgray",
    signature: "light blue"
  }
};
```

Объект внутри объекта определяется так же, как и свойство с использованием скобочного синтаксиса для установки значения свойства для объекта. Если мы хотим добавить свойство во вложенный объект, то можем для этого использовать те же только что полученные знания.

Допустим, мы хотим добавить свойство `frame` во вложенный объект `content`. Сделать мы это можем, например, так:

```
colors.content.frame = "yellow";
```

Начинаем с объекта `colors`, переходим к объекту `content`, а затем определяем свойство и значение, которые нам нужны. Если же для обращения к свойству `content` вы предпочтете использовать скобочную нотацию, то сделаете так:

```
colors["content"]["frame"] = "yellow";
```

Если вы хотите одновременно использовать оба вида нотации, то это тоже возможно:

```
colors.content["frame"] = "yellow";
```

В начале я говорил, что существует ряд способов для добавления свойств объекту. Мы рассмотрели один из них. Более сложный способ задействует методы `Object.defineProperty` и `Object.defineProperties`. Эти методы также позволяют вам устанавливать свойство и его значение, но при этом дают и другие возможности. Например, возможность указать, может ли свойство быть пронумеровано или может ли оно быть перенастроено и т. д. Это однозначно выходит за рамки того, что мы будем делать 99 % времени в начале обучения, но если вам это нужно, то упомянутые два метода вполне пригодятся. Документация MDN (https://mdn.dev/) приводит хорошие примеры их использования для добавления одного или нескольких свойств объекту.

Удаление свойств

Если добавление свойств могло показаться вам занятным, то их удаление несколько муторно. Но при этом оно проще. Продолжим работать с объектом `colors`:

```
let colors = {
  header: "blue",
  footer: "gray",
  content: {
    title: "black",
    body: "darkgray",
    signature: "light blue"
  }
};
```

Требуется удалить свойство `footer`. Для этого используем один из двух способов в зависимости от того, хотим мы обратиться к свойству посредством скобочной нотации или точечной:

```
delete colors.footer;

// или

delete colors["footer"];
```

Главную роль при этом играет ключевое слово `delete`. Просто используйте его, сопроводив свойством, которое хотите удалить.

Но JavaScript не был бы собой, если бы тут не содержался подвох. В данном случае он связан с производительностью. Если вы будете часто удалять большое количество свойств во множестве объектов, то использование `delete` окажется намного медленнее, чем определение значений свойств как `undefined`:

```
colors.footer = undefined;

// или

colors["footer"] = undefined;
```

Оборотная же сторона определения свойства как `undefined` в том, что оно по-прежнему остается в памяти. Вам потребуется взвесить все за и против (скорость или память) для каждой отдельной ситуации, чтобы выбрать оптимальный вариант.

Что же происходит под капотом?

Мы научились создавать объекты и производить с ними некоторые простые модификации. Так как объекты — это сердце всех возможностей JavaScript, то важно как можно лучше разобраться в происходящем. И не ради расширения багажа знаний, хоть это и помогло бы впечатлить друзей или родственников за ужином. Главная часть работы в JavaScript — это создание объектов на основе других объектов и выполнение традиционных, присущих объектному программированию действий. Все эти действия будут для вас гораздо понятнее, когда мы разберемся в том, что же происходит при работе с объектами.

Давайте вернемся к нашему объекту `funnyGuy`:

```
let funnyGuy = {};
```

Итак, что мы можем сделать с пустым объектом, не имеющим свойств? Неужели наш объект `funnyGuy` совсем одинок и изолирован от всего происходящего? В ответ эхом — *нет*. Причина скрыта в том, как со-

здаваемые в JS объекты автоматически связываются с более крупным `Object` и всей присущей ему функциональностью. Лучшим способом понять эту связь будет визуализация. Сосредоточьтесь и внимательно рассмотрите рис. 18.1.

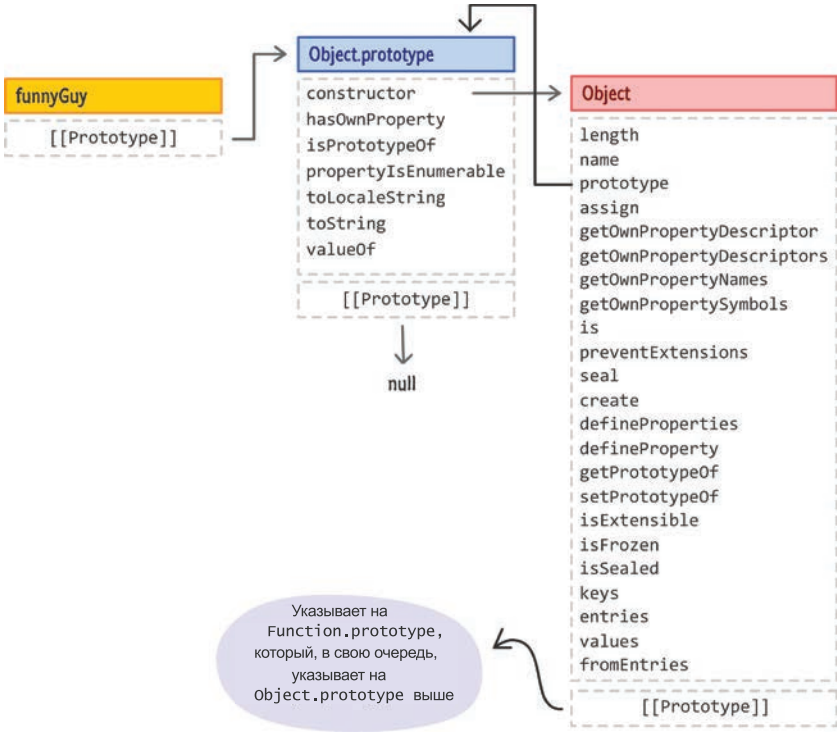


РИС. 18.1.
Что на самом деле происходит с простым, казалось бы, объектом funnyGuy

Здесь отображено, что именно происходит за кадром, когда мы создаем пустой объект `funnyGuy`.

Рассмотрение этого представления начнем с самого объекта. Здесь все по-прежнему, а вот остальное уже отличается. Мы видим, что наш `funnyGuy` — это просто пустой объект. У него нет свойств, которые мы могли бы ему определить, но есть свойства, которые определяются по умолчанию. Эти свойства связывают объект `funnyGuy` с лежащим в основе типом `Object`, не требуя для этого нашего вмешательства. Эта связь позволяет вызывать стандартные свойства `Object` для `funnyGuy`:

```
let funnyGuy = {};
funnyGuy.toString(); // [объект Object]
```

Для ясности еще раз скажу, что именно эта связь позволяет вызвать `toString` для нашего кажущегося пустым объекта `funnyGuy`. Однако называть эту связь связью не совсем точно. Эта связь в действительности известна как прототип (и зачастую представлена как `[[Prototype]]`), который в итоге указывает на другой объект. Другой объект может иметь свой собственный `[[Prototype]]`, который будет также указывать на другой объект, и т. д. Такой род связи называется цепочкой прототипов. Перемещение по цепочке прототипов — это существенная часть того, что делает JavaScript при поиске вызываемого вами свойства. В нашем случае это вызов `toString` для объекта `funnyGuy`, который визуально представлен на рис. 18.2.

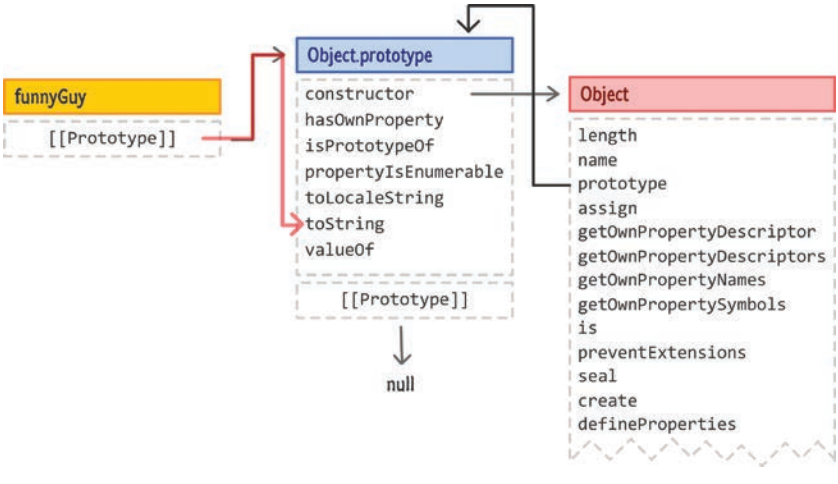


РИС. 18.2.
Переход по цепочке прототипов в поиске нужного свойства

В цепочке прототипов, даже если в нашем объекте не определено конкретное свойство, которое мы ищем, JavaScript все равно продолжит поиск по цепочке в попытке найти его в каждом последующем пункте. В нашем случае цепочка прототипов объекта `funnyGuy` состоит из прототипа самого этого объекта и `Object.prototype`, то есть является весьма простой. Когда же мы будем работать с более сложными объектами, цепочки будут становиться намного длиннее и сложнее. И вскоре мы это увидим.

ОБЪЕКТ НЕ ЯВЛЯЕТСЯ ЧАСТЬЮ ЦЕПОЧКИ ПРОТОТИПА

В предыдущих визуализациях объекта мы видели выделенные точки соединения и линии, соединяющие его свойства с `Object.prototype`. Здесь стоит заметить, что объект не является частью цепочки прототипов. Он играет роль в том, как объекты реализуют связь между их конструктором и неудачно названным свойством `prototype` (не связанным с нашим `[[Prototype]]`), и мы еще коснемся этой его роли позднее. Я продолжу показывать роль объекта в будущих реализациях объектов, но помните, что он не принимает участия в проходе по цепочке прототипов.

Далее, как мы видим, наш объект `funnyGuy` очень прост. Давайте для интереса добавим в него свойства `firstName` и `lastName`:

```
let funnyGuy = {
  firstName: "Conan",
  lastName: "O'Brien"
};
```

На рис. 18.3 показано, как будет выглядеть наша прежняя визуализация при участии добавленных свойств.

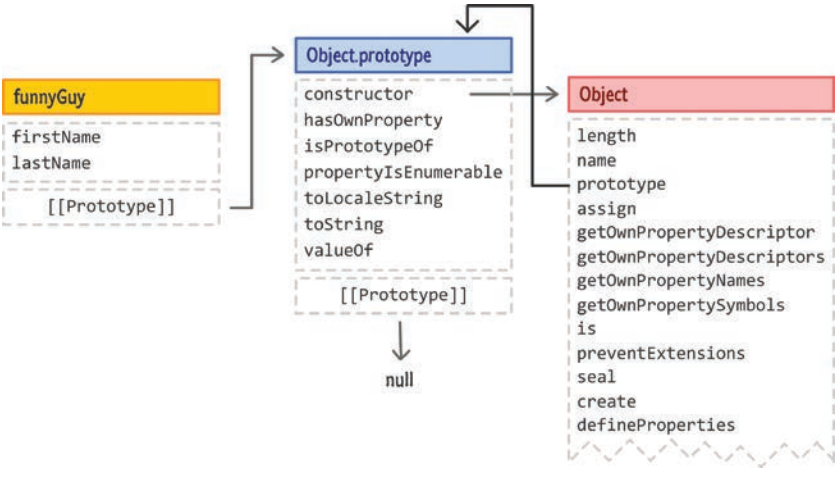


РИС. 18.3.
Поздоровайтесь с нашими старыми знакомыми firstName и lastName

Свойства `firstName` и `lastName` являются частью объекта `funnyGuy` и также представлены. Покончив с рассмотрением этих основ объекта, мы можем переходить к подробностям.

Создание пользовательских объектов

Работа с обобщенным объектом `Object` и добавление в него свойств служит определенной цели, но вся его прелесть быстро исчезает, когда мы создаем много одинаковых в основе объектов. Взгляните на этот фрагмент:

```
let funnyGuy = {
  firstName: "Conan",
  lastName: "O'Brien",

  getName: function () {
    return "Name is: " + this.firstName + " " + this.lastName;
  }
};

let theDude = {
  firstName: "Jeffrey",
  lastName: "Lebowski",

  getName: function () {
    return "Name is: " + this.firstName + " " + this.lastName;
  }
};

let detective = {
  firstName: "Adrian",
  lastName: "Monk",

  getName: function () {
    return "Name is: " + this.firstName + " " + this.lastName;
  }
};
```

Этот код создает объект `funnyGuy` и вводит два новых очень похожих на него объекта `theDude` и `detective`. Наша визуализация всего этого теперь будет выглядеть, как показано на рис. 18.4.

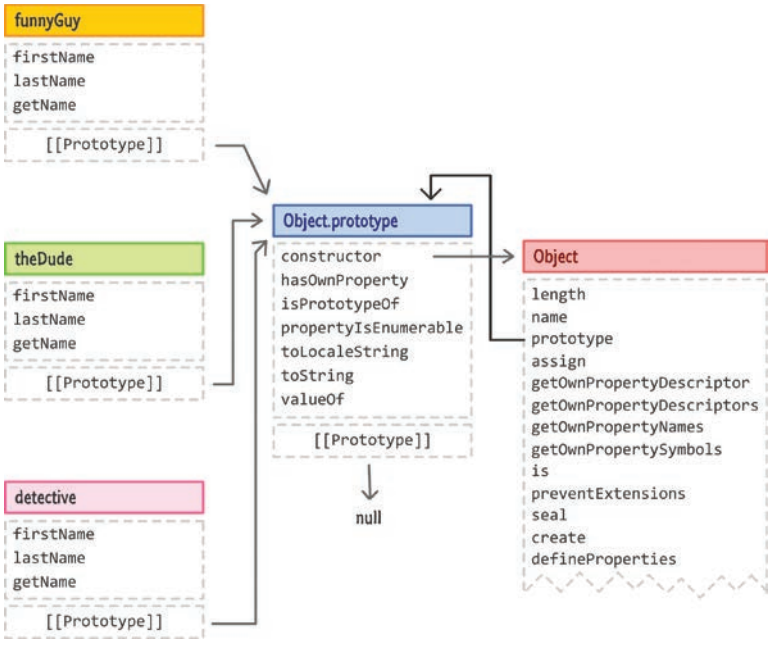


РИС. 18.4.
Каждый вновь созданный объект расширяется от Object.prototype

На первый взгляд кажется, что здесь многовато повторений. Каждый из только что созданных объектов содержит свою собственную копию свойств `firstName`, `lastName` и `getName`. Итак, все же повторение — это не всегда плохо. Да, есть противоречие тому, что я утверждал ранее, но дайте-ка я все объясню. В случае с объектами нужно выяснить, какие свойства имеет смысл повторять, а какие нет. В нашем примере свойства `firstName` и `lastName` будут, как правило, уникальны для каждого объекта, а значит, это повторение имеет смысл. А вот свойство `getName` хоть и выступает в роли помощника, но не содержит ничего, что отдельный объект мог бы определить уникально:

```
getName: function () {
  return "Name is: " + this.firstName + " " + this.lastName;
}
```

В этом случае его повторение ни к чему, следовательно, нам стоит сделать его общедоступным и избежать повторения. И как же?

Что ж... Для этого есть прямой путь, а именно создание промежуточного *родительского* объекта, содержащего общие свойства. В свою очередь,

наши *дочерние* объекты смогут наследовать от этого родительского объекта вместо наследования напрямую от `Object`. Для большей конкретики мы создадим объект `person`, содержащий свойство `getName`. Наши объекты `funnyGuy`, `theDude` и `detective` станут наследниками `person`. Упорядоченная таким образом структура обеспечит, чтобы все свойства, требующие повторения, были повторены, а требующие совместного использования использовались совместно. Лучше понять все сказанное поможет рис. 18.5, где эти действия изображены наглядно.

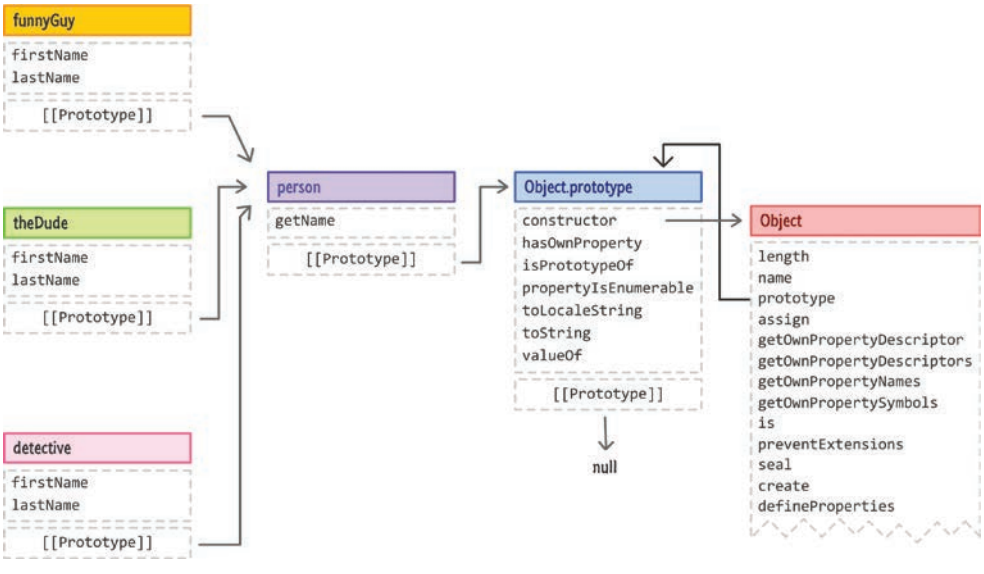


РИС. 18.5.
Добавление промежуточного объекта person со свойством (теперь используемым совместно) getName

Заметьте, что теперь `person` стал частью цепочки прототипов, удачно расположившись между `Object.prototype` и нашими дочерними объектами. Как же это делается? Один из подходов мы уже видели ранее, и в нем мы опираемся на `Object.create`. При использовании `Object.create` мы можем указать объект, на основе которого требуется создать новый объект. Например:

```
let myObject = Object.create(fooObject);
```

Когда мы это делаем, за кадром происходит следующее: *прототип* нашего объекта `myObject` теперь будет `fooObject`. При этом он становится

частью цепочки прототипов. Теперь, когда мы сделали крюк и расширили наше понимание `Object.create`, освоив содержание этой главы. Давайте вернемся к изначальному вопросу о том, как же именно наши объекты `funnyGuy`, `theDude` и `detective` наследуют от `person`.

Код, осуществляющий все это, будет таким:

```
let person = {
  getName: function () {
    return "The name is " + this.firstName + " " + this.lastName;
  }
};

let funnyGuy = Object.create(person);
funnyGuy.firstName = "Conan";
funnyGuy.lastName = "O'Brien";

let theDude = Object.create(person);
theDude.firstName = "Jeffrey";
theDude.lastName = "Lebowski";

let detective = Object.create(person);
detective.firstName = "Adrian";
detective.lastName = "Monk";
```

Принцип работы цепочки прототипов позволяет нам вызывать `getName` для любого из наших объектов `funnyGuy`, `theDude` или `detective`, что приведет к ожидаемому результату:

```
detective.getName(); // Имя Adrian Monk
```

Если мы решим расширить объект `person`, то достаточно сделать это всего один раз, и это также отразится на всех наследующих от него объектах, не требуя дополнительного повторения. Предположим, мы хотим добавить метод `getInitials`, возвращающий первую букву из имени и фамилии:

```
let person = {
  getName: function () {
    return "The name is " + this.firstName + " " + this.lastName;
  },
  getInitials: function () {
    if (this.firstName && this.lastName) {
      return this.firstName[0] + this.lastName[0];
    }
  }
};
```

Мы добавляем метод `getInitials` в объект `person`. Чтобы использовать этот метод, можем вызвать его для любого объекта, расширяющего `person`, например `funnyGuy`:

```
funnyGuy.getInitials(); // CO
```

Такая возможность создавать промежуточные объекты, помогающие разделять функциональность кода, является мощным инструментом. Она повышает эффективность создания объектов и добавления в них функциональности. Неплохо, правда?

Ключевое слово this

В предыдущих фрагментах кода вы могли заметить использование ключевого слова `this`, особенно в случае с объектом `person`, где мы задействовали его для обращения к свойствам, созданным в его потомках, а не к его собственным. Давайте вернемся к этому объекту, а в частности к его свойству `getName`:

```
let person = {
  getName: function () {
    return "The name is " + this.firstName + " " + this.lastName;
  },
  getInitials: function () {
    if (this.firstName && this.lastName) {
      return this.firstName[0] + this.lastName[0];
    }
  }
};
```

Когда мы вызываем `getName`, то возвращаемое имя будет зависеть от того, из какого объекта мы это делаем. Например, если мы сделаем следующее:

```
let spaceGuy = Object.create(person);
spaceGuy.firstName = "Buzz";
spaceGuy.lastName = "Lightyear";

console.log(spaceGuy.getName()); // Buzz Lightyear
```

При выполнении этого кода мы увидим в консоли `Buzz Lightyear`. Если мы еще раз взглянем на свойство `getName`, то увидим, что там нет свойств `firstName` и `lastName` в объекте `person`. Но как мы видели ранее, если свойство не существует, мы переходим далее по цепочке от родителя к родителю, как показано на рис. 18.6.

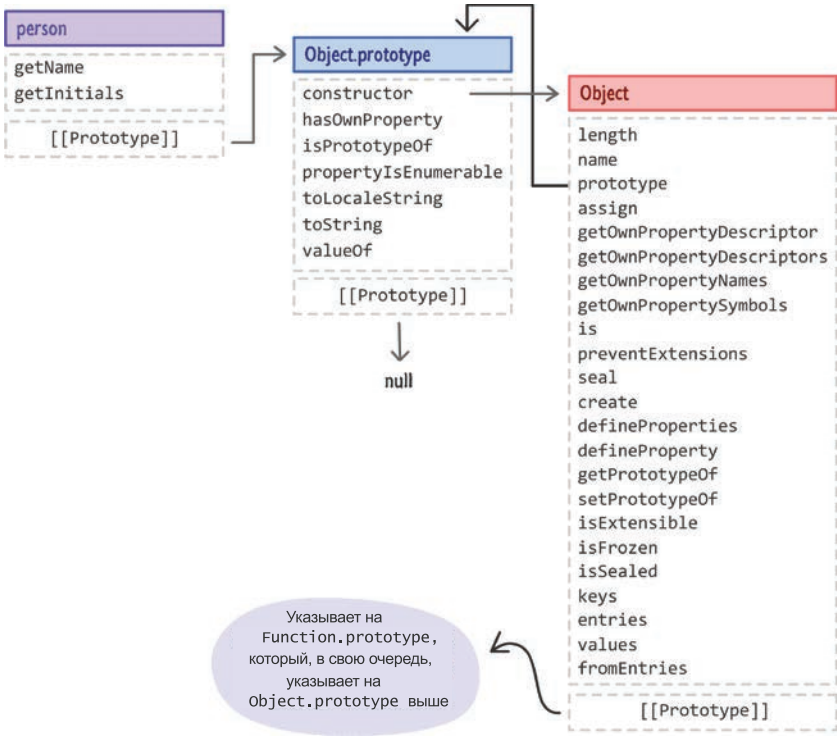


РИС. 18.6.
Цепочка прототипов для объекта person

В нашем случае единственной остановкой в цепочке будет `Object.prototype`, но в нем также не обнаруживаются свойства `firstName` и `lastName`. Как же тогда метод `getName` умудряется сработать и вернуть нужные значения?

Ответ заключается в ключевом слове `this`, предшествующем `firstName` и `lastName` в инструкции `return` метода `getName`:

```
let person = {
  getName: function () {
    return "The name is " + this.firstName + " " + this.lastName;
  },
  getInitials: function () {
    if (this.firstName && this.lastName) {
      return this.firstName[0] + this.lastName[0];
    }
  }
};
```

Ключевое слово `this` ссылается на объект, к которому привязан наш метод `getName`. В данном случае объектом является `spaceGuy`, так как именно его мы используем в качестве *точки входа* в этот совершенный процесс навигации между прототипами (рис. 18.7).

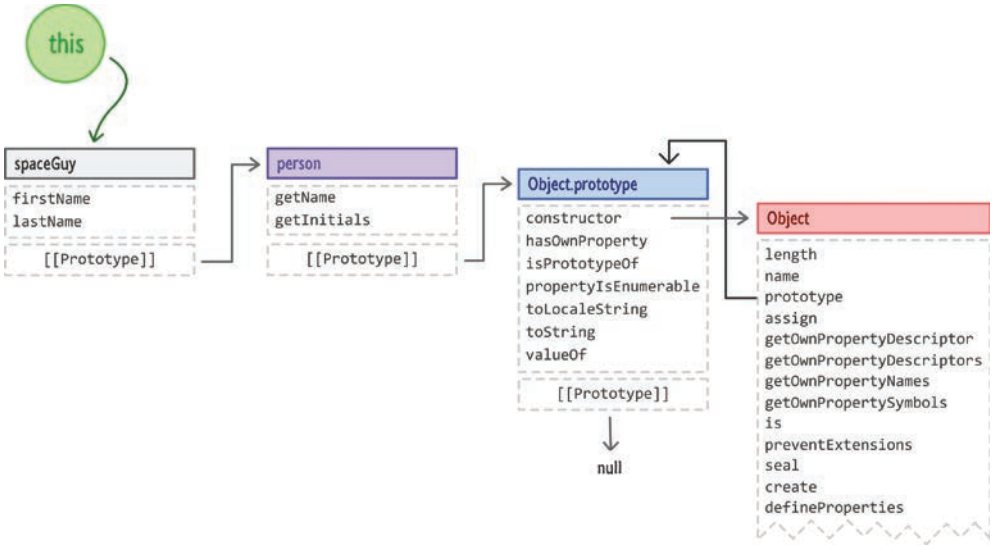


РИС. 18.7.
Ключевое слово this ссылается на spaceGuy!

Когда происходит вычисление метода `getName` и свойства `firstName` и `lastName` должны разрешиться, поиск начинается там, куда указывает ключевое слово `this`. Это означает, что наш поиск начинается с объекта `spaceGuy`, который, как выясняется, содержит свойства `firstName` и `lastName`. Именно поэтому мы получаем верный результат при вызове кода для `getName` (а также и `getInitials`).

Понимание, на что ссылается ключевое слово `this`, скрыто под галлонами пролитых чернил, и полноценное рассмотрение этого вопроса выходит далеко за рамки того, о чем мы собираемся говорить. Но хорошо то, что пройденного материала вам уже будет достаточно, чтобы решать большинство задач.

КОРОТКО О ГЛАВНОМ

Из-за неразберихи вокруг объектной ориентированности в JavaScript разумным было сделать рассмотрение этой темы глубоким и обширным, как мы и поступили. Многое из того, что было затронуто здесь, прямо или косвенно связано с наследованием — когда объекты разветвляются и основываются на других объектах. В отличие от классических языков, использующих классы как шаблоны для объектов, в JavaScript понятия классов, строго говоря, не существует. Здесь используется так называемая *модель наследования прототипов*. Вы не инстанцируете объекты из шаблона. Вместо этого вы создаете их либо заново, либо, чаще всего, копированием или клонированием другого объекта. JavaScript попадает в ту самую серую область, где не соответствует классической форме языка, но при этом имеет подобные классам конструкции (некоторые из них вы увидите позже), которые позволяют ему сидеть за одним столом с классическими языками. Не хочу здесь увлекаться навешиванием ярлыков.

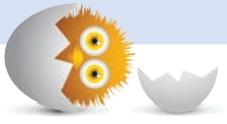
Среди всего этого множества страниц я постарался сгруппировать новую функциональность JavaScript для работы с объектами и их расширения для ваших дальнейших нужд. Тем не менее еще многое предстоит рассмотреть, поэтому сделайте перерыв, и мы в ближайшем будущем затронем более интересные темы, которые дополнят пройденное более мощными и выразительными возможностями.

Дополнительные ресурсы и примеры:

- Понимание прототипов в JS: http://bit.ly/kirupaJSPrototypes
- Простое английское руководство по прототипам JS: http://bit.ly/kirupaPrototypesGuide
- Как работает `prototype`? http://bit.ly/kirupaPrototypeWork
- Это большая и странная тема, поэтому обращайтесь на форум https://forum.kirupa.com, если столкнетесь с какими-либо сложностями.

В ЭТОЙ ГЛАВЕ:

- научимся расширять функциональность объектов;
- лучше разберемся в цепочке прототипов.



19

РАСШИРЕНИЕ ВСТРОЕННЫХ ОБЪЕКТОВ

Как нам уже хорошо известно, JavaScript поставляется с богатым арсеналом встроенных объектов. Эти объекты обеспечивают некоторую базовую функциональность для работы с текстом, числами, коллекциями данных, датами и многим другим. Однако по мере углубления в этот язык, когда вы уже начинаете реализовывать более интересные и продуманные вещи, возникает желание выйти за рамки возможностей встроенных объектов.

Давайте взглянем на пример, демонстрирующий подобную ситуацию. В нем показано, как мы можем перемешивать содержимое массива:

```
function shuffle(input) {
  for (let i = input.length – 1; i >= 0; i--) {

    let randomIndex = Math.floor(Math.random() * (i + 1));
    let itemAtIndex = input[randomIndex];

    input[randomIndex] = input[i];
    input[i] = itemAtIndex;
  }
  return input;
}
```

Мы используем функцию `shuffle`, просто вызвав ее и передав массив, чье содержимое нужно перемешать:

```
let shuffleArray = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];
shuffle(shuffleArray);

// и результат…
console.log(shuffleArray);
```

После выполнения этого кода конечным результатом будет перегруппировка содержимого. Такая функциональность весьма полезна. Я бы даже сказал, что слишком полезна. Возможность производить перемешивание должна быть частью объекта `Array` и являться легко доступной наряду с такими его методами, как `push`, `pop`, `slice` и др.

Если бы функция `shuffle` была частью объекта `array`, то мы могли бы с легкостью использовать ее следующим образом:

```
let shuffleArray = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];
shuffleArray.shuffle();
```

В этом примере мы расширяем встроенный объект (`Array`) заданной нами функциональностью (`shuffle`). В нескольких последующих разделах мы конкретно рассмотрим, как это делается, как работает и почему расширение встроенных объектов является спорным решением.

Поехали!

И снова приветствуем прототип!

Расширение встроенного объекта новой функциональностью звучит сложно, но на деле, как только вы поймете, что нужно сделать, это окажется достаточно просто. Для простоты усвоения этого материала мы рассмотрим комбинацию образца кода и диаграмм с участием дружелюбно настроенного объекта `Array`:

```
let tempArray = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];
```

Если бы мы построили диаграмму всей иерархии объекта `tempArray`, то выглядела бы она, как показано на рис. 19.1.

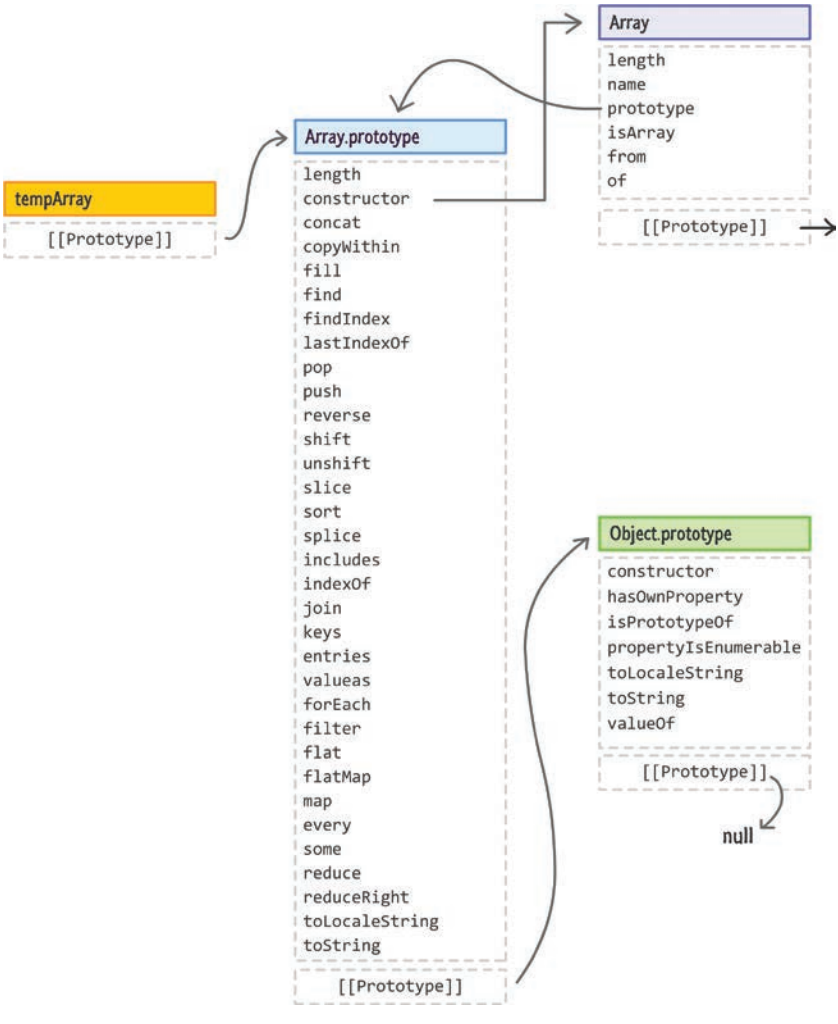


РИС. 19.1.
Паутина объектов (или лжи!), которые существуют под поверхностью

Слева у нас объект `tempArray`, являющийся экземпляром `Array.prototype`, который, в свою очередь, является экземпляром основного `Object.prototype`. Теперь нам нужно расширить возможности нашего массива функцией `shuffle`. Это означает, что нужно найти способ внедрить эту функцию в `Array.prototype`, как показывает рис. 19.2.

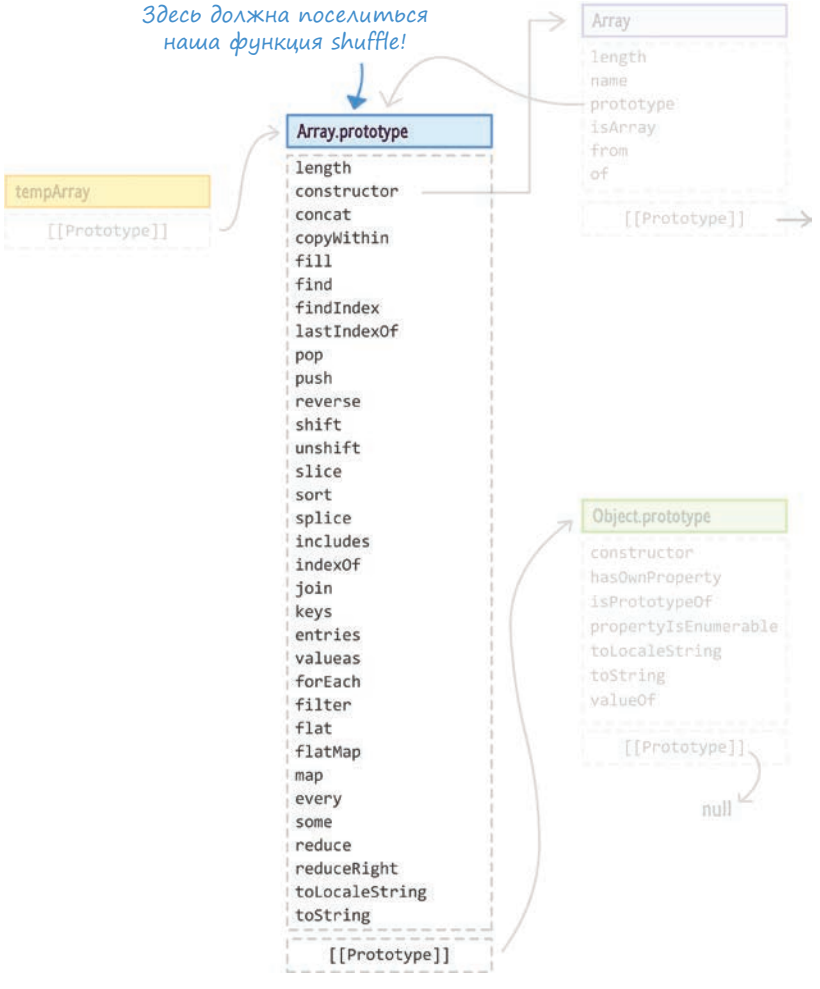


РИС. 19.2.
Здесь должна поселиться наша функция shuffle!

Здесь мы сталкиваемся с проявлением пресловутой странности JavaScript. У нас нет доступа к коду, формирующему функциональность массива. Мы также не можем найти функцию или объект, формирующие сам `Array`, и внедрить `shuffle` в них, как это делается в случае с пользовательским объектом. Наши встроенные объекты, подобные `Array`, определены в вулканических глубинах браузера, куда ни одно человеческое существо не может попасть. Поэтому здесь нам нужен иной подход.

При этом *другом подходе* мы тайком прокрадываемся и прикрепляем нужную функциональность к свойству `prototype` объекта `Array`. Выглядит это примерно так:

```
Array.prototype.shuffle = function () {
  let input = this;

  for (let i = input.length – 1; i >= 0; i--) {

    let randomIndex = Math.floor(Math.random() * (i + 1));
    let itemAtIndex = input[randomIndex];

    input[randomIndex] = input[i];
    input[i] = itemAtIndex;
  }
  return input;
}
```

Обратите внимание, что наша функция `shuffle` объявлена в `Array.prototype`. Как часть этого прикрепления мы внесли небольшое изменение в работу функции. Теперь она не получает аргумент для обращения к массиву, который нужно перемешать:

```
function shuffle(input) {
  .
  .
  .
  .
  .
}
```

Вместо этого, так как отныне функция является частью `Array`, на этот массив указывает ключевое слово `this` внутри ее тела:

```
Array.prototype.shuffle = function () {
  let input = this;
  .
  .
  .
  .
}
```

Возвращаясь к предыдущему шагу, как только этот код будет запущен, функция `shuffle` окажется бок о бок со встроенными методами, которые объект `Array` выражает через `Array.prototype`, как показано на рис. 19.3.

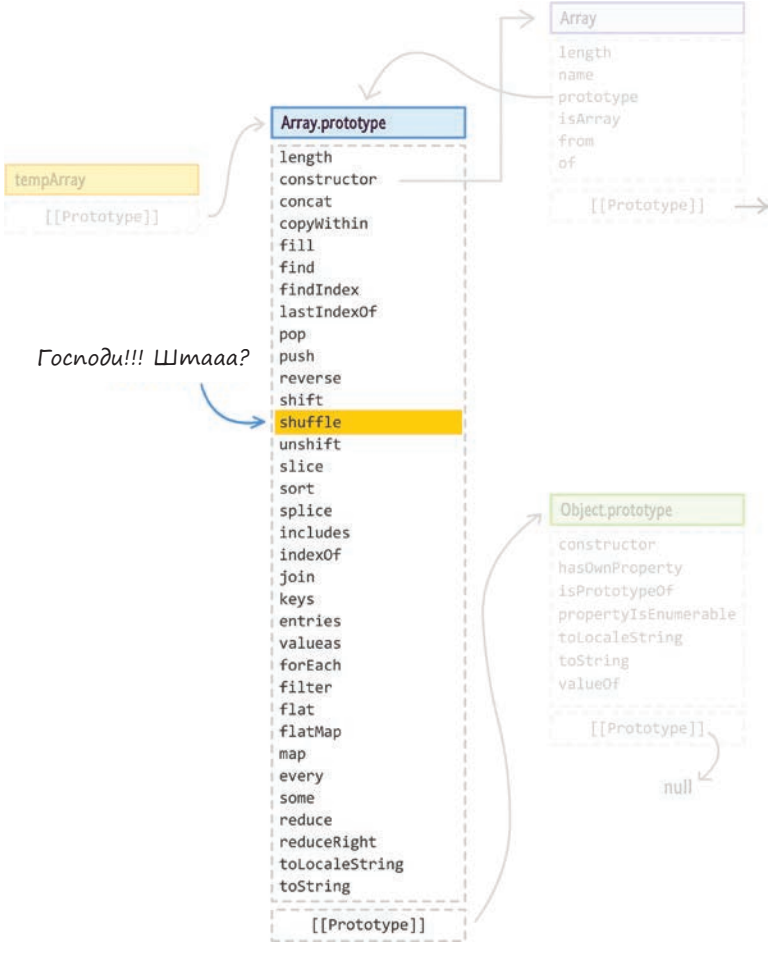


РИС. 19.3.
Великий успех! Теперь функция shuffle на своем месте

С этого момента, если нам понадобится обратиться к возможностям `shuffle`, мы можем использовать для этого изначально желаемый подход:

```
let numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];
numbers.shuffle();
```

Самое лучшее в этом то, что создаваемые нами массивы будут также по умолчанию иметь доступ к функциональности `shuffle` благодаря принципам работы наследования прототипов.

Спорность расширения встроенных объектов

Учитывая, насколько просто расширить функциональность встроенного объекта, объявляя методы и свойства с помощью свойства `prototype`, легко представить себе, что все обожают такую возможность. Но как выясняется, расширение встроенных объектов отчасти спорно. Причины этого витают рядом.

Вы не контролируете будущее встроенного объекта

Ничто не мешает будущей реализации JavaScript включить собственную версию `shuffle`, применимую к объектам `Array`. В таком случае у вас возникнет коллизия, когда ваша версия `shuffle` окажется в конфликте с браузерной версией `shuffle`, особенно если их поведение или производительность сильно различаются.

Некоторую функциональность не следует расширять или переопределять

Ничто не мешает вам использовать полученные здесь знания для изменения существующих методов и свойств. Например, в следующем примере я меняю поведение `slice`:

```
Array.prototype.slice = function () {
  let input = this;
  input[0] = "This is an awesome example!";

. return input;
}

let tempArray = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10] tempArray.slice();

// и результат будет...
console.log(tempArray);
```

Несмотря на то что это ужасный пример, он прекрасно показывает, как легко оказалось нарушить существующую функциональность.

ЧТО ПОЧИТАТЬ

Подробное обсуждение этого противоречия ищите на ветке StackOverflow: http://stackoverflow.com/questions/8859828/.

КОРОТКО О ГЛАВНОМ: ЧТО ЖЕ МНЕ ДЕЛАТЬ?

Мой ответ на этот вопрос будет прост: *пользуйтесь здравым смыслом*! Я обозначил всего лишь два случая из множества, которые люди обсуждают в связи с темами, касающимися расширения встроенных объектов. По большей части возражения имеют реальные основания. Вам же при этом стоит себя спросить: «Относятся ли эти возражения к моему сценарию?» Смею предположить, что нет.

Лично я никогда не имел проблем при расширении встроенных объектов нужной мне функциональностью. Эту функцию перемешивания я написал много лет назад, и ни один браузер по сей день даже близко не реализовал ее альтернативы. При этом я не жалуюсь. Я тестирую всю добавляемую функциональность и убеждаюсь, что она полноценно работает в интересующих меня браузерах, на которые я нацелен. До тех пор пока вы будете проводить обширное тестирование (для одной или двух последних версий наиболее популярных браузеров), наверняка все будет в порядке.

Если же вы беспокоитесь о будущем вашего приложения, называйте свойства или методы таким образом, чтобы их могло использовать только ваше приложение. Например, шансы, что функция `Array.prototype.kirupaShuffle` будет введена в какой-либо браузер, стремятся к нулю.

Теперь же, когда мы подробно изучили некоторые темы, касающиеся объектов, давайте вернемся к рассмотрению и других типов, с которыми вам предстоит работать, и уже затем будем переходить к действительно потрясным вещам.

Если у вас есть вопросы по расширению объектов или вы просто хотите поговорить о жизни, обращайтесь на форум https://forum.kirupa.com.

В ЭТОЙ ГЛАВЕ:

- узнаем, что такое классы в мире JavaScript;
- научимся проще создавать объекты с помощью синтаксиса классов;
- поймем роль конструктора и других конструкций класса.

20

ИСПОЛЬЗОВАНИЕ КЛАССОВ

Мы уже рассмотрели множество основных аспектов работы с объектами. Мы видели, как они создаются, изучили наследование прототипов и даже взглянули на темное искусство расширения объектов. При этом мы работали на очень низком уровне и были замешаны в процессе изготовления самого объекта. Это здорово для качественного понимания происходящего, но не так здорово, когда в вашем приложении появляется сложный объект. В целях упрощения всего этого в ES6-версии JavaScript появилась поддержка так называемых *классов*.

Те из вас, у кого есть опыт работы в других объектно ориентированных языках, вероятно, знакомы с этим термином. Если же нет, то не стоит беспокоиться. В мире JavaScript классы не представляют собой ничего особенного. Здесь они не более чем горстка новых ключевых слов и условных конструкций, *упрощающих набор команд* при работе с объектами. В ближайших разделах мы опробуем все это на себе.

Поехали!

Синтаксис классов и создание объектов

Будем осваивать синтаксис классов дедовским способом — через написание кода. Так как рассмотреть предстоит многое, не будем хвататься за все сразу, а начнем с применения синтаксиса классов при создании объектов. Как вы увидите, здесь замешано множество всего, и нам будет над чем потрудиться.

Создание объекта

Вы можете рассматривать класс как шаблон — шаблон, на который ссылаются объекты при создании. Предположим, что мы хотим создать класс `Planet`. Максимально простая версия этого класса будет выглядеть так:

```
class Planet {

}
```

Мы используем ключевое слово `class`, сопровождаемое именем, которое мы хотим задать нашему классу. Тело этого класса будет содержаться внутри фигурных скобок `{ }`. Очевидно, что на данный момент класс пуст. Пока это нормально, так как начинаем мы с самого простого.

Для создания объекта на основе этого класса вам всего лишь нужно сделать следующее:

```
let myPlanet = new Planet();
```

Мы объявляем имя нашего объекта и используем ключевое слово `new` для создания (то есть инстанцирования) объекта на основе класса `Planet`. Рисунок 20.1 демонстрирует наглядно, что именно происходит за кадром.

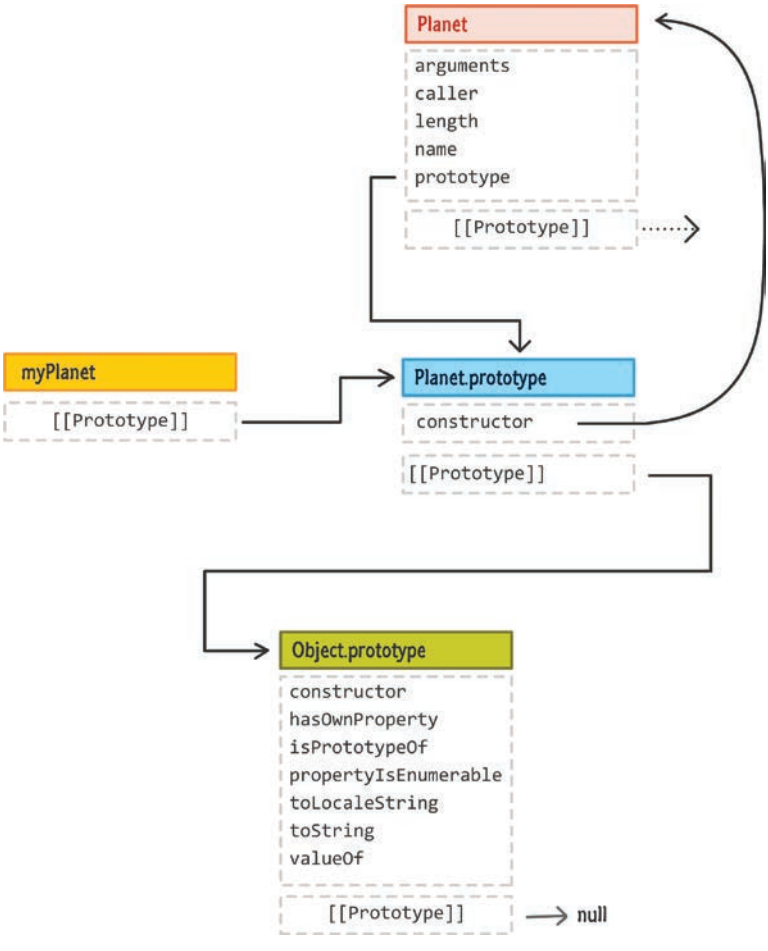


РИС. 20.1.
Внутренний процесс при создании myPlanet

Это представление несколько отличается от того, что мы видели при создании объектов с помощью `Object.create()`. Разница заключается в создании объекта `myPlanet` с помощью ключевого слова `new`. При создании объектов с помощью `new` происходит следующее:

1. Новый объект имеет тип `Planet`.
2. `[[Prototype]]` нашего нового объекта является новой функцией или свойством класса `prototype`.
3. Выполняется функция-конструктор, которая занимается инициализацией нашего созданного объекта.

Не стану утомлять вас излишними дополнительными деталями, но среди них есть одна важная, с которой мы далее познакомимся. Она связана с так называемым *конструктором*, упомянутым в пункте 3.

Знакомьтесь с конструктором

Конструктор — это функция (или метод), существующий внутри тела класса. Он отвечает за инициализацию создаваемых объектов, и делает он это, выполняя содержащийся в нем код, во время самого процесса создания. Эта деталь является обязательной. Все классы должны быть оснащены функцией-конструктором. Если у вашего класса таковой не имеется (как у `Planet`), JavaScript автоматически создаст пустой конструктор за вас.

Теперь давайте определим конструктор для нашего класса `Planet`. Взгляните на следующую модификацию:

```
class Planet {
  constructor(name, radius) {
    this.name = name;
    this.radius = radius;
  }
}
```

Для определения конструктора мы используем особое ключевое слово `constructor`, чтобы создать то, что по сути является функцией. Так как это функция, вы можете, как обычно, указать любые аргументы, которые хотите использовать. В нашем случае в виде аргументов мы указываем значения `name` и `radius` и используем их, чтобы установить свойства `name` и `radius` в нашем объекте:

```
class Planet {
  constructor(name, radius) {
    this.name = name;
    this.radius = radius;
  }
}
```

Вы можете совершать гораздо больше (или меньше) интересных действий изнутри конструктора, главное не забывать, что этот код будет выполняться каждый раз, когда мы будем создавать новый объект, используя класс `Planet`. Кстати говоря, вот как вы можете вызвать класс `Planet` для создания объекта:

```
let myPlanet = new Planet("Earth", 6378);
console.log(myPlanet.name); // Earth
```

Обратите внимание, что два аргумента, которые нам нужно указать в конструкторе, в действительности указаны в самом классе `Planet`. Когда создается наш объект `myPlanet`, запускается конструктор и значения `name` и `radius`, переданные ранее, устанавливаются в этом объекте. Рисунок 20.2 показывает, как это выглядит.

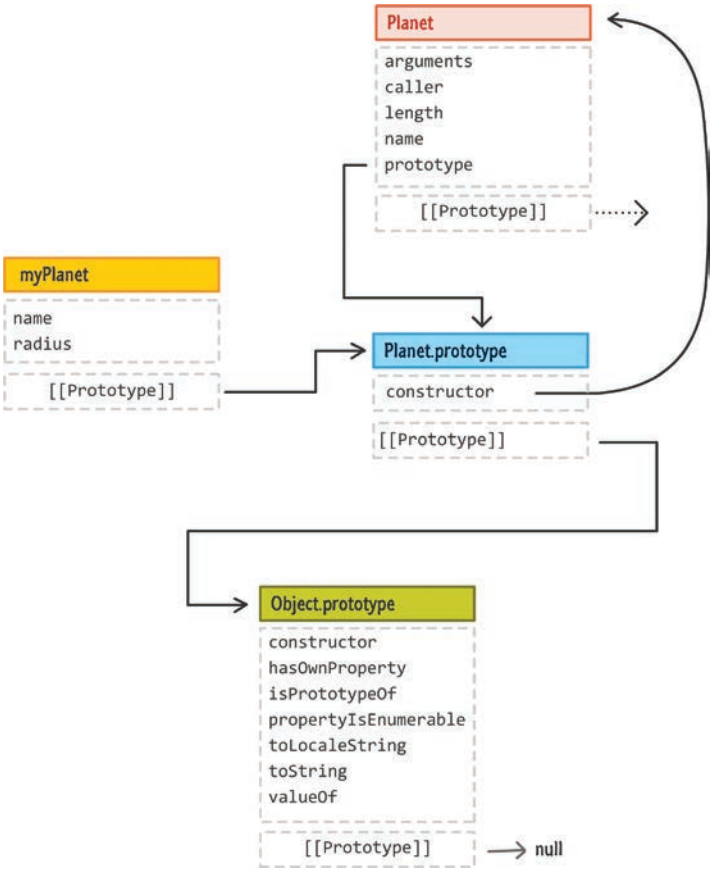


РИС. 20.2.
Наш объект myPlanet содержит свойства name и radius

Хоть мы и изучаем синтаксис `class` и окружающие его детали, всегда помните, что все это лишь посыпка — изысканный синтаксический сахар, разработанный для облегчения вашей жизни. Если не использовать синтаксис `class`, то можно сделать так, например:

```
function Planet(name, radius) {
  this.name = name;
  this.radius = radius;
};

let myPlanet = new Planet("Earth", 6378);
console.log(myPlanet.name); // Земля
```

Конечный результат почти что идентичен тому, что мы получили с помощью `class`. Единственное отличие — в средствах достижения этого результата. Тем не менее не дайте этому сравнению сбить вас с верного пути, так как другие полезные варианты использования синтаксиса `class` уже не получится столь же легко преобразовать с помощью традиционных подходов, как мы сделали в этом примере.

Что помещается в класс

Объекты `class` очень похожи на функции, но имеют свои причуды. Один из помещаемых внутрь класса элементов мы уже видели — это особая функция `constructor`. Помимо нее в него можно поместить только другие *функции* и *методы*, а также *геттеры* и *сеттеры*. Все. Никаких объявлений и инициализаций переменных не допускается.

Чтобы все это увидеть в действии, давайте добавим функцию `getSurfaceArea`, которая выводит в консоль площадь нашей планеты. Внесите в код следующие изменения:

```
class Planet {
  constructor(name, radius) {
    this.name = name;
    this.radius = radius;
  }

  getSurfaceArea() {
    let surfaceArea = 4 * Math.PI * Math.pow(this.radius, 2);
    console.log(surfaceArea + " square km!");
    return surfaceArea;
  }
}
```

Вызовите `getSurfaceArea` из созданного объекта, чтобы увидеть ее в деле:

```
let earth = new Planet("Earth", 6378);
earth.getSurfaceArea();
```

После выполнения этого кода вы увидите в консоли что-то вроде 511 миллионов квадратных километров. Хорошо. Поскольку мы упомянули, что в тело класса могут быть помещены геттеры и сеттеры, давайте их также добавим. Используем же мы их, чтобы представить гравитацию планеты:

```
class Planet {
  constructor(name, radius) {
    this.name = name;
    this.radius = radius;
  }

  getSurfaceArea() {
    let surfaceArea = 4 * Math.PI * Math.pow(this.radius, 2);
    console.log(surfaceArea + " square km!");
    return surfaceArea;
  }

  set gravity(value) {
    console.log("Setting value!");
    this._gravity = value;
  }

  get gravity() {
    console.log("Getting value!");
    return this._gravity;
  }
}

let earth = new Planet("Earth", 6378);
earth.gravity = 9.81;
earth.getSurfaceArea();

console.log(earth.gravity) // 9.81
```

Вот и все. Такое добавление элементов в тело класса хорошо тем, что они *не будут существовать в созданном объекте*. Вместо этого они будут находиться в прототипе (`Planet.prototype`), как показано на рис. 20.3.

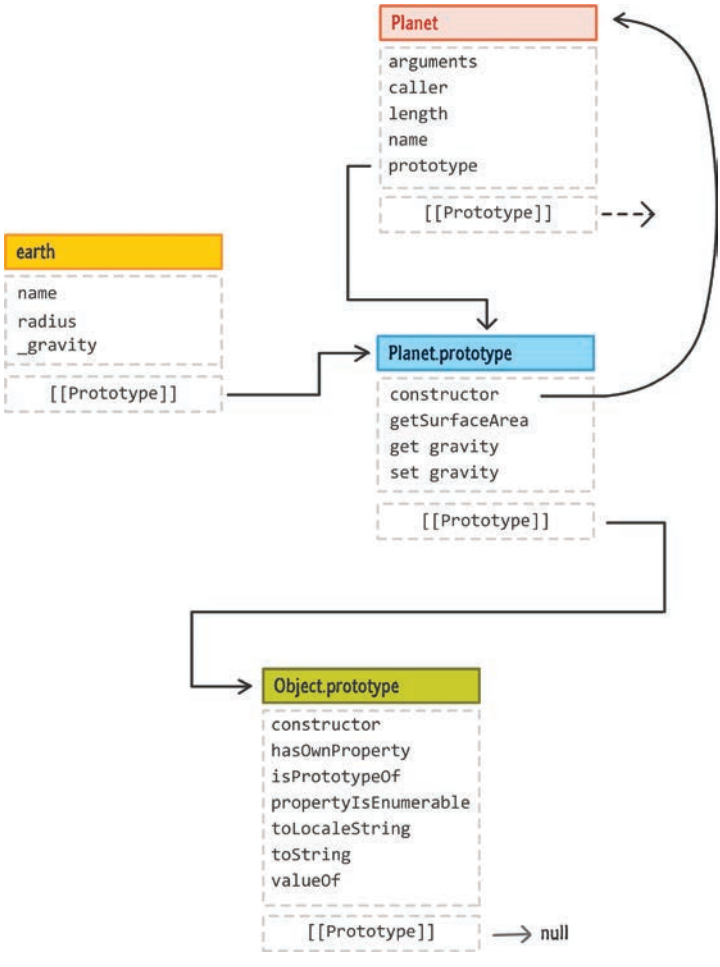


РИС. 20.3.
Нам не нужно делать ничего особенного, чтобы обратиться к объекту-прототипу

Это очень хорошо, так как нам не нужно, чтобы каждый объект без необходимости носил в себе копию содержимого класса, когда с этим прекрасно справляется совместно используемый экземпляр. Наши геттер и сеттер `gravity` наряду с функцией `getSurfaceArea` полностью существуют в прототипе.

ПОЧЕМУ ФУНКЦИИ ВНУТРИ КЛАССА ВЫГЛЯДЯТ СТРАННО?

Вы могли заметить, что функции внутри класса выглядят несколько необычно. К примеру, в них не достает ключевого слова `function`. Эта странность (в данном случае) не связана с самими классами. Дело в том, что при определении функций внутри объектов можно использовать упрощенный синтаксис.

Вместо написания, например, этого:

```
let blah = {
  zorb: function() {
    // что-то интересное
  }
};
```

вы можете сократить определение функции `zorb` до следующего:

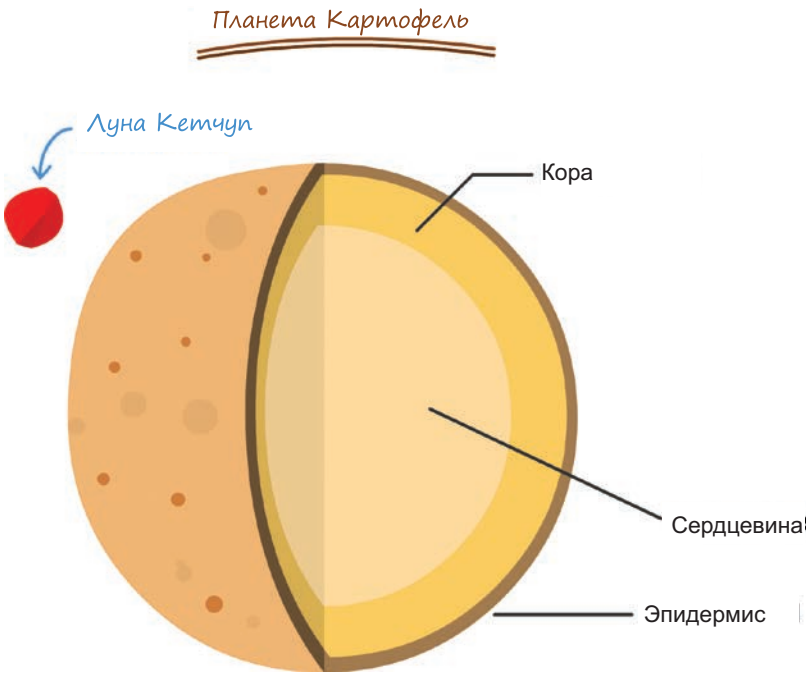
```
let blah = {
  zorb() {
    // что-то интересное
  }
};
```

Именно такую сокращенную форму вы будете встречать и использовать при определении функций внутри тела класса.

Расширение объектов

Последнее, что мы рассмотрим, связано с расширением объектов в мире классов. Чтобы разобраться в этой теме, мы будем работать с совершенно новым типом планеты, известным как `Potato Planet` (планета Картофель).

Планета Картофель содержит все, что присуще обычной планете, но состоит она полностью из картофеля, в противоположность расплавленным камням и газу, составляющим другие виды планет. Наша задача определить планету Картофель как класс. Ее функциональность будет, по большому счету, отражать представленную в классе `Planet`, но мы также добавим некоторые дополнительные элементы вроде аргумента `potatoType` в конструкторе и метода `getPotatoType`, выводящего в консоль значение `potatoType`.



Не самым лучшим подходом было бы определить класс Картофеля так:

```
class PotatoPlanet {
  constructor(name, radius, potatoType) {
    this.name = name;
    this.radius = radius;
    this.potatoType = potatoType;
  }

  getSurfaceArea() {
    let surfaceArea = 4 * Math.PI * Math.pow(this.radius, 2);
    console.log(surfaceArea + " square km!");
    return surfaceArea;
  }

  getPotatoType() {
    var thePotato = this.potatoType.toUpperCase() + "!!1!!!";
    console.log(thePotato);
    return thePotato;
  }

  set gravity(value) {
    console.log("Setting value!");
    this._gravity = value;
  }
```

```
  get gravity() {
    return this._gravity;
  }
}
```

У нас есть класс `PotatoPlanet`, и он содержит не только новые связанные с Картофелем элементы, но также всю функциональность класса `Planet`. Плохо в этом подходе то, что мы повторяем код. А что, если бы вместо повторения кода у нас была возможность расширить функциональность, предоставляемую нашим классом `Planet`, функциональностью, необходимой для `PotatoPlanet`? Такой подход будет однозначно лучше. К счастью, эта возможность у нас есть, и предоставлена она в виде ключевого слова `extends`. Расширив класс `Planet` классом `PotatoPlanet`, мы можем сделать следующее:

```
class Planet {
  constructor(name, radius) {
    this.name = name;
    this.radius = radius;
  }

  getSurfaceArea() {
    let surfaceArea = 4 * Math.PI * Math.pow(this.radius, 2);
    console.log(surfaceArea + " square km!");
    return surfaceArea;
  }

  set gravity(value) {
    console.log("Setting value!");
    this._gravity = value;
  }

  get gravity() {
    return this._gravity;
  }
}
class PotatoPlanet extends Planet {
  constructor(name, width, potatoType) {
    super(name, width);
    this.potatoType = potatoType;
  }

  getPotatoType() {
    let thePotato = this.potatoType.toUpperCase() + "!!1!!!";
    console.log(thePotato);
    return thePotato;
  }
}
```

Обратите внимание, как мы объявляем класс `PotatoPlanet` — используем ключевое слово `extends` и указываем класс, который расширяем, то есть `Planet`:

```
class PotatoPlanet extends Planet {
.
.
.
.
}
```

Здесь нужно помнить кое-что, связанное с `constructor`. Если мы хотим просто расширить класс и не нуждаемся в изменении конструктора, то можем полностью пропустить определение конструктора в этом классе:

```
class PotatoPlanet extends Planet {
  sayHello() {
    console.log("Hello!");
  }
}
```

В нашем же случае, поскольку мы изменяем действия конструктора, добавляя свойство для типа картошки, то мы снова определяем его с одним важным дополнением:

```
class PotatoPlanet extends Planet {
  constructor(name, width) {
    super(name, width);

    this.potatoType = potatoType;
  }

  getPotatoType() {
    var thePotato = this.potatoType.toUpperCase() + "!!1!!!";
    console.log(thePotato);
    return thePotato;
  }
}
```

Мы производим явный вызов конструктора родителя (`Planet`) с помощью ключевого слова `super` и передачи соответствующих необходимых аргументов. Вызов `super` обеспечивает срабатывание всей необходимой функциональности части `Planet` нашего объекта.

Чтобы использовать `PotatoPlanet`, мы создаем объект и заполняем его свойства или вызываем для него методы так же, как и в случае с простым, не расширенным объектом. Вот пример создания объекта типа `PotatoPlanet` с именем `spudnik`:

```
let spudnik = new PotatoPlanet("Spudnik", 12411, "Russet");
spudnik.gravity = 42.1;
spudnik.getPotatoType();
```

При этом хорошо то, что `spudnik` имеет доступ не только к функциональности, определенной нами как часть класса `PotatoPlanet`, но и всей функциональности, предоставляемой классом `Planet`, который мы расширяем. Мы можем понять, почему это происходит, еще раз обратившись к нашим прототип-объектным связям (рис. 20.4).

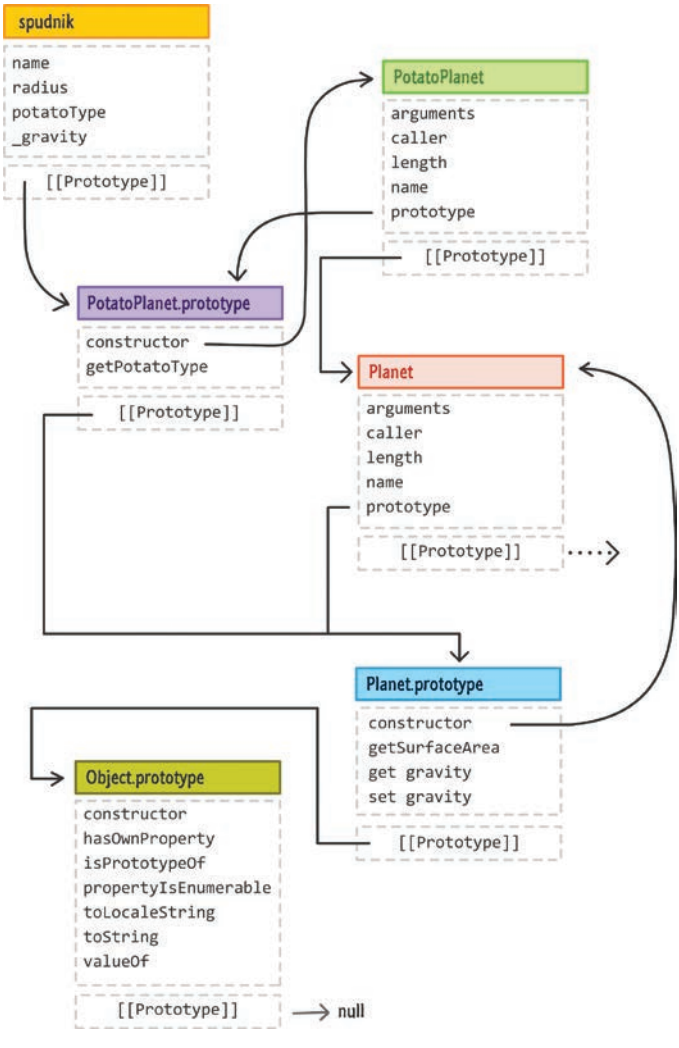


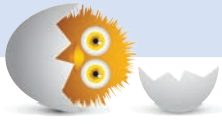
РИС. 20.4.
Так выглядит расширение объекта

Если мы проследуем по цепочке прототипов, то от объекта `spudnik` перейдем к `PotatoPlanet.prototype`, оттуда — к `Planet.prototype`, а закончим в `Object.prototype`. Объект `spudnik` имеет доступ к любому свойству или методу, определенному в каждом этом прототипе, что и дает ему возможность вызывать эти элементы для `Object` или `Planet`, несмотря на то что большая их часть не определена в `PotatoPlanet`. В этом заключается удивительная мощь расширения объектов.

КОРОТКО О ГЛАВНОМ

Синтаксис класса значительно упрощает работу с объектами. Вы можете уловить отголоски этого в текущей главе, но главное вы увидите позднее. Суть этого синтаксиса в том, что он позволяет нам больше фокусироваться на том, *что* мы хотим сделать, вместо того чтобы разбираться, *как* это сделать. Несмотря на то что, работая со свойствами `Object.create` и `prototype`, мы получали существенный контроль, этот контроль зачастую был не нужен. Работая с классами, мы размениваем сложность на простоту. И это совсем не плохо, когда простое решение оказывается верным… в большинстве случаев!

Есть вопросы? Не откладывайте. Обращайтесь на форум **https://forum.kirupa.com**.



В ЭТОЙ ГЛАВЕ:

- узнаем, что именно происходит за кадром `true` и `false`;
- поймем, что делают логические объекты и функции;
- выясним разницу между простыми операторами неравенства и строгими.

21

ЛОГИЧЕСКИЕ ТИПЫ И СТРОГИЕ ОПЕРАТОРЫ === И !==

Из вежливости можно сказать, что все типы одинаково интересны и занятны, но и вы, и я знаем, что это неправда. Некоторые из них весьма скучны. Одним из таких примеров является логический тип данных, и вот почему. Мы создаем логический тип каждый раз, когда инициализируем переменную, используя `true` либо `false`:

```
let sunny = false;
let traffic = true;
```

Примите мои поздравления! Если вам это известно, то вы уже на 80 % достигли полного понимания функционирования логических типов. Конечно, если задуматься, то 80 % недостаточно. Это как есть хот-дог без соуса, уйти с концерта, не дождавшись выхода на бис, или не дописать предложение.

Мы же собираемся заполнить эти недостающие 20 %, которые состоят из различных особенностей логических типов, объекта `Boolean`, функции `Boolean` и очень важных операторов `===` и `!==`.

Поехали!

Объект Boolean

Логические типы рождены для использования в качестве примитивов. Я не стану бороться с ленью и просто приведу пример, который вы только что видели, чтобы продемонстрировать, как выглядит этот примитив:

```
let sunny = false;
let traffic = true;
```

Как вы уже видели много раз, в тени каждого примитива скрывается его объектная форма. Создание логического объекта происходит с помощью ключевого слова `new`, имени конструктора `Boolean` и начального значения:

```
let boolObject = new Boolean(false);
let anotherBool = new Boolean(true);
```

В виде начального значения вы можете передать логическому конструктору либо `true`, либо `false`. Но при этом вы вполне можете передать и нечто иное, что в итоге будет вычислено как `true` или `false`. Расскажу немного о том, какие виды значений будут предсказуемо становиться `true` или `false`, но относительно этого подхода есть обязательное предостережение: *используйте логические объекты только в исключительных случаях, в остальных старайтесь придерживаться примитивов*.

Логическая функция

Конструктор `Boolean` предоставляет одну существенную выгоду, которая связана с возможностью передачи любого произвольного значения или выражения в процессе создания объекта `Boolean`:

```
let boolObject = new Boolean(< arbitrary expression >);
```

Выгодно же это, потому что вам может понадобиться вычислить логическое выражение, в котором итоговые данные оказываются не чистыми `true` или `false`. Это особенно актуально, когда вы имеете дело с внешними данными или кодом и не контролируете получение значения `false` или `true`. Вот пример из головы:

```
let isMovieAvailable = getMovieData()[4];
```

Значение `isMovieAvailable`, вероятно, `true` или `false`. Когда дело доходит до обработки данных, у вас зачастую нет уверенности, что в какой-то момент что-либо вдруг не даст сбой или не вернет иное значение. Как и в реальной жизни, простая вера в то, что все будет работать как надо, неразумна, если не предпринять действенные меры. Одной из таких мер и является функция `Boolean`.

Создание специальной функции для разрешения двусмысленности может быть излишним, но у конструктора `Boolean` есть побочный эффект — у вас остается логический объект, что нежелательно. К счастью, есть способ получить гибкость конструктора `Boolean` совместно с легковесностью логического примитива, причем достаточно легко. Этот способ основывается на функции `Boolean`:

```
let bool = Boolean(true);
```

Логическая функция позволяет передавать произвольные значения и выражения, при этом по-прежнему возвращая *примитивное логическое значение* `true` *либо* `false`. Главным же отличием этого подхода от использования конструктора является то, что вы не используете ключевое слово `new`. Как бы то ни было, давайте приостановимся и рассмотрим, что именно вы можете передать в логическую функцию. Имейте в виду, что все это также можно передавать в логический конструктор, который мы видели в предыдущем разделе.

Для возвращения `false` вы можете передать следующие значения: `null`, `undefined`, пусто или ничего, `0`, пустую строку и, конечно же, `false`:

```
let bool;

bool = Boolean(null);
bool = Boolean(undefined);
bool = Boolean();
bool = Boolean(0);
bool = Boolean("");
bool = Boolean(false);
```

Во всех этих примерах переменная `bool` вернет `false`. Чтобы вернуть `true`, мы можем передать значение `true` или *что угодно*, что не приведет к одному из перечисленных выше значений `false`:

```
let bool;

bool = Boolean(true);
bool = Boolean("hello");
bool = Boolean(new Boolean()); // Внедрение!!!
bool = Boolean("false"); // "false" — это строка
bool = Boolean({});
bool = Boolean(3.14);
bool = Boolean(["a", "b", "c"]);
```

В этих примерах переменная `bool` вернет `true`. Это может показаться немного странным, учитывая некоторые варианты инструкций, поэтому давайте обратим внимание на имеющиеся нюансы. Если то, что мы вычисляем, является объектом, как `new Boolean(new Boolean())`, то вычисляться всегда будет `true`. Причина в том, что простое существование объекта уже приводит к срабатыванию `true`, а вызов `new Boolean()` создает именно новый объект. Если дополнить логику происходящего, это означает, что следующая инструкция `if` также будет вычислена как `true`:

```
let boolObject = new Boolean(false);

if (boolObject) {
  console.log("Bool, you so crazy!!!");
}
```

При этом не важно, если вычисляемый нами объект скрывает в себе значение `false`... или объект `String` или `Array` и т. д. Правила, касающиеся примитивов, гораздо проще. Если мы передаем примитив (или то, что вычисляется как примитив), то все, за исключением `null`, `undefined`, `0`, пустой строки, `NaN` или `false`, будет вычисляться как `true`.

Операторы строгого равенства и неравенства

Последнее, что мы рассмотрим, объединит наши знания о типах, в том числе и логических, и привнесет разнообразие в условные операторы, изученные ранее. Итак, мы знаем об операторах `==` и `!=` и, вероятно,

видели их пару раз в деле. Это операторы равенства и неравенства, которые позволяют понять, являются ли два элемента равными или нет. А вот и сюжетный поворот. Они демонстрируют утонченное, отклоняющееся от нормы поведение, о котором мы можем не знать.

Вот пример:

```
function theSolution(answer) {
  if (answer == 42) {
    console.log("You have nothing more to learn!");
  }
}

theSolution("42"); // 42 передано как строка
```

В этом примере выражение `answer == 42` будет вычислено как `true`. Так происходит несмотря на то, что переданное значение `42` является строкой, мы же производим сравнение с `42`, являющимся числом. Что здесь происходит? Неужели мы попали в мир, где числа и строки равны? При использовании операторов `==` и `!=` такое поведение вполне ожидаемо. В этом случае значением обоих сравниваемых элементов будет `42`. Для этого JavaScript осуществляет нужные операции, и оба значения в итоге рассматриваются как одинаковые. Формально это называется *приведением типа*.

Проблема в том, что такое поведение иногда мешает — особенно когда так происходит у нас за спиной. Во избежание подобных ситуаций у нас есть более строгие версии операторов равенства/неравенства, а именно `===` и `!==` соответственно. Задача этих операторов заключается в сравнении *как значения, так и типа*. При этом они не делают приведения типов. Они ведут к тому, что все заботы по обеспечению равенства или неравенства ложатся непосредственно на нас, и это хорошо.

Теперь давайте исправим предыдущий пример, заменив оператор `==` на `===`:

```
function theSolution(answer) {
  if (answer === 42) {
    console.log("You have nothing more to learn!");
  }
}

theSolution("42"); // 42 передано как строка
```

На сей раз условное выражение будет вычислено как `false`. В этом более строгом мире строка и число — это разные типы, несмотря на то

что их значения одинаковы. Так как приведение типа не производится, то и результат в итоге `false`.

Общее правило гласит: *всегда используйте более строгую форму операторов равенства/неравенства*. Помимо всего прочего, их использование поможет обнаруживать ошибки в коде — ошибки, которые в противном случае может быть сложно распознать.

Если мы сравниваем два разных объекта, то строгий оператор равенства (и менее строгий тоже) не будет работать ожидаемым образом. Например, все приведенные ниже случаи будут вычислены как `false`:

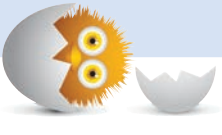
```
console.log(new String("A") == new String("A"));
console.log([1, 2, 3] == [1, 2, 3]);
console.log({ a: 1 } == { a: 1 });
```

Имейте это в виду при выяснении равенства/неравенства двух отдельных самостоятельных объектов.

КОРОТКО О ГЛАВНОМ

Логические типы являются одними из наиболее часто используемых типов при написании кода. Несмотря на внешнюю простоту, они играют ключевую роль в разветвлении кода. Хоть я и могу посчитать на одной руке количество раз, когда мне приходилось использовать функцию `Boolean` или строгие операторы равенства и неравенства, мне не хватит рук и пальцев, чтобы счесть все случаи, когда я сталкивался с этими странными вещами в сторонних проектах.

Если у вас возникнут вопросы, добро пожаловать на форум **https://forum.kirupa.com**.



В ЭТОЙ ГЛАВЕ:

- узнаем, когда значения не существуют;
- поймем, что делать с null и undefined.

22

NULL И UNDEFINED

Одна из величайших загадок мира JS витает вокруг `null` и `undefined`. Зачастую код буквально напичкан этими значениями, и вы, возможно, уже с ними встречались. Но как только спадает завеса тайны, оказывается, что `null` и `undefined` не такое уж странное явление. Они просто ужасно скучные. Возможно, скучнейшие (но важные) элементы JavaScript из всех, с какими вам когда-либо предстоит познакомиться.

Поехали!

Null

Начнем с `null`. Ключевое слово `null` — это примитив, который выполняет особую роль в мире JavaScript. Он является явным определением, обозначающим *отсутствие значения*. Если вам доводилось просматривать чужой код, то, вероятно, вы видели, что `null` встречается достаточно часто. Этот элемент весьма популярен, так как имеет преимущество в виде определенности. Вместо работы с переменными, содержащими устаревшие значения или таинственные неопределенные значения, вы можете установить их как `null`, однозначно указав, что *значение существовать не должно*.

Такая возможность важна, когда вы пишите код и хотите инициализировать или освободить переменную, чтобы она ничего не представляла.

Вот пример:

```
let name = null;

if (name === null) {
  name = "Peter Griffin";
} else {
  name = "No name";
}
```

Примитив `null` не появляется сам собой. Его вы присваиваете сознательно, поэтому предстоит часто встречаться с ним в объявлениях переменных или среди аргументов, передаваемых в вызовы функций. Использовать `null` легко. Проверить его наличие также несложно:

```
if (name === null) {
  // делает что-нибудь интересное или нет
}
```

Имейте в виду, что при этом нужно использовать более строгий оператор `===` вместо `==`. Хоть от использования `==` конец света и не наступит, но при работе с `null` лучше производить проверку как значения, так и типа.

Undefined

А вот здесь уже интереснее. Чтобы представить что-то, что не определено, вы используете примитив `undefined`. Он пригождается в нескольких случаях. Чаще всего это происходит, когда вы пытаетесь обратиться к переменной, которая не была инициализирована или когда обращаетесь к значению функции, которая ничего не возвращает.

Следующий фрагмент кода приводит несколько реальных случаев с `undefined`:

```
let myVariable;
console.log(myVariable); // undefined

function doNothing() {
  // watch paint dry
  return;
}
```

```
let weekendPlans = doNothing();
console.log(weekendPlans); // undefined

let person = {
  firstName: "Isaac",
  lastName: "Newton"
}
console.log(person.title); // undefined
```

В своем коде вы, скорее всего, не станете присваивать `undefined` чему-либо. Вы уделите время проверке, не является ли значение или что-либо еще `undefined`. Такую проверку можно выполнить несколькими способами. Первый из них очень прост, но практически всегда работает:

```
if (myVariable === undefined) {
  // делает что-нибудь
}
```

Оборотная сторона этого подхода связана с истинной природой `undefined`. Держитесь крепче: `undefined` — это глобальная переменная, которая определяется за нас автоматически. Это означает, что потенциально мы можем ее переопределить, например, на `true` или что-либо другое, что нам нужно. Если `undefined` будет переопределена, то нарушит работу кода в случае проверки только с оператором `===` или `==`. Чтобы избежать подобного безобразия, наиболее безопасным способом выполнения проверки на `undefined` будет использование `typeof` и затем уже оператора `===`:

```
let myVariable;

if (typeof myVariable === "undefined") {
  console.log("Define me!!!");
}
```

Это гарантирует выполнение проверки на `undefined` и возвращение верного ответа.

NULL == UNDEFINED, НО NULL !== UNDEFINED

Продолжая тему странности `==` и `===`: если вы когда-нибудь проверите `null == undefined`, то ответом будет `true`. Если же вы используете `===`, то есть `null === undefined`, то ответом будет `false`.

Причина в том, что `==` производит приведение, присваивая значениям такие типы, какие JS посчитает целесообразными. Используя `===`, вы проверяете и тип, и значение. Это уже полноценная проверка, которая определяет, что `undefined` и `null` на деле являются двумя разными вещами.

Монету в шляпу шестиглазому (то есть Тревору Маккаули) за то, что указал на это!

КОРОТКО О ГЛАВНОМ

Я неспроста отложил напоследок эти встроенные типы. `null` и `undefined` — наименее интересные члены коллектива, но при этом зачастую самые недопонятые. Умение использовать `null`, а также обнаруживать его и `undefined` — это очень важные навыки, которыми следует овладеть. Иначе вы рискуете столкнуться с ошибками, которые будет очень сложно обнаружить.

Если у вас появились вопросы о `null` и `undefined` или вы просто хотите пообщаться с самыми дружелюбно настроенными разработчиками на планете, пишите на https://forum.kirupa.com.

В ЭТОЙ ГЛАВЕ:

- разберем, что такое JSON;
- рассмотрим содержимое объектов JSON;
- узнаем, как производить чтение и запись данных JSON.

23

ВСЕ О JSON (ОБЪЕКТНАЯ НОТАЦИЯ JAVASCRIPT)

Когда дело доходит до хранения, извлечения или передачи данных, то в нашем распоряжении оказывается множество форматов файлов и структур данных. Вы наверняка уже использовали текстовые файлы, документы Word, электронные таблицы Excel и т. д. Что же касается фронтенд-разработки, то здесь лидирует один формат — JSON, *JavaScript Object Notation*.

В этой главе мы узнаем все о том, что делает объекты JSON столь прекрасными, рассмотрим в деталях происходящее внутри них и узнаем, как можно считывать их значения для ваших собственных реализаций.

Поехали!

Что такое JSON?

В JavaScript для определения объектов вы используете специальный синтаксис объектного литерала:

```
let funnyGuy = {
  firstName: "Conan",
  lastName: "O'Brien",

  getName: function () {
    return "Name is: " + this.firstName + " " + this.lastName;
  }
};

let theDude = {
  firstName: "Jeffrey",
  lastName: "Lebowski",

  getName: function () {
    return "Name is: " + this.firstName + " " + this.lastName;
  }
};

let detective = {
  firstName: "Adrian",
  lastName: "Monk",

  getName: function () {
    return "Name is: " + this.firstName + " " + this.lastName;
  }
};
```

Если вы не понимаете этот синтаксис, то настоятельно рекомендую почитать раздел «Об объектах подробней». Это существенно упростит понимание объектов JSON и работу с ними.

Внешне синтаксис объектного литерала выглядит как куча скобок, фигурных скобок и двоеточий, которые определяют свойства и значения объекта. Несмотря на такую внешнюю странность, изнутри все достаточно наглядно. Вы вольны использовать большинство из распространенных типов данных и обстоятельно представлять их свойства и значения в виде пар ключей и значений, разделенных двоеточием. Помимо всего перечисленного не менее важно то, что этот синтаксис также позволяет создавать структуру и допускает вложенные значения. В целом это достаточно приятный способ представления объектов JavaScript в литеральном виде.

Формат JSON очень многое заимствует от синтаксиса объектного литерала. Ниже приведен пример реальных данных JSON, возвращенных API WeatherUnderground для отображения погоды в моем родном городе Сиэтле:

```
{
  "response": {
    "version": "0.1",
    "termsofService":
"http://www.wunderground.com/weather/api/d/terms.html",
    "features": {
      "conditions": 1
    }
  },
  "current_observation": {
    "image": {
      "url": "http://icons.wxug.com/graphics/wu2/logo_130x80.png",
      "title": "Weather Underground",
      "link": "http://www.wunderground.com"
    },
    "display_location": {
      "full": "Seattle, WA",
      "city": "Seattle",
      "state": "WA",
      "state_name": "Washington",
      "country": "US",
      "country_iso3166": "US",
      "zip": "98101",
      "magic": "1",
      "wmo": "99999",
      "latitude": "47.61167908",
      "longitude": "-122.33325958",
      "elevation": "63.00000000"
    },

    "observation_location": {
      "full": "Herrera, Inc., Seattle, Washington",
      "city": "Herrera, Inc., Seattle",
      "state": "Washington",
      "country": "US",
      "country_iso3166": "US",
      "latitude": "47.616558",
      "longitude": "-122.341240",
      "elevation": "121 ft"
    },
    "estimated": {},
    "station_id": "KWASEATT187",
    "observation_time": "Last Updated on August 28, 9:28 PM PDT",
    "observation_time_rfc822": "Fri, 28 Aug 2015 21:28:12 -0700",
    "observation_epoch": "1440822492",
    "local_time_rfc822": "Fri, 28 Aug 2015 21:28:45 -0700",
    "local_epoch": "1440822525",
    "local_tz_short": "PDT",
    "local_tz_long": "America/Los_Angeles",
    "local_tz_offset": "-0700",
```

```
    "weather": "Overcast",
    "temperature_string": "68.0 F (20.0 C)",
    "temp_f": 68.0,
    "temp_c": 20.0,
    "relative_humidity": "71%",
    "wind_string": "Calm",
    "wind_dir": "NNW",
    "wind_degrees": 331,
    "wind_mph": 0.0,
    "wind_gust_mph": "10.0",
    "wind_kph": 0,
    "wind_gust_kph": "16.1",
    "pressure_mb": "1008",
    "pressure_in": "29.78",
    "pressure_trend": "-",
    "dewpoint_string": "58 F (15 C)",
    "dewpoint_f": 58,
    "dewpoint_c": 15,
    "heat_index_string": "NA",
    "heat_index_f": "NA",
    "heat_index_c": "NA",
    "windchill_string": "NA",
    "windchill_f": "NA",
    "windchill_c": "NA",
    "feelslike_string": "68.0 F (20.0 C)",
    "feelslike_f": "68.0",
    "feelslike_c": "20.0",
    "visibility_mi": "10.0",
    "visibility_km": "16.1",
    "solarradiation": "--",
    "UV": "0",
    "precip_1hr_string": "0.00 in ( 0 mm)",
    "precip_1hr_in": "0.00",
    "precip_1hr_metric": " 0",
    "precip_today_string": "0.00 in (0 mm)",
    "precip_today_in": "0.00",
    "precip_today_metric": "0",
    "icon": "cloudy",
    "icon_url": "http://icons.wxug.com/i/c/k/nt_cloudy.gif",
    "nowcast": ""
  }
}
```


Если не обращать внимания на размер, то данные JSON, которые вы видите, имеют много схожего с синтаксисом объектного литерала, виденным вами ранее. Также нужно знать и о некоторых существенных их различиях, но эту занудную тему мы затронем несколько позже. Сначала давайте посмотрим, из чего именно состоит объект JSON.

Объект JSON изнутри

Объект JSON — не более чем комбинация имен свойств и их значений. Звучит очень просто, но есть в этом важные детали, которые мы рассмотрим в текущем разделе.

Имена свойств

Имена свойств являются идентификаторами, которые вы будете использовать для обращения к значению. Располагаются они слева от двоеточия:

```
{
  "firstName": "Kirupa",
  "lastName": "Chinnathambi",
  "special": {
    "admin": true,
    "userID": 203
  },
  "devices": [
    {
      "type": "laptop",
      "model": "Macbook Pro 2015"
    },
    {
      "type": "phone",
      "model": "iPhone 6"
    }
  ]
}
```

В этом фрагменте JSON имена свойств — это `firstName`, `lastName`, `special`, `admin`, `userID`, `devices`, `type` и `model`. Обратите внимание на то, как они определены, то есть представлены в виде строчных значений, заключенных в кавычки. Кавычки — это важная деталь, которую вам не требуется использовать для имен свойств в случае с объектными литералами. Поэтому имейте их в виду, когда будете работать в мире JSON.

Значения

Каждое имя свойства отображается в значение. Сами же значения могут иметь следующие типы:

- числа;
- строки;

- логические типы (`true` или `false`);
- объекты;
- массивы;
- `Null`.

Давайте сопоставим эти типы с примером, который видели недавно.

Строки

Ниже выделены именно строчные значения:

```
{
  "firstName": "Kirupa",
  "lastName": "Chinnathambi",
  "special": {
    "admin": true,
    "userID": 203
  },
  "devices": [
    {
      "type": "laptop",
      "model": "Macbook Pro"
    {
    },
      "type": "phone",
      "model": "iPhone XS"
    }
  ]
}
```

Двойные кавычки являются убедительным признаком того, что представленные значения — это строки. Помимо привычных букв, чисел и символов можно включать в них *экранирующие символы* вроде \', \", \\, \/ и т. д., чтобы определять те знаки строки, которые в противном случае будут считаны как операция JSON.

Числа

В нашем примере представлен единственный член семейства чисел — это значение свойства `userID`:

```
{
  "firstName": "Kirupa",
  "lastName": "Chinnathambi",
  "special": {
    "admin": true,
    "userID": 203
  },
```

```
  "devices": [
    {
      "type": "laptop",
      "model": "Macbook Pro"
    },
    {
      "type": "phone",
      "model": "iPhone XS"
    }
  ]
}
```

Вы можете указывать как десятичные значения (например `0.204`, `1200.23`, `45`), так и экспоненциальные (`2e16`, `3e+4`, `1.5e-2`). При этом нужно помнить о том, что нельзя использовать префикс, начинающийся с нуля, сопровождаемого числом. Например, значение `03.14` недопустимо.

Логический тип

Логические значения весьма просты:

```
{
  "firstName": "Kirupa",
  "lastName": "Chinnathambi",
  "special": {
    "admin": true,
    "userID": 203
  },
  "devices": [

    {
      "type": "laptop",
      "model": "Macbook Pro"
    },
    {
      "type": "phone",
      "model": "iPhone XS"
    }
  ]
}
```

Эти значения могут быть либо `true`, либо `false`. Здесь важно помнить, что регистр важен. Оба упомянутых значения должны использоваться в нижнем регистре. Печать в режиме предложения (`True` или `False`) либо использование только верхнего регистра (`TRUE` или `FALSE`) запрещено.

Объекты

А вот тут ситуация немного интереснее:

```
{
  "firstName": "Kirupa",
  "lastName": "Chinnathambi",
  "special": {
    "admin": true,
    "userID": 203
  },
  "devices": [
    {
      "type": "laptop",
      "model": "Macbook Pro"
    },
    {
      "type": "phone",
      "model": "iPhone XS"
    }
  ]
}
```

Объекты содержат коллекцию имен свойств и значений, отделяясь от остального содержимого фигурными скобками. Видите? Разве это не было *немного* интереснее?

Массивы

Наше свойство `devices` представляет массив:

```
{
  "firstName": "Kirupa",
  "lastName": "Chinnathambi",
  "special": {
    "admin": true,
    "userID": 203
  },
  "devices": [
    {
      "type": "laptop",
      "model": "Macbook Pro"
    },
    {
      "type": "phone",
      "model": "iPhone XS"
    }
  ]
}
```

Массивы хранят упорядоченную коллекцию значений, которые вы можете итерировать. Отделяются они скобочной нотацией. Внутри массива допускается использование любого типа JSON, который мы встречали до сих пор, включая и другие массивы.

Null

Последний тип данных самый унылый:

```
{
  "foo": null
}
```

Ваши значения JSON могут быть `null`, что означает пустое значение.

Чтение данных JSON

Признаю, что предыдущий раздел был занудный, но есть и хорошие новости! На фоне ужасной скукоты предыдущего раздела этот покажется куда более захватывающим, чем есть на самом деле.

Как бы то ни было, практически всегда ваше взаимодействие с JSON будет связано с чтением данных. Когда дело доходит до чтения данных JSON, главное помнить, что это очень похоже на чтение значений, хранящихся в типичном объекте JavaScript. Вы можете либо обратиться к необходимому значению через точку (`property.propertyFoo`), либо использовать для этого подход массива (`property["propertyFoo"]`).

Следующий пример продемонстрирует это:

```
let exampleJSON = {
  "firstName": "Kirupa",
  "lastName": "Chinnathambi",
  "special": {
    "admin": true,
    "userID": 203
  },
  "devices": [
    {
      "type": "laptop",
      "model": "Macbook Pro"
    },
```

```
    {
      "type": "phone",
      "model": "iPhone XS"
    }
  ]
};
```

Чтобы считать значение, хранящееся в `firstName`, вы можете сделать одно из следующего:

```
exampleJSON.firstName;
exampleJSON["firstName"];
```

Обе строки вернут значение `Kirupa`. При этом нет принципиальной разницы, получите ли вы нужное значение посредством точечной нотации или с помощью массива. Можете выбирать тот способ, который будет более удобен, но лично я предпочитаю именно точечную нотацию. От передачи имен свойств в виде строк меня немного мутит, поэтому во всех будущих фрагментах кода я буду выделять именно точечную нотацию.

Аналогично тому, что вы видели ранее, для обращения к значению, хранящемуся в `lastName`, вы можете сделать следующее:

```
exampleJSON.lastName;
```

В случае простых свойств, хранящих простые значения, все достаточно просто. Есть, правда, очень и очень маленькая трудность, с которой вы можете столкнуться, и связана она с более сложными значениями, состоящими из объектов и массивов. Чтобы считать значение, содержащееся в объекте, просто продолжайте через точку обращаться к каждому свойству, пока не достигните того, которое и хранит нужное вам значение.

Вот пример того, как выглядит попытка обращения к значению, содержащемуся в свойстве `userID`:

```
exampleJSON.special.userID;
```

Массивы ничем не отличаются, но вам придется переключиться на нотацию массива, как только вы достигните свойства, которое хранит значения массива. Если бы мы хотели обратиться к значению `model` первого устройства в массиве `devices`, то могли бы написать, например, следующее:

```
exampleJSON.devices[0].model;
```

Так как свойство `devices` относится к массиву, вы также можете производить стандартные присущие массивам операции вроде такой:

```
let devicesArray = exampleJSON.devices;

for (let i = 0; i < devicesArray.length; i++) {
  let type = devicesArray[i].type;
  let model = devicesArray[i].model;

  // Делает что-нибудь интересное с этими данными.
}
```

Напоминая пройденное в предыдущем разделе, скажу, что ваши значения JSON могут быть строками, числами, объектами, массивами, логическими типами или null. Все, что JavaScript поддерживает для заданного типа данных, встреченного вами в объекте JSON, вы можете использовать в своих интересах.

Парсинг JSON-подобных данных в действительный JSON

В нашем примере данные JSON были точно определены внутри переменной `exampleJSON`. Ни у кого не возникнет сомнений, что мы имеем дело именно с реальным объектом JS, представленным посредством семантики JSON.

В реальной жизни подобные сценарии будут не всегда. Ваши данные JSON могут поступать из множества различных ресурсов, не все из которых будут возвращать их в удобном для работы формате, аналогичном тому, что мы только что видели. Многие будут возвращать данные JSON в виде *сырого текста*. В итоге у вас будет что-то похожее на объект JSON, но вы не сможете взаимодействовать с данными так, как это возможно при работе с реальным объектом JSON.

Для таких случаев существует метод `JSON.parse`, который получает ваши «недействительные» данные JSON в виде аргумента:

```
function processRequest(e) {
  if (xhr.readyState == 4 && xhr.status == 200) {
    let response = JSON.parse(xhr.responseText);
    selectInitialState(response.region);
  }
}
```

Как видно по выделенной строке, этот метод получает любые JSON-подобные конечные данные и преобразует их в реальный объект JSON, с которым уже гораздо легче работать. Лично я при работе с внешними JSON-данными всегда использую JSON.parse просто для безопасности.

Запись данных JSON?

Текущий раздел был целиком посвящен чтению значений из данных JSON. Было бы логичным посвятить аналогичный раздел их записи. Но как выясняется, запись данных JSON вовсе не так популярна, если только вы не сохраняете их в файл или не работаете с веб-службами. Если вы выполняете какую-либо из этих задач, то очевидно, что это либо разработка на Node, либо написание кода на не JavaScript.

В случае же с фронтенд-разработкой я не представляю распространенных случаев, где запись данных JSON была бы актуальна. Если вы столкнетесь с какой-либо редкой ситуацией, в которой ваши потребности не ограничатся только чтением данных JSON, то гугл в помощь!

КОРОТКО О ГЛАВНОМ

Раньше эта глава была бы посвящена XML. Даже сегодня XML по-прежнему широко используется как формат файлов для хранения информации и ее обмена. JSON же чрезвычайно популярен только в том мире, где браузер является королем (то есть в мире, где живем мы с вами). Вне веб-сайтов, веб-приложений и веб-сервисов, основанных на REST, работа с данными в формате JSON не приветствуется. Вам следует иметь это в виду, если вы столкнетесь с более старыми, менее веб-ориентированными ситуациями.

Если у вас появились какие-либо связанные с JSON или чем-то другим вопросы, отправляйтесь за ответами на форум **https://forum.kirupa.com**.

В ЭТОЙ ГЛАВЕ:

- узнаем, как взаимодействует JavaScript со всей остальной частью страницы;
- поймем, откуда столько суеты вокруг DOM (объектная модель документа);
- разберемся с нечеткими границами между HTML, CSS и JavaScript.

24

JS, БРАУЗЕР И DOM

До сих пор мы изучали JavaScript обособленно. Мы многое узнали о его базовой функциональности, но все это было мало связано с реальным положением вещей в реальном мире — мире, представленном браузером и наполненном множеством мелких HTML-тегов и CSS-стилей. Эта глава послужит введением в этот мир, а последующие главы помогут погрузиться в него уже глубже.

В ближайших разделах вы узнаете о таинственной структуре данных и интерфейсе программирования, известном как *DOM* (*объектная модель документа*). Вы поймете, что это такое, чем она полезна и как она связана со всем, что вам предстоит делать в будущем.

Поехали!

Что делают HTML, CSS и JavaScript

Прежде чем перейти к делу и начать обсуждать смысл жизни... гм, то есть DOM, давайте быстренько взглянем на кое-что, возможно, вам

известное. Для начала скажу, что весь материал, помещаемый в HTML-документы, касается HTML, CSS и JavaScript. Мы рассматриваем эти три компонента как равных партнеров в создании того, что вы знаете как браузер (рис. 24.1).

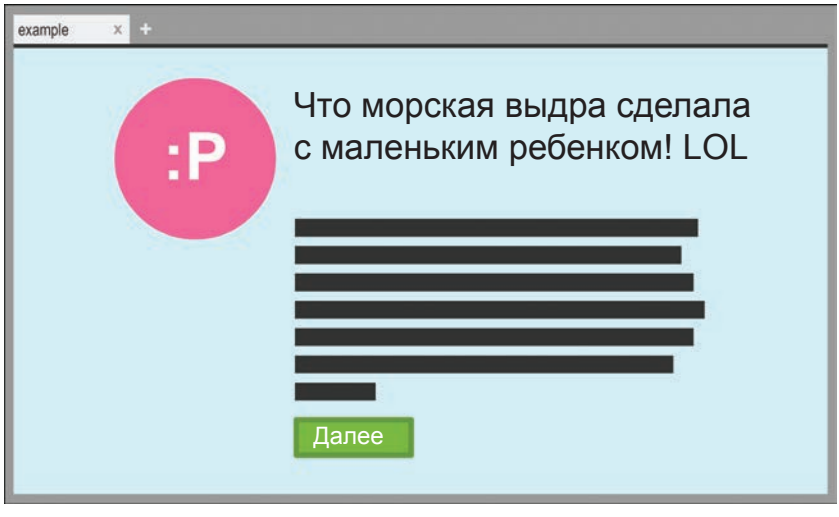


РИС. 24.1.
Типичная веб-страница состоит из HTML, CSS и JavaScript

Каждый из этих партнеров играет свою уникальную важную роль.

HTML определяет структуру

HTML определяет структуру страницы и по большей части содержит материал, который вы видите:

```
<!DOCTYPE html>
<html>

<head>
  <meta content="sea otter, kid, stuff" name="keywords">
  <meta content="Sometimes, sea otters are awesome!"
name="description">
  <title>Example</title>
  <link href="foo.css" rel="stylesheet" />
</head>
```

```
<body>
  <div id="container">
    <img src="seaOtter.png" />

    <h1>What This Sea Otter Did to This Little Kid Will Make You
    LOL!</h1>

    <p class="bodyText">
      Nulla tristique, justo eget semper viverra,
      massa arcu congue tortor, ut vehicula urna mi
      in lorem. Quisque aliquam molestie dui, at tempor
      turpis porttitor nec. Aenean id interdum urna.
      Curabitur mi ligula, hendrerit at semper sed,
      feugiat a nisi.
    </p>

    <div class="submitButton">
      more
    </div>
  </div>
  <script src="stuff.js"></script>
</body>

</html>
```

HTML — это кто-то вроде Мег Гриффин, и он весьма скучен. Если вы не знакомы с Мег Гриффин и вам лень гуглить, кто это, то рис. 24.2 приблизительно показывает, на что она похожа.

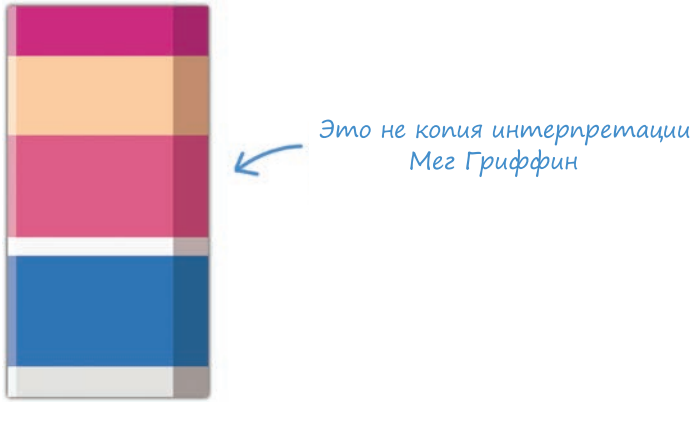


РИС. 24.2.
Творческая интерпретация Мег Гриффин

В любом случае, у нас есть скучный HTML-документ, а чтобы преобразовать его содержимое из чего-то простого и однообразного, можно использовать CSS.

Приукрась мой мир, CSS!

CSS — это ваш главный язык стиля, который позволяет придавать HTML-элементам столь необходимую эстетическую и схематичную привлекательность:

```
body {
  font-family: "Arial";
  background-color: #CCCFFF;
}
#container {
  margin-left: 30%;
}
#container img {
  padding: 20px;
}
#container h1 {
  font-size: 56px;
  font-weight: 500;
}
#container p.bodyText {
  font-size: 16px;
  line-height: 24px;
}
.submitButton {
  display: inline-block;
  border: 5px solid #669900;
  background-color: #7BB700;
  padding: 10px;
  width: 150px;
  font-weight: 800;
}
```

На протяжении долгого времени HTML и CSS предоставляли все необходимое для создания прекрасных и функциональных страниц. Имелись и структура, и макет страницы, были навигация и даже простые взаимодействия вроде наведения мыши. Жизнь была прекрасна.

Настало время JavaScript!

Несмотря на все чудесные возможности, которые предоставляли HTML и CSS, они были ограничены в возможностях интерактивности. Люди хотели делать в веб-документе существенно больше, чем просто сидеть и наблюдать за происходящим. Они хотели и чтобы сами эти документы могли делать больше, чтобы они помогали работать с медиаматериалом, запоминали, на каком моменте происходил выход, обрабатывали клики мыши, нажатия клавиш и пальцев, использовали вычурные меню для навигации и изящные программные анимации, взаимодействовали с веб-камерами/микрофонами, не требовали перезагрузки страницы в случае каждого действия и многое-многое другое.

Конечно же, помогло то, что веб-разработчики и дизайнеры (как вы и я) также прилагали все усилия в поиске способа создания всех этих запросов.

Чтобы заполнить пробел между тем, что предлагали HTML/CSS, и тем, что требовалось пользователям, применялись сторонние компоненты вроде Java и Flash, которые благополучно развивались на протяжении многих лет. Так продолжалось до недавних пор. Это было следствием многих технических и политических причин, но одна из таких причин была в том, что JavaScript просто был не готов в течение долгих лет. Для эффективности ему не хватало либо того, что предоставляли базовые языки, либо того, что поддерживали браузеры.

Сегодня это уже не важно. Теперь JavaScript является крайне способным языком, который позволяет добавлять нужные людям виды интерактивности. Доступ же ко всем этим возможностям обеспечивает именно DOM.

Знакомьтесь с объектной моделью документа

То, что отображается в вашем браузере, является веб-документом. Если говорить более конкретно, суммируя материал предыдущих разделов, то вы видите то, что сформировано взаимодействием HTML, CSS и JavaScript. Если еще немного углубиться, то изнутри действует иерархическая структура, которую ваш браузер использует, чтобы разобраться во всем, что происходит.

Эта структура известна как объектная модель документа. Друзья зовут ее просто DOM. На рис. 24.3 показан упрощенный вид DOM для нашего предыдущего примера.

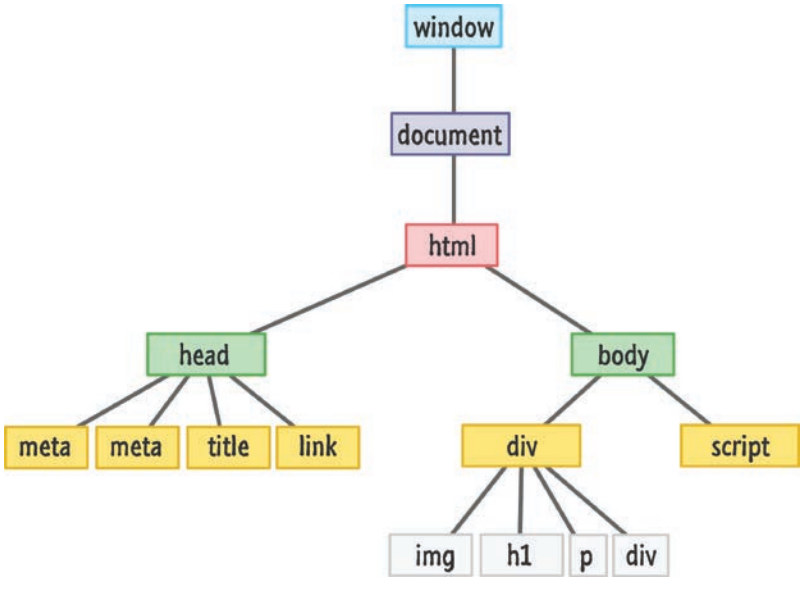


РИС. 24.3.
DOM для всего HTML, приведенного в начале главы, выглядит примерно так

Несмотря на простоту, нужно разобрать кое-какие вещи, применимые ко всем структурам DOM. В действительности DOM кроме HTML содержит и многие другие компоненты. Все эти компоненты более широко известны как *узлы*.

Эти узлы могут быть элементами (что не должно вас удивлять), атрибутами, текстом, комментариями, документным материалом и чем-то еще, о чем вы даже и не задумываетесь. Эта деталь может быть для кого-то и важна, но не для нас с вами. Практически всегда единственным интересующим нас узлом будет элемент, так как именно с ними мы будем иметь дело в 99 % случаев. Хотя на скучном техническом уровне узлы по-прежнему играют определенную роль в нашем сконцентрированном на элементе представлении.

Каждый HTML-элемент, к которому вы хотите обратиться, имеет конкретный, ассоциированным с ним тип. Все эти типы расширяют основу *Node* (*узел*), которая представляет их все, как показано на рис. 24.4.

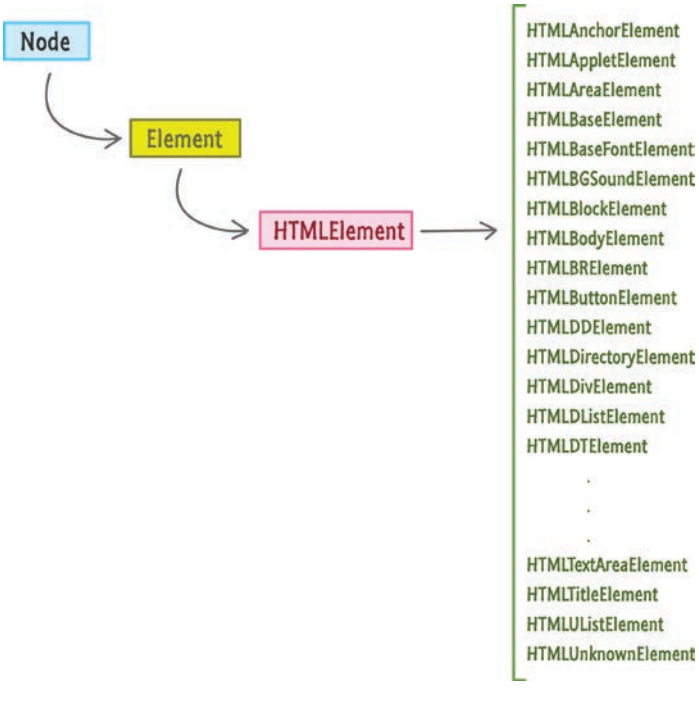


РИС. 24.4.
Структурное расположение элементов, которые мы обычно видим

Ваши HTML-элементы находятся в конце цепочки, которая начинается с узла и продолжается `Element` и `HTMLElement`, заканчиваясь типом (`HTMLDivElement`, `HTMLHeadingElement` и т. д.), совпадающим с самим элементом. Свойства и методы, используемые для управления HTML-элементами, представлены в одной из частей этой цепочки.

Прежде чем перейти к использованию DOM для модифицирования HTML-элементов, давайте сначала поговорим о двух особых объектах, которые встают у нас на пути.

Объект window

В браузере основой всей иерархии выступает объект `window`, который содержит много свойств и методов, помогающих в работе (рис. 24.5).

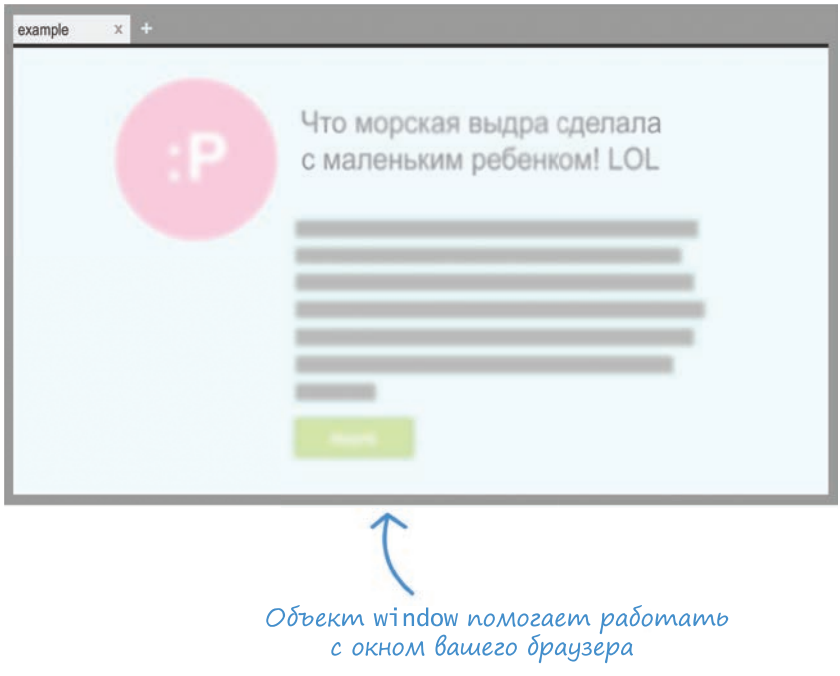


РИС. 24.5.
Window — это весьма значимый компонент процесса

Действия, которые вы можете совершать с помощью объекта `window`, включают обращение к текущему URL, получение информации о любых

рамках страницы, использование локального хранилища, просмотр информации об экране, работу с полосой прокрутки, установку текста строки состояния и различные другие действия, применимые к контейнеру, в котором отображена страница.

Объект document

Теперь мы переходим к объекту `document`, выделенному на рис. 24.6. Здесь уже будет поинтереснее, и именно на нем мы будем фокусироваться.

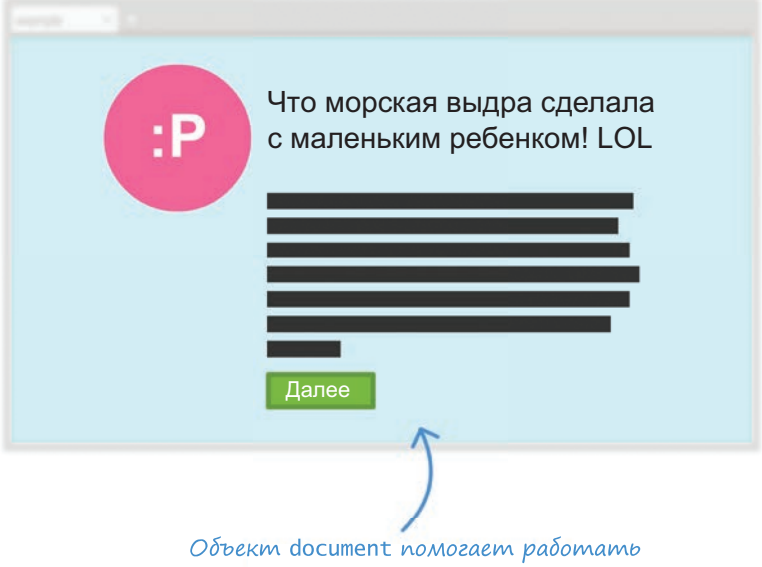


РИС. 24.6.
Объект document — тоже по-своему важный компонент

Объект `document` определяет доступ ко всем HTML-элементам, составляющим отображение. Важная деталь, которую нужно помнить (и которая станет более ясна в последующих главах), — это то, что объект `document` не просто представляет версию HTML-документа только для чтения. Здесь предусмотрено двухстороннее взаимодействие, при котором вы можете не только считывать документ, но также и манипулировать им по своему усмотрению.

Любые изменения, производимые вами в DOM через JavaScript, отражаются на том, что показывается в браузере. Это значит, что вы можете динамически добавлять элементы, удалять их, перемещать, изменять их атрибуты, устанавливать встроенные стили CSS и выполнять различные другие выкрутасы. За исключением самого основного HTML-кода, необходимого в виде тега `script` для запуска JavaScript-кода в HTML-документе, вы можете построить полноценную страницу, используя исключительно JavaScript, если вам того захочется. При правильном использовании это очень мощная возможность.

Другой аспект импорта объекта — `document` связан с событиями. Я поясню детали вкратце, но если вы хотите использовать реакцию на наведение или клик мыши, отметку в графе для галочки, обнаружение, когда была нажата клавиша, и т. д., то будете полагаться на функциональность, предоставляемую объектом `document` для прослушивания событий и реагирования на них.

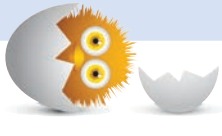
Есть и другие наборы функций, которые обеспечивает DOM, но о них позже, когда дойдем до соответствующей темы.

КОРОТКО О ГЛАВНОМ

DOM — это наиболее важный компонент функциональности, доступный вам для работы с HTML-документами. Он является тем недостающим звеном, которое связывает HTML и CSS с JavaScript. Помимо этого, он также повышает уровень доступа к браузеру.

Вообще, знание DOM — это только часть веселья. В действительности использование ее функциональности для взаимодействия с веб-документом — это гораздо более объемная и более веселая другая часть. Когда будете готовы, перелистывайте страницу, и будем продолжать.

Если у вас есть вопросы по этой или другой теме, задавайте их на форуме https://forum.kirupa.com.



В ЭТОЙ ГЛАВЕ:

- научимся находить элементы в DOM;
- используем синтаксис селекторов CSS для более грамотного обнаружения элементов.

25

ПОИСК ЭЛЕМЕНТОВ В DOM

Как мы видели в предыдущей главе, DOM — это не более чем древовидная структура (рис. 25.1), состоящая из всех элементов нашего HTML-документа.

Но эта деталь важна только отчасти. Действительно же важно то, что, имея все эти связанные с HTML элементы, нужно обращаться к ним для считывания или изменения данных. Есть разные способы отыскать эти элементы. В конечном итоге все они выстроены в древоподобную структуру, а программисты любят придумывать безумные способы просмотра дерева снизу вверх и наоборот в поиске чего-либо.

Не буду принуждать вас к подобной пытке. Пока что. В текущей главе вы научитесь использовать встроенные функции `querySelector` и `querySelectorall`, которые удовлетворят большую часть ваших поисковых нужд в DOM.

Поехали!

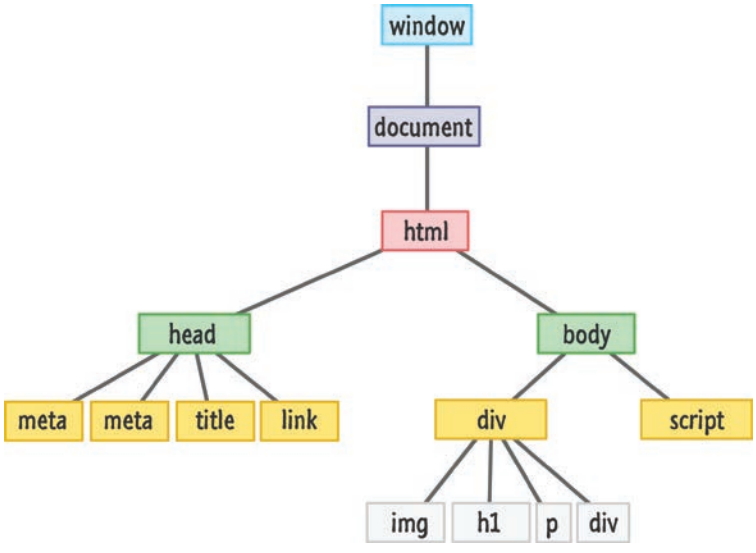


РИС. 25.1.
Действительно выглядит как древовидная структура

Знакомьтесь с семейством querySelector

Чтобы лучше понять всю прелесть возможностей, предоставляемых `querySelector` и `querySelectorAll`, взгляните на следующий HTML-код:

```
<div id="main">
  <div class="pictureContainer">
    <img class="theImage" src="smiley.png" height="300" width="150" />
  </div>
  <div class="pictureContainer">
    <img class="theImage" src="tongue.png" height="300" width="150" />
  </div>
  <div class="pictureContainer">
    <img class="theImage" src="meh.png" height="300" width="150" />
  </div>
  <div class="pictureContainer">
    <img class="theImage" src="sad.png" height="300" width="150" />
  </div>
</div>
```

В этом примере у вас есть один `div` с `id main`, а также четыре элемента `div` и `img`, каждый из которых имеет значение класса `pictureContainer` и `theImage` соответственно. В нескольких следующих разделах мы задействуем функции `querySelector` и `querySelectorAll` в этом HTML-документе и посмотрим, что это даст.

querySelector

На базовом уровне функция `querySelector` работает так:

```
let element = document.querySelector("CSS selector");
```

Она получает аргумент, который является строкой, представляющей селектор CSS для искомого элемента. `querySelector` возвращает первый найденный элемент, даже если существуют и другие, на которые может быть нацелен селектор. В этом смысле описываемая функция довольно упряма.

Например, если мы захотим обратиться к `div` с `id main` из недавнего примера, то напишем следующее:

```
let element = document.querySelector("#main");
```

Так как `main` является `id`, синтаксис селектора для нацеливания на него будет `#main`. Аналогичным образом давайте определим селектор для класса `pictureContainer`:

```
let element = document.querySelector(".pictureContainer");
```

Вернется при этом первый `div`, чье значение класса будет `pictureContainer`. Остальные элементы `div` с тем же значением класса `pictureContainer` будут просто проигнорированы.

Синтаксис селектора не изменяется и не становится особенным из-за того, что вы работаете в JavaScript. Вы можете использовать для селекторов в точности тот синтаксис, который используете в таблице стилей или в области стилей.

querySelectorAll

Функция `querySelectorAll` возвращает все найденные элементы, которые совпадают с предоставленным вами селектором:

```
let elements = document.querySelectorAll("CSS selector");
```

Все описанное мной относительно `querySelector` также относится и к `querySelectorAll`, за исключением числа возвращаемых элементов. Эта важная деталь определяет то, что вы в итоге получаете, используя `querySelectorAll`. При этом возвращается не единичный элемент, а целый массивоподобный контейнер элементов.

Продолжая недавний пример HTML, то представим, как выглядел бы наш код JavaScript, если бы мы хотели использовать `querySelector` для помощи в отображении атрибута `src` среди элементов `img`, содержащих значение класса `theImage`:

```
let images = document.querySelectorAll(".theImage");

for (let i = 0; i < images.length; i++) {
  let image = images[i];
  console.log(image.getAttribute("src"));
}
```

Видите? Здесь все достаточно просто. Главное — это запомнить, как работать с массивами, чему к данному моменту вы уже должны были научиться. Еще одна (немного странная) особенность — это загадочная функция `getAttribute`. Если вы с ней не знакомы и не знаете, как считывать значения элементов, то волноваться не стоит. Мы все это вскоре подробно рассмотрим. Пока просто знайте, что она позволяет вам считывать значение любого HTML-атрибута, который может иметь рассматриваемый HTML-элемент.

Таков синтаксис селектора CSS

Когда я впервые использовал `querySelector` и `querySelectorAll`, то меня удивило, что в качестве аргумента они, по сути, получают всевозможные вариации синтаксиса селектора CSS. При этом вам не обязательно делать их простыми, как я показывал до сих пор.

Если вам понадобится нацелиться на все элементы `img` без необходимости указывать значение класса, то вот как может выглядеть вызов `querySelectorAll`:

```
let images = document.querySelectorAll("img");
```

Если вы захотите нацелиться только на изображение, чей атрибут `src` установлен как `meh.png`, то можете сделать следующее:

```
let images = document.querySelectorAll("img[src='meh.png']");
```

Обратите внимание, что я просто указал *селектор атрибута*[1] в качестве аргумента для `querySelectorAll`. Практически любые сложные выражения, которые вы можете определить для селектора в CSS-документе, также могут быть определены в качестве аргумента для `querySelector` или `querySelectorAll`.

Однако есть и некоторые подвохи, о которых стоит знать.

Не все селекторы псевдоклассов допустимы. Селекторы, состоящие из `:visited`, `:link`, `::before` и `::after`, будут проигнорированы, и элементы не будут найдены.

Допустимый диапазон возможностей предоставления вами селекторов зависит от поддержки CSS браузером. Internet Explorer 8 поддерживает `querySelector` и `querySelectorAll`, но не поддерживает CSS3. Это значит использование всего, что новее селекторов, определенных в CSS2, не будет работать с `querySelector` и `querySelectorAll` в IE8. Скорее всего, это вас не коснется, так как вы наверняка используете более новые версии браузеров, в которых эта проблема с IE8 абсолютно не актуальна.

Селектор, который вы указываете, применяется только к наследникам стартового элемента, с которого начинается поиск. При этом сам этот стартовый элемент в поиск не включается. Не все вызовы `querySelector` и `querySelectorAll` должны производиться из `document`.

[1] http://bit.ly/kirupaAttribute

КОРОТКО О ГЛАВНОМ

Функции `querySelector` и `querySelectorAll` чрезвычайно полезны в сложных документах, где нацелиться на конкретный элемент зачастую не так просто. Полагаясь на грамотно организованный синтаксис селектора CSS, мы можем охватывать как малое, так и большое количество нужных нам элементов. Если мне требуются все элементы изображений, я просто могу написать `querySelectorAll("img")`. Если мне нужен только непосредственно элемент `img`, содержащийся внутри его родителя `div`, то я могу написать `querySelector("div + img")`. Это все очень круто.

Прежде чем завершить тему: есть еще кое-что важное, о чем хочется сказать. Во всем этом захватывающем процессе поиска элементов недостает функций `getElementById`, `getElementsByTagName` и `getElementsByClassName`. В свое время именно они использовались для поиска элементов в DOM. Функции `querySelector` и `querySelectorAll` — это настоящее и будущее решение этой задачи, поэтому не стоит беспокоиться о перечисленных функциях `getElement*`. На данный момент единственным их преимуществом перед `querySelector` и `querySelectorAll` является производительность. `getElementById` весьма быстра, и вы можете своими глазами увидеть ее в сравнении здесь: https://jsperf.com/getelementbyid-vs-queryselector/11.

Однако как сказал один мудрый человек: «Жизнь слишком коротка, чтобы тратить ее на изучение старых функций JavaScript, даже если они немного быстрее!»

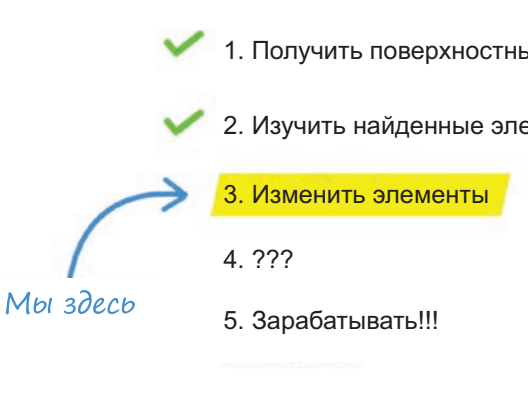
В ЭТОЙ ГЛАВЕ:

- поймем, как можно использовать JavaScript для модифицирования DOM;
- познакомимся с элементами HTML;
- научимся изменять атрибуты.

26

МОДИФИЦИРОВАНИЕ ЭЛЕМЕНТОВ DOM

На данном этапе вы уже знаете, что такое DOM. Вы также видели, как осуществляется поиск элементов с помощью `querySelector` и `querySelectorAll`. Теперь мы изучим, как изменять эти найденные элементы.



В конце концов, разве весело иметь гигантский кусок глины (или теста) и при этом не иметь возможности приложить к нему руки и сделать мощный замес? Как бы то ни было, кроме развлечений мы будем постоянно работать с модифицированием DOM. Независимо от того, используем мы JavaScript для изменения текста элемента, смены изображения, перемещения элемента по документу, установки встроенного стиля или внесения любого желаемого изменения из миллиона возможных, *для всего этого мы модифицируем DOM*. Этот урок научит вас основам этого процесса.

Поехали!

Элементы DOM — они как объекты

Возможность использовать JavaScript для изменения, отображаемого в браузере, стала доступна благодаря одной основной детали. Заключается она в том, что каждый HTML-тег, правило стиля и другие относящиеся к странице компоненты также представлены и в DOM.

Чтобы наглядно изобразить все сказанное, предположим, что у нас есть элемент изображения, определенный в разметке:

```
<img src="images/lol_panda.png" alt="Sneezing Panda!" width="250"
height="100"/>
```

Когда браузер считывает документ и доходит до этого элемента, он создает узел в DOM, который представляет его, как это показано на рис. 26.1.

Это представление в DOM дает нам возможность делать все, что мы могли делать в разметке. На деле оказывается, что представление DOM позволяет производить даже больше действий в отношении HTML-элементов, чем при использовании самой разметки. Этого мы в некоторой степени коснемся в текущей главе и существенно серьезнее рассмотрим в дальнейшем. Причина такой гибкости HTML-элементов при их рассмотрении через DOM кроется в том, что они имеют много общего с обычными объектами JavaScript. Наши элементы DOM содержат свойства, позволяющие получать и устанавливать значения и вызывать методы. Они имеют форму наследования, рассмотренную нами несколько ранее, при которой функциональность, предоставля-

емая каждым элементом DOM, распространяется по базовым типам `Node`, `Element` и `HTMLElement` (рис. 26.2).

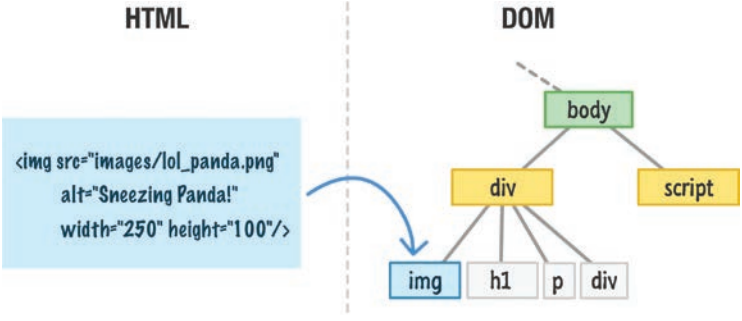


РИС. 26.1.
Все наши HTML-элементы в итоге будут иметь представление в DOM

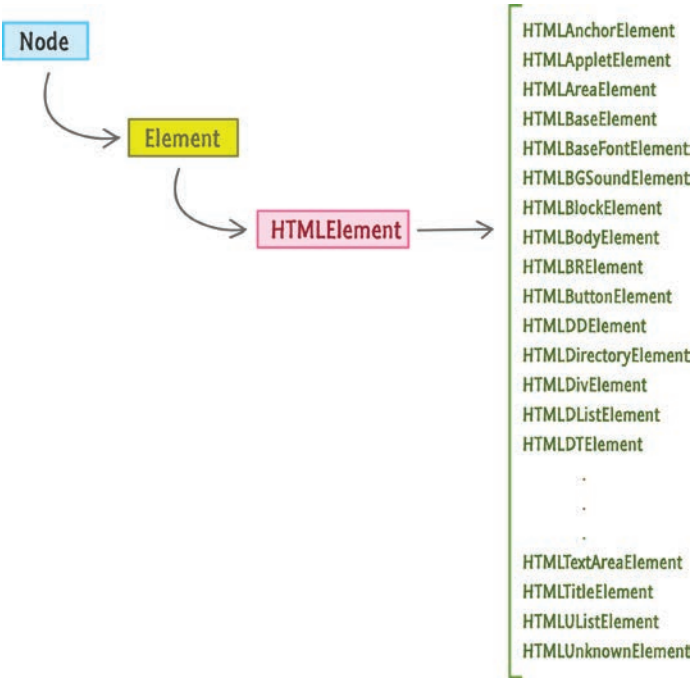


РИС. 26.2.
Иерархия элементов представления, с которыми мы обычно сталкиваемся в HTML

Наверняка элементы DOM даже пахнут, как `Object`, когда забегают в дом после прогулки под дождем.

Несмотря на все схожести, в связи с требованием закона и в целях сохранения ментального здоровья я должен предоставить следующий дисклеймер: *DOM разрабатывалась не для того, чтобы имитировать принцип работы объекта*. Многое из того, что мы можем делать с объектами, мы определенно можем делать и с DOM, но это лишь потому, что поставщики браузеров обеспечили такую возможность. Спецификации W3C не предписывают DOM вести себя в точности так, как компоненты ведут себя при работе с обычными объектами. Хоть я бы и не стал об этом особо волноваться, но если вы когда-нибудь решите расширить элементы DOM или произвести более продвинутые действия, связанные с объектами, то обязательно выполните тесты во всех браузерах, просто чтобы убедиться, что все работает, как вам надо.

Теперь, когда это нескладное обсуждение позади, давайте уже начнем именно модифицировать DOM.

Пора модифицировать элементы DOM

Несмотря на то что мы определенно можем откинуться на стуле и освоить изменения элементов DOM в пассивном режиме, это один их тех случаев, когда будет интереснее рассмотреть тему, следуя за одним простым примером. Если вы не против, то будем использовать следующий HTML в качестве песочницы для отработки изучаемых техник:

```
<!DOCTYPE html>
<html>

<head>
  <title>Hello...</title>

  <style>
    .highlight {
      font-family: "Arial";
      padding: 30px;
    }

    .summer {
      font-size: 64px;
```

```
      color: #0099FF;
    }
  </style>

</head>

<body>

  <h1 id="bigMessage" class="highlight summer">What's happening?</h1>

  <script>

  </script>
</body>

</html>
```

Просто поместите весь этот код в HTML-документ и следуйте дальнейшим указаниям. Если просмотреть этот HTML в браузере, то вы увидите изображение, аналогичное рис. 26.3.

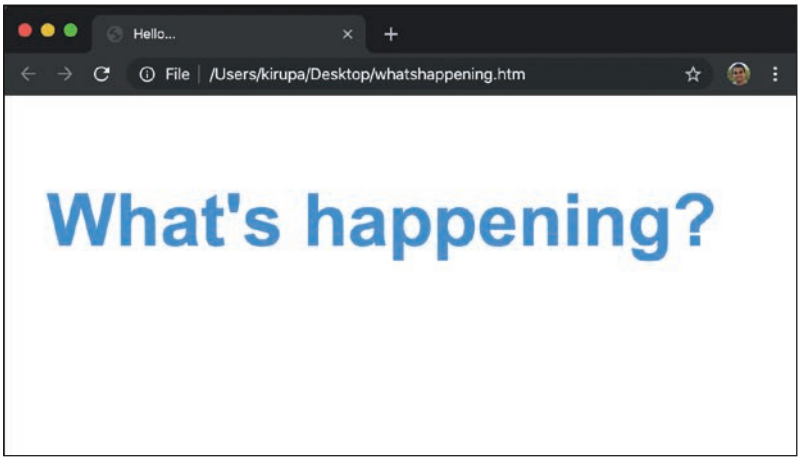


РИС. 26.3.
Что происходит?

На самом деле происходит не так уж много. Основная часть содержимого — это тег `h1`, отображающий текст `What's happening?`:

```
<h1 id="bigMessage" class="highlight summer">What's happening?</h1>
```

Теперь если переключиться на обзор с позиции DOM, то на рис. 26.4 можно увидеть, как выглядит текущий пример при отображении всех HTML-элементов и узлов вроде `document` и `window`.

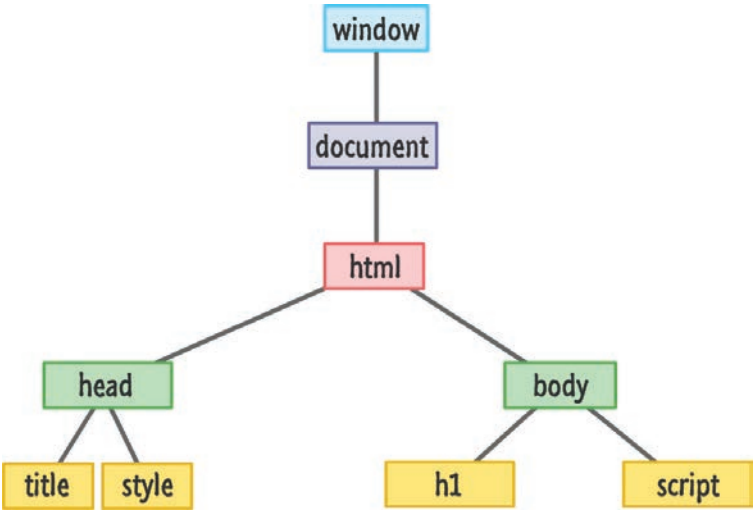


РИС. 26.4.
Так выглядит структура DOM нашего примера

В ближайших разделах мы рассмотрим кое-какие возможности модифицирования элементов DOM.

Изменение значения текста элемента

Начнем с самого простого. Многие HTML-элементы могут отображать текст. Примером таких элементов являются заголовки, абзацы, разделы, вводные данные, кнопки и многое другое. У всех них есть одна общая деталь, а именно то, что изменение значения текста вы производите, устанавливая свойство `textContent`.

Предположим, нужно изменить текст, появляющийся в элементе `h1` нашего примера. Следующий фрагмент кода показывает, как это будет выглядеть:

```
<body>
  <h1 id="bigMessage" class="highlight summer">What's happening?</h1>

  <script>
    let headingElement = document.querySelector("#bigMessage");
    headingElement.textContent = "Oppa Gangnam Style!";
  </script>
</body>
```

Если вы внесете эти изменения и снова запустите код, то в браузере вы увидите то, что показано на рис. 26.5.

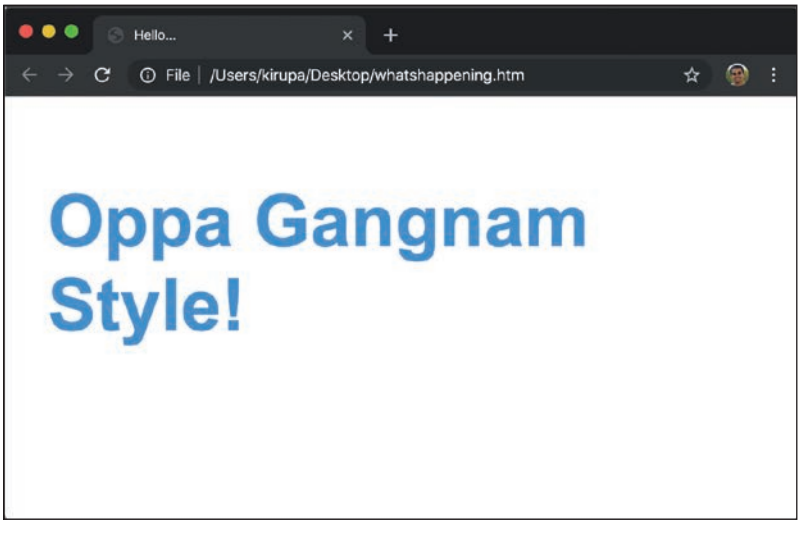


РИС. 26.5.
Изменение значения текста заголовка

Теперь давайте посмотрим, что же конкретно мы сделали, чтобы внести эти изменения. Первым шагом модифицирования любого HTML-элемента в JavaScript будет получение ссылки на него:

```
let headingElement = document.querySelector("#bigMessage");
```

Именно здесь появляются наши старые знакомые `querySelector` и `querySelectorAll`. Как мы увидим позже, есть и второстепенные способы сослаться на элемент. Тем не менее прямой подход, приведенный здесь, мы будем использовать, когда у вас будет очень конкретное понимание того, на какой элемент или элементы вы хотите нацелиться.

Как только у нас появилась ссылка на элемент, мы можем просто установить в нем свойство `textContent`:

```
headingElement.textContent = "Oppa Gangnam Style!";
```

Чтобы увидеть текущее значение свойства `textContent`, вы можете считать его, как и любую переменную. Мы также можем установить свойство для изменения текущего значения, как мы сделали это здесь. После выполнения этой строки кода наше оригинальное значение `What's happening?` в разметке будет заменено в DOM на то, что установлено в JavaScript.

Значения атрибутов

Один из основных способов отличия HTML-элементов заключается в их атрибутах и хранимых в этих атрибутах значениях. Например, следующие три элемента изображений отличаются по атрибутам `src` и `alt`:

```
<img src="images/lol_panda.png" alt="Sneezing Panda!"/>
<img src="images/cat_cardboard.png" alt="Cat sliding into box!"/>
<img src="images/dog_tail.png" alt="Dog chasing its tail!"/>
```

К каждому HTML-атрибуту (включая пользовательские `data-*`) можно обратиться через свойства, предоставляемые DOM. Чтобы облегчить нам работу с атрибутами, элементы предлагают самоочевидные методы `aetAttribute` и `aetAttribute`.

Метод `getAttribute` позволяет указать имя атрибута в элементе, где он существует. Если атрибут найден, этот метод вернет значение, ассоциированное с этим атрибутом. Вот пример:

```
<body>
  <h1 id="bigMessage" class="highlight summer">What's happening?</h1>

  <script>
    let headingElement = document.querySelector("h1");
    console.log(headingElement.getAttribute("id")); // bigMessage
  </script>
</body>
```

В этом фрагменте кода стоит обратить внимание на то, что мы получаем значение атрибута `id` в элементе `h1`. Если мы укажем имя несуществующего атрибута, то получим значение `null`. В противоположность получению значения существует его установка. Чтобы установить значение, мы используем соответственно названный метод `setAttribute`. Делается это вызовом `setAttribute` для элемента, на

который нужно воздействовать, и указанием как имени атрибута, так и значения, которое он должен в итоге хранить.

Вот пример использования `setAttribute`:

```
<body>
  <h1 id="bigMessage" class="highlight summer">What's happening?</h1>

  <script>
    let headingElement = document.querySelector("h1");
    headingElement.setAttribute("class", "bar foo");
  </script>
</body>
```

Мы устанавливаем (на самом деле перезаписываем) атрибут `class` в элементе `h1` на `bar foo`. Функция `setAttribute` не производит никакой проверки, чтобы убедиться, что мы устанавливаем в элемент допустимый для него атрибут. Поэтому ничто не мешает нам сделать какую-нибудь глупость вроде этой:

```
<body>
  <h1 id="bigMessage" class="highlight summer">What's happening?</h1>

  <script>
    let headingElement = document.querySelector("h1");
    headingElement.setAttribute("src", "http://www.kirupa.com");
  </script>
</body>
```

Элемент `h1` не содержит атрибут `src`, но мы можем его указать. Когда код будет запущен, элемент `h1` даже примерит на себя этот атрибут `src`, но ему наверняка будет неудобно.

Прежде чем мы продолжим, хочу кое-что прояснить. В этих примерах использования `setAttribute` и `getAttribute` я выбрал `id` и `class`. Для этих двух атрибутов у нас есть и другой способ установки. В связи с тем что процесс установки атрибутов `id` и `class` очень распространен, наши HTML-элементы открыто выражают свойства `id` и `className`:

```
<body>
  <h1 id="bigMessage" class="highlight summer">What's happening?</h1>

  <script>
    let headingElement = document.querySelector("h1");
    console.log(headingElement.id); // bigMessage

    headingElement.className = "bar foo";
  </script>
</body>
```

Возвращаясь к нашему примеру, обратите внимание, что я переключился с использования `getAttribute` и `setAttribute` на использование `id` и `className`. Конечный результат полностью одинаков. Единственное отличие в том, что у вас появился прямой путь установки этих атрибутов без необходимости использования `getAttribute` или `setAttribute`. Теперь, прежде чем продолжить, я должен прояснить одну особенность: мы не можем использовать `class` в JavaScript для обращения к атрибуту класса, так как `class` имеет совершенно другое назначение, имеющее отношение к работе с объектами. Поэтому мы используем `className`.

СОВЕТ

Есть существенно лучший способ указания значений класса, чем использование `className`. Этот способ реализуется через куда более интересное свойство `classList`, о котором вы подробно узнаете в следующей главе.

КОРОТКО О ГЛАВНОМ

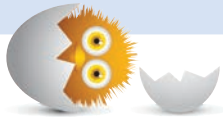
Может показаться странным закончить разговор об элементах DOM на этой ноте. Несмотря на то что изменение текста элемента и значений атрибутов очень распространено, это далеко не единственные модификации, которые вы можете производить. Управление DOM и использование свойств элементов и методов для выполнения наших задач являются основой всего, что мы будем рассматривать. В последующих главах вы увидите намного больше, чем здесь.

Основные выводы из текущей темы в том, что производимые вами изменения DOM практически всегда будут принимать одну из двух форм:

- установка свойства;
- вызов метода.

Методы `textContent`, `setAttribute` и `getAttribute`, рассмотренные нами, покрывают оба этих подхода, и вскоре вы часто будете встречать не только их самих, но и их друзей.

Это весьма увесистая тема! Если у вас есть вопросы, не откладывайте и скорее обращайтесь на форум https://forum.kirupa.com.



В ЭТОЙ ГЛАВЕ:

- научимся изменять CSS с помощью JavaScript;
- разберем плюсы и минусы прямой настройки стилей в противоположность настройке значений классов;
- используем `classList`, чтобы облегчить работу со значениями класса элемента.

27

СТИЛЬ КОНТЕНТА

В предыдущей главе мы рассмотрели, как можно модифицировать содержимое DOM, используя JavaScript. Однако HTML-элементы также отличаются и внешностью, а именно стилем. Когда дело доходит до стилизации содержимого, наиболее распространены создание правила стиля и нацеливание его селектора на элемент(ы). Выглядит правило стиля так:

```
.batman {
  width: 100px;
  height: 100px;
  background-color: #333;
}
```

Сам же элемент, на который будет действовать это правило, может выглядеть так:

```
<div class="batman"></div>
```

На любой веб-странице мы повсюду встречаем от нескольких до многих правил стиля, красиво перемежающиеся друг с другом, стилизуя все, что мы видим. Тем не менее это не единственный подход, который можно использовать для стилизации содержимого с помощью CSS. Это бы не был HTML, если бы в нем ни присутствовало несколько способов решения одной задачи.

Без учета встроенных стилей другой подход для CSS-стилизации элементов заключается в использовании JavaScript. Мы можем использовать JS, чтобы *устанавливать стиль элемента напрямую*, а также чтобы *добавлять или удалять значения класса элементов*, которые будут определять применение тех или иных правил стиля.

В этой главе мы изучим оба этих подхода.

Поехали!

Зачем устанавливать стили с помощью JavaScript?

Для начала будет нелишним пояснить, откуда вообще возникает желание использовать JavaScript для воздействия на стиль элемента. В обыденных случаях, когда мы используем правила стиля или встроенные стили для определения внешнего вида элемента, стилизация подключается после загрузки страницы. Это прекрасно, и именно так нам в большинстве случаев и нужно.

Однако существует множество ситуаций, когда содержимое становится более интерактивным, и нам уже нужно, чтобы стили загружались динамически на основе пользовательского ввода, фонового выполнения кода и по другим причинам. В подобных сценариях CSS-модель, использующая правила стиля или встроенные стили, уже не сгодится. Несмотря на то что псевдоселекторы вроде *hover* предоставляют некоторую поддержку, мы по-прежнему ограничены в возможностях.

В данном случае полноценным решением будет использование JavaScript. Он позволит нам не только стилизовать элемент, с которым мы взаимодействуем, но и, что еще важнее, стилизовать элементы по всей странице. Такая свобода действий имеет огромный потенциал и вы-

ходит далеко за рамки ограниченных возможностей CSS, позволяющих стилизовать содержимое только внутри себя или в непосредственной близости.

Два подхода стилизации

Из начала главы мы знаем, что есть два способа изменять стиль элемента с помощью JavaScript. Первый — это установка свойства CSS непосредственно в элементе. Второй — добавление или удаление значений класса из элемента, что может приводить к применению или игнорированию определенных правил стиля. Рассмотрим оба случая подробнее.

Установка стиля напрямую

Каждый HTML-элемент, к которому вы обращаетесь через JavaScript, имеет объект `style`. Этот объект позволяет определять свойство CSS и устанавливать его значение. Например, вот как будет выглядеть установка фонового цвета HTML-элемента со значением `id` — `superman`:

```
let myElement = document.querySelector("#superman");
myElement.style.backgroundColor = "#D93600";
```

Чтобы воздействовать на несколько элементов, вы можете сделать следующее:

```
let myElements = document.querySelectorAll(".bar");

for (let i = 0; i < myElements.length; i++) {
  myElements[i].style.opacity = 0;
}
```

Коротко говоря, для прямой стилизации элементов с помощью JavaScript первым шагом будет обращение к элементу. При этом будет очень удобно использовать знакомый нам метод `querySelector`. Вторым шагом будет нахождение интересующего вас свойства CSS и указание для него значения. Помните, что многие значения в CSS являются строками. Также помните, что многим значениям для распознания требуются единицы измерения вроде `px`, `em` или аналогичные.

И последнее. Некоторые свойства CSS требуют, чтобы более сложные значения предоставлялись в виде текста, сопровождаемого значени-

ем, которое нужно вам самим. Одно из наиболее популярных свойств в этом списке — это свойство преобразования. В качестве подхода для установки сложного значения можно использовать старую добрую конкатенацию:

```
myElement.style.transform = "translate3d(" + xPos + ", " + yPos + "px, 0)";
```

Но такой вариант быстро надоест, поскольку отслеживание расстановки кавычек и прочие действия весьма утомительны и могут вызвать множество ошибок. Другое решение уже не так сложно и подразумевает использование синтаксиса шаблонного литерала:

```
myElement.style.transform = `translate3d(${xPos}px, ${yPos}px, 0)`;
```

Обратите внимание, что этот подход позволяет вам по-прежнему применять пользовательские значения, избегая при этом всей сложности конкатенации строк.

ВЫНУЖДЕННОЕ ИЗМЕНЕНИЕ ИМЕН НЕКОТОРЫХ СВОЙСТВ CSS

JavaScript очень требователен к содержанию допустимого имени свойства. Большинство имен в CSS будут одобрены JS, поэтому вы можете использовать их в изначальном виде. Тем не менее кое-что стоит помнить.

Чтобы указать в JavaScript свойство CSS, содержащее тире, просто удалите это тире. Например, `background-color` станет `backgroundColor`, а `border-radius` будет `borderRadius` и т. д.

Кроме того, некоторые слова в JavaScript зарезервированы и не могут быть использованы как есть. Один из примеров свойств CSS, подпадающих под эту категорию, является `float`. В CSS это свойство макета. В JavaScript же оно обозначает нечто иное. Чтобы использовать свойство, чье имя зарезервировано, используйте для него префикс `css`, то есть напишите не `float`, а `cssfloat`.

Добавление и удаление классов с помощью JavaScript

Второй подход подразумевает добавление и удаление значений классов, которые, в свою очередь, определяют, какие правила стиля будут применены. Предположим, у нас есть следующее правило стиля:

```
.disableMenu
  { display: none;
}
```

В HTML у нас есть меню, чей `id` — `dropDown`:

```
<ul id="dropDown">
  <li>One</li>
  <li>Two</li>
  <li>Three</li>
  <li>Four</li>
  <li>Five</li>
  <li>Six</li>
</ul>
```

Теперь если мы захотим применить стиль `.disableMenu` к этому элементу, то нам всего лишь понадобится добавить `disableMenu` в качестве значения `class` к элементу `dropDown`:

```
<ul class="disableMenu" id="dropDown">
  <li>One</li>
  <li>Two</li>
  <li>Three</li>
  <li>Four</li>
  <li>Five</li>
  <li>Six</li>
</ul>
```

Одним из вариантов сделать это будет установление свойства `className` элемента. Этот подход мы уже видели ранее. Проблема же с `className` в том, что мы становимся ответственны за поддержание текущего списка применяемых значений классов. Помимо этого, сам этот список возвращается в виде строки. Если у нас есть несколько значений классов, которые мы хотим добавить, удалить или просто включать и выключать, то понадобится проделать множество трюков со строками, что спровоцирует появление ошибок.

Эти неудобства может сгладить гораздо более приятный API, который делает добавление и удаление значений классов из элемента до смешного легким. Этот новый API известен как `classList`. Он предоставляет набор методов, которые упрощают работу со значениями классов:

- `add`;
- `remove`;
- `toggle`;
- `contains`.

Назначение этих методов угадывается из их имен, но все же давайте их изучим внимательнее.

Добавление значений классов

Чтобы добавить значение класса элементу, получите ссылку на этот элемент и вызовите для него метод `add` через `classList`:

```
let divElement = document.querySelector("#myDiv");
divElement.classList.add("bar");
divElement.classList.add("foo");
divElement.classList.add("zorb");
divElement.classList.add("baz");

console.log(divElement.classList);
```

После выполнения этого кода элемент `div` будет иметь следующие значения класса: `bar`, `foo`, `zorb`, `baz`. API `classList` берет на себя добавление пробелов между значениями классов. Если же мы укажем недопустимое значение класса, API `classList` будет ругаться и добавлять его не станет. Если мы прикажем методу `add` добавить класс, уже существующий в элементе, то код по-прежнему будет выполняться, но повторяющееся значение класса добавлено не будет.

Удаление значений классов

Для удаления значения класса мы можем вызвать метод `remove` также через `classList`:

```
let divElement = document.querySelector("#myDiv");
divElement.classList.remove("foo");

console.log(divElement.classList);
```

После выполнения этого кода значение класса `foo` будет удалено и останутся только `bar`, `baz` и `zorb`. Несложно, правда?

Переключение значений класса

Для многих сценариев стилизации применяется стандартный рабочий процесс. Во-первых, мы проверяем, существует ли значение класса в элементе. Если да, мы удаляем его из элемента. Если же нет, то мы, наоборот, его добавляем. Чтобы упростить этот тривиальный шаблон переключения, API `classList` предоставляет метод `toggle`:

```
let divElement = document.querySelector("#myDiv");
divElement.classList.toggle("foo"); // удаляет foo
divElement.classList.toggle("foo"); // добавляет foo
divElement.classList.toggle("foo"); // удаляет foo

console.log(divElement.classList);
```

Метод `toggle` при каждом вызове добавляет указанное значение класса в элемент или удаляет его. В нашем случае класс `foo` при первом вызове метода `toggle` удаляется. Во второй раз класс `foo`, наоборот, добавляется. В третий раз снова удаляется. Думаю, суть вы уловили.

Проверка наличия значения класса

Последним рассмотрим метод `contains`:

```
let divElement = document.querySelector("#myDiv");

if (divElement.classList.contains("bar") == true) {
  // делает что-нибудь
}
```

Этот метод проверяет, существует ли указанное значение класса в элементе. Если да, то возвращается `true`, если нет — `false`.

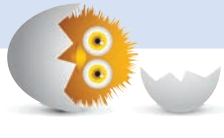
Углубление

Как вы видите, API `classList` предоставляет почти все, что нужно для удобного добавления, удаления или проверки значений классов в элементе. Ключевое слово «почти». О том немногом, что этот API все же не предлагает по умолчанию, можно почитать в моей статье http://bit.ly/kClassList, рассказывающей о многих других возможностях `classList`.

КОРОТКО О ГЛАВНОМ

Итак, теперь у вас есть два отличных основанных на JavaScript подхода для стилизации элементов. Если есть возможность модифицировать CSS, рекомендую стилизовать элементы посредством добавления и удаления классов. Причина связана с тем, что такой подход существенно упрощает обслуживание кода. Гораздо легче добавлять или удалять свойства стилей из правила стиля в CSS, чем добавлять и удалять строки кода JavaScript.

Остались вопросы? Дружелюбные форумчане ответят вам. Обращайтесь на https://forum.kirupa.com!



В ЭТОЙ ГЛАВЕ:

- освоим навигацию по дереву DOM;
- используем различные API, доступные для перемещения и переподчинения элементов;
- научимся находить братьев элементов, а также их родителей и потомков.

28

ПЕРЕМЕЩЕНИЕ ПО DOM

Как вы уже поняли, DOM — это огромное дерево с ветвями, на которых висит множество элементов. Если выразиться более точно, элементы в DOM упорядочены иерархически (рис. 28.1), определяя то, что мы в итоге видим в браузере.

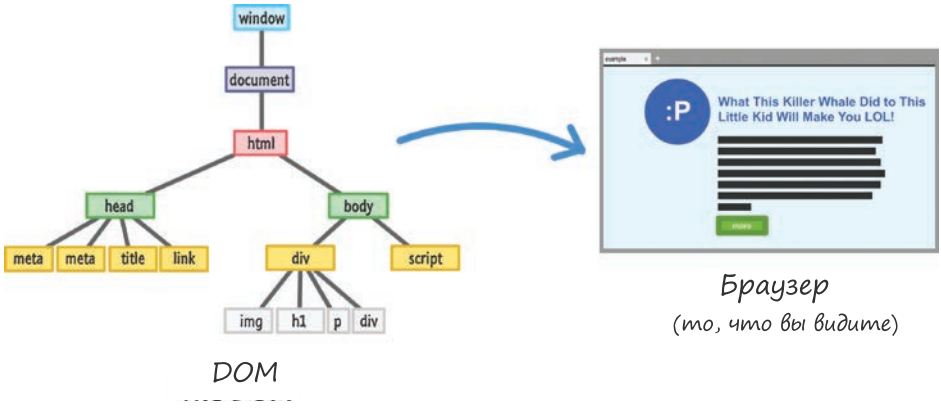


РИС. 28.1.
DOM и браузер неразрывно связаны

Эта иерархия помогает нам организовывать HTML-элементы. Помимо этого, она объясняет CSS, какие стили и к чему применять. С точки же зрения JavaScript эта иерархия привносит некоторую сложность, и нам придется тратить немало времени, определяя, где именно в DOM мы находимся в данный момент и куда хотим попасть. Это станет более очевидно, когда мы взглянем на создание новых элементов или их перемещение. С этой сложностью важно научиться ладить и чувствовать себя комфортно.

Этому и будет посвящена данная глава. Чтобы понять, как с легкостью переходить от ветви к ветви, DOM предоставляет ряд свойств, которые можно совмещать с уже известными техниками.

Поехали!

Поиск пути

Прежде чем найти элементы и работать с ними, важно понять, где они находятся. Самый простой способ решить эту проблему — начать сверху и постепенно спускаться, что мы и сделаем.

Вид вершины нашей DOM представлен элементами `window`, `document` и `html` (рис. 28.2).

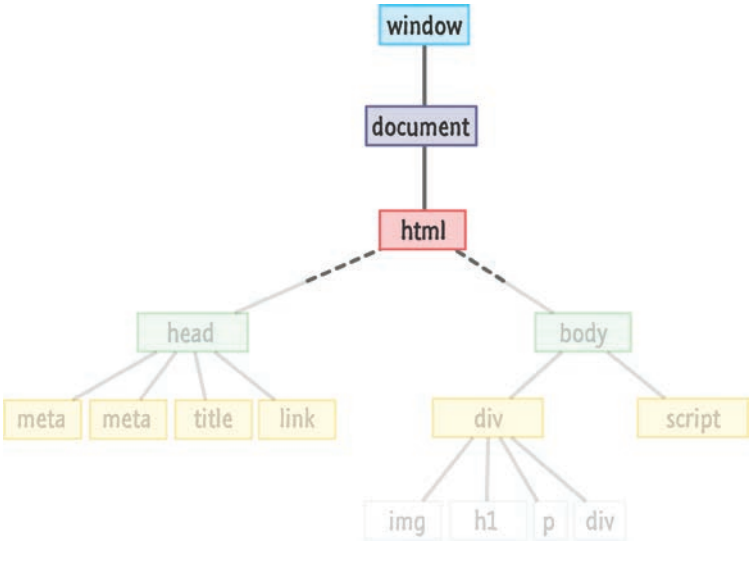


РИС. 28.2.
Вид вершины этого дерева всегда одинаков

В связи с повышенной важностью этих трех элементов DOM предоставляет к ним легкий доступ посредством `window`, `document` и `document.documentElement`:

```
let windowObject = window; // хм-м....
let documentObject = document; // Это наверняка необязательно
let htmlElement = document.documentElement;
```

Здесь стоит отметить, что и `window`, и `document` являются глобальными свойствами. Нам не обязательно явно объявлять их подобно тому, как сделал я. Используйте их сразу.

Как только мы спускаемся ниже уровня HTML-элемента, наша DOM начинает ветвление и становится гораздо интереснее. С этого места можно использовать несколько способов навигации. Один из них мы уже видели многократно, и связан он с использованием `querySelector` и `querySelectorAll` для получения в точности тех элементов, которые нам нужны. На практике зачастую эти два метода слишком ограничены.

Иногда мы не знаем, куда именно нужно направиться. В этом случае методы `querySelector` и `querySelectorAll` уже не помогут. Нам просто нужно сесть за руль и ехать в надежде, что удастся найти то, что мы ищем. Когда дело доходит до навигации по DOM, мы зачастую будем оказываться в подобном положении. Именно здесь нам и помогут различные свойства, предлагаемые DOM, которые мы изучим далее.

Разобраться нам поможет знание того, что все элементы в DOM имеют по меньшей мере одну комбинацию *родителей*, *братьев* (*соседних элементов*) и *потомков*, на которых можно ориентироваться. Для наглядного представления посмотрите на ряд, содержащий элементы `div` и `script`, как показано на рис. 28.3.

Элементы `div` и `script` являются братьями, так как имеют одного родителя — элемент `body`. Элемент `script` не имеет потомков, но у `div`, напротив, они есть. `Img`, `h1`, `p` и `div` являются потомками элемента `div`, при этом все потомки одного родителя между собой являются братьями. Как и в реальной жизни, положение родителя, потомка и брата зависит от того, на какой части дерева мы фокусируемся. То есть практически каждый элемент в зависимости от угла обзора может выступать в различных ролях.

Для упрощения работы с этим всем у нас есть несколько свойств, на которые мы и будем полагаться. Ими являются `firstChild`, `lastChild`, `parentNode`, `children`, `previousSibling` и `nextSibling`. Просто глядя на их названия, вы можете догадаться, какую именно роль они играют. Дьявол в деталях, поэтому рассмотрим все подробно.

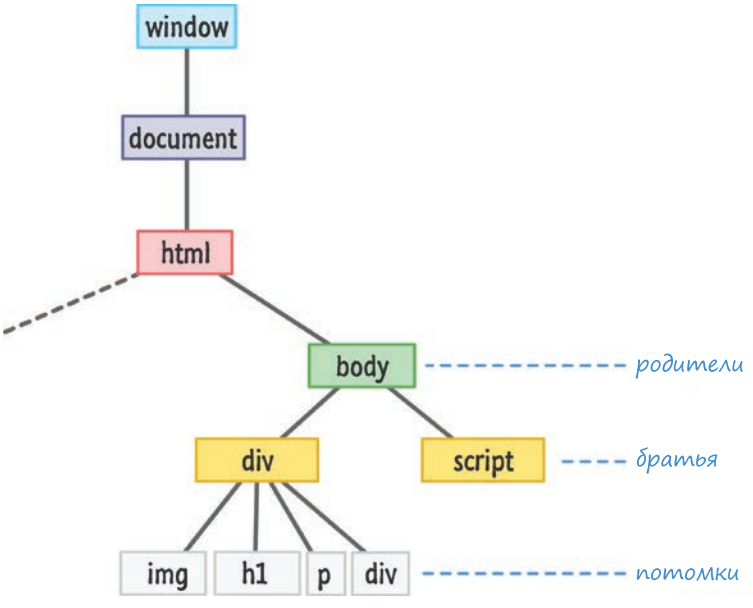


РИС. 28.3.
Пример нашего дерева с родителями, братьями и потомками

Работа с братьями и родителями

Из всех свойств легче всего работать с относящимися к родителям и братьям, а именно `parentNode`, `previousSibling` и `nextSibling`. Схема на рис. 28.4 дает представление о том, как эти свойства работают.

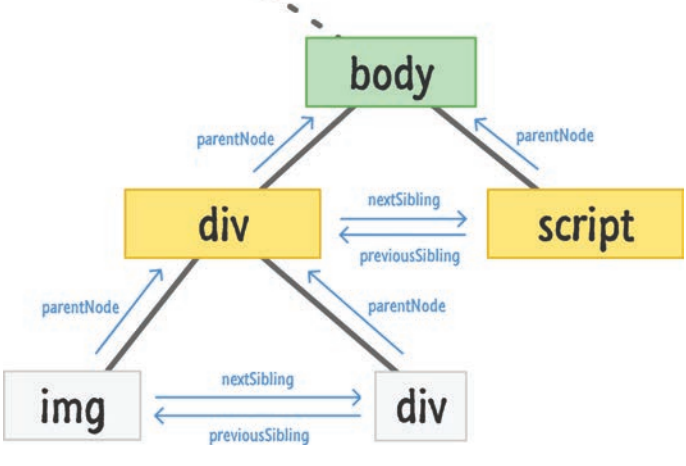


РИС. 28.4.
Связь между братьями и родителями с позиции DOM

Схема несколько перегружена, но понять, что на ней происходит, можно. Свойство `parentNode` указывает на родителя элемента. Свойства `previousSibling` и `nextSibling` позволяют элементу найти его предыдущего или следующего брата — если мы будем следовать по стрелкам на рисунке, то увидим это. В нижней строке `nextSibling` нашего `img` это `div`. `previousSibling` для `div` — это `img`. Обращение к `parentNode` в любом из этих элементов приведет вас к родителю `div` во втором ряду. Здесь все достаточно понятно.

Давай заведем детей!

Менее понятно, как во все это вписываются потомки. Поэтому давайте взглянем на свойства `firstChild`, `lastChild` и `children`, показанные на рис. 28.5.

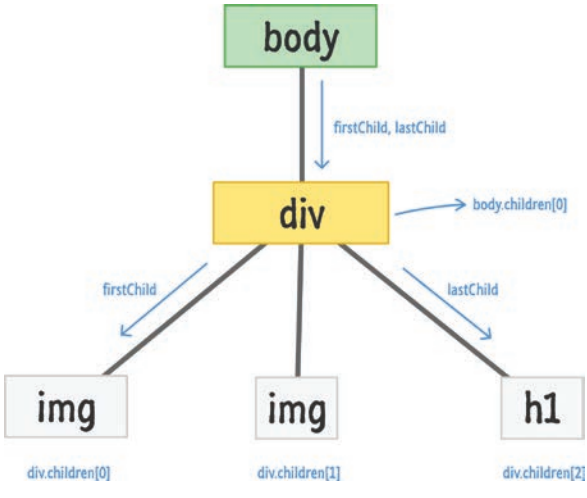


РИС. 28.5.
Представление потомков и их потомков

Свойства `firstChild` и `lastChild` относятся к первому и последнему дочерним элементам родителя. Если у родителя есть всего один потомок, как в случае с элементом `body` из нашего примера, тогда и `firstChild`, и `lastChild` будут указывать на одно и то же. Если у элемента нет потомков, то эти свойства будут возвращать `null`.

Самое хитрое из всех этих свойств — свойство `children`. Когда вы обращаетесь к свойству `children` в родителе, то по умолчанию получаете коллекцию дочерних элементов, которые у него есть. Эта коллекция не является `Array`, но при этом имеет некоторые присущие массиву возмож-

ности. Как и в случае с массивом, вы можете перебирать эту коллекцию или обращаться к ее потомкам по отдельности. У этой коллекции есть свойство `length`, которое сообщает, с каким количеством потомков взаимодействует родитель. Если у вас уже голова пошла кругом, не переживайте. Код из следующего раздела прояснят сказанное.

Складываем все воедино

Теперь, когда вы разобрались во всех важных свойствах, имеющихся для перемещения по DOM, давайте рассмотрим листинги, которые свяжут все диаграммы и слова в несколько приятных строк JavaScript.

Проверка наличия потомка

Чтобы проверить, есть ли у элемента потомок, мы можем сделать следующее:

```
let bodyElement = document.querySelector("body");

if (bodyElement.firstChild) {
  // Делает что-нибудь интересное
}
```

Эта инструкция `if` вернет `null`, если потомков не существует. Мы могли бы также использовать `bodyElement.lastChild` или `bodyElement.children.count`, если бы любили много печатать, но я предпочитаю простые варианты.

Обращение ко всем потомкам

Если нужно обратиться ко всем потомкам родителя, всегда можно прибегнуть к старому доброму циклу `for`:

```
let bodyElement = document.body;

for (let i = 0; i < bodyElement.children.length; i++) {
  let childElement = bodyElement.children[i];

  document.writeln(childElement.tagName);
}
```

Обратите внимание, что мы используем свойства `children` и `length` так же, как делали бы это в случае с `Array`. Стоит отметить, что эта

коллекция по сути не является `Array`. Практически все методы `Array`, которые мы можем захотеть использовать, не будут доступны для этой коллекции, возвращенной свойством `children`.

Прогулка по DOM

Последний фрагмент затрагивает все, что мы видели до сих пор. Он рекурсивно проходится по DOM и касается каждого HTML-элемента, который находит:

```
function theDOMElementWalker(node) {
  if (node.nodeType == Node.ELEMENT_NODE) {

    console.log(node.tagName);
    node = node.firstChild;

    while (node) {
      theDOMElementWalker(node);
      node = node.nextSibling;
    }
  }
}
```

Чтобы увидеть эту функцию в деле, мы просто вызываем ее, передавая узел, с которого хотим начать путь:

```
let texasRanger = document.querySelector("#texas");
theDOMElementWalker(texasRanger);
```

В этом примере мы вызываем функцию `theDOMElementWalker` для элемента, на который ссылается переменная `texasRanger`. Если вы хотите выполнить некий код для элемента, найденного этим скриптом, замените закомментированную строку на то, что хотите сделать.

КОРОТКО О ГЛАВНОМ

Нахождения пути внутри DOM — это один из тех навыков, которым должен обладать каждый JavaScript-разработчик. Этот урок предоставляет обзор доступных техник. Применение же всего этого материала на практике уже полностью ложится на вас… или хорошего друга, который поможет со всем этим разобраться. В последующих уроках мы углубимся в эту тему еще больше. Разве это не звучит захватывающе?

Появился вопрос? Обращайтесь на https://forum.kirupa.com за ответом от дружественных разработчиков, таких же, как мы с вами.

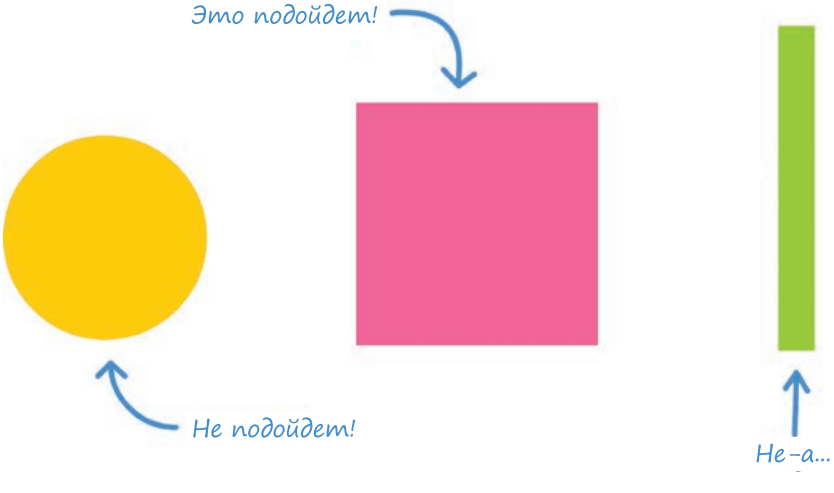
В ЭТОЙ ГЛАВЕ:

- поймем, как легко с помощью Javascript создавать элементы DOM буквально из ничего;
- научимся клонировать существующие элементы DOM, а также удалять уже не нужные.

29

СОЗДАНИЕ И УДАЛЕНИЕ ЭЛЕМЕНТОВ DOM

Здесь реально может заштормить. Поэтому держитесь крепче во время чтения следующих разделов:



Независимо от того, какое представление сформировалось у вас на основе предыдущих обсуждений DOM, наша DOM не обязана состоять только из HTML-элементов, существующих в разметке. Есть возможность создавать HTML-элементы прямо из воздуха и добавлять их в DOM, используя всего несколько строк JavaScript. Также есть возможность перемещать элементы, удалять их и проделывать с ними многие другие богоподобные действия. Сделайте паузу, осмыслите прочитанное, дайте ему осесть в своем сознании.

Помимо изначальной крутизны всего этого возможность динамического создания и изменения элементов в DOM является важной деталью, благодаря которой работают многие из наших любимых сайтов и приложений. Если об этом как следует подумать, то все встает на свои места. Если в нашем HTML все будет предопределено, то это сильно ограничит его возможности. Нам же нужно, чтобы содержимое изменялось и адаптировалось при поступлении новых данных, при взаимодействии со страницей, при прокручивании или при выполнении множества других действий.

В этой главе мы познакомимся с основами того, что позволяет всему этому работать. Мы рассмотрим создание элементов, удаление элементов, их переподчинение и клонирование. И эта глава будет последней, где изучаются особенности DOM. Так что можете звать друзей и надувать шары к празднику!

Поехали!

Создание элементов

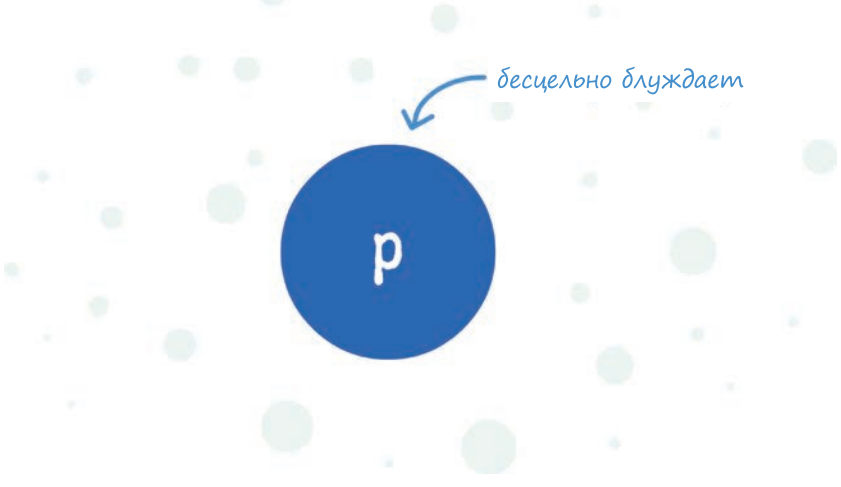
Для интерактивных сайтов и приложений динамическое создание HTML-элементов и помещение их в DOM — вполне обыденная процедура. Если вы впервые слышите о том, что такое возможно, то этот раздел точно вам понравится!

Создавать элементы можно с помощью метода `createElement`, который работает достаточно просто. Он вызывается через объект `document`, и ему передается имя HTML-тега элемента, который нужно создать. В следующем фрагменте кода создается элемент абзаца, представленный буквой `p`:

```
let myElement = document.createElement("p");
```

Переменная `myElement` содержит ссылку на только что созданный элемент.

Если мы запустим приложение, добавив эту строку, то при своем выполнении она создаст элемент `p`. Создание элемента — это простая часть. Усилия придется приложить, чтобы сделать его веселым и ответственным членом нашей DOM. Требуется поместить этот элемент в определенное место DOM, и пока что наш динамически созданный элемент `p` где-то совершенно бесцельно блуждает:



Причина в том, что DOM не знает о существовании этого элемента, и, чтобы он стал полноценной частью DOM, нужно сделать две вещи:

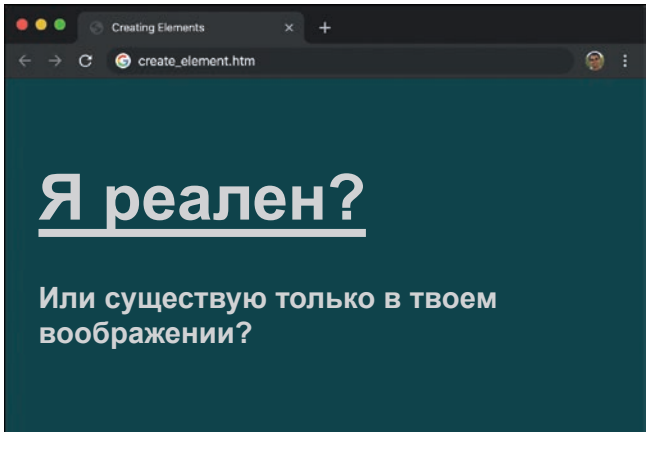
1. Найти элемент, который выступит в качестве его родителя.
2. Использовать `appendChild` и присоединить элемент к этому родителю.

Легче всего это будет понять на примере, который все объединяет. Если вы хотите проработать это самостоятельно, то создайте HTML-документ и добавьте в него следующий код HTML, CSS и JS:

```
<!DOCTYPE html>
<html>

<head>
  <title>Creating Elements</title>

  <style>
    body {
      background-color: #0E454C;
      padding: 30px;
    }
```

```
    h1 {
      color: #14FFF7;
      font-size: 72px;
      font-family: sans-serif;
      text-decoration: underline;
    }

    p {
      color: #14FFF7;
      font-family: sans-serif;
      font-size: 36px;
      font-weight: bold;
    }
  </style>
</head>

<body>
  <h1>Am I real?</h1>
  <script>
    let newElement = document.createElement("p");
    let bodyElement = document.querySelector("body");

    newElement.textContent = "Or do I exist entirely in your
                             imagination?";

    bodyElement.appendChild(newElement);
  </script>
</body>

</html>
```

Сохраните файл и просмотрите его в браузере. Если все сработало, то вы увидите что-то похожее на следующий скриншот:



Отступим на шаг назад и разберем этот пример подробней. Все, что нам нужно для создания элемента и его добавления в DOM, находится между тегами `script`:

```
let newElement = document.createElement("p");
let bodyElement = document.querySelector("body");

newElement.textContent = "Or do I exist entirely in your
imagination?";

bodyElement.appendChild(newElement);
```

`newElement` хранит ссылку на наш созданный тег `p`. В `bodyElement` хранится ссылка на элемент `body`. В только что созданном элементе (`newElement`) мы устанавливаем свойство `textContent`, указав, что в итоге нужно отобразить.

В конце мы берем бесцельно блуждающий `newElement` и присоединяем его к элементу-родителю `body` с помощью функции `appendChild`.

На рис. 29.1 показано, как выглядит DOM для нашего простого примера.

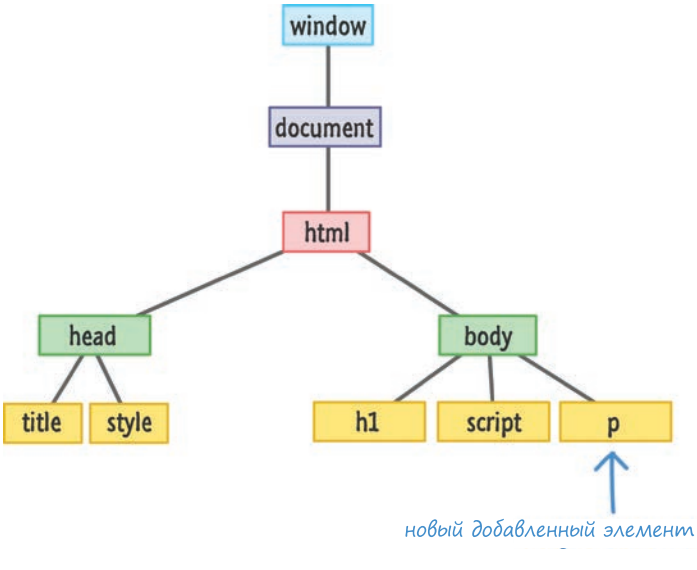


РИС. 29.1.
Как выглядит DOM после выполнения приведенного кода

Следует учитывать, что функция `appendChild` всегда добавляет элемент после всех потомков, которые могут быть у родителя. В нашем случае

элемент `body` уже имеет в качестве потомков элементы `h1` и `script`. Элемент `p` присоединяется после них как самый новый. В итоге появлятся контроль над позицией, в которой будет размещен конкретный элемент под родителем.

Если мы хотим вставить `newElement` сразу после тега `h1`, то можем сделать это, вызвав функцию `insertBefore` для родителя. Функция `insertBefore` получает два аргумента. Первый из них является элементом-вставкой, а второй — ссылкой на брата (то есть другого потомка родителя), которому этот элемент должен предшествовать. Далее приведен измененный пример, где наш `newElement` помещен после элемента `h1` (и перед элементом `script`):

```
let newElement = document.createElement("p");
let bodyElement = document.querySelector("body");
let scriptElement = document.querySelector("script");

newElement.textContent = "I exist entirely in your imagination.";

bodyElement.insertBefore(newElement, scriptElement);
```

Обратите внимание, что мы вызываем `insertBefore` для `bodyElement` и указываем, что `newElement` должен быть вставлен перед элементом `script`. Наша DOM в этом случае будет выглядеть так, как показано рис. 29.2.

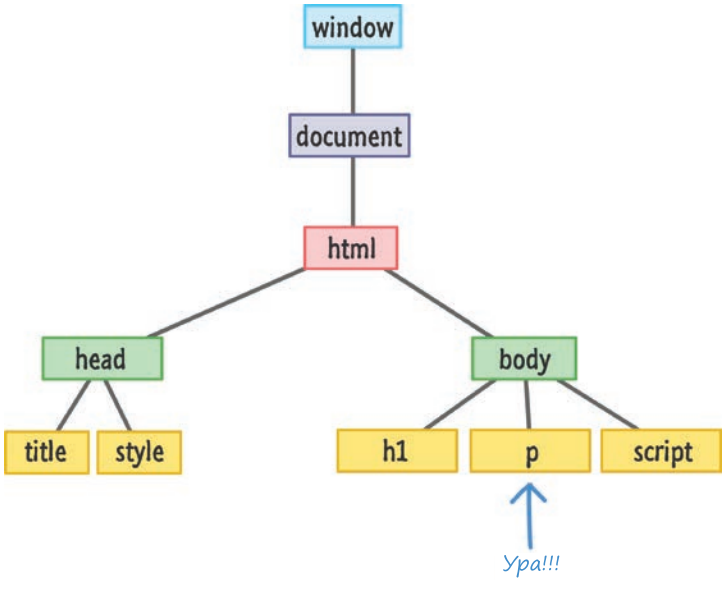


РИС. 29.2.
Вставленный элемент находится между элементами h1 и script

Вы можете подумать, что если есть метод `insertBefore`, то должен быть и метод `insertAfter`. Но на самом деле это не так. Здесь нет встроенного способа для вставки элемента после, а не до другого элемента. Мы можем только перехитрить функцию `insertBefore`, сказав ей вставить элемент *перед элементом, следующим за нужным*. Но будет ли в этом смысл? Сначала я покажу пример и затем все объясню:

```
let newElement = document.createElement("p");
let bodyElement = document.querySelector("body");
let h1Element = document.querySelector("h1");

newElement.textContent = "I exist entirely in your imagination.";

bodyElement.insertBefore(newElement, h1Element.nextSibling);
```

Обратите внимание на выделенные строки, а затем взгляните на рис. 29.3, где видно положение до и после выполнения кода.

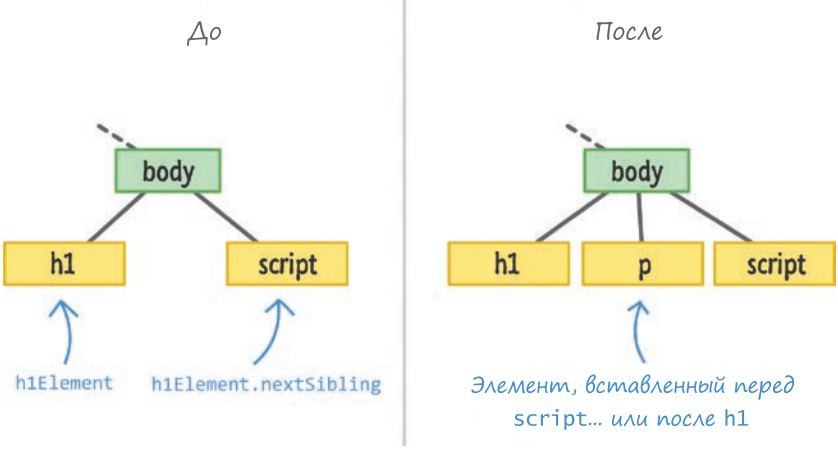


РИС. 29.3.
Трюк для имитирования действия insertAfter

Вызов `hiElement.nextSibling` находит элемент `script`. Вставка `newElement` перед элементом `script` удовлетворяет нашу цель вставить элемент после `h1`. А что, если нет элемента-брата, на который можно указать? В таком случае функция `insertBefore` достаточно умна и просто автоматически вставляет элемент в конец.

УДОБНАЯ ФУНКЦИЯ

Если по какой-то причине вам нужно все время вставлять элементы после другого потомка, то можете использовать эту функцию, чтобы немного упростить себе жизнь:

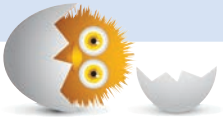
```
function insertAfter(target, newElement) {
  target.parentNode.insertBefore(newElement,
                                 target.nextSibling);
}
```

Да, я понимаю, что это окольный путь, но он работает, и весьма прилично. Вот пример этой функции в действии:

```
let newElement = document.createElement("p");
let bodyElement = document.querySelector("body");
let h1Element = document.querySelector("h1");

newElement.textContent = "I exist entirely in your imagination.";

function insertAfter(target, element) {
  target.parentNode.insertBefore(element, target.nextSibling);
}

insertAfter(bodyElement, newElement);
```

Можно пойти еще дальше и расширить этой функцией HTML-элемент, чтобы предоставить ее функциональность для всех HTML-элементов. В главе 19 «Расширение встроенных объектов» рассматривается, как сделать нечто подобное. Имейте в виду, что некоторые разработчики не одобряют расширение DOM, поэтому заготовьте какие-нибудь остроумные отговорки на случай, если эти некоторые начнут вам досаждать.

Более обобщенный подход к добавлению потомков родителю основан на понимании, что элементы-родители рассматривают потомков как точки входа в массив. Чтобы обратиться к этому массиву потомков, у нас есть свойства `children` и `childNodes`. Свойство `children` возвращает только HTML-элементы, а `childNodes` возвращает более обобщенные узлы, представляющие много того, до чего нам нет дела. Да, я осознаю, что повторяюсь, и вы можете пересмотреть главу 28 «Перемещение по DOM», чтобы лучше разобраться в способах точного указания на элемент.



Удаление элементов

Мне кажется, что следующая фраза принадлежит какому-то умному человеку: «*То, что имеет силу создавать, имеет силу и удалять*». В предыдущем разделе мы видели, как можно использовать метод `createElement` для создания элементов. В текущем разделе мы рассмотрим метод `removeChild`, который, несмотря на пугающее имя, занимается именно удалением элементов.

Взгляните на следующий фрагмент кода, который можно создать для работы со знакомым нам примером:

```
let newElement = document.createElement("p");
let bodyElement = document.querySelector("body");
let h1Element = document.querySelector("h1");

newElement.textContent = "I exist entirely in your imagination.";

bodyElement.appendChild(newElement);

bodyElement.removeChild(newElement);
```

Элемент `p`, хранящийся в `newElement`, добавляется к элементу `body` с помощью метода `appendChild`. Это мы уже видели раньше. Чтобы удалить этот элемент, вызывается `removeChild` для элемента `body` и передается указатель на элемент, который нужно удалить. Конечно же, это элемент `newElement`. Как только будет выполнен метод `removeChild`, все станет так, как будто DOM никогда не знала о существовании `newElement`.

Главное, обратите внимание, что нужно вызывать `removeChild` из родителя потомка, которого мы хотим удалить. Этот метод не будет отыскивать элемент для удаления по всей DOM. А теперь предположим, что у нас нет прямого доступа к родителю элемента и тратить время на его поиск мы не хотим. При этом все равно с легкостью можно удалить его с помощью свойства `parentNode`:

```
let newElement = document.createElement("p");
let bodyElement = document.querySelector("body");
let h1Element = document.querySelector("h1");

newElement.textContent = "I exist entirely in your imagination.";

bodyElement.appendChild(newElement);

newElement.parentNode.removeChild(newElement);
```

В этом варианте мы удаляем `newElement`, вызывая `removeChild` для его родителя, указав `newElement.parentNode`. Выглядит замысловато, но работает.

Теперь познакомимся с новым и лучшим способом удаления элемента, подразумевающим прямой вызов метода `remove` для элемента, который нужно удалить. Вот пример его использования:

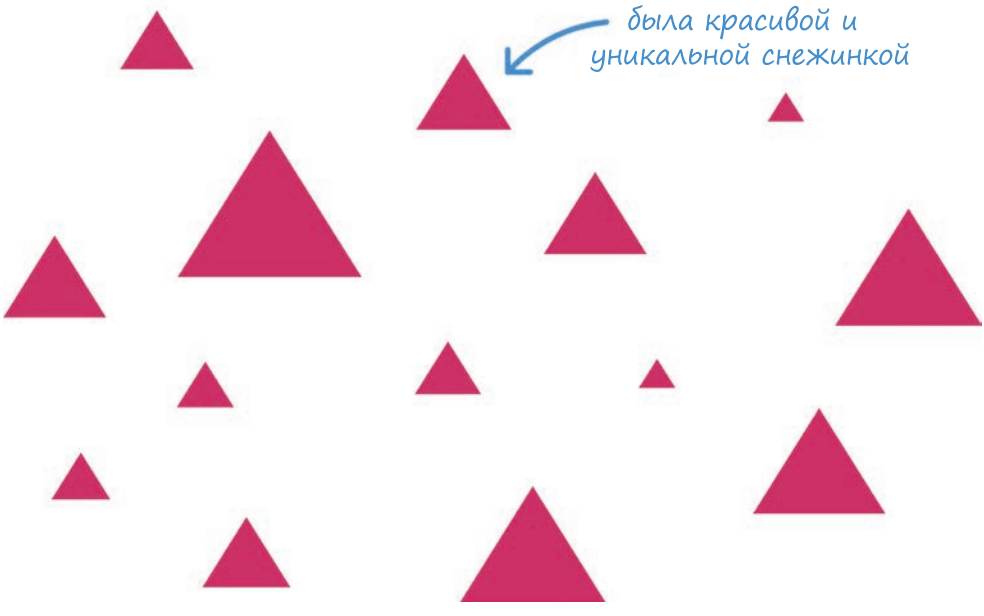
```
let newElement = document.createElement("p");
let bodyElement = document.querySelector("body");
let h1Element = document.querySelector("h1");

newElement.textContent = "I exist entirely in your imagination.";

bodyElement.appendChild(newElement);

newElement.remove();
```

Я не собираюсь заканчивать тему удаления элементов на этом методе `remove`. Почему? Все дело в поддержке браузера. Этот подход все еще нов, поэтому более старые версии браузеров вроде Internet Explorer его не поддерживают. Если для вас принципиальна поддержка IE, то подойдут другие рассмотренные подходы.

Если вы ищите универсальный способ удаления элементов, то функция `removeChild`, несмотря на ее причуды, весьма эффективна. Если нужен более прямолинейный способ, присмотритесь к `remove`. Оба этих подхода успешно справляются с удалением элементов DOM, включая те, что были изначально созданы в разметке. При этом мы не ограничены возможностью удаления динамически добавленных элементов. Если удаляемый элемент DOM имеет несколько уровней потомков и их потомков, то все они будут также удалены.

Клонирование элементов

По мере продвижения эта глава становится все запутаннее, но, к счастью, мы уже дошли до последнего раздела. Оставшаяся техника управления DOM, о которой стоит знать, связана с клонированием элементов, а именно с созданием их идентичных копий:



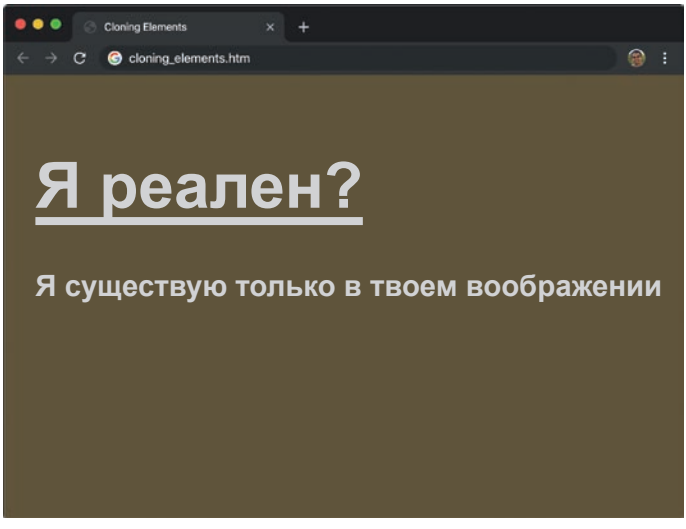
Клонирование производится с помощью вызова функции `cloneNode` для нужного элемента с аргументом `true` или `false` — это определяет, хотим мы клонировать только сам элемент или еще и всех его потомков. Вот как будет выглядеть код для клонирования элемента (и добавления его в DOM):

```
let bodyElement = document.querySelector("body");
let item = document.querySelector("h1");

let clonedItem = item.cloneNode(false);

// добавление клонированного элемента в DOM

bodyElement.appendChild(clonedItem);
```

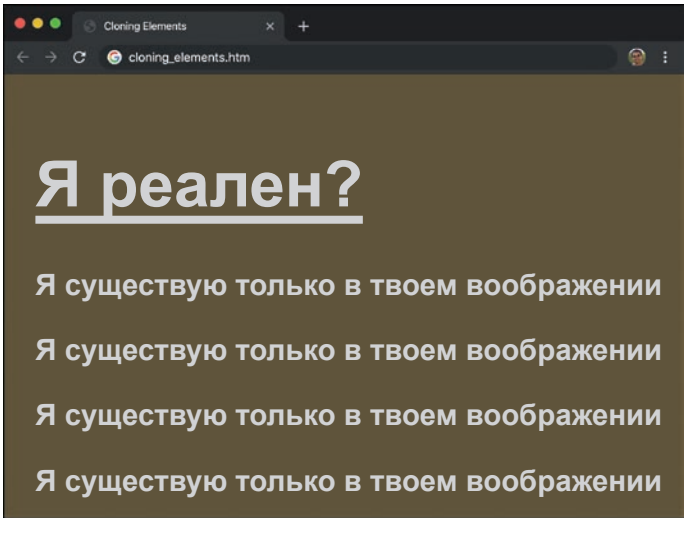
Как только клонированные элементы будут добавлены в DOM, можно применить уже изученные нами техники. Клонирование элементов является весьма важной функцией, поэтому давайте перейдем от рассмотрения фрагмента к более полному примеру:

```
<!DOCTYPE html>
<html>

<head>
  <title>Cloning Elements</title>

  <style>
    body {
      background-color: #60543A;
      padding: 30px;
    }

    h1 {
      color: #F2D492;
      font-size: 72px;
      font-family: sans-serif;
      text-decoration: underline;
    }

    p {
      color: #F2D492;
      font-family: sans-serif;
      font-size: 36px;
      font-weight: bold;
    }
  </style>
</head>

<body>
  <h1>Am I real?</h1>
  <p class="message">I exist entirely in your imagination.</p>

  <script>
    let bodyElement = document.querySelector("body");
    let textElement = document.querySelector(".message");

    setInterval(sayWhat, 1000);

    function sayWhat() {
      let clonedText = textElement.cloneNode(true);
      bodyElement.appendChild(clonedText);
    }
  </script>
</body>

</html>
```

Если вы поместите весь этот код в HTML-документ и просмотрите его в браузере, то увидите нечто напоминающее недавний пример:



Но спустя пару секунд вы заметите, что этот пример несколько отличается — тем, что его сообщение продолжает повторяться:



Секрет происходящего кроется в самом коде. Давайте вернемся назад, взглянем на код внутри тега `script` и попытаемся разобраться в происходящем:

```
let bodyElement = document.querySelector("body");
let textElement = document.querySelector(".message");
```

На самом верху есть переменная `bodyElement`, которая ссылается на элемент `body` в нашем HTML. Также есть переменная `textElement`, ссылающаяся на элемент `p` со значением класса `message`. Здесь нет ничего необычного.

А дальше уже интереснее. Есть функция-таймер `setInterval`, вызывающая функцию `sayWhat` каждые 1000 миллисекунд (1 секунду):

```
setInterval(sayWhat, 1000);
```

Сам процесс клонирования происходит внутри функции `sayWhat`:

```
function sayWhat() {
  let clonedText = textElement.cloneNode(true);
  bodyElement.appendChild(clonedText);
}
```

Мы вызываем `cloneNode` для `textElement`. В результате этого создается копия `textElement`, которая хранится как часть переменной `clonedText`. Последним шагом добавляем созданный элемент в DOM, чтобы он начал отображаться. Благодаря функции `setTimer` весь код после `sayWhat` повторяется и продолжает добавлять клонированный элемент на страницу.

Вы могли заметить, что мы клонируем следующий элемент абзаца:

```
<p class="message">I exist entirely in your imagination.</p>
```

В коде же мы указали следующее:

```
let clonedText = textElement.cloneNode(true);
```

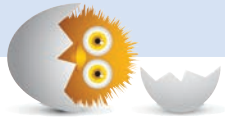
Мы вызываем `cloneNode` с флагом `true`, обозначая, что хотим клонировать и всех потомков. Зачем? Ведь у нашего элемента абзаца, кажется, нет потомков. Что ж, именно здесь и проявляется различие между `elements` и `nodes`. Тег абзаца не имеет дочерних `elements`, но текст, обернутый тегом `p`, является дочерним `node`. Этот нюанс важно учитывать, когда вы клонируете что-либо и в итоге получаете не то, что хотели, указав, что потомков клонировать не нужно.

КОРОТКО О ГЛАВНОМ

Подытожим: DOM можно не только использовать, но и всячески изменять. Мы уже мимоходом обсуждали, как все в DOM может быть изменено, но именно здесь впервые увидели глубину и ширину доступных изменений, которые можно производить с помощью таких методов, как `createElement`, `removeElement`, `remove` и `cloneNode`.

Изучив весь этот материал, вы сможете начать с абсолютно чистой страницы и с помощью всего нескольких строк JavaScript-кода заполнить ее всем необходимым:

```
<!DOCTYPE html>
<html>

<head>
  <title>Look what I did, ma!</title>
</head>

  <body>
    <script>
      let bodyElement = document.querySelector("body");

      let h1Element = document.createElement("h1");
      h1Element.textContent = "Do they speak English
                              in 'What'?";

      bodyElement.appendChild(h1Element);

      let pElement = document.createElement("p");
      pElement.textContent = "I am adding some text here...
                             like a boss!";

      bodyElement.appendChild(pElement);
    </script>
</body>

</html>
```



В ЭТОЙ ГЛАВЕ:

- узнаем, как можно сэкономить время с помощью браузерных инструментов разработчика;
- познакомимся с возможностями инструментов разработки Chrome.

30

БРАУЗЕРНЫЕ ИНСТРУМЕНТЫ РАЗРАБОТЧИКА

Серьезные браузеры — Google Chrome, Apple Safari, Mozilla Firefox и Microsoft Edge (бывший Internet Explorer) — не только отображают страницы. Они дают разработчикам доступ к богатой и интересной функциональности, позволяющей разобраться в том, что именно происходит на отображаемой странице. Эта возможность реализована в так называемых *инструментах разработчика*, которые встроены в браузер и позволяют работать с HTML, CSS и JavaScript всяческими изощренными способами.

Рассмотрим здесь эти инструменты и узнаем, как с их помощью облегчить себе жизнь.

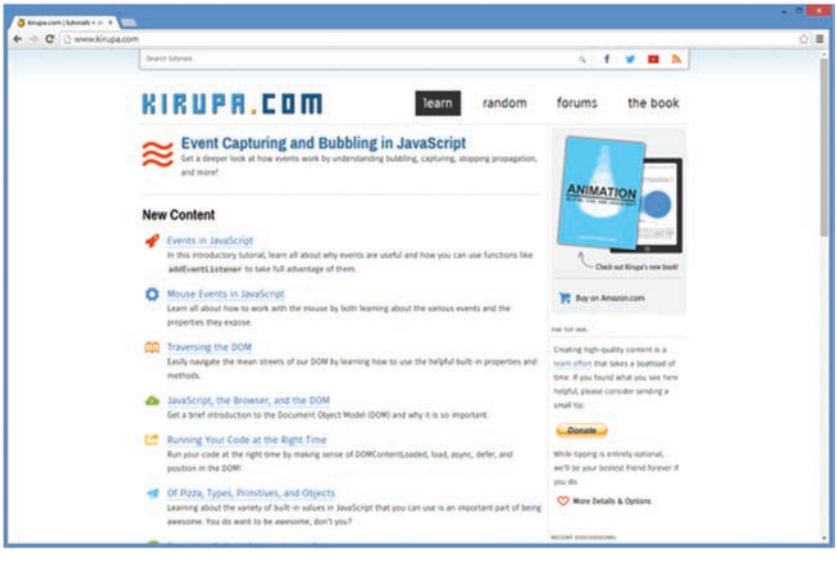
Поехали!

Я ИСПОЛЬЗУЮ GOOGLE CHROME

В следующих примерах я буду использовать браузер Google Chrome. Хотя в любом браузере есть аналогичные функции, интерфейс и шаги, которые необходимо выполнить, будут различаться. Просто помните об этом, а также обратите внимание, что версия Chrome может быть новей, чем та, которая используется в этой главе.

Знакомство с инструментами разработчика

Начнем с азов. Когда вы переходите на веб-страницу, браузер загружает тот документ, который ему велено загружать:



Выглядит знакомо, так как эта часть функционала особо не менялась с момента выпуска первого браузера в 1800-х или когда его там выпустили. Если используете Chrome, нажмите Cmd-Opt-I для Mac или F12 (или Ctrl + Shift + I) для Windows.

Нажмите нужные клавиши и посмотрите, что произойдет. Тревожная музыка, сопровождаемая землетрясением и лазерными выстрелами, скорее всего, не зазвучит, но совершенно точно изменится макет страницы браузера. Он покажет нечто загадочное. Обычно это изменение происходит в нижней или правой части экрана, как показано на рис. 30.1.

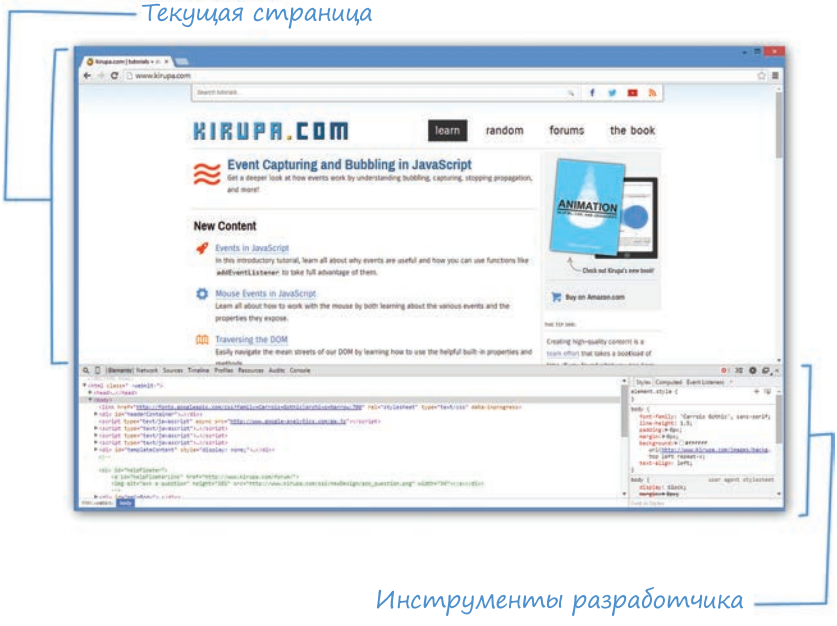


РИС. 30.1.
Браузер с активированными инструментами разработчика внизу экрана

Браузер разделится на две части. Одна из них отвечает за отображение веб-страниц. Мы с ней уже знакомы, и она нам очень нравится. Вторая же — новая часть — предоставляет доступ к информации о странице, отображаемой в данный момент. Ее уже может оценить только разработчик вроде нас с вами. Эта часть и называется инструментами разработчика.

Инструменты разработчика предоставляют следующие возможности:

- Просмотр DOM.
- Отладка JavaScript.
- Проверка объектов и просмотр сообщений через консоль.
- Определение производительности и обнаружение проблем с памятью.
- Просмотр сетевого трафика

...и многое другое!

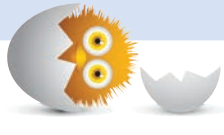
В целях экономии времени (скоро начнется очередная серия «Игры престолов», и я верю, что в этом эпизоде Нед Старк вернется в виде лютоволка) я сконцентрируюсь на первых трех пунктах, так как они непосредственно относятся к тому, что вы изучаете в этой книге.

Просмотр DOM

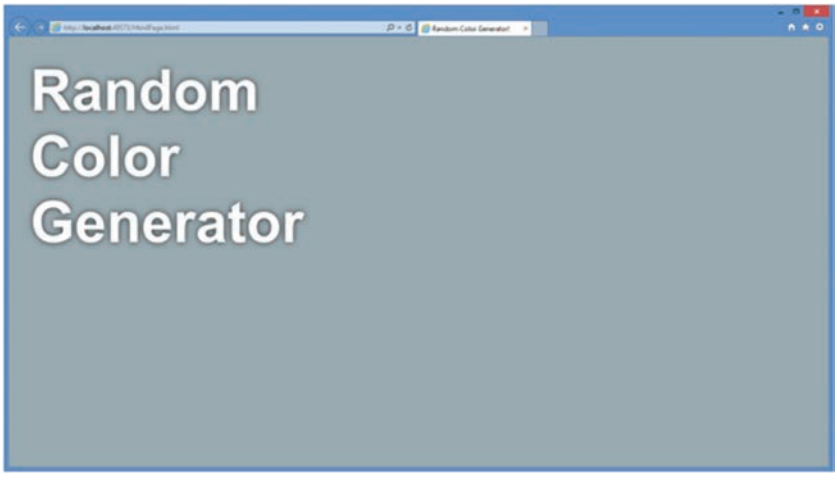
Первый инструмент разработчика, который мы рассмотрим, позволяет просматривать и даже манипулировать содержимым DOM. Запустив Chrome, пройдите по ссылке http://bit.ly/kirupaDevTool.

НЕТ БРАУЗЕРА? БЕЗ ПРОБЛЕМ!

Если у вас нет браузера под рукой или вы просто не можете получить доступ к этой ссылке, не волнуйтесь. Я буду объяснять, что происходит на каждом этапе пути, чтобы вы не остались в стороне.

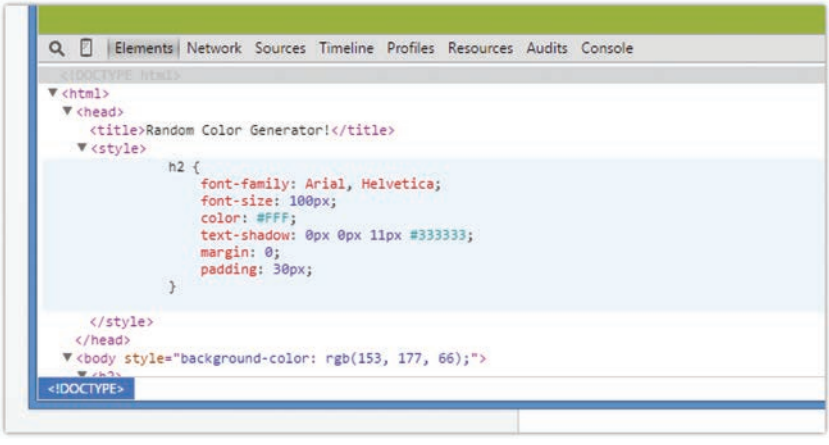


При загрузке этой страницы вы увидите цветной фон и отображенный на нем текст:



Если вы обновите эту страницу, то увидите уже другой цвет фона. Как можно догадаться, при каждой загрузке страницы генерируется случайный цвет фона.

Первое, что мы сделаем с этим примером, — это проверим DOM и посмотрим, что здесь происходит. Убедитесь, что инструменты разработчика видимы и при этом выбрана вкладка Elements:



Здесь вы увидите *текущее* представление разметки страницы. Говоря более конкретно, это *представление вашей DOM*. Важность этого отличия в том, что это представление демонстрирует текущую версию того, как выглядит страница. Любые особенности, которые могли привнести в DOM браузер или JavaScript, будут здесь отображены.

Результат команды View source (просмотр исходного кода) в нашем примере будет выглядеть примерно так:

```
<!DOCTYPE html>
<html>

<head>
  <title>Random Color Generator!</title>
  <style>
    h2 {
      font-family: Arial, Helvetica;
      font-size: 100px;
      color: #FFF;
      text-shadow: 0px 0px 11px #333333;
      margin: 0;
      padding: 30px;
    }
  </style>
</head>

<body>
  <h2>Random
    <br />Color
    <br />Generator</h2>
  <script src="js/randomColor.js"> </script>
  <script>
```

```
    let bodyElement = document.querySelector("body");
    bodyElement.style.backgroundColor = getRandomColor();
  </script>
</body>

</html>
```

Команда View source просто выводит представление разметки, хранящейся в HTML-странице. Иначе говоря, View source предоставляет вам (устаревшую) версию разметки, существующую на сервере, но не версию DOM.

Если вы используете инструмент разработчика для просмотра представления DOM, то увидите это представление на основе текущей версии страницы:

```
<!DOCTYPE html>
<html>

<head>
  <title>Random Color Generator!</title>
  <style>
    h2 {
      font-family: Arial, Helvetica;
      font-size: 100px;
      color: #FFF;
      text-shadow: 0px 0px 11px #333333;
      margin: 0;
      padding: 30px;
    }
  </style>
<body style="background-color: rgb(75, 63, 101);">
  <h2>Random
    <br>Color
    <br>Generator</h2>
  <script src="js/randomColor.js"> </script>
  <script>
    let bodyElement = document.querySelector("body");
    bodyElement.style.backgroundColor = getRandomColor();
  </script>

</body>

</html>
```

Если присмотреться, то можно заметить едва уловимые различия в представлении некоторых элементов. Самое серьезное различие — это выделенный стиль `background-color` в элементе `body`, который

есть только в представлении DOM, но не традиционном представлении исходного кода (View source). Причина в том, что есть JavaScript, который динамически устанавливает встроенный стиль в элементе `body`. Следующая врезка поясняет, почему так происходит.

ОТЛИЧИЕ ПРЕДСТАВЛЕНИЯ DOM ОТ VIEW SOURCE

Причина связана с тем, что представляет DOM. Еще раз повторю, что DOM — это результат завершения работы вашего браузера и JavaScript. Она показывает вам самое свежее представление, которое имитирует то, что видит браузер.

View Source — это просто статическое представление документа в том виде, в котором он находится на сервере (или на вашем компьютере). Оно не отображает текущих изменений выполняемой страницы, которые отражены в представлении DOM. Если вы взглянете на наш JavaScript код, то увидите, что я установил динамическое получение `backgroundColor` элементом `body`:

```
let bodyElement = document.querySelector("body");
bodyElement.style.backgroundColor = getRandomColor();
```

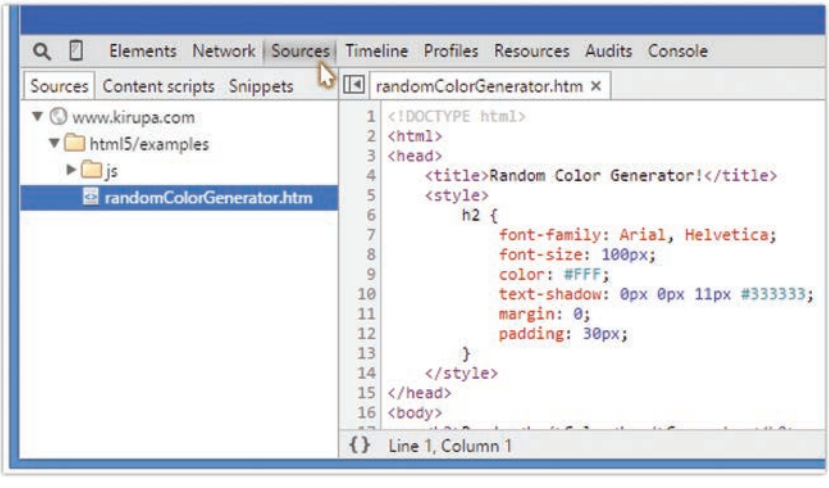
Когда этот код запускается, он изменяет DOM, чтобы установить свойство `backgroundColor` в элементе `body`. Используя View Source, вы бы никогда не увидели этого. Вообще никогда. Вот почему представление DOM из инструментов разработчика — ваш самый лучший друг.

Пример, выделяющий различия между исходником и DOM, был весьма прост. Чтобы увидеть реальную пользу представления DOM, стоит поэкспериментировать с переподчинением элементов, а также их созданием и удалением. Таким образом, вы отлично поймете различия между просмотром исходника и инспектированием DOM. Некоторые из примеров в прошлых главах, касающиеся управления DOM, могут быть полезны для аналогичных проверок.

Отладка JavaScript

Следующим важным инструментом разработчика является *отладка*. Не уверен, что использовал подходящее выражение, но инструменты разработчика позволяют проверить код вдоль и поперек, чтобы выяснить, есть ли в нем неполадки и в чем они. Этот процесс по-другому называется *отладкой*.

В панели инструментов разработчика перейдите на вкладку Sources:

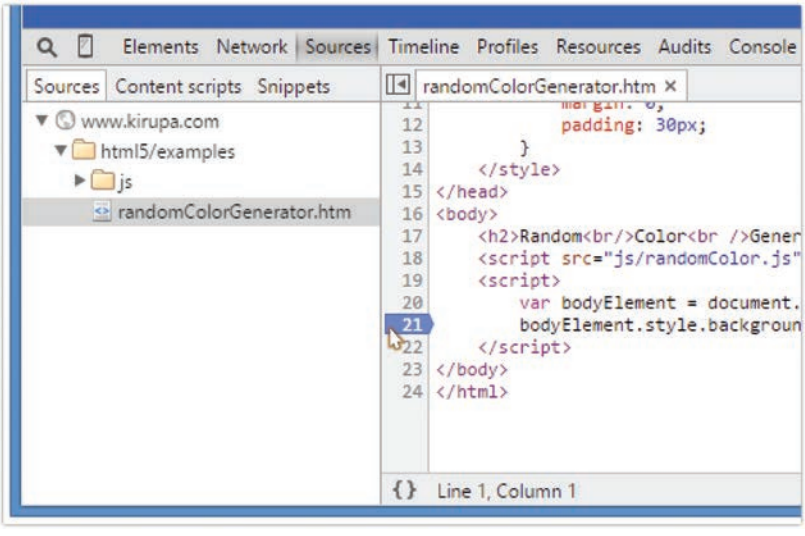


Sources дает доступ ко всем файлам вашего документа. Вы видите сырое содержимое этих файлов, а не сгенерированную версию DOM, которая, как вы помните, ваш лучший друг.

Убедитесь, что в представлении дерева слева выбран файл randomColorGenerator.htm. Это гарантия того, что содержимое этого файла будет отображено справа и вы сможете его просмотреть. Промотайте вниз до тега `script`, содержащего две строки кода, которые вы видели ранее. Судя по нумерации строк, указанной в столбце слева, наши строки идут под номерами 20 и 21.

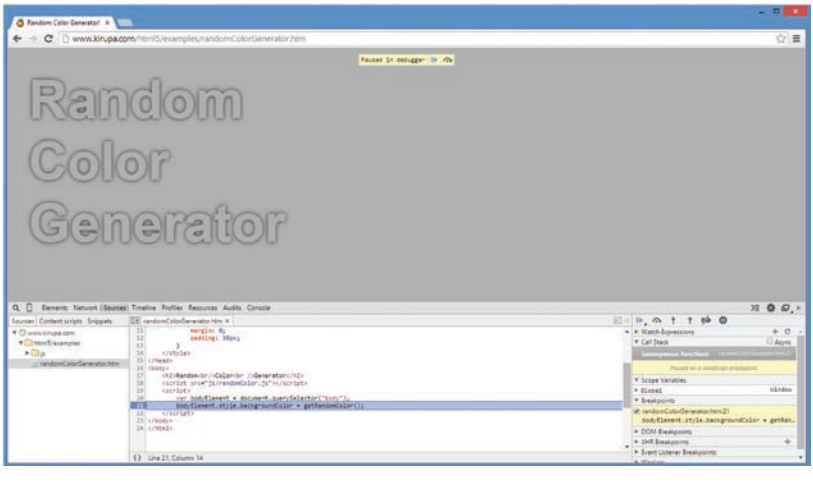
Нужно проверить, что происходит, когда код в строке 21 начинает выполнение. Чтобы это сделать, нужно сообщить браузеру о необходимости остановиться перед началом выполнения кода этой строки. Делается это с помощью установки так называемой *точки останова* (*прерывания*). Щелкните на ярлык строки 21 в столбце слева, чтобы установить точку останова.

Так вы выделите строку 21:



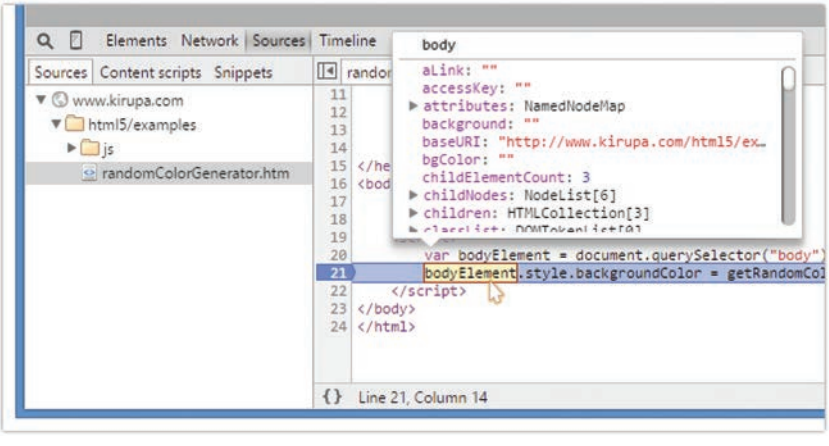
Теперь точка останова установлена. В следующем шаге браузер должен наткнуться на эту точку или, говоря более мягко, достиг точки прерывания. В нашем случае достаточно нажать F5, чтобы обновить страницу.

Если все сработало ожидаемым образом, то вы увидите загрузку страницы, которая внезапно остановится, выделив строку 21:



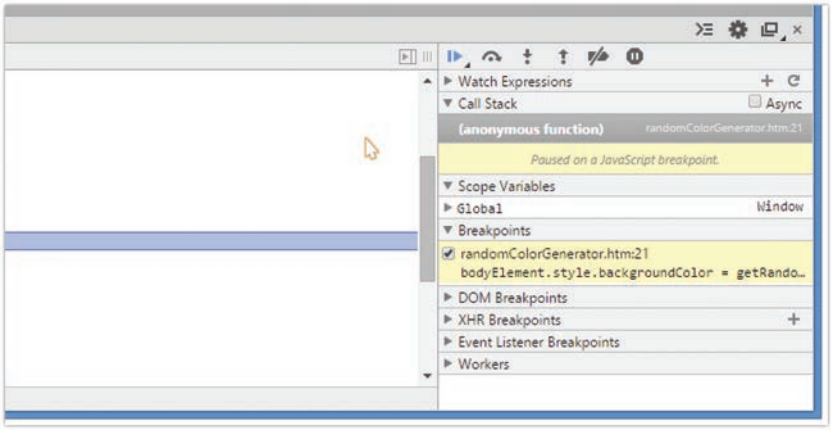
Сейчас вы находитесь в *режиме отладки*. Точка останова, которую вы установили на строке 21, была достигнута, и вся страница с визгом остановила свою загрузку. Браузер попадает в режим приостановки анимации, и у вас есть возможность разобраться со всем происходящим на странице. Это как если бы остановилось время и только вы имели возможность двигаться, изучать и изменять окружение. Если об этом еще не сняли кино, то кто-нибудь обязательно должен этим заняться.

Находясь в этом режиме, вернитесь к строке 21 и наведите курсор на переменную `bodyElement`. При наведении вы увидите подсказку, содержащую различные свойства и значения, существующие в этом конкретном объекте:



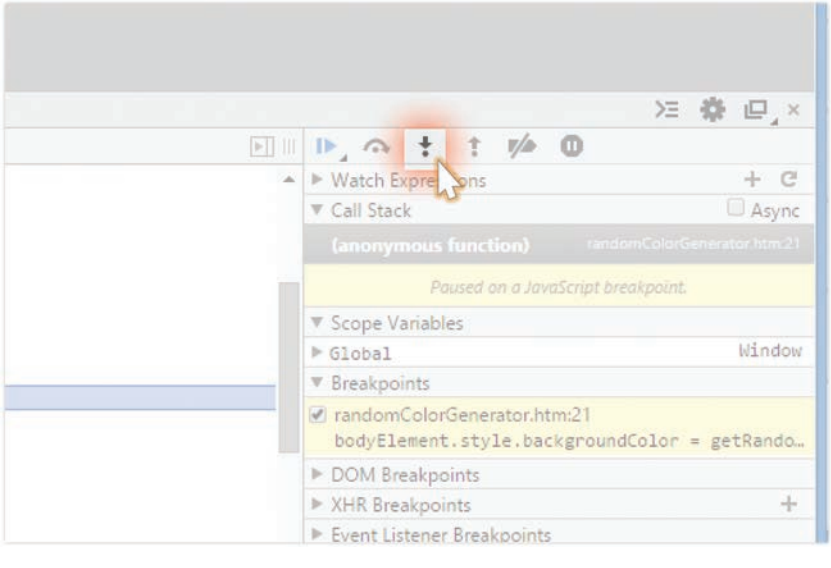
Можно взаимодействовать с этой справкой, делать прокрутку по объектам и даже углубиться в сложные объекты, содержащие в себе другие объекты. Так как `bodyElement` по сути является JavaScript/DOM-представлением элемента `body`, то вы увидите множество свойств, с которыми косвенно столкнулись при рассмотрении `HTMLElement` несколько глав назад.

Справа от представления исходного кода видно, с каких еще ракурсов его можно изучить:



Не стану объяснять возможности всех этих категорий, но просто скажу, что с их помощью можно изучать текущее состояние всех переменных и объектов JavaScript.

Помимо этого, точка останова предоставляет очень полезную возможность проследовать по коду так же, как это делает браузер. Мы остановились на строке 21. Чтобы проследовать далее, щелкните по кнопке Step into function call справа:

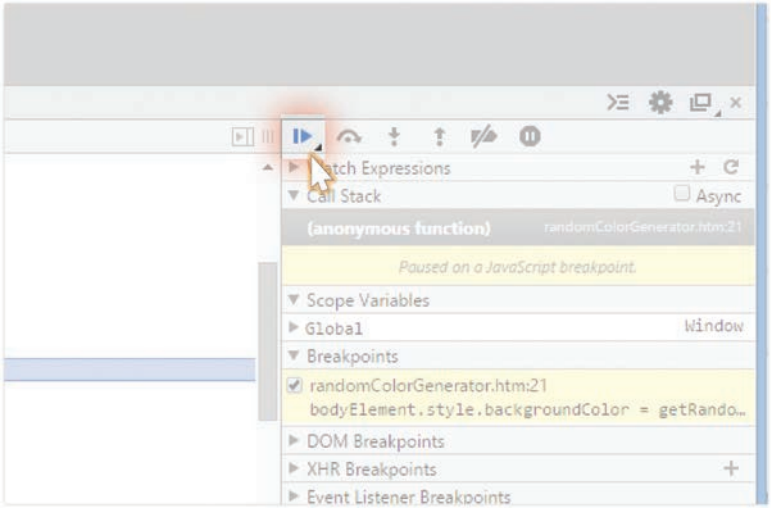


Помните, что вы сейчас находитесь на этой строке:

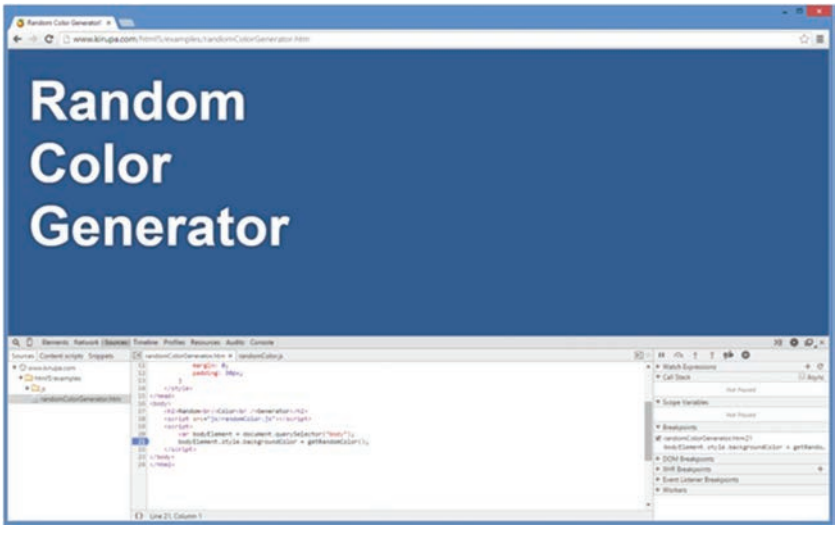
```
bodyElement.style.backgroundColor = getRandomColor();
```

Щелкните по указанной кнопке и посмотрите, что произойдет. Вы окажетесь внутри randomColor.js, где была определена функция `getRandomColor`. Нажмайте на Step into function call, чтобы двигаться по коду и проходить каждую строку функции `getRandomColor`. Обратите внимание, что теперь вы видите, как объекты в памяти браузера обновляются по мере выполнения кода при вашем продвижении от строки к строке. Если вы устали от этого, сделайте шаг назад, щелкнув на кнопку Step out of current function (она расположена справа от предыдущей кнопки), которая выведет вас из этой функции. В нашем случае выход произойдет обратно на строку 21 в randomColorGenerator.htm.

Если вы просто хотите запустить приложение, не продолжая прогулку по коду, то щелкните на кнопке Play, расположенной несколькими пикселями левее кнопки Step into:



Когда вы нажмете Play, код продолжит выполнение. Если у вас где-то еще определена точка останова, то она также сработает. При остановке на каждой такой точке можно совершать шаг внутрь, выход или просто возвращаться к выполнению, нажав Play. Так как мы установили только одну точку останова, то нажатие Play приведет к выполнению кода до завершения и появлению случайного цвета фона для элемента `body`:



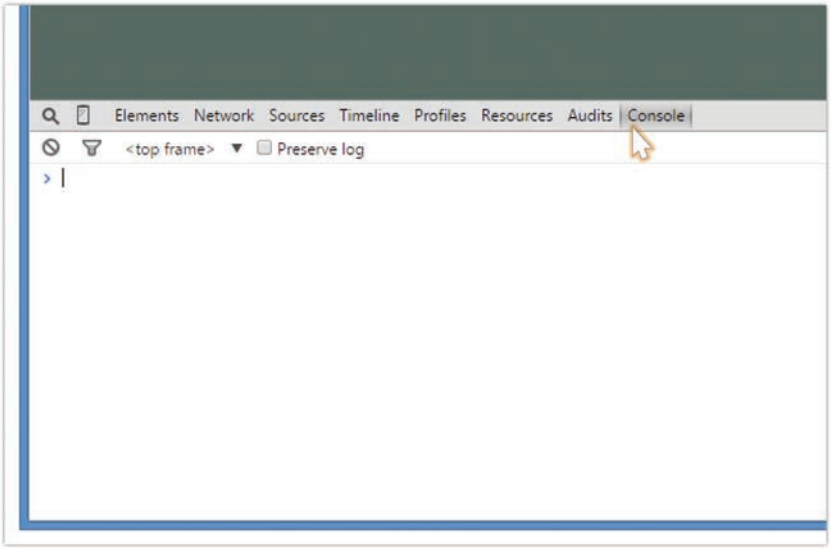
Чтобы удалить точку останова, еще раз щелкните на номер строки, для которой ее определяли. В данном случае если вы нажмете на ярлык строки 21, то точка останова будет отключена и можно выполнять приложение, не попадая в режим отладки.

На этом заканчивается наша быстрая экскурсия, посвященная использованию функции отладки. Еще раз повторюсь: я показал все достаточно поверхностно. Ресурсы в конце главы помогут продвинуться дальше.

Знакомство с консолью

Еще один важный инструмент отладки — это *консоль*. Она позволяет совершать несколько действий — видеть сообщения, журналируемые кодом, а также передавать команды и проверять любой объект, находящийся в текущей области видимости.

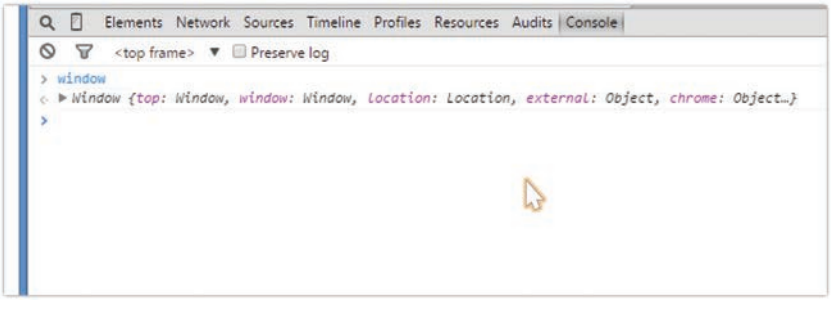
Чтобы вызвать ее, просто перейдите на вкладку Console:



Не пугайтесь пустоты, которую видите, лучше просто насладитесь этой свободой и свежим воздухом.

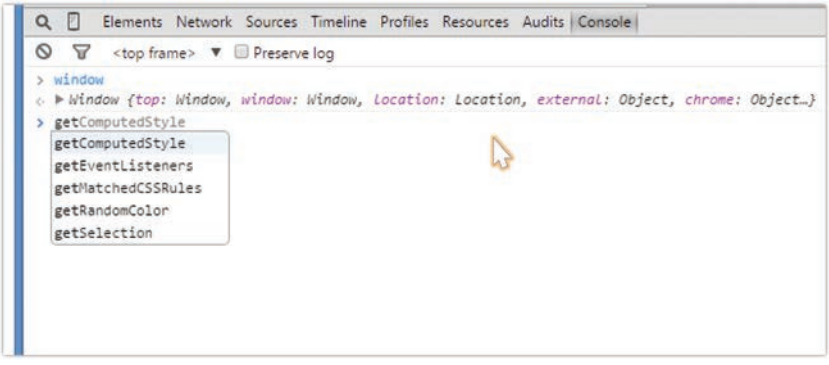
Инспектирование объектов

Там, где находится ваш курсор, напечатайте `window` и нажмите Enter:



Вы увидите интерактивный список всех элементов, находящихся в объекте `window`. Можно ввести любое действующее имя объекта или

свойства, и если он окажется в области видимости, то к нему можно обратиться, изучить его значение или даже выполнить его:



Это ни в коем случае не песочница «только для чтения». Здесь легко можно учинить всяческие беспорядки. Например, если набрать `document.body.remove()` и нажать Ввод, то весь ваш документ просто исчезнет. Если вы случайно удалили `body`, тогда просто обновите страницу, чтобы вернуться к предыдущему состоянию. Инструменты разработчика работают напрямую с представлением страницы в оперативной памяти и не производят запись в исходный код. Поэтому ваши эксперименты без ущерба для всех останутся в промежуточной реальности.

Консоль позволяет проверять или вызывать любой объект, существующий в любой области, где в данный момент выполняется приложение. Если не установлено точек останова, тогда запуск консоли переводит вас в глобальное состояние.

Журналирование сообщений

Мы уже почти закончили с темой инструментов разработчика. В финале рассмотрим возможность консоли журналировать сообщения из кода. Помните, как мы уже делали нечто подобное?

```
function doesThisWork() {
  console.log("It works!!!");
}
```

Это «нечто» мы делали, когда использовали инструкцию `alert` для вывода значения или подтверждения выполнения кода. Что ж, теперь можем перестать это делать. Консоль предоставляет нам гораздо менее

раздражающий способ вывода сообщений, без прерывания всего процесса всплывающими диалоговыми окнами. Вы можете использовать функцию `console.log` для вывода нужной информации в консоль:

```
function doesThisWork() {
  console.log("It works!!!")
}
```

ЕЩЕ РАЗ ОБ ОБЛАСТИ ВИДИМОСТИ И СОСТОЯНИИ

Я уже несколько раз упоминал о том, что консоль позволяет проверять текущую область видимости. По сути, все то, что мы обсуждали в главе 8 «Область видимости переменных», также относится и к поведению консоли.

Предположим, вы установили точку останова на следующей выделенной строке:

```
let oddNumber = false;

function calculateOdd(num) {
  if (num % 2 == 0) {
    oddNumber = false;
  } else {
    oddNumber = true;
  }
}
calculateOdd(3);
```

Когда код при выполнении достигает этой точки, значение `oddNumber` по-прежнему остается `false`. Строка, на которой произошло прерывание, еще не была выполнена, и вы можете проверить это, протестировав значение `oddNumber` в консоли. А теперь предположим, что вы выполняете код, достигаете точки останова и совершаете шаг к следующей строке.

В этот момент значение `oddNumber` уже установлено как `true`. Теперь консоль будет отражать уже новое значение, так как именно оно определено для переменной `oddNumber` согласно представлению в оперативной памяти. Главный вывод здесь в том, что представление в консоли напрямую связано с той частью кода, на которой вы в данный момент фокусируетесь. Это становится особенно заметно, когда вы шагаете по коду и области видимости сменяются часто.

При выполнении этого кода вы увидите указанные вами данные в консоли:

Далее вы увидите, что в некоторых случаях я начну использовать `console.log` вместо `alert`.

КОРОТКО О ГЛАВНОМ

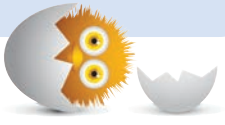
Если вы еще не использовали инструменты разработчика, то настоятельно рекомендую выделить время на ознакомление с ними. JavaScript — это один из тех языков, где проблемы могут возникнуть даже тогда, когда все вроде бы в порядке. В очень простых примерах, которые вы встретите в книге, обнаружить ошибки легко. Но когда вы сами начнете работать над крупными и более сложными приложениями, наличие правильных инструментов для определения неполадок сэкономит вам много времени и усилий.

Чтобы узнать больше об инструментах разработчика (или **Dev Tools**, как их называют крутые ребята), обратитесь к следующим ресурсам:

- Обзор Chrome Dev Tools: **http://bit.ly/kirupaChromeDevTools**
- Обзор IE/Edge F12 Dev Tools: **http://bit.ly/kirupaIEDevTools**
- Обзор Firefox Dev Tools: **http://bit.ly/kirupaFFDevTools**
- Обзор Safari Web Inspector: **http://bit.ly/kirupaSafariDevTools**

В ЭТОЙ ГЛАВЕ:

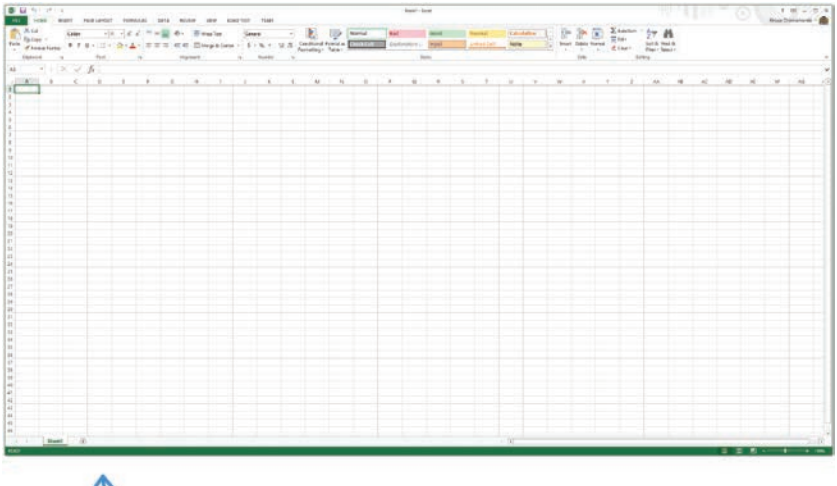
- поймем, как осуществляется связь между вами и вашим приложением;
- познакомимся с событиями;
- используем аргументы событий для лучшей обработки связанных с событиями сценариев.



31

СОБЫТИЯ

Многие приложения и веб-сайты, оставленные без внимания, становятся очень скучными. Запускаются они с шиком и помпой, но вся эта напыщенность очень быстро пропадает, если мы перестаем взаимодействовать с ними:



Боже! Просто сделай что-нибудь... что угодно!

Причина проста. Приложения существуют, чтобы реагировать на наши в них действия. Хоть они и имеют определенную базовую мотивацию, проявляющуюся при запуске, которая вытаскивает их из постели и приводит в рабочее состояние, все дальнейшие их действия уже по большей части диктуются нами. Тут все и становится интересно.

Чтобы сообщить приложениям, что нужно делать, мы используем так называемые *события*, на которые они в итоге реагируют. В этой главе узнаем, чем являются эти события и как их можно использовать.

Поехали!

Что такое события?

На высоком уровне все создаваемое нами может быть смоделировано следующей инструкцией:

Когда__ совершится, то___

Мы можем заполнить пробелы этой инструкции великим множеством способов. Первый пробел описывает то, что происходит, а второй уже описывает реакцию на это. Вот некоторые примеры с заполненными пробелами:

Когда страница будет загружена, начнется воспроизведение видео с кошкой, скользящей на картоне.

Когда произойдет щелчок, покупка будет отправлена.

Когда ЛКМ будет освобождена, вылетит гигантская / не очень довольная птичка.

Когда клавиша удаления будет нажата, файл отправится в корзину.

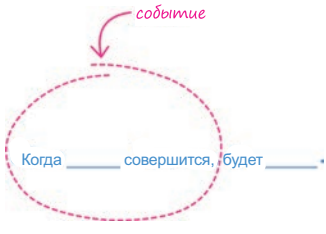
Когда произойдет касание, к фотографии будет применен старый фильтр.

Когда файл будет загружен, обновится индикатор выполнения.

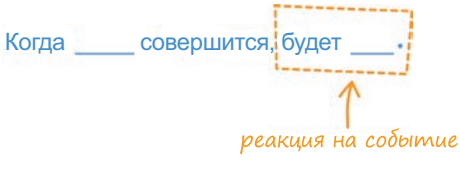
Эта обобщенная модель применима ко всему коду, который мы с вами написали. Она также применима ко всему коду, который написали для своих приложений наши друзья-разработчики/дизайнеры. От этой модели никуда не деться, поэтому и смысла противиться ей нет. Наоборот, следует хорошенько познакомиться с важным и очень талантливым представителем этой модели — *событием*.

Событие — это не более чем сигнал. Оно сообщает о том, что что-то только что произошло. Это что-то может быть щелчком мыши, нажатием клавиши клавиатуры, изменением размера окна или загрузкой документа. Главное здесь — понять, что этим сигналом могут послужить сотни встроенных в JavaScript явлений, равно как бессчетное множество отдельных пользовательских решений, созданных под конкретное приложение.

Вернемся к нашей модели, где события составляют первую ее половину:



Событие определяет то, что происходит, и *передает сигнал*. Вторая часть модели определяет реакцию на это событие:



При этом какая польза от сигнала, если кто-то где-то не ожидает его, чтобы предпринять соответствующее действие? Хорошо, теперь, когда у нас есть высокоуровневое представление о том, что такое событие, давайте поглядим на их жизнь в природном заповеднике JavaScript.

События и JavaScript

Учитывая важность событий, вас не должно удивлять, что JavaScript дает богатые возможности для работы с ними. Всего для их использования нам понадобится делать две вещи:

1. Прослушивать события.
2. Реагировать на события.

Оба этих шага кажутся простыми, но не стоит забывать, что мы имеем дело с JavaScript. Простота в этом случае является всего лишь завесой, скрывающей печальные последствия, которые JS обрушит на нас, сделай мы один неверный шаг. Но возможно, я драматизирую.

1. Прослушивание событий

Практически все, что мы делаем внутри приложения, заканчивается срабатыванием событий. Иногда наше приложение будет запускать события автоматически, например при своей загрузке. В других случаях оно будет запускать их в виде реакции на наши действия. Здесь стоит отметить, что наше приложение постоянно бомбардируется событиями независимо от того, хотим мы, чтобы они срабатывали, или нет. Наша же задача — указать приложению прослушивать только те события, которые нас интересуют.

Неблагодарная работа по прослушиванию нужных событий полностью обрабатывается функцией, называемой `addEventListener`. Эта функция всегда начеку, поэтому может в нужное время уведомить другую часть приложения о том, что произошло некоторое событие.

Используется она следующим образом:

```
source.addEventListener(eventName, eventHandler, false);
```

Наверняка так не совсем понятно, поэтому давайте разберем, что именно означает каждая часть этой функции.

Source (источник)

Мы вызываем `addEventListener` через элемент или объект, в котором мы хотим прослушивать события. Как правило, это будет элемент DOM, но им также может быть `document`, `window` или другой объект, специально созданный для запуска событий.

Event Name (имя события)

В виде первого аргумента для функции `addEventListener` мы указываем имя события, которое хотим прослушивать. Весь список событий, доступных нам, слишком велик, чтобы полностью его здесь привести, но некоторые из наиболее распространенных перечислены в табл. 31.1.

ТАБЛ. 31.1. Распространенные события

Событие	Событие срабатывает...
`Click`	...когда вы нажимаете и отпускаете основную кнопку мыши, сенсорной панели и т. п.
`Mousemove`	...при движении курсора мыши
`Mouseover`	...при наведении курсора мыши на элемент. Это событие используется для обнаружения наведения
`Mouseout`	...при перемещении курсора мыши за границы элемента
`Dblclick`	...при быстром двойном клике
`DOMContentLoaded`	...когда DOM документа полностью загрузилась. Подробнее об этом событии в следующей главе
`Load`	...когда загрузился весь документ (DOM, внешние изображения, сценарии и т. п.)
`Keydown`	...при нажатии клавиши клавиатуры
`Keyup`	...при прекращении нажатия клавиши клавиатуры
`Scroll`	...при прокрутке элемента
`wheel and DOMMouseScroll`	...при использовании колесика мыши для прокрутки

В последующих главах мы рассмотрим многие из этих событий более подробно. На данный момент просто взгляните на событие `click`. Именно его мы будет использовать в скором времени.

Обработчик событий

Во втором аргументе нужно указать функцию, которая будет вызвана, когда событие будет услышано. Эта функция (а иногда объект) в кругу семьи и друзей зовется *обработчик событий*. Через несколько мгновений мы узнаем о ней гораздо больше.

Захватывать или не захватывать, вот в чем вопрос!

Последний аргумент состоит из `true` либо `false`. Чтобы полноценно понять последствия определения того или иного значения, нам придется подождать до главы «Всплытие и погружение событий», которая как раз будет следующей.

Обобщая все сказанное

Теперь, когда мы разобрали функцию `addEventListener`, давайте все это свяжем в полноценном заполненном примере:

```
document.addEventListener("click", changeColor, false);
```

В этом примере `addEventListener` прикреплена к объекту `document`. Когда будет услышано событие `click`, она вызовет функцию `changeColor` (то есть обработчик событий), которая отреагирует на это событие. Теперь самое время перейти к следующему разделу, посвященному как раз реагированию на события.

2. Реагирование на события

Как мы видели в предыдущем разделе, за прослушивание событий отвечает `addEventListener`. За действия, производимые после того, как событие будет услышано, отвечает обработчик событий. И я не шутил про то, что обработчик событий — это не более чем функция или объект:

```
function normalAndBoring() {
  // я люблю походы, щенков и многие другие вещи!
}
```

Единственным отличием между типичной функцией и установленной мной в качестве обработчика событий в том, что функция обработчика событий специально вызывается по имени в вызове `addEventListener` (и получает объект `Event` в виде аргумента):

```
document.addEventListener("click", changeColor, false);

function changeColor(event) {
  // I am important!!!
}
```

Любой код, который мы поместим в обработчик событий, будет выполнен, когда событие, за которым следит наша функция `addEventListener`, будет услышано.

Простой пример

Лучшим способом разобраться в этом материале будет рассмотрение его в действии. Чтобы попутно проработать пример самим, добавьте в HTML-документ следующую разметку и код:

```
<!DOCTYPE html>
<html>

<head>
  <title>Click Anywhere!</title>
</head>

<body>
  <script>
    document.addEventListener("click", changeColor, false);

    function changeColor() {
      document.body.style.backgroundColor = "#FFC926";
    }
  </script>
</body>

</html>
```

Если мы просмотрим наш документ в браузере, то изначально увидим пустую страницу (рис. 31.1).

Но все изменится, если щелкнуть в любом месте этой страницы. По завершении щелчка (то есть как только вы отпустите кнопку мыши) цвет фона страницы изменится на желтый (рис. 31.2).

Причина в коде:

```
document.addEventListener("click", changeColor, false);

function changeColor() {
  document.body.style.backgroundColor = "#FFC926";
}
```

Вызов `addEventListener` идентичен тому, что мы уже видели, поэтому его можно пропустить. Вместо этого давайте обратим внимание на обработчика событий `changeColor`.

```
document.addEventListener("click", changeColor, false);

function changeColor() {
  document.body.style.backgroundColor = "#FFC926";
}
```

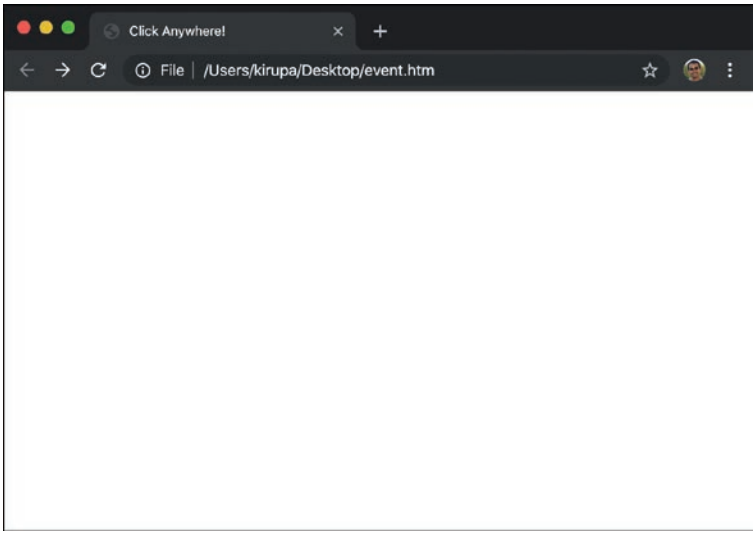


РИС. 31.1.
Просто пустая страница

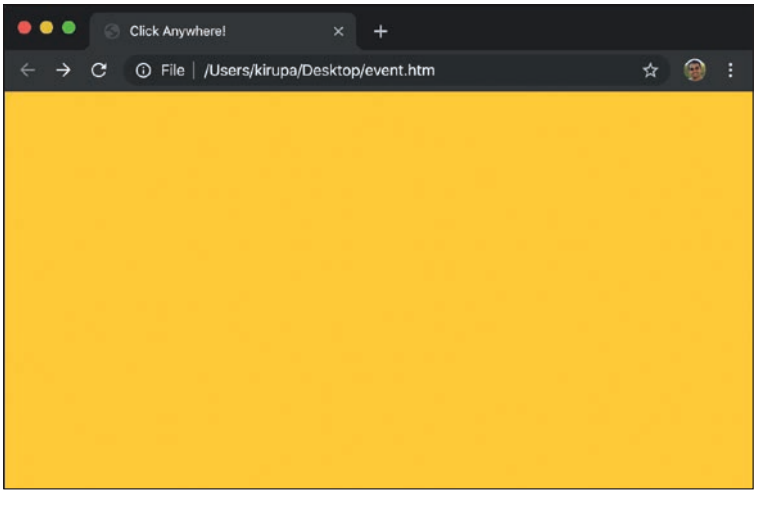
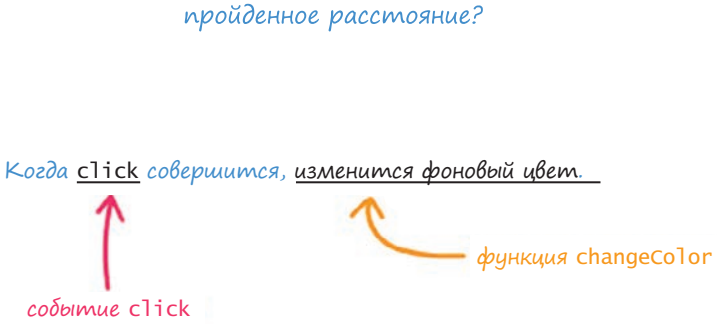


РИС. 31.2.
Страница становится желтой после щелчка

Эта функция вызывается, как только событие `click` в `document` будет услышано. Когда происходит вызов этой функции, она устанавливает фоновый цвет как желтый. Если связать это с самым началом, где мы в общих чертах обсуждали принцип работы приложений, то вот как будет выглядеть наш пример:



Надеюсь, что теперь все стало понятно. Вы только что изучили один из важнейших принципов работы, с которым предстоит сталкиваться постоянно. Но это еще не все.

Аргументы и типы событий

Наш обработчик событий помимо вызова при срабатывании события совершает и другие действия — обеспечивает доступ к основному объекту события в виде своего аргумента. Чтобы с легкостью обращаться к объекту события, нужно изменить сигнатуру обработчика событий этого аргумента.

Вот пример, в котором мы указываем имя `event`, чтобы сослаться на наш аргумент события:

```
function myEventHandler(event) {
  // материал для обработки событий для поддержки
}
```

Обработчик событий — это по-прежнему простая скучная функция, которая просто получает один аргумент — аргумент события. Можно использовать любой доступный идентификатор для этого аргумента, но я склоняюсь к использованию `event` или просто `e`, потому что именно

так делают все крутые ребята. Тем не менее технически будет верен и следующий идентификатор:

```
function myEventHandler(isNyanCatReal) {
  // материал для обработки событий
}
```

Нюанс в том, что аргумент события указывает на объект события и этот объект передается как часть срабатывания события. Есть причина, по которой мы уделяем внимание такому, казалось бы, типичному и скучному явлению. Этот объект события содержит свойства, *которые связаны со сработавшим событием*. Событие, запущенное кликом мыши, будет иметь отличные свойства в сравнении с событием, запущенным нажатием клавиши клавиатуры, загрузкой страницы, анимацией и многим другим. Большинство событий будут иметь свое собственное уникальное поведение, на которое мы будем опираться, а объект события — это наше окно в эту их уникальность.

Несмотря на разнообразие событий и итоговых объектов событий, которые мы можем получить, у них есть некоторые общие свойства. Эта общая часть определяется тем, что все объекты событий происходят от основного типа `Event` (технически подразумевается `Interface`). Ниже перечислены некоторые из известных свойств типа `Event`, которые мы будем использовать:

1. `currentTarget`
2. `target`
3. `preventDefault`
4. `stopPropagation`
5. `type`

Чтобы как следует понять назначение этих свойств, нужно углубиться в понимание самих событий. Мы пока еще не достигли нужной глубины, поэтому просто знайте, что они существуют. В ближайших главах мы с ними встретимся.

УДАЛЕНИЕ СЛУШАТЕЛЯ СОБЫТИЙ

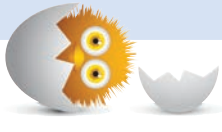
Иногда понадобится удалять слушателя событий из элемента. Делается это с помощью злейшего врага `addEventListener` функции `removeEventListener`:

```
something.removeEventListener(eventName, eventHandler, false);
```

Как видно из примера, эта функция получает в точности такие же типы аргументов, что и функция `addEventListener`. Причина тому проста. Когда мы прослушиваем событие в элементе или объекте, JavaScript использует `eventName`, `eventHandler` и значение `true` / `false`, чтобы опознать слушателя событий. Чтобы удалить этого слушателя событий, нам нужно указать в точности такие же аргументы. Вот пример:

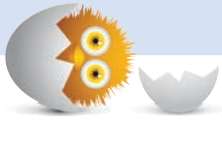
```
document.addEventListener("click", changeColor, false);
document.removeEventListener("click", changeColor, false);

function changeColor() {
  document.body.style.backgroundColor = "#FFC926";
}
```

Слушатель событий, которого мы добавили в первой строке, полностью нейтрализован вызовом `removeEventListener` в выделенной второй строке. Если вызов `removeEventListener` использует аргументы, отличные от указанных в соответствующем вызове `addEventListener`, то его усилия будут проигнорированы и прослушивание продолжится.



КОРОТКО О ГЛАВНОМ

Что ж, это все, что нужно для ознакомления с событиями. Помните: у вас есть функция `addEventListener`, которая позволяет зарегистрировать функцию обработчика событий. Эта функция обработчика событий будет вызываться при запуске события, которое примет прослушиватель событий. Хотя мы коснулись некоторых других тем, они станут более понятными, если рассмотреть их в контексте более сложных примеров, связанных с событиями.



В ЭТОЙ ГЛАВЕ:

- узнаем, как события перемещаются по DOM;
- поймем разницу между погружением событий и их всплытием;
- научимся прерывать события.

32

ВСПЛЫТИЕ И ПОГРУЖЕНИЕ СОБЫТИЙ

В предыдущей главе вы узнали, как использовать функцию `addEventListener` для прослушивания событий, на которые нужно среагировать. Но кроме основ мы также затронули важную тему того, как именно события срабатывают. Событие — это не обособленное возмущение. Подобно бабочке, машущей крыльями, землетрясению, падению метеорита или визиту Годзиллы, многие события расходятся волнами и воздействуют на другие элементы, которые лежат у них на пути.

Сейчас я превращусь в Шерлока Холмса и раскрою вам тайну, что же именно происходит при срабатывании события. Вы узнаете о двух фазах, в которых пребывают события, поймете, почему это важно знать, а также познакомитесь с некоторыми трюками, которые помогут лучше контролировать события.

Событие опускается. Событие поднимается

Для наглядности оформим все в виде простого примера:

```
<!DOCTYPE html>
<html>

<head>
  <title>Events!</title>
</head>

<body id="theBody" class="item">
  <div id="one_a" class="item">
    <div id="two" class="item">
      <div id="three_a" class="item">
        <button id="buttonOne" class="item">one</button>
      </div>
      <div id="three_b" class="item">
        <button id="buttonTwo" class="item">two</button>
        <button id="buttonThree" class="item">three</button>
      </div>
    </div>
  </div>
  <div id="one_b" class="item">

  </div>
  <script>

  </script>
</body>

</html>
```

Вроде бы ничего особенного здесь не происходит. HTML должен выглядеть достаточно понятно, и его представление DOM приведено на рис. 32.1.

Отсюда и начнем наше расследование. Давайте представим, что щелкаем по элементу `buttonOne`. Исходя из пройденного материала, вы знаете, что при этом запустится событие клика. Интересная же часть, которую до этого я опускал, заключается в том, откуда конкретно будет запущено это событие. Оно (как почти каждое событие в JavaScript) не возникает в элементе, с которым произошло взаимодействие. Иначе все бы было слишком просто и логично.

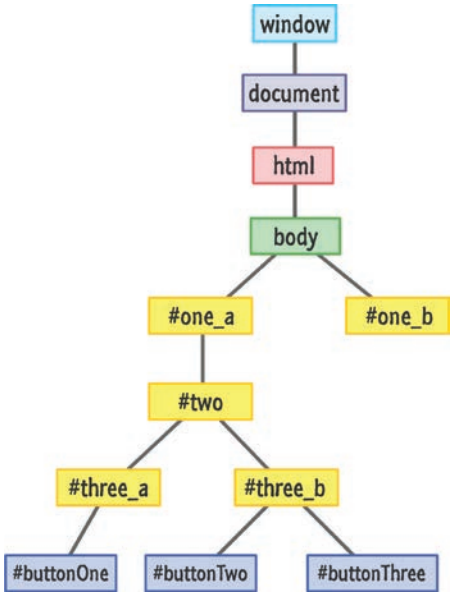
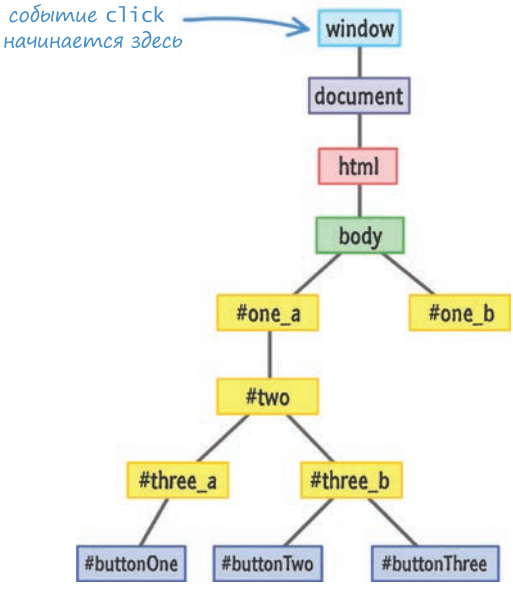
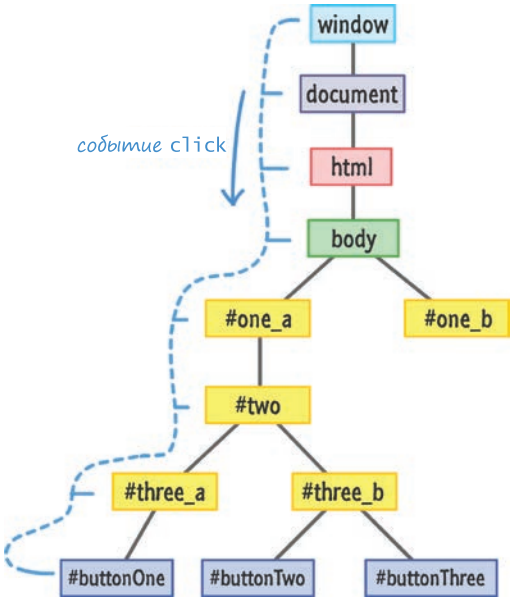


РИС. 32.1.
Так выглядит DOM для разметки, приведенной выше

Вместо этого событие стартует из корня вашего документа:

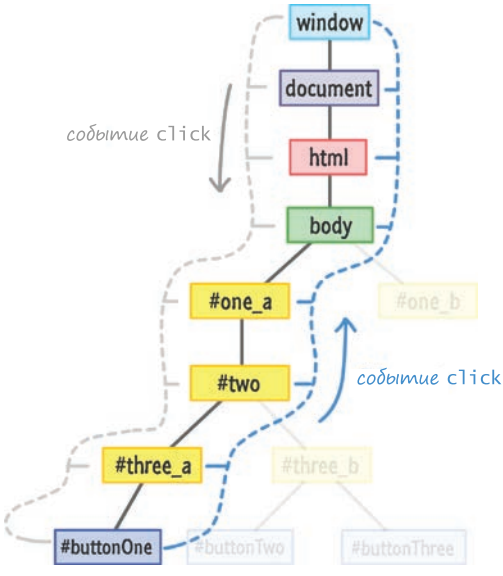


Начиная с корня, событие проделывает путь по узким тропкам DOM и останавливается у элемента, который его вызвал, а именно `buttonOne` (также известного как *целевое событие*):



Как показано на рисунке, событие совершает прямой путь, но при этом наглым образом уведомляет каждый элемент на своем пути. Это означает, что если бы вы прослушивали событие клика в `body`, `one_a`, `two` или `three_a`, то сработал бы связанный с ними обработчик событий. Это важная деталь, к которой мы еще вернемся.

Как только наше событие достигнет своей цели, оно не остановится и, как кролик из известной рекламы батареек, продолжает движение по своим следам обратно к корню:

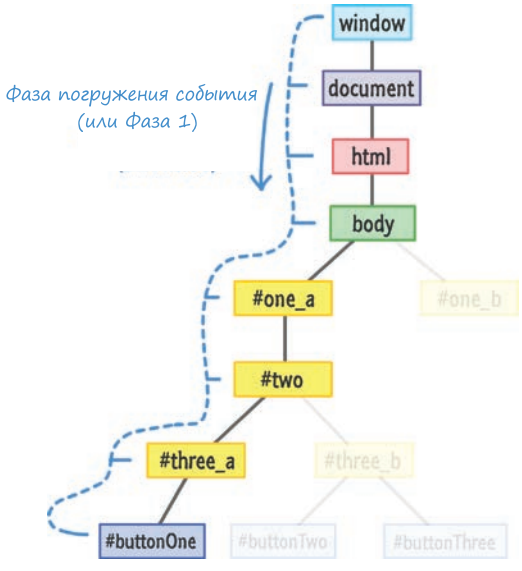


Как и прежде, каждый элемент на пути события будет уведомлен о его присутствии.

Знакомьтесь с фазами

Важно заметить, что не имеет значения, где в DOM инициируется событие. Оно всегда начинает движение от корня, спускается вниз до встречи с целью, а затем возвращается к корню. Этот путь имеет официальное определение, поэтому давайте рассмотрим его с этой позиции.

Часть, в которой вы инициируете событие и оно, начиная с корня, совершает спуск вниз по DOM, называется *фазой погружения события*.



Менее продвинутые люди иногда называют его *фаза 1*, так что имейте в виду, что в реальной жизни вы будет встречать верное название вперемешку с названием фазы. Следующей будет *фаза 2*, во время которой событие всплывает обратно к корню.

Эта фаза также известна как *фаза всплытия события*.

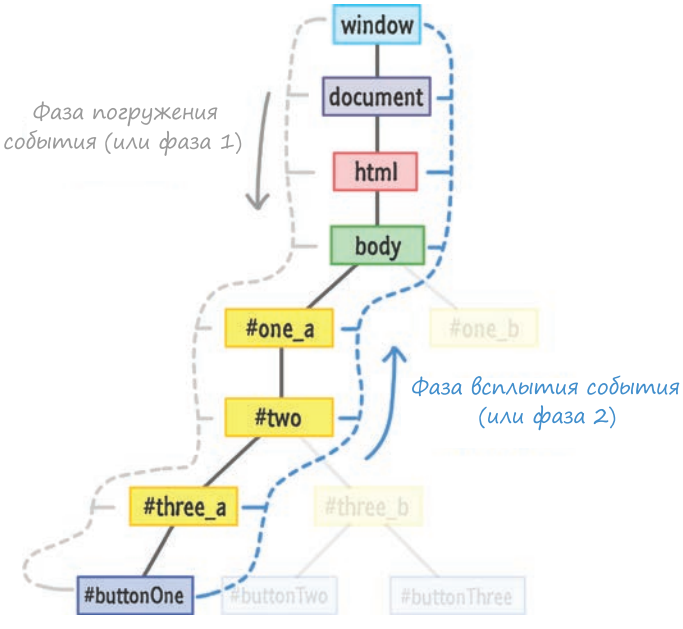
Как бы то ни было, но всем элементам на пути события в некотором смысле повезло. Судьба наградила их возможностью двойного уведомления при срабатывании события. Это может повлиять на код, который вы пишете, так как каждый раз, когда мы прослушиваем события, решаем, в какой фазе нужно это делать. Слушаем ли мы событие во время его спуска в фазе погружения или же тогда, когда оно взбирается обратно в фазе всплытия?

Выбор фазы — это тонкая деталь, которую вы определяете с помощью `true` или `false` в вызове `addEventListener`:

```
item.addEventListener("click", doSomething, true);
```

Если вы помните, в предыдущей главе я вскользь упомянул третий аргумент — `addEventListener`. Этот аргумент указывает, хотите ли вы прослушивать событие во время фазы погружения. В этом смысле значение `true` означает, что именно так вы и хотите. И наоборот,

аргумент `false` будет означать, что нужно прослушивать событие во время фазы всплытия.



Чтобы прослушивать его в обеих фазах, можно сделать следующее:

```
item.addEventListener("click", doSomething, true);
item.addEventListener("click", doSomething, false);
```

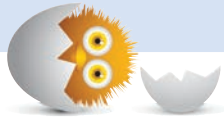
Я не могу представить, зачем вам это может понадобиться, но если вдруг понадобится, вы знаете, что делать.

НЕ УКАЗАНА ФАЗА

Можно возмутиться и вообще не указывать этот третий аргумент для фазы:

```
item.addEventListener("click", doSomething);
```

Если вы не укажете третий аргумент, то поведением по умолчанию будет прослушивание вашего события во время фазы восходящей цепочки. Это эквивалентно передаче в качестве аргумента ложного значения.



Кому это важно?

Вы можете спросить: «А почему это все важно?» Такой вопрос вдвойне справедлив, если вы и так давно работали с событиями и только сейчас обо всем этом прочитали. Выбор в пользу прослушивания события во время погружения или всплытия по большей части не зависит от того, что вы делаете. Очень редко может возникнуть путаница, когда код, отвечающий за прослушивание и обработку событий, делает не то, что нужно, так как вы случайно указали `true` вместо `false` в вызове `addEventListener`.

Этим я лишь хочу сказать, что в жизни может возникнуть ситуация, когда потребуется разбираться в фазах погружения/всплытия и работать с ними. Ошибка прокрадется в ваш код и выльется в многочасовое чесание затылка в поисках решения. Я могу привести список ситуаций из своей практики, когда мне пришлось осознанно выбирать фазу, в которой я наблюдал за событиями:

1. Перетаскивание элемента по экрану и обеспечение продолжения перетаскивания, даже если перемещаемый элемент выскользнет из-под курсора.
2. Вложенные меню, открывающие подменю при наведении на них указателя.
3. Есть несколько обработчиков событий в обеих фазах, а вы хотите сфокусироваться только на обработчиках в фазе погружения или всплытия.
4. Сторонний компонент и библиотека элементов управления имеют свою логику событий, и вы хотите обойти ее, чтобы использовать собственную настройку поведения.
5. Вы хотите переназначить предустановленное поведение браузера, например, когда вы нажимаете на полосу прокрутки или переключаетесь на текстовое поле.

За мои уже почти 105 лет работы с JavaScript могу привести только такие примеры. И даже они уже не столь однозначны, поскольку некоторые браузеры вообще некорректно работают с различными фазами.

Прерывание события

Последнее, о чем поговорим, — это о предотвращении распространения события. У события не обязательно должна быть полноценная жизнь,

в которой оно начинается и заканчивается в корне. Бывает, что лучше не давать ему счастливо дожить до старости.

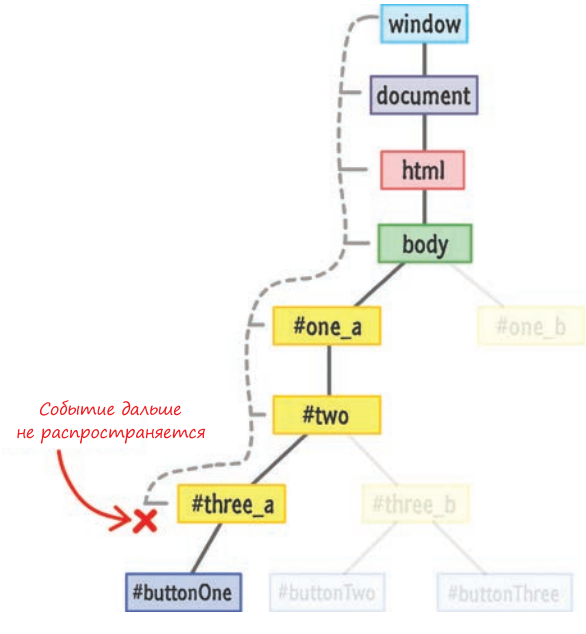
Чтобы прекратить существование события, можно использовать метод `stopPropagation` в объекте `Event`:

```
function handleClick(e) {
  e.stopPropagation();

  // что-нибудь делает
}
```

Метод `stopPropagation` прекращает движение события по фазам. Обратившись к предыдущему примеру, давайте предположим, что вы прослушиваете событие `click` в элементе `three_a` и хотите помешать этому событию распространиться. В этом случае код будет выглядеть так:

```
let theElement = document.querySelector("#three_a");
theElement.addEventListener("click", doSomething, true);

function doSomething(e) {
  e.stopPropagation();
}
```

В данном случае при нажатии на `buttonOne` путь нашего события будет выглядеть так:



Событие `click` начнет быстрое движение вниз по дереву DOM, уведомляя каждый элемент на своем пути к `buttonOne`. Так как элемент `three_a` прослушивает событие `click` во время фазы погружения, будет вызван связанный с ним обработчик событий:

```
function doSomething(e) {
  e.stopPropagation();
}
```

Как правило, события не продолжают распространение, пока взаимодействие с активированным обработчиком событий не будет завершено. Поскольку обработчик событий для `three_a` настроен реагировать на событие `click`, происходит вызов обработчика событий `doSomething`. Событие попадает в состояние задержки до тех пор, пока обработчик событий `doSomething` не будет выполнен и возвращен.

В данном случае событие не будет распространяться. Обработчик событий `doSomething` оказывается его последним клиентом благодаря функции `stopPropagation`, которая притаилась в тени, чтобы разделаться с событием раз и навсегда. Событие `click` не достигнет элемента `buttonOne` и не получит возможности вернуться к корню, как бы печально это ни было.

СОВЕТ

В вашем объекте события существует еще одна функция, с которой вы можете ненароком встретиться, и называется она `preventDefault`:

```
function overrideScrollBehavior(e) {
  e.preventDefault();

  // делает что-нибудь
}
```

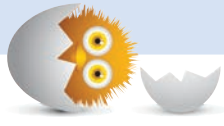
Действия этой функции немного загадочны. Многие HTML-элементы при взаимодействии с ними демонстрируют стандартное поведение. Например, щелчок по текстовой рамке производит переключение на нее и вызывает появление мигающего курсора. Использование колесика мыши в области, допускающей прокрутку, приведет к прокрутке в соответствующем направлении. Щелчок в графе для галочки переключит состояние отметки в положение да/нет. Браузер по умолчанию знает, как обработать встроенные реакции на все приведенные события.

Если нужно отключить это встроенное поведение, можно вызвать функцию `preventDefault`. Ее нужно вызывать во время реагирования на событие в элементе, чью встроенную реакцию вы хотите проигнорировать. Мой пример применения этой функции можно посмотреть здесь: http://bit.ly/kirupaParallax.

КОРОТКО О ГЛАВНОМ

Ну и как вам эта тема про события с их погружением и всплытием? Лучшим способом освоить принципы работы погружения и всплытия событий будет написание кода и наблюдение за перемещением события по DOM.

На этом мы завершили техническую часть этой темы, но если у вас есть несколько свободных минут, я предлагаю вам посмотреть связанный с ней эпизод *Comedians in Cars Getting Coffee*, метко названный *It's Bubble Time, Jerry!*. Это, вероятно, их лучший эпизод, в котором Майкл Ричардс и Джерри Сайнфелд попивают кофе и беседуют о событиях, фазе всплытия и прочих, на мой взгляд, важных вещах.



В ЭТОЙ ГЛАВЕ:

- научимся использовать события мыши для прослушивания ее действий;
- разберем объект MouseEvent;
- поработаем с колесиком мыши.

33

СОБЫТИЯ МЫШИ

Один из наиболее распространенных способов взаимодействия с компьютером — это использование *мыши* (рис. 33.1).

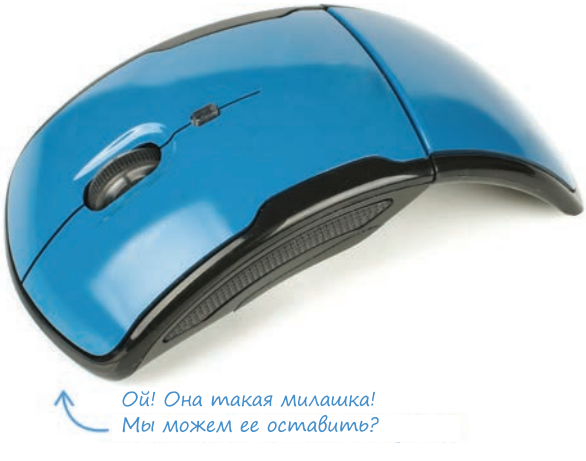


РИС. 33.1.
Кошки их тоже наверняка любят

Перемещая это волшебное устройство и нажимая на его кнопки, можно совершать множество полезных действий. При этом использование мыши простым пользователем — это одно. Для разработчика же согласование работы кода с действиями мыши — уже совсем другое. Здесь мы и начнем очередную главу.

Знакомьтесь с событиями мыши

В JavaScript основным способом работы с мышью являются события. Для этой задачи предусмотрено их великое множество, но в этой теме мы не станем рассматривать все подряд, а вместо этого сфокусируемся на самых популярных:

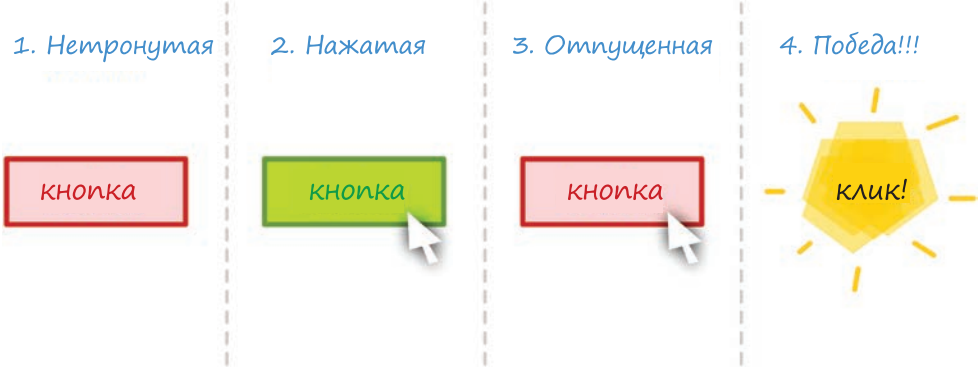
- `click`
- `dblclick`
- `mouseover`
- `mouseout`
- `mouseenter`
- `mouseleave`
- `mousedown`
- `mouseup`
- `mousemove`
- `contextmenu`
- `mousewheel` и `DOMMouseScroll`

Названия этих событий могут уже дать некоторое представление об их назначении, но мы, тем не менее, рассмотрим каждое из них подробнее. Хочу сразу предупредить вас, что некоторые из событий ужасно скучны.

Одинарный или двойной клик

Начнем с наиболее часто используемого события мыши — *клика*. Это событие срабатывает, когда вы щелкаете по элементу. Иначе можно сказать, что событие `click` срабатывает, когда вы используете мышь для нажатия на элемент, а затем отпускаете нажатие, сохраняя курсор на этом элементе.

Ниже — абсолютно бесполезная визуализация сказанного:



Вы уже несколько раз видели код для работы с событием клика, но его никогда не бывает достаточно, поэтому вот вам еще один пример:

```
let button = document.querySelector("#myButton");
button.addEventListener("click", doSomething, false);

function doSomething(e) {
  console.log("Mouse clicked on something!");
}
```

Прослушивание события `click` аналогично практически любому другому событию, поэтому не стану утомлять вас подробностями нашей старой знакомой функции `addEventListener`. Вместо этого утомлю вас подробностями, связанными с событием `dblclick`.

`dblclick` срабатывает, когда вы быстро щелкаете мышью дважды. Код для использования этого события будет следующим:

```
let button = document.querySelector("#myButton");
button.addEventListener("dblclick", doSomething, false);

function doSomething(e) {
  console.log("Mouse clicked on something...twice!");
}
```

Время между кликами, определяющее срабатывание события `dblclick`, заложено в ОС, где происходит выполнение кода. Оно не зависит от браузера и не определяется (не считывается) пользователем с помощью JavaScript.

НЕ ПЕРЕГРУЖАЙТЕ

Если вдруг случится прослушивать и событие `click`, и событие `dblclick` для элемента, обработчики событий будут вызваны три раза при двойном щелчке. Вы получите два события щелчка, которые будут соответствовать каждому щелчку. После второго щелчка вы также получите событие `dblclick`.

Наведение и отведение курсора

Классические сценарии наведения и отведения курсора обрабатываются логично названными событиями `mouseover` и `mouseout` соответственно:

Вот фрагмент кода с применением этих событий:

```
let button = document.querySelector("#myButton");
button.addEventListener("mouseover", hovered, false);
button.addEventListener("mouseout", hoveredOut, false);

function hovered(e) {
  console.log("Hovered!");
}

function hoveredOut(e) {
  console.log("Hovered Away!");
}
```

Это все, что касается этих событий. По большому счету, они весьма скучны, что, как вы уже наверняка поняли, даже хорошо, когда дело касается принципов программирования.

ЧТО НАСЧЕТ ДВУХ ДРУГИХ ПОХОЖИХ СОБЫТИЙ?

Мы рассмотрели только два события (`mouseover` и `mouseout`), которые срабатывают при наведении курсора на что-либо и его отведении. На деле же оказывается, что есть еще два события, которые делают то же самое, — `mouseenter` и `mouseleave`. Уникальность этих событий обусловливается одной важной деталью, а именно тем, что они не всплывают.

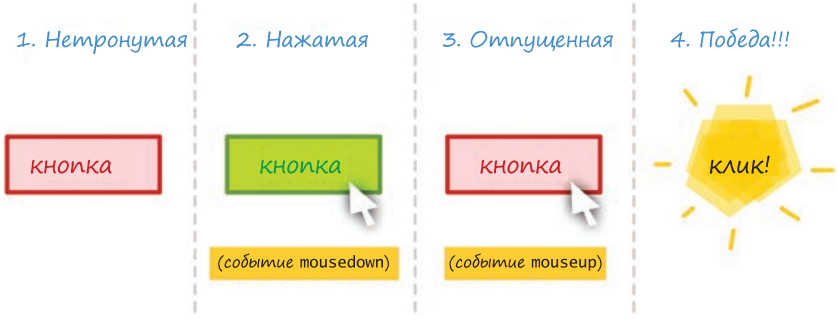
Это важно, только если интересующий вас элемент имеет потомков. Все эти четыре события ведут себя одинаково, если в процессе не присутствуют потомки. Если же таковые присутствуют, тогда:

- `mouseover` и `mouseout` будут срабатывать каждый раз, когда вы наводите курсор на потомка. Это значит, что можно увидеть срабатывание многих ненужных событий, несмотря на то что курсор двигается внутри одной области.
- `mouseenter` и `mouseleave` будут срабатывать только единожды. При этом не важно, через сколько потомков вы переместите курсор мыши.

В 90 % случаев вам вполне подойдут `mouseover` и `mouseout`. В остальных случаях, которые зачастую связаны с более сложными сценариями UI, вас порадует, что существуют такие события, как `mouseenter` и `mouseleave`.

События mousedown и mouseup

Два события, которые практически являются субкомпонентами события `click`, — это `mousedown` и `mouseup`. Следующая диаграмма поясняет почему:



Когда вы нажимаете на кнопку мыши, срабатывает событие `mousedown`. Когда вы отпускаете нажатие, срабатывает событие `mouseup`. Если нажатие и отпускание произошло на одном и том же элементе, тогда также сработает событие `click`.

Все это показано в следующем фрагменте кода:

```
let button = document.querySelector("#myButton");
button.addEventListener("mousedown", mousePressed, false);
button.addEventListener("mouseup", mouseReleased, false);
button.addEventListener("click", mouseClicked, false);

function mousePressed(e) {
  console.log("Mouse is down!");
}

function mouseReleased(e) {
  console.log("Mouse is up!");
}

function mouseClicked(e) {
  console.log("Mouse is clicked!");
}
```

Справедливый вопрос: «Зачем заморачиваться этими двумя событиями?» Кажется, что событие `click` идеально подходит для большинства случаев, в которых может понадобиться использовать `mousedown` и `mouseup`. Вы правы, можно не замарачиваться. При этом будет полезно пояснить, что события `mousedown` и `mouseup` просто дают больше контроля, когда он нужен. Некоторые взаимодействия (вроде перетаскиваний или отпадных приемов в видеоиграх, когда вы задерживаете нажатие, чтобы зарядить смертоносный удар молнии) подразумевают совершение действий, когда сработало только событие `mousedown`, но не `mouseup`.

Событие услышано снова… и снова… и снова!

Одним из самых болтливых событий, с которыми вам предстоит работать, является событие `mousemove`. Оно срабатывает огромное количество раз во время движения курсора по элементу, в котором вы прослушиваете событие `mousemove`:

Далее приведен пример использования `mousemove` в коде:

```
let button = document.querySelector("#myButton");
button.addEventListener("mousemove", mouseIsMoving, false);

function mouseIsMoving(e) {
  console.log("Mouse is on the run!");
}
```

Ваш браузер контролирует частоту, с которой срабатывает событие `mousemove`, при этом оно срабатывает, когда курсор мыши смещается даже всего на один пиксель. Это событие хорошо для многих интерактивных сценариев, в которых, к примеру, важно отслеживать текущую позицию курсора.

Контекстное меню

Последним связанным с мышью событием, которое мы рассмотрим, является `contextmenu`. Как вам наверняка хорошо известно, когда вы по обыкновению кликаете правой кнопкой мыши в различных приложениях, появляется меню:

Оно называется *контекстное меню*. Как раз перед появлением этого меню срабатывает событие `contextmenu`.

Честно говоря, есть всего одна весомая причина для прослушивания этого события. Она связана с предотвращением появления этого меню при правом клике, использовании связанной с ним клавиши клавиатуры или просто горячей клавиши.

Вот пример того, как вы можете *предотвратить встроенное поведение*, при котором появляется контекстное меню:

```
document.addEventListener("contextmenu", hideMenu, false);

function hideMenu(e) {
  e.preventDefault();
}
```

Метод `preventDefault` в любом типе `Event` предотвращает любое его встроенное действие. Так как событие `contextmenu` срабатывает до появления меню, вызов `preventDefault` гарантирует, что оно показано не будет. Да, я уже второй раз упоминаю это свойство, но вы же знаете, что мне платят за количество символов (ха-ха).

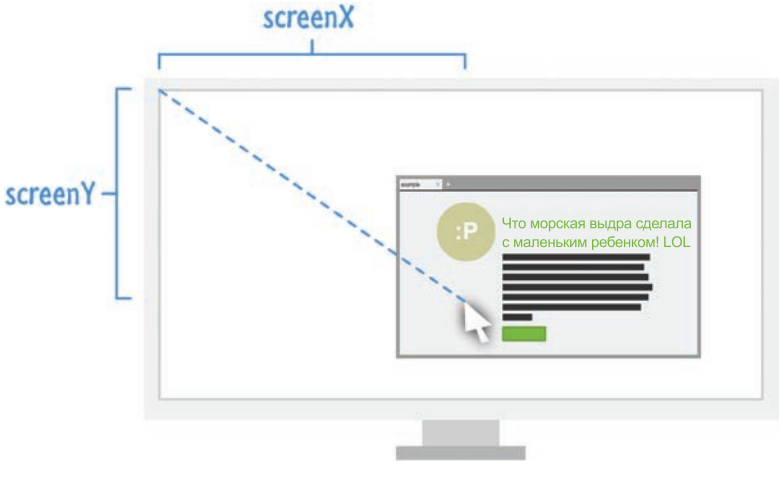
Учитывая все сказанное, я могу придумать множество альтернативных способов для предотвращения появления контекстного меню без помощи событий, но пока что дела обстоят именно так.

Свойства MouseEvent

Давайте перейдем к конкретике. Все события мыши, которые мы видели до сих пор, основаны на `MouseEvent`. Обычно эту разновидность фактоида[1] вы храните исключительно для торжественных случаев и игнорируете. Тем не менее в данном случае эта деталь для нас важна, так как `MouseEvent` несет в себе набор свойств, упрощающих работу с мышью. Давайте на них посмотрим.

Глобальная позиция мыши

Свойства `screenX` и `screenY` возвращают расстояние, на котором находится курсор мыши от левого верхнего угла основного монитора:



Вот очень простой пример использования `screenX` и `screenY`:

```
document.addEventListener("mousemove", mouseMoving, false);

function mouseMoving(e) {
  console.log(e.screenX + " " + e.screenY);
}
```

[1] Фактóид — недостоверное или ложное утверждение, которое облекается в форму достоверного и выдается за достоверное. — *Примеч. ред.*

При этом не важно, какие еще действия с отступами, заполнением или сдвигами макета производятся на странице. Возвращаемые значения всегда будут отражать расстояние между текущим положением курсора и левым верхним углом основного монитора.

Позиция курсора мыши в браузере

Свойства `clientX` и `clientY` возвращают позиции `x` и `y` курсора относительно левого верхнего угла браузера (технически его области просмотра):

Код в данном случае достаточно прост:

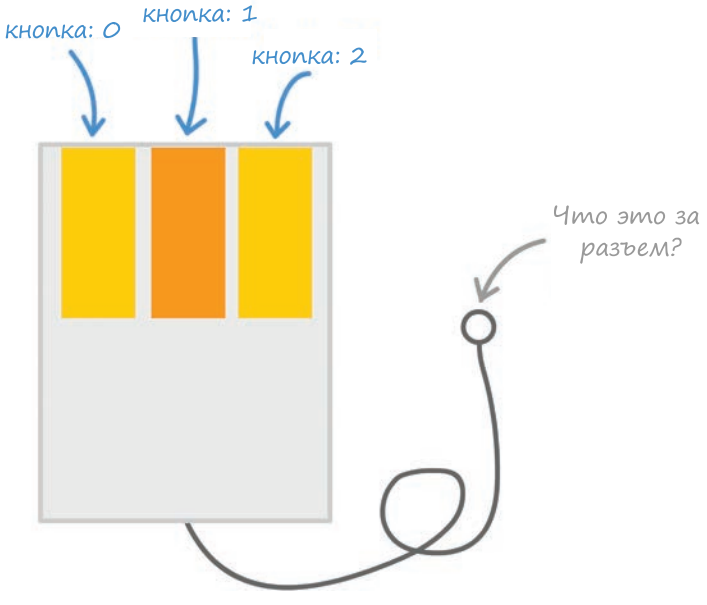
```
let button = document.querySelector("#myButton");
document.addEventListener("mousemove", mouseMoving, false);

function mouseMoving(e) {
  console.log(e.clientX + " " + e.clientY);
}
```

Вы просто вызываете свойства `clientX` и `clientY` аргумента события, переданного в обработчик события, чтобы получить их значения.

Определение нажатой кнопки

Мыши зачастую оборудованы несколькими кнопками или предоставляют возможность их имитировать. Наиболее распространенная кон-

фигурация состоит из левой, правой и средней (обычно это нажатие на колесико мыши) кнопок. Для определения, какая из кнопок была нажата, существует свойство `button`. Это свойство возвращает `0` при нажатии левой кнопки, `1` — при нажатии средней и `2` — при нажатии правой:



Код для использования этого свойства выглядит вполне ожидаемо:

```
document.addEventListener("mousedown", buttonPress, false);

function buttonPress(e) {
  if (e.button == 0) {
    console.log("Left mouse button pressed!");
  } else if (e.button == 1) {
    console.log("Middle mouse button pressed!");
  } else if (e.button == 2) {
    console.log("Right mouse button pressed!");
  } else {
    console.log("Things be crazy up in here!!!");
  }
}
```

В дополнение к свойству `button` существует свойство `buttons`, а также другие, которые делают нечто похожее для помощи в определении нажатой кнопки. Я не буду много говорить об этих свойствах, просто имейте в виду, что они существуют (вы всегда можете погуглить их).

Работа с колесиком мыши

Колесико мыши отличается от всего, что мы рассмотрели до этого момента. Очевидная разница в том, что здесь мы уже имеем дело с колесиком, а не кнопкой. Менее же очевидное, но при этом более важное отличие в том, что в данном случае вы используете уже два события для работы. Первое — это `mousewheel`, используемое в Internet Explorer и Chrome, а второе — это `DOMMouseScroll`, используемое в Firefox.

Прослушивание этих событий производится обычным образом:

```
document.addEventListener("mousewheel", mouseWheeling, false);
document.addEventListener("DOMMouseScroll", mouseWheeling, false);
```

А вот после уже есть нюансы. События `mousewheel` и `DOMMouseScroll` будут срабатывать в момент прокручивания колесика в любом направлении. Но для любой практической цели будет важно, в каком направлении происходит прокрутка. Чтобы получить эту информацию, пороемся в обработчике событий и найдем аргумент события.

Аргументы события для события `mousewheel` содержат свойство под названием `wheelDelta`. В случае же с `DOMMouseScroll` в аргументе события присутствует свойство `detail`. Оба этих свойства похожи в том, что их значения изменяются на положительные или отрицательные в зависимости от направления прокрутки колесика. Здесь стоит отметить, что они не согласованы в трактовке положительного и отрицательного значения. Свойство `wheelDelta`, связанное с событием `mousewheel`, становится положительным при прокрутке вверх и отрицательным при прокрутке вниз. В точности наоборот происходит в случае со свойством `DOMMouseScroll`. При прокрутке вверх оно дает отрицательное значение, а при прокрутке вниз — положительное.

Из следующего примера видна обработка этой несогласованности свойств `wheelDelta` и `detail`, которая весьма проста:

```
function mouseWheeling(e) {
  let scrollDirection;
  let wheelData = e.wheelDelta;

  if (wheelData) {
    scrollDirection = wheelData;
  } else {
    scrollDirection = -1 * e.detail;
  }
```

```
  if (scrollDirection > 0) {
    console.log("Scrolling up! " + scrollDirection);
  } else {
    console.log("Scrolling down! " + scrollDirection);
  }
}
```

Переменная `scrollDirection` хранит значение, содержащееся в свойстве `wheelData` или `detail`. Вы можете определить особое поведение в зависимости от того, является значение положительным или отрицательным.

КОРОТКО О ГЛАВНОМ

Если вы умеете работать с одним событием, то, значит, понимаете основу работы со всеми остальными. Главное — знать, какое событие соответствует нужным вам действиям. Знакомство с событиями мыши — это хорошее начало для освоения работы с событиями в принципе, так как ничего сложного в этом нет. Они не отличаются беспорядочностью, и то, что вы о них узнаете, вы будете использовать практически во всех своих приложениях.

Дополнительные ресурсы и примеры, которые могут вас заинтересовать:

- Перемещение элемента в место клика: http://bit.ly/kirupaElementClickPosition
- Вы используете сенсорное устройство? http://bit.ly/kirupaTouchEnabled

Если у вас есть какие-либо вопросы, уделите им время и обратитесь на форум https://forum.kirupa.com.

В ЭТОЙ ГЛАВЕ:

- научимся прослушивать клавиатуру и реагировать на нее;
- поймем, как работать с событиями клавиатуры;
- рассмотрим работу некоторых популярных сценариев клавиатуры.

34

СОБЫТИЯ КЛАВИАТУРЫ

Работая в приложениях, очень много внимания мы уделяем клавиатуре. Если вдруг вам интересно, как выглядит это устройство, то на рис. 34.1 показан один из экземпляров, вероятно, столетней давности.

РИС. 34.1.
Так выглядит музейный экспонат клавиатуры

Как бы то ни было, наши компьютеры (а именно выполняемые на них приложения) умеют взаимодействовать с этими усыпанными клави-

шами пластиковыми досками. Вас же этот процесс интересует редко. Однако в нашем случае в зависимости от выполняемой задачи думать об этом взаимодействии придется регулярно. Если быть точнее, то придется не только думать о нем, но и научиться его налаживать. Поэтому лучше отмените все ближайшие планы, так как текущая глава будет насыщенной.

К моменту ее завершения вы узнаете все о том, как прослушивать события клавиатуры, что каждое из них делает, и познакомитесь с рядом примеров, демонстрирующих удобные трюки, которые порой могут очень пригодиться.

Поехали!

Знакомьтесь с событиями клавиатуры

Для работы с клавиатурой в HTML-документе вам потребуется познакомиться с тремя событиями, а именно:

- `keydown`;
- `keypress`;
- `keyup`.

Глядя на названия этих событий, вы наверняка уже догадались, за что они отвечают. `keydown` срабатывает при нажатии клавиши, а `keyup` — при ее отпускании. Оба этих события работают для любой клавиши, с которой вы взаимодействуете.

Событие же `keypress` — это отдельный случай. На первый взгляд может показаться, что оно срабатывает при нажатии клавиши. Но, несмотря на название, срабатывает оно, только когда вы нажимаете на клавишу, которая отображает знак (букву, цифру и т. п.). Это может показаться не совсем понятным, но своеобразный смысл здесь все-таки присутствует.

Если вы нажмете и отпустите клавишу знака вроде буквы `y`, то увидите, что по порядку сработали события `keydown`, `keypress` и `keyup`. В данном случае `keydown` и `keyup` сработали потому, что клавиша `y` для них — просто клавиша. Событие же `keypress` сработало, так как клавиша `y` — это клавиша знака. Если вы нажмете и отпустите клавишу, которая на экране ничего не отображает (например, пробел, стрелка

или функциональные клавиши), то увидите, что сработали только события `keydown` и `keyup`.

Это неявное отличие, но оно очень важно, чтобы нажатия клавиш были услышаны приложением.

ЧТО СКАЗАЛ?

Странно, что событие под названием `keypress` не срабатывает при нажатии любой клавиши. Может быть, это событие следует назвать как-то иначе, например `characterkeypress`, но это, скорее всего, проблема МУ (все равно, что мнение коровы и ее мнение никого не волнует). (Подробнее о проблеме МУ здесь: http://bit.ly/kirupaMoo)

Использование событий

Прослушивание событий `keydown`, `keyup` и `keypress` аналогично любым другим событиям, которые мы прослушиваем и на которые реагируем. Вы вызываете `addEventListener` для элемента, который будет работать с этим событием, указываете событие, которое нужно прослушать, указываете функцию обработчика событий, которая вызывается, когда событие услышано, а также указываете значение `true` или `false`, определяя, должно ли оно прослушиваться в фазе всплытия.

Вот пример прослушивания трех событий клавиатуры в объекте `window`:

```
window.addEventListener("keydown", dealWithKeyboard, false);
window.addEventListener("keypress", dealWithKeyboard, false);
window.addEventListener("keyup", dealWithKeyboard, false);

function dealWithKeyboard(e) {
  // вызывается, когда услышано любое из событий клавиатуры
}
```

Как только любое из этих событий будет услышано, последует вызов обработчика событий `dealWithKeyboard`. На деле же при нажатии клавиши знака этот обработчик будет вызван трижды. Все это вполне по-

нятно, поэтому давайте увеличим уровень сложности и в последующих разделах уже выйдем за рамки основ.

Свойства события Keyboard

Когда происходит вызов обработчика событий, передается аргумент события `Keyboard`. Давайте вернемся к обработчику событий `dealWithKeaboard` из предыдущего раздела. В нем событие клавиатуры представлено передаваемым аргументом `e`:

```
function dealWithKeyboard(e) {
  // вызывается, когда услышано любое событие клавиатуры
}
```

Этот аргумент содержит несколько свойств:

- `KeyCode`. Каждая клавиша клавиатуры имеет связанное с ней число. Это число возвращается свойством только для чтения.
- `CharCode`. Это свойство существует только в аргументах события, возвращенных событием `keypress`, и содержит код ASCII для любой нажатой клавиши знака.
- `ctrlKey`, `altKey`, `shiftKey`. Эти три свойства возвращают `true`, если нажата клавиша Ctrl, Alt или Shift.
- `MetaKey`. Это свойство похоже на `ctrlKey`, `altKey` и `shiftKey` тем, что возвращает `true`, если нажата клавиша Meta на клавиатурах Windows или клавиша Command на клавиатурах Apple.

Событие `Keyboard` содержит и другие свойства, но приведенные выше являются наиболее интересными. Используя их, вы можете проверять, какая была нажата клавиша, и реагировать соответственно. В двух последующих разделах вы увидите примеры.

ВНИМАНИЕ

Свойства `charCode` и `keyCode` считаются специалистами по веб-стандартам W3C устаревшими. Их альтернативой может стать мало поддерживаемое свойство `code`. Просто имейте это в виду и будьте готовы к обновлению кода, как только на трон взойдет преемник этих свойств.

Примеры

Теперь, когда скучные основы работы с событиями клавиатуры позади, пора рассмотреть некоторые примеры, которые прояснят (а может, и запутают) все пройденное.

Проверка нажатия конкретной клавиши

Следующий пример показывает, как использовать свойство `keyCode` для проверки нажатия конкретной клавиши:

```
window.addEventListener("keydown", checkKeyPressed, false);

function checkKeyPressed(e) {
  if (e.keyCode == 65) {
    console.log("The 'a' key is pressed.");
  }
}
```

Здесь я проверяю клавишу `a`. Внутренне эта клавиша отображается значением `65` свойства `keyCode`. В случае если вы так и не зазубрили их в школе, можете обратиться к подручному списку кодов для всех клавиш и знаков по следующей ссылке: http://bit.ly/kirupaKeyCode.

Пожалуйста, не заучивайте все коды из этого списка, так как есть гораздо более интересные вещи для запоминания.

Прошу отметить следующее. Значения `charCode` и `keyCode` для конкретной клавиши не будут одинаковы. Помимо этого, `charCode` возвращается, только если обработчик событий сработал в ответ на `keypress`. В нашем примере событие `keydown` не содержало бы ничего полезного для свойства `charCode`.

Если вы захотите проверить `charCode` и использовать событие `keypress`, то предыдущий пример будет выглядеть так:

```
window.addEventListener("keypress", checkKeyPressed, false);

function checkKeyPressed(e) {
  if (e.charCode == 97) {
    console.log("The 'a' key is pressed.");
  }
}
```

`charCode` для клавиши `a` — это `97`. Опять же напомню, подробности можно взять из таблицы, приведенной по ссылке выше.

Совершение действий при нажатии клавиш стрелок

Чаще всего мы встречаемся с этим в играх, где нажатие стрелок приводит к выполнению действий. В следующем листинге показана реализация:

```
window.addEventListener("keydown", moveSomething, false);

function moveSomething(e) {
  switch (e.keyCode) {
    case 37:
      // нажатие влево
      break;
    case 38:
      // нажатие вверх
      break;
    case 39:
      // нажатие вправо
      break;
    case 40:
      // нажатие вниз
      break;
  }
}
```

Здесь все вполне понятно. Помимо прочего, это актуальный пример использования инструкции `switch`, которую вы изучили еще в далекой главе 4.

Определение нажатия нескольких клавиш

А вот теперь эпичная часть! Самое интересное связано с определением нажатия нескольких клавиш и соответственного реагирования. Далее показано, как это можно сделать:

```
window.addEventListener("keydown", keysPressed, false);
window.addEventListener("keyup", keysReleased, false);

let keys = [];

function keysPressed(e) {
  // сохраняет запись о каждой нажатой клавише
  keys[e.keyCode] = true;
```

```
  // Ctrl + Shift + 5
  if (keys[17] && keys[16] && keys[53]) {
    // делает что-нибудь
  }

  // Ctrl + f
  if (keys[17] && keys[70]) {
    // делает что-нибудь

    // предотвращает встроенное поведение браузера
     e.preventDefault();
  }
}

function keysReleased(e) {
  // отмечает отпущенные клавиши
  keys[e.keyCode] = false;
}
```

Чтобы разобрать все, что здесь происходит, понадобится отдельная глава, поэтому давайте пробежимся по верхам.

Во-первых, имеется массив клавиш, хранящий каждую клавишу, которую вы нажимаете:

```
let keys = [];
```

По мере нажатия клавиш происходит вызов обработчика событий `keysPressed`:

```
function keysPressed(e) {
  // хранит запись о каждой нажатой клавише
  keys[e.keyCode] = true;

  // Ctrl + Shift + 5
  if (keys[17] && keys[16] && keys[53]) {
    // делает что-нибудь
  }
  // Ctrl + f
  if (keys[17] && keys[70]) {
    // делает что-нибудь

    // предотвращает встроенное поведение браузера
    e.preventDefault();
  }
}
```

Когда клавиша отпускается, происходит вызов обработчика событий `keysRealesed`:

```
function keysReleased(e) {
  // помечает отпущенные клавиши
  keys[e.keyCode] = false;
}
```

Обратите внимание, как эти два обработчика событий взаимодействуют. При нажатии клавиш для них создается запись в массиве `keys` со значением `true`. При отпускании же клавиш их значения меняются на `false`. Существование в массиве нажимаемых вами клавиш поверхностно, важны лишь хранимые ими значения.

До тех пор пока ничего вроде всплывающего окна уведомления не мешает правильному вызову ваших обработчиков событий, с позиции массива `keys` будет возникать взаимно однозначное соответствие между нажатыми и отпущенными клавишами. Учитывая это, проверка определения комбинации нажатых клавиш производится в обработчике событий `keysPressed`. Выделенные в следующем коде строки показывают, как это работает:

```
function keysPressed(e) {
  // хранит запись о каждой нажатой клавише
  keys[e.keyCode] = true;

  // Ctrl + Shift + 5
  if (keys[17] && keys[16] && keys[53]) {
    // делает что-то
  }

  // Ctrl + f
  if (keys[17] && keys[70]) {
    // делает что-то

    // предотвращает стандартное поведение браузера
    e.preventDefault();
  }
}
```

Важно учитывать одну деталь: некоторые комбинации клавиш приводят к реагированию браузера. Чтобы избежать выполнения браузером ненужных вам действий, используйте метод `preventDefault`, как это выделено, при проверке использования Ctrl + F:

```
function keysPressed(e) {
  // хранит запись о каждой нажатой клавише
  keys[e.keyCode] = true;

  // Ctrl + Shift + 5
  if (keys[17] && keys[16] && keys[53]) {
    // делает что-то
 }
```

```
  // Ctrl + f
  if (keys[17] && keys[70]) {
    // делает что-то

    // предотвращает стандартное поведение браузера
    e.preventDefault();
  }
}
```

Метод `preventDefault` предотвращает стандартную реакцию браузера на событие. В данном случае он не дает браузеру показать диалоговое окно поиска. Различные комбинации клавиш будут запускать различные реакции браузера, поэтому держите этот метод на вооружении, чтобы исключить их.

В любом случае, при рассмотрении кода в совокупности у вас есть схема для удобной проверки нажатия нескольких клавиш.

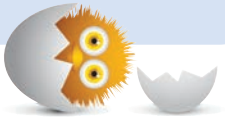
КОРОТКО О ГЛАВНОМ

Клавиатура выступает важным элементом взаимодействия пользователей с компьютерами и аналогичными им устройствами. Несмотря на это, не всегда приходится иметь с ней дело напрямую. Ваш браузер, различные относящиеся к тексту элементы управления и аналогичные им компоненты уже справляются с этим по умолчанию. Тем не менее есть определенные приложения, где может потребоваться непосредственно поработать с клавиатурой. Этому и была посвящена текущая глава.

Она началась максимально скучно с объяснения принципа работы событий `Keyboard` и их аргументов событий. По мере продвижения все становилось интереснее: вы увидели некоторые примеры кода, где были показаны действия с клавиатурой. Если у вас есть какие-либо вопросы по этой или другой теме, не стесняйтесь обращаться с ними на форум https://forum.kirupa.com.

В ЭТОЙ ГЛАВЕ:

- познакомимся со всеми событиями, срабатывающими при загрузке страницы;
- поймем, что происходит при этом за кадром;
- поиграем с различными атрибутами элемента script, контролирующими точное время запуска кода.



35

СОБЫТИЯ ЗАГРУЗКИ СТРАНИЦЫ И ПРОЧЕЕ

Важной частью работы с JavaScript является обеспечение выполнения кода в нужное время. И часто все не так просто, чтобы лишь поместить код вниз страницы и ждать, что все начнет работать, как только страница загрузится. Здесь мы еще раз вернемся к некоторым моментам из главы 10. Во многих случаях вам может понадобиться добавить дополнительный код, чтобы гарантировать, что код не будет запущен, пока страница не будет к этому готова. А иногда может даже понадобиться поместить код именно в начало страницы.

Есть множество факторов, влияющих на выбор «подходящего времени» для запуска кода. В этой главе мы рассмотрим эти факторы и сформируем из пройденного материала небольшое полезное руководство.

Поехали!

Что происходит в процессе загрузки страницы

Начнем с самого начала. Вы щелкаете по ссылке либо нажимаете Enter после набора URL, и если сошлись звезды, загружается страница. Все кажется очень простым и занимает мало времени:

За этот короткий отрезок времени происходит множество связанных с ним интересных процессов, о которых следует знать подробнее. Одним из примеров таких процессов является выполнение заданного для страницы кода. Точный момент выполнения этого кода зависит от сочетания следующих компонентов, которые оживают в определенный момент загрузки страницы:

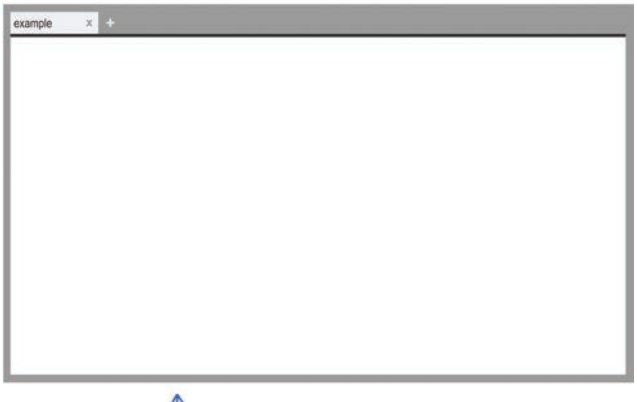
- событие `DOMContentLoaded`;
- событие `load`;
- атрибут `async` элементов сценария;
- атрибут `defer` элементов сценария;
- место расположения сценария в DOM.

Не переживайте, если перечисленное вам не знакомо. Очень скоро вы узнаете о назначении всех этих штук. Но сначала рассмотрим три стадии загрузки страницы.

Стадия первая

Первая стадия охватывает момент, когда браузер вот-вот начнет загрузку страницы.

Стадия 1. Ничего особенного



Браузер ожидает загрузки нового содержимого страницы

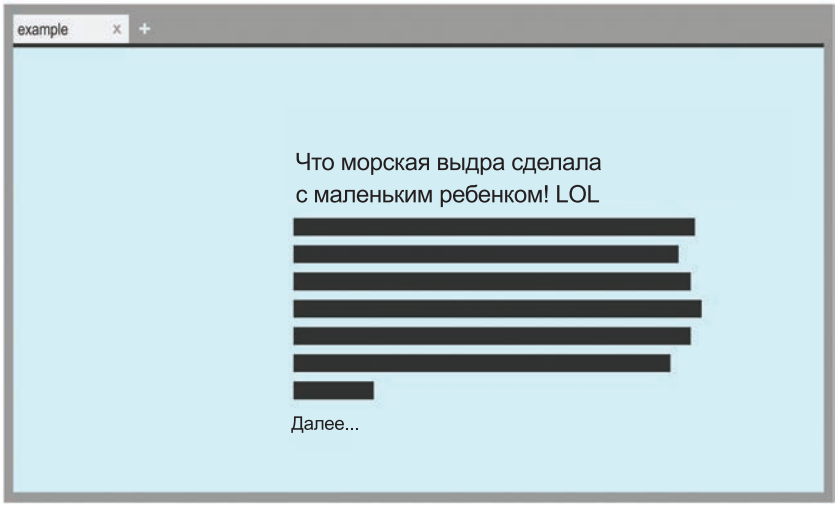
Во время этой стадии не происходит ничего интересного. Запрос на загрузку страницы уже был сделан, но еще ничего не загружено.

Стадия вторая

Здесь уже более насыщенный процесс, во время которого происходят загрузка и обработка сырой разметки, а также DOM-страницы.

Здесь стоит отметить, что внешние ресурсы вроде изображений и связанных с ними таблиц стилей еще не считаны. Вы видите только сырое содержимое, определенное разметкой страницы или документа.

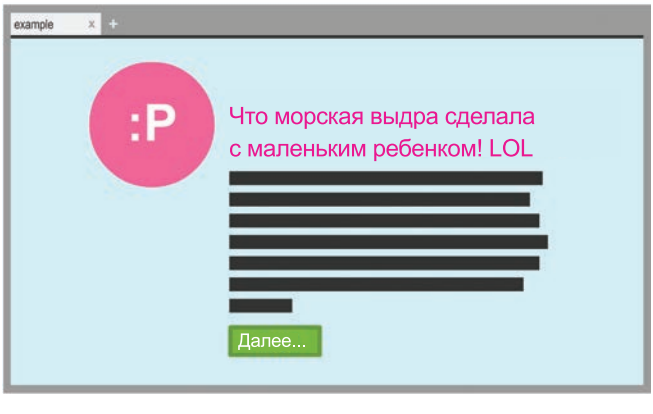
Стадия 2. DOM готов



Стадия третья

Во время заключительной стадии страница уже полностью загружена со всеми изображениями, таблицами стилей, сценариями и прочими внешними ресурсами, преобразованными в то, что вы видите:

Стадия 3. Страница полностью загружена



На этой стадии индикаторы загрузки браузера прекращают анимацию, и именно здесь вы почти всегда оказываетесь при взаимодействии

с HTML-документом. Учитывая все сказанное, иногда страница может оказаться в промежуточном состоянии, когда 99 % контента загрузилось, но какой-то случайный элемент застревает в загрузке навечно. Если вы посещали один из вирусных, или информационных, или фид-сайтов, то прекрасно поймете, о чем я.

Теперь, когда у вас есть представление о трех стадиях, которые проходит документ при загрузке содержимого, перейдем к более интересной части. А к трем стадиям мы вернемся в ближайших разделах.

DOMContentLoaded и load Events

Есть два события, представляющих два основных ориентира в процессе загрузки страницы: `DOMContentLoaded` и `load`. `DOMContentLoad` срабатывает в конце стадии 2, когда DOM страницы полностью обработан. Событие `load` срабатывает в конце стадии 3, как только страница полностью завершает загрузку. Вы можете использовать эти события для выбора времени выполнения кода.

Ниже приведен фрагмент кода с использованием этих двух событий:

```
document.addEventListener("DOMContentLoaded", theDomHasLoaded,
false);
window.addEventListener("load", pageFullyLoaded, false);

function theDomHasLoaded(e) {
  // делает что-нибудь
}

function pageFullyLoaded(e) {
  // снова делает что-нибудь
}
```

Вы используете эти события так же, как и любые другие, но при этом важно учесть, что вам надо прослушивать `DOMContentLoaded` из элемента `document`, а `load` — из элемента `window`.

Теперь, когда со всеми скучными техническими деталями покончено, подумаем, почему эти события важны? Очень просто. Если у вас есть код, опирающийся на работу с DOM вроде всего того, что использует `querySelector` или `querySelectorAll`, то вам нужно обеспечить, чтобы этот код запускался только после полной загрузки DOM. Если вы попробуете обратиться к DOM до этого момента, то либо получите неполные результаты, либо не получите их вообще.

Вот прекрасный радикальный пример от Кайла Мюррея:

```
<!DOCTYPE html>
<html>

<head>
  <script>
    // попытайтесь проанализировать здесь содержимое книги
  </script>
</head>

<body>
  [Вставьте здесь полную копию /Войны и мира/]
</body>

</html>
```

Верный способ избежать ситуации, в которой код запускается до момента готовности DOM, — это прослушать событие `DOMContentLoaded` и установить запуск кода, опирающегося на DOM только тогда, когда это событие будет услышано:

```
document.addEventListener("DOMContentLoaded", theDomHasLoaded, false);

function theDomHasLoaded(e) {
  let headings = document.querySelectorAll("h2");

  // делает что-нибудь с изображениями
}
```

Для случаев, когда нужно, чтобы код запускался только после полной загрузки страницы, используйте событие `load`. За все годы использования JavaScript мне не так часто приходилось использовать это событие на уровне документа, за исключением проверки итоговых размеров загруженного изображения или создания простых индикаторов прогресса.

Сценарии и их расположение в DOM

В главе 8 мы рассмотрели различные способы для определения положения сценариев внутри документа. Вы видели, что положение элементов в DOM влияет на момент запуска. В этом разделе подтвердим эту простую истину и немного углубимся.

Вспомним, что простой элемент сценария может быть встроенным кодом в какой-то части документа:

```
<script>
  let number = Math.random() * 100;
  console.log("A random number is: " + number);
</script>
```

Он также может быть чем-то, что ссылается на некий код во внешнем файле:

```
<script src="/foo/something.js"></script>
```

А теперь важная деталь относительно этих элементов. Ваш браузер считывает DOM последовательно сверху вниз. Любые элементы сценария, встречающиеся на его пути, будут считаны в том порядке, в каком они расположены в DOM.

Ниже приведен очень простой пример со множеством элементов сценария:

```
<!DOCTYPE html>
<html>

<body>
  <h1>Example</h1>
  <script>
    console.log("inline 1");
  </script>
  <script src="external1.js"></script>
  <script>
    console.log("inline 2");
  </script>
  <script src="external2.js"></script>
  <script>
    console.log("inline 3");
  </script>
</body>

</html>
```

Не важно, содержит ли сценарий встроенный код или ссылается на внешний источник, — все сценарии рассматриваются одинаково и запускаются в том порядке, в котором расположены в документе. В верхнем примере порядок выполнения сценариев будет следующим: `inline 1`, `external 1`, `inline 2`, `external 2` и в конце `inline 3`.

А вот еще одна, но уже очень важная деталь, которую необходимо учитывать. Так как DOM считывается сверху вниз, ваш элемент сценария имеет доступ ко всем элементам DOM, которые уже были считаны. И наоборот, ваш сценарий не имеет доступа к еще не считанным элементам DOM. Как вам такое?

Предположим, есть элементы сценария, расположенные внизу страницы чуть выше закрывающего тело элемента:

```
<!DOCTYPE html>
<html>

<body>
  <h1>Example</h1>

  <p>
  Quisque faucibus, quam sollicitudin pulvinar dignissim, nunc
  velit sodales leo, vel vehicula odio lectus vitae mauris. Sed
  sed magna augue. Vestibulum tristique cursus orci, accumsan
  posuere nunc congue sed. Ut pretium sit amet eros non consectetur.
  Quisque tincidunt eleifend justo, quis molestie tellus venenatis
  non. Vivamus interdum urna ut augue rhoncus, eu scelerisque
  orci dignissim. In commodo purus id purus tempus commodo.
  </p>

  <button>Click Me</button>

  <script src="something.js"></script>
</body>

</html>
```

Когда выполняется something.js, он может обратиться ко всем элементам DOM, находящимся над ним, вроде `h1`, `p` и `button`. Если ваш элемент сценария расположен в верхней части документа, он не будет знать о других элементах DOM, расположенных ниже него:

```
<!DOCTYPE html>
<html>

<body>
  <script src="something.js"></script>

  <h1>Example</h1>

  <p>
  Quisque faucibus, quam sollicitudin pulvinar dignissim, nunc
  velit sodales leo, vel vehicula odio lectus vitae mauris. Sed
  sed magna augue. Vestibulum tristique cursus orci, accumsan
  posuere nunc congue sed. Ut pretium sit amet eros nonconsectetur.
  Quisque tincidunt eleifend justo, quis molestie tellus venenatis
  non. Vivamus interdum urna ut augue rhoncus, eu scelerisque
  orci dignissim. In commodo purus id purus tempus commodo.
  </p>
```

```
  <button>Click Me</button>

</body>

</html>
```

При размещении элемента сценария внизу страницы, как было показано ранее, его конечное поведение будет таким же, будто есть код, явно слушающий событие `DOMContentLoaded`. Если вы сможете сделать так, что сценарии появятся ближе к концу документа, после всех элементов DOM, то полностью избежите использования подхода `DOMContentLoaded`, описанного в предыдущем разделе. Итак, если вам действительно нужно расположить элементы сценария в верхней части DOM, обеспечьте, чтобы весь код, опирающийся на DOM, выполнялся после срабатывания события `DOMContentLoaded`.

В этом вся суть. Я большой поклонник размещения элементов сценария в нижней части DOM. Есть и еще одна причина кроме упрощенного доступа к DOM, почему я рекомендую располагать сценарии внизу страницы. Когда элемент сценария считывается, браузер приостанавливает выполнение всего остального на странице на время выполнения его кода. Если речь идет о длительно выполняемом сценарии или внешнем сценарии, требующем время на загрузку, HTML-страница будет попросту заморожена. Если же на этот момент ваша DOM будет считана лишь частично, то страница помимо остановки еще и будет выглядеть незавершенной. А это вряд ли кому-то понравится.

Элементы сценария async и defer

В предыдущем разделе я объяснил, как расположение элементов сценария в DOM определяет время их запуска. Все это относится только к тем элементам, которые я называю простыми. Чтобы стать частью непростого мира, элементы сценария, указывающие на внешние сценарии, могут содержать атрибуты `defer` и `async`:

```
<script async src="myScript.js"></script>
<script defer src="somethingSomethingDarkSide.js"></script>
```

Эти атрибуты изменяют время запуска сценария вне зависимости от того, где в DOM они фактически расположены. Посмотрим, как они это делают.

async

Атрибут `async` позволяет сценарию выполняться асинхронно:

```
<script async src="someRandomScript.js"></script>
```

Если вспомнить предыдущий раздел, то на время считывания элемента браузер может заблокироваться и стать недееспособным. Установив атрибут `async` в элементе сценария, вы полностью избегаете этой проблемы. Сценарий выполнит все, что должен, и при этом ничто не помешает браузеру заниматься своими делами.

Такая беспечность в выполнении кода довольно удивительна, но важно понимать, что сценарии, отмеченные как `async`, не всегда будут запускаться по порядку. Может случиться, что несколько таких сценариев будут запущены в последовательности, отличной от указанной в разметке. Точно известно лишь то, что сценарии с `async` начнут выполнение в некоей загадочной точке до срабатывания события `load`.

defer

Атрибут `defer` несколько отличен от `async`:

```
<script defer src="someRandomScript.js"></script>
```

Сценарии, помеченные `defer`, запускаются в том порядке, в каком были определены, но выполняются только в самом конце, за несколько мгновений до срабатывания события `DOMContentLoaded`. Взгляните на следующий пример:

```
<!DOCTYPE html>
<html>

<body>
  <h1>Example</h1>
  <script defer src="external1.js"></script>
  <script>
    console.log("inline 1");
  </script>
  <script src="external2.js"></script>
  <script>
    console.log("inline 2");
  </script>
  <script defer src="external3.js"></script>
  <script>
    console.log("inline 3");
```

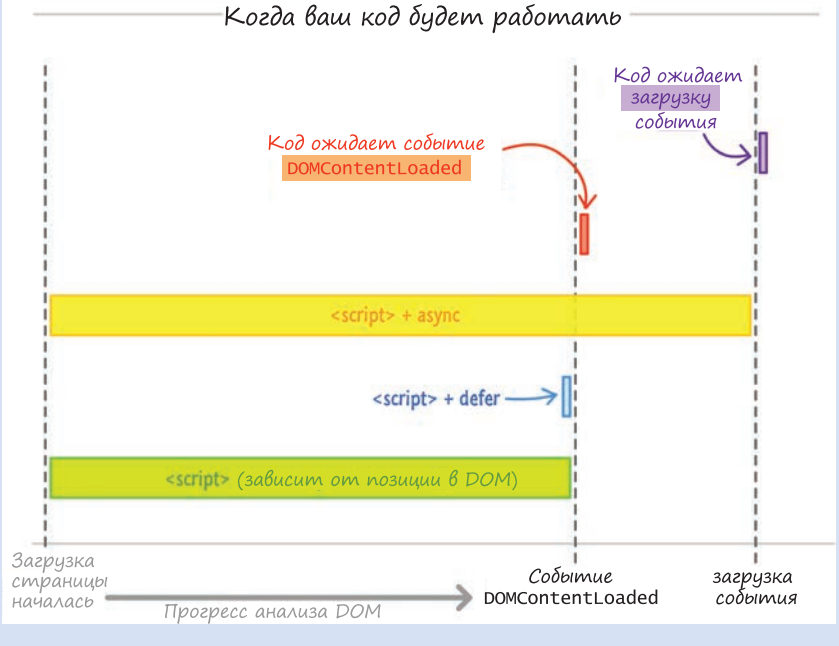
```
  </script>
</body>

</html>
```

Задумайтесь на секунду и расскажите находящемуся рядом человеку (или животному), в каком порядке эти сценарии будут запущены. При этом можете не пояснять контекст, ведь если они вас любят, то обязательно поймут.

Запустятся они в такой последовательности: `inline 1`, `external 2`, `inline 2`, `inline 3`, `external 3`, а затем `external 1`. Сценарии `external 3` и `external 1` помечены как `defer`, именно поэтому они оказываются в конце, несмотря на свое положение в разметке.

КОРОТКО О ГЛАВНОМ

В последних разделах мы рассмотрели факторы, влияющие на время запуска кода. Схема ниже объединяет весь этот материал:



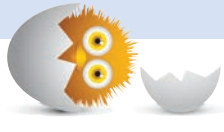
Теперь перейдем к актуальному для вас вопросу. Какое время будет наилучшим для выполнения вашего кода JavaScript? Важно добиться следующего:

1. Ссылки на сценарии располагайте ниже DOM, сразу над закрывающим `body` элементом.
2. Если вы не создаете библиотеку для других пользователей, не усложняйте код прослушиванием событий `DOMContentLoaded` или `load`. Прочтите предыдущий пункт.
3. Помечайте сценарии, ссылающиеся на внешние файлы, атрибутом `defer`.
4. Если у вас есть код, не зависящий от загрузки DOM и выполняемый как часть разветвления других сценариев в документе, его можно поместить вверх страницы, снабдив атрибутом `async`.

Вот и все. Думаю, что этих четырех рекомендаций хватит, чтобы в 90 % случаев обеспечить своевременный запуск кода. Для более продвинутых сценариев следует рассмотреть сторонние библиотеки вроде require.js, которые дают больший контроль над временем запуска кода. Если у вас возникнут какие-либо сложности с загрузкой, обращайтесь на https://forum.kirupa.com.

Дополнительные ресурсы и примеры:

- Загрузка модулей с помощью RequireJS: http://bit.ly/kirupaRequireJS
- Предварительная загрузка изображений: http://bit.ly/kirupaPreloadImages



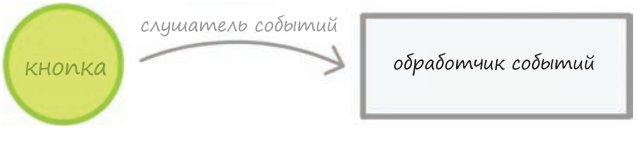
В ЭТОЙ ГЛАВЕ:

- научимся эффективно реагировать на несколько событий;
- напоследок вспомним, как работают события.

36

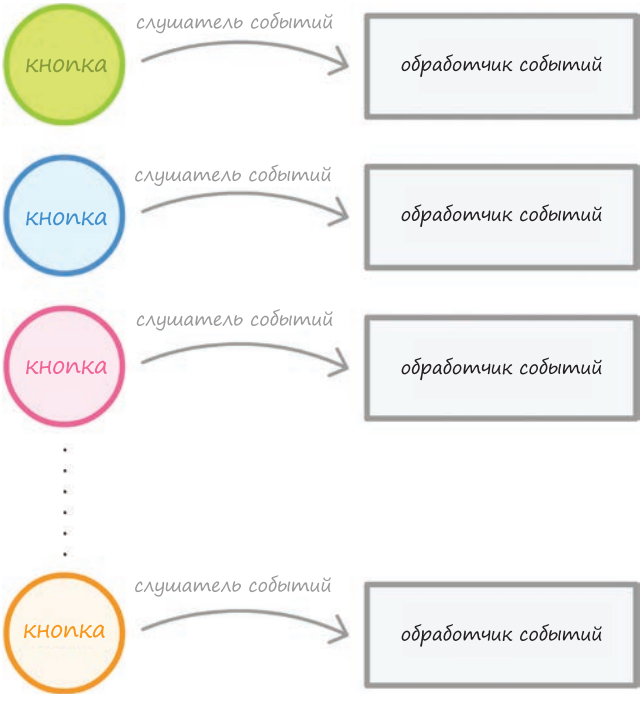
ОБРАБОТКА СОБЫТИЙ ДЛЯ НЕСКОЛЬКИХ ЭЛЕМЕНТОВ

Базово слушатель событий работает с событиями, запускаемыми одним элементом:

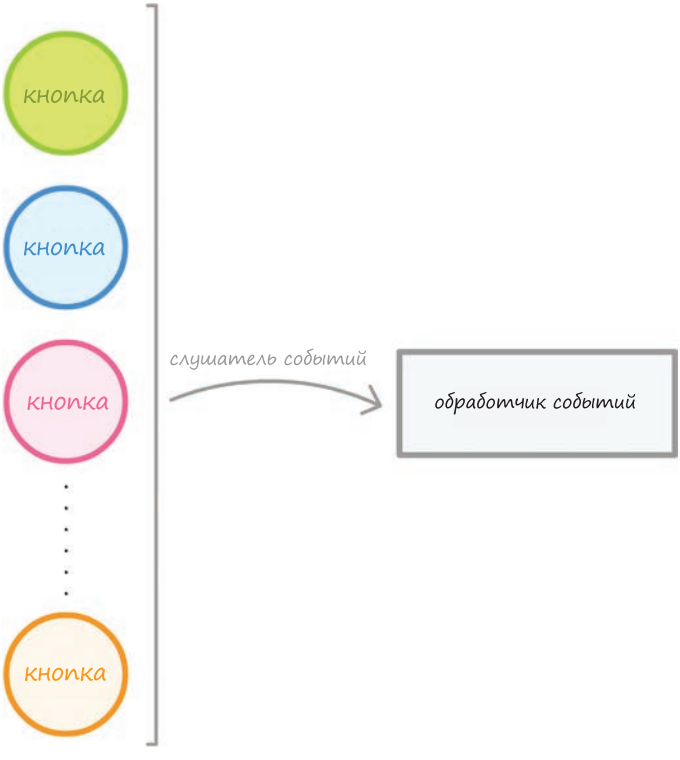


Однако по мере создания более сложных программ отображение «одного обработчика событий для одного элемента» уже не подойдет. Причина в динамическом создании элементов посредством JavaScript. Эти элементы могут запускать события, которые вам может понадобиться прослушать и на которые соответственно среагировать. При этом вам может потребоваться обработка событий как для нескольких элементов, так и для их множества.

Вряд ли вы захотите делать так:



Вам не захочется создавать слушателя событий для каждого элемента, *если слушатель событий для всех них одинаков*. Причина в том, что это непродуктивно. Каждый из этих элементов несет в себе данные об одном и том же слушателе событий и его свойствах, что может существенно увеличить потребление памяти при добавлении большого количества содержимого. Вам, наоборот, нужен чистый и быстрый способ обработки событий для множества элементов с минимальным количеством повторений и ненужных компонентов. Предпочтительный вариант в этом случае будет выглядеть примерно так:



Все это может звучать несколько нереально, не так ли? Что ж, в этой главе вы узнаете, что это вполне нормально, и научитесь реализовывать подобное, используя всего несколько строк кода JavaScript.

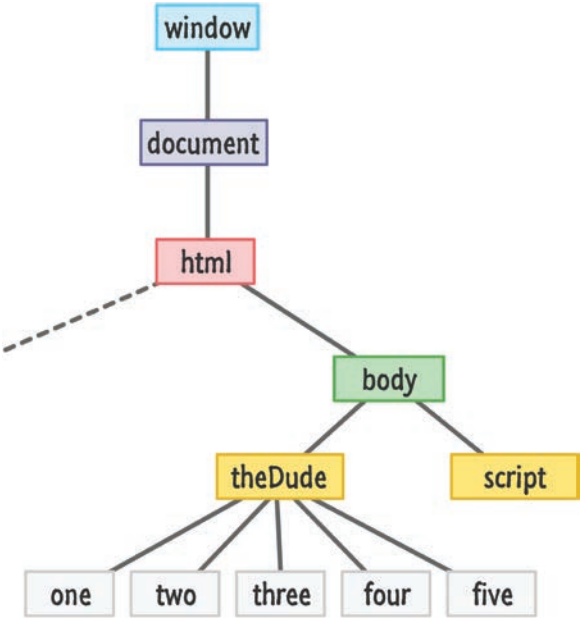
Поехали!

Как все это делается?

Суть в следующем. Вы знаете, как работает простая обработка событий, когда у вас есть один элемент, один слушатель событий и один обработчик событий. Несмотря на то что случай обработки нескольких элементов может казаться иным, воспользовавшись разрывностью событий, разрешить эту проблему достаточно просто.

Представьте, что есть случай, в котором вы хотите прослушивать событие клика в любом из элементов-братьев со значениями `id one`,

two, three, four и five. Давайте дорисуем картину, изобразив DOM следующим образом:



В самом низу расположены элементы, в которых мы хотим прослушивать события. Все они имеют одного родителя в виде элемента с id, равным theDude. Чтобы разрешить проблему обработки этих событий, давайте рассмотрим сначала плохое решение, а затем его удачную альтернативу.

Плохое решение

Так делать не нужно. Мы не хотим создавать слушателя событий для каждой из кнопок:

```
let oneElement = document.querySelector("#one");
let twoElement = document.querySelector("#two");
let threeElement = document.querySelector("#three");
let fourElement = document.querySelector("#four");
let fiveElement = document.querySelector("#five");

oneElement.addEventListener("click", doSomething, false);
twoElement.addEventListener("click", doSomething, false);
```

```
threeElement.addEventListener("click", doSomething, false);
fourElement.addEventListener("click", doSomething, false);
fiveElement.addEventListener("click", doSomething, false);

function doSomething(e) {
  let clickedItem = e.target.id;
  console.log("Hello " + clickedItem);
}
```

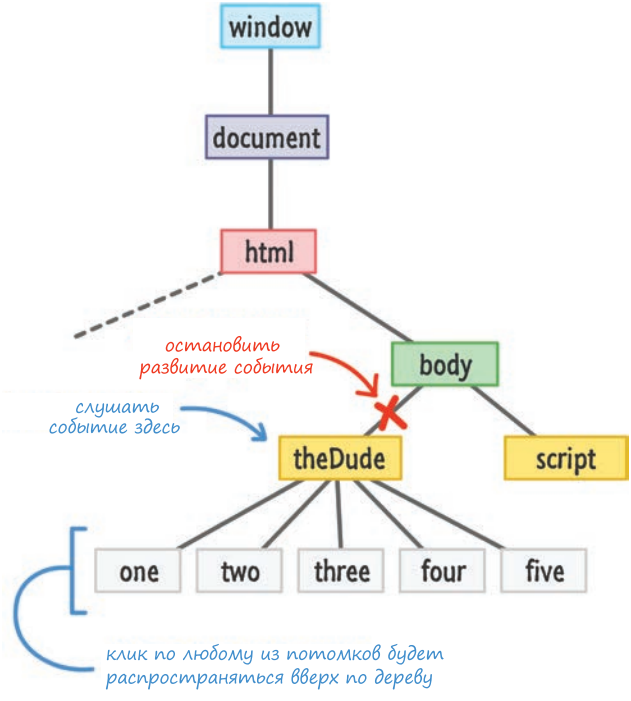
Очевидная причина так не делать — в нежелании повторять код. Другая причина состоит в том, что для каждого элемента теперь установлено свойство `addEventListener`. В случае с пятью элементами это не так страшно. Однако все становится куда серьезнее, когда вы работаете с десятками или сотнями элементов, каждый из которых задействует частичку памяти. Еще одна причина в том, что число элементов может варьировать в зависимости от степени адаптивности или динамичности UI. Ваше приложение может добавлять или удалять элементы в зависимости от действий пользователя, что затруднит отслеживание всех индивидуальных слушателей событий, которые могут потребоваться объекту. Наличие же одного всеобщего обработчика событий существенно упрощает весь этот процесс.

Хорошее решение

Хорошее решение вторит схеме, приведенной ранее, где мы используем всего один слушатель событий. Сначала я вас немного запутаю описанием того, как это работает, а затем попытаюсь распутать, приведя пример кода и подробно пояснив все происходящее. Простое и запутывающее решение:

1. Создать один слушатель событий в родительском элементе `theDude`.
2. Когда произойдет щелчок по любому из элементов `one`, `two`, `three`, `four` или `five`, опереться на поведение распространения, присущее событиям, и прерывать их, когда они достигают элемента `theDude`.
3. (По желанию) Остановить распространение события в родительском элементе, чтобы оно не отвлекало нас своей безудержной беготней по дереву DOM вверх и вниз.

Не знаю, как вы, но я после прочтения этих пунктов точно запутался. Давайте начнем распутываться, обратившись для начала к схеме, более наглядно представляющей описанные действия:



Последним этапом нашего квеста по распутыванию будет код, подробно расписывающий содержимое схемы и все три шага:

```
let theParent = document.querySelector("#theDude");
theParent.addEventListener("click", doSomething, false);

function doSomething(e) {
  if (e.target != e.currentTarget) {
    let clickedItem = e.target.id;
    console.log("Hello " + clickedItem);
  }
  e.stopPropagation();
}
```

Уделите время и внимательно прочитайте и проанализируйте этот код. Приняв во внимание наши изначальные цели и схему, мы будем слушать событие в родительском элементе `theDude`:

```
let theParent = document.querySelector("#theDude");
theParent.addEventListener("click", doSomething, false);
```

Обработкой этого события занимается один обработчик, которым является функция `doSomething`:

```
function doSomething(e) {
  if (e.target != e.currentTarget) {
    let clickedItem = e.target.id;
    console.log("Hello " + clickedItem);
  }
  e.stopPropagation();
}
```

Этот слушатель событий будет вызван каждый раз, когда будет происходить щелчок как в самом элементе `theDude`, так и в любом из его потомков. Нас же интересуют только события щелчка потомков. Правильным способом игнорировать щелчки по родительскому элементу будет просто избежать выполнения любого кода, если элемент, на котором произошел щелчок (то есть целевое событие), совпадает со слушателем событий (то есть элементом `theDude`):

```
function doSomething(e) {
  if (e.target != e.currentTarget) {
    let clickedItem = e.target.id;
    console.log("Hello " + clickedItem);
  }
  e.stopPropagation();
}
```

Цель события представлена `e.target`, а целевой элемент, к которому прикреплен слушатель событий, — `e.currentTarget`. Простая проверка равенства этих событий даст гарантию, что обработчик событий не среагирует на ненужные вам события, запущенные из родительского элемента.

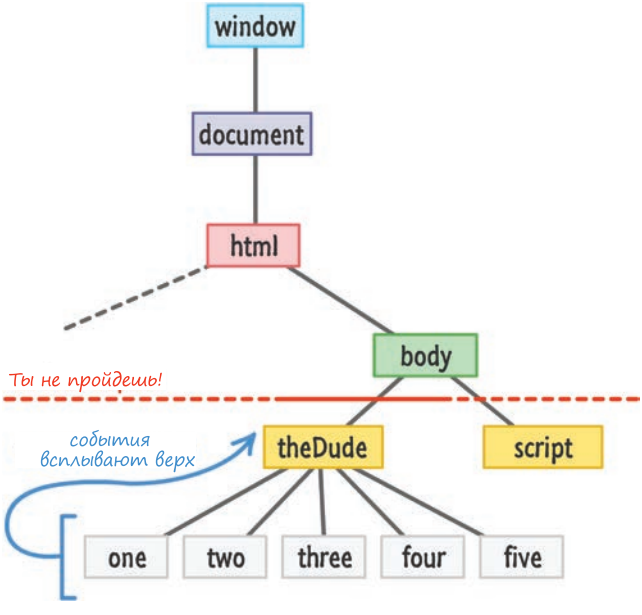
Чтобы остановить распространение события, мы просто вызываем метод `stopPropagation`:

```
function doSomething(e) {
  if (e.target != e.currentTarget) {
    let clickedItem = e.target.id;
    console.log("Hello " + clickedItem);
  }
  e.stopPropagation();
}
```

Обратите внимание, что этот код располагается вне инструкции `if`. Я сделал так, чтобы остановить перемещение события по DOM во всех случаях, как только оно будет услышано.

Объединяя все сказанное

В результате выполнения всего этого кода вы можете щелкнуть по любому потомку `theDude` и прослушать событие при его распространении вверх:



Поскольку все аргументы событий по-прежнему уникальны для элемента, с которым мы взаимодействуем (то есть источника события), мы также можем распознать и выделить нажатый элемент изнутри обработчика событий, несмотря на то что `addEventListener` активна только в родителе. Главное в этом подходе то, что он решает обозначенные проблемы. Вы создали всего один обработчик событий, и не важно, сколько в итоге будет потомков у `theDude`. Этот подход достаточно универсален и способен справиться со всеми ими, не требуя для этого дополнительного изменения кода. Это также значит, что понадобится произвести строгую фильтрацию, если потомками элемента `theDude` в итоге будут не только кнопки, но и другие важные для вас элементы.

КОРОТКО О ГЛАВНОМ

Некоторое время назад я предложил решение загадки этого многоэлементного троеборья (крутые ребята говорят: MEEC), которое оказалось непродуктивным, но при этом не требовало повторения множества строк кода. До тех пор пока многие разработчики не указали мне на его непродуктивность, я считал его вполне рабочим.

В этом решении использовался цикл `for` для прикрепления слушателей событий ко всем потомкам родителя (или массива, содержащего HTML-элементы). Вот как выглядел его код:

```
let theParent = document.querySelector("#theDude");

for (let i = 0; i < theParent.children.length; i++) {
  let childElement = theParent.children[i];
  childElement.addEventListener('click', doSomething, false);
}

function doSomething(e) {
  let clickedItem = e.target.id;
  console.log("Hello " + clickedItem);
}
```

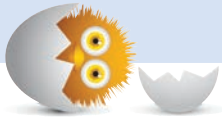
В итоге этот подход позволял прослушивать события щелчка непосредственно в потомках. Единственным, что мне пришлось прописывать вручную, был один вызов слушателя событий, который параметризировался для соответствующего дочернего элемента в зависимости от того, где в цикле находился код:

```
childElement.addEventListener('click', doSomething, false);
```

Причина несостоятельности этого подхода в том, что каждый дочерний элемент имеет связанный с ним слушатель событий. Это возвращает нас к вопросу об эффективности, которая в данном случае страдает от неоправданных затрат памяти.

Если у вас возникнет ситуация, в которой элементы будут разбросаны по DOM, не имея рядом общего родителя, использование этого подхода для массива HTML-элементов будет неплохим способом решения проблемы MEEC.

Как бы то ни было, когда вы начинаете работать с большим количеством элементов UI в играх, приложениях для визуализации данных и прочих насыщенных элементами программах, то будете вынуждены использовать все описанное в этой главе по меньшей мере один раз. Надеюсь, что если все остальное не сработает, то эта глава послужит на славу. При этом весь материал, касающийся перемещения и перехвата событий, рассмотренный ранее, здесь оказался весьма кстати.



В ЭТОЙ ГЛАВЕ:

- похлопаем себя по плечу за проделанную работу;
- еще раз похлопаем себя по плечу просто потому, что это приятно.

37

ЗАКЛЮЧЕНИЕ

Итак, вы справились! Вы наверняка читали без передышки и достигли конца. Как вы себя чувствуете, понимая, что до начала следующего сезона у вас не будет нового материала?

Если усердно читали книгу с самого начала, то согласитесь, что рассмотрели мы очень многое. Начали с такого:

```
<script>
  console.log("hello, world!");
</script>
```

А закончили куда более сложными, полезными и крутыми примерами кода, состоящими из гораздо большего числа строк.

Помните, что писать код легко, а вот писать *элегантный* код, решающий задачи, уже сложнее. Лучше всего это отражено в одном из моих любимых фильмов «Лицо со шрамом», где Тони Монтана произнес следующие слова (за точность не ручаюсь, поскольку зачастую его сложно понять — ну сами знаете, если смотрели фильм):

Эта книга целиком посвящена основам. Следующим шагом станет написание кода, изучение новых приемов и продолжение учебы. В этой книге описаны различные инструменты и приведены примеры их совместного использования для создания небольших программ. Вам осталось лишь вооружиться этими знаниями и применить их для создания более крутых и крупных проектов на JavaScript. Если вам такое по душе, ищите на моем сайте новые продвинутые уроки: **https://www.kirupa.com**.

Так что до скорой встречи, и не стесняйтесь черкнуть пару строк на **kirupa@kirupa.com** или на фейсбук и твиттер (**@Kirupa**). Как я уже говорил, мне нравится получать обратную связь от читателей, поэтому смело обращайтесь. Если у вас есть вопросы, не откладывайте и пишите на форум **https://forum.kirupa.com**.

И еще. Я понимаю, что выбор книг по JavaScript огромен. Поэтому хочу поблагодарить за то, что выбрали именно мою, позволив мне косвенно поприсутствовать в вашем редакторе кода.

Всех вам благ,

ГЛОССАРИЙ

Термины, которые вы будете встречать как в этой книге, так и в работе.

А

Активизировать. Изощренный способ сказать «вызвать функцию».

Аргументы. Значения, которые вы задаете (или передаете) в функцию.

Б

Браузер. Сложное приложение, которое в своем минимуме позволяет взаимодействовать с интернетом и отображает веб-страницы.

В

Возврат. Ключевое слово, производящее выход из функции или блока. В случае функций оно зачастую используется для возврата данных вызвавшему функцию компоненту.

Всплытие события. Фаза, в которой событие начинает подъем от элемента, его инициировавшего, обратно к корню DOM.

Г

Глобальная область видимости. Что-либо объявленное вне функции и доступное для всего приложения.

З

Замыкание. Внутренняя функция, имеющая доступ к переменным внешней функции (помимо своих собственных и глобальных).

Значения. Официальное имя для различных типов данных, с которыми вы работаете.

И

Инструкция if. Условная инструкция, выполняющая заданный код, если условие `true`.

Инструкция if…else. Условная инструкция, выполняющая различные части кода в зависимости от того, является ли условие `true` или `false`.

Инструкция switch. Условная инструкция, проверяющая конкретное состояние в отношении списка случаев. Если один из случаев совпадает с условием, то выполняется связанный с ним код.

Инструменты разработчика. В контексте браузеров они являются расширениями, помогающими инспектировать, отлаживать и диагностировать происходящее на веб-странице изнутри.

К

Каскадная таблица стилей (CSS). Язык таблиц стилей, использующийся в основном для изменения внешнего вида содержимого HTML-страницы.

Комментарии. Текст для людей (зачастую отделенный знаками `//` или `/*` и `*/`), указываемый в коде, который абсолютно игнорируется самим JavaScript.

Л

Логический тип. Структура данных, представляющая `true` или `false`.

Локальная область. Что-либо доступное только внутри функции или блока.

М

Массив. Структура данных, позволяющая хранить последовательность значений и обращаться к ним.

О

Область. Термин, описывающий видимость чего-либо.

Область переменной. Выражение, описывающее видимость переменной в участке кода.

Объект. Очень гибкая и повсеместная структура данных, которую можно использовать для хранения свойств, их значений и даже объектов.

Объектная модель документа (DOM). JavaScript-представление (зачастую древоподобная структура) HTML-страницы и всего ее содержимого.

Операторы. Встроенный функционал вроде `+`, `-`, `*`, `/`, `for`, `while`, `if`, `do`, `=` и других слов.

П

Переменная. Именованный контейнер для хранения данных.

Погружение события. Фаза, в которой событие, начиная от корня DOM, перемещается вниз до элемента, его инициировавшего.

Примитивы. Базовый тип, не состоящий из других типов.

С

Слабая проверка неравенства (`!=`). Проверяет на предмет неравенства только значения двух элементов.

Слабая проверка равенства (`==`). Проверяет на предмет равенства только значения двух элементов.

Слушатель событий. Функция, слушающая событие и затем выполняющая определенный код, как только событие услышано.

Событие. Сигнал, перемещающийся через DOM, сообщая, что что-то произошло.

Строгая проверка неравенства (`!==`). Проверяет, являются ли значение и тип двух элементов неравными.

Строгая проверка равенства (`===`). Проверяет, являются ли значение и тип двух элементов равными.

Строка. Последовательность знаков, составляющих текст. Также является названием типа для работы с текстом в JavaScript.

Т

Тип. Классификация, помогающая распознать ваши данные и значения, которые можно использовать.

У

Узел. Обобщенное имя для элемента DOM.

Ф

Функции-таймеры. Функции, выполняющие код с периодическим интервалом. Наиболее распространенными являются `setTimeOut`, `setInterval` и `requestAnimationFrame`.

Функция. Переиспользуемый блок кода, который получает аргументы, группирует инструкции и может быть вызван для выполнения содержащегося в нем кода.

Ц

Цель события. Элемент, отвечающий за инициацию (запуск) события.

Цикл. Оператор управления для повторяющегося выполнения кода.

Цикл do...while. Оператор управления, выполняющий некий код до тех пор, пока заданное условие не вернет `false`. (Отлично подходит, когда вы не знаете, сколько раз нужно выполнить цикл.)

Цикл for. Оператор управления, выполняющий определенный код ограниченное число раз.

Цикл while. Оператор управления, который продолжительно выполняет заданный код, пока указанное условие не вернет `false`.

J

JavaScript. Капризный и зачастую непоследовательный язык создания сценариев, который, ко всеобщему удивлению, с течением лет набрал популярность в среде разработки приложений как в Сети, так и на сервере.

I

IIFE (функция-выражение, вызываемая сразу после создания). Способ написания кода на JS, позволяющий выполнять заданный код в его собственном диапазоне, не оставляя следов его существования